UNIVERSITY OF CALIFORNIA
SANTA CRUZ

FOUNDATIONS OF ACCESS CONTROL FOR SECURE STORAGE

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Avik Chaudhuri

December 2008

The Dissertation of Avik Chaudhuri
is approved:

_____
Professor Martín Abadi, Chair

_____
Professor Scott Brandt

_____
Professor Cormac Flanagan

_____
Professor Luca de Alfaro

_____
Dean Lisa Sloan
Vice Provost and Dean of Graduate Studies

Table of Contents

Abstract

Foundations of Access Control for Secure Storage

by

Avik Chaudhuri

Over the years, formal techniques have played a significant role in the study of secure communication. Unfortunately, secure storage has received far less attention. In particular, the uses and consequences of dynamic access control for security in file systems, operating systems, and other distributed systems are seldom well-understood. In this dissertation, we develop and apply formal techniques to understand the foundations of access control for security in such systems. Our case studies include the security designs of some state-of-the-art storage systems and operating systems. Our techniques are derived from ideas in programming languages and logic.

To Pops & Mumma.

Acknowledgments

I am very fortunate to have been advised by Martín Abadi in the course of this degree. Martín has been not only a never-ending source of inspiration, but also a remarkably astute coach. I can only hope that his technique, taste, and attitude in research continue to influence my future work.

Moreover, I am very grateful to Sriram Rajamani for making possible an internship at Microsoft Research India, around half-way through my degree; and to Bruno Blanchet for making possible a collaboration almost entirely over email, towards the end. Their enthusiasm and persistence in these ventures have been extraordinary.

Most of the material in this dissertation is based on collaborative papers; I thank my co-authors, including Martín, Sriram, Bruno, as well as Ganesan Ramalingam, Prasad Naldurg, and Lakshmisubrahmanyam Velaga, for their contributions. Moreover, various discussions have influenced the presentation of this material; I thank my dissertation committee members, including Martín, as well as Scott Brandt, Cormac Flanagan, and Luca de Alfaro, for their suggestions. This work has been supported in part by the National Science Foundation under Grants CCR-0204162, CCR-0208800, and CCF-0524078, and by Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under Contract B554869.

Several others have been part of this endeavor in their own ways. My friends—especially those of the unparalleled "baskin-$\star$" groups—have refreshed me with coffee, parties, and random debates whenever I needed them. My lovely wife, Reema, has taken care of pretty much everything else.

Finally, I cannot even begin to describe what I owe to my parents. With all my love, I therefore dedicate this dissertation to them.

Chapter 1

Introduction

Formal techniques have played a significant role in the study of secure communication in recent years. Specifically, there has been much research in developing process calculi, type systems, logics, and other foundations for the rigorous design and analysis of secure communication protocols [Abadi and Fournet, 2001; Abadi and Blanchet, 2003; Gordon and Jeffrey, 2003b; Fournet et al., 2005; Burrows et al., 1989; Blanchet, 2001a]. In comparison, the study of secure storage has received far less formal attention. Yet, over the years storage has assumed a pervasive role in modern computing. Now storage is a fundamental part of most computer systems that we rely on—and understanding secure storage is as important as understanding secure communication.

One might wonder whether the foundations of secure communication already provide those of secure storage—after all, storage is a form of communication. Indeed, one can think of a file f with content M as a channel f that is ready to send message M; then f may be read and written by receiving and sending messages on f. Certainly it would be nice if techniques developed for the study of secure communication could also be applied to study secure storage. In particular, previous work on asymmetric channels (*i.e.*, channels with separate read and write capabilities) should be relevant [Abadi and Blanchet, 2003]. Moreover the use of cryptography for secure communication on untrusted channels is close to its use for secure storage on untrusted servers [Kallahalla et al., 2003]. In general, one might expect at least verification concepts and tools developed for the analysis of communication systems to be useful for the analysis

of storage systems as well. Still, one must be careful about carrying the analogies too far. For example, some notions of forward secrecy in communication via channels may not apply in communication via storage. Undoubtedly there are other examples.

Further, some distinctive features of storage pose problems for security that seem to go beyond those explored in the context of communication protocols. Perhaps the most striking of these features is access control. Indeed, computer systems typically feature access control on store operations, for various reasons linked with security. On the other hand, several aspects of access control do not arise in typical communication protocols. For example, channel communication seldom relies on dynamic access control (such as revocation of permissions). Not surprisingly, such aspects of access control have been largely ignored in formal studies of secure communication. Yet, access control is indispensable for security in a typical storage design. Perhaps the primary reason for this dependence is the potential role of access control as a flexible run-time mechanism for enforcing dynamic specifications. We see an intriguing and challenging research opportunity in understanding the foundations of access control for security in such systems. Briefly, we propose and defend the following thesis:

> A formal understanding of the foundations of access control for secure storage can significantly help in articulating, evaluating, and improving the security of computer systems.

In the remainder of this chapter, we outline how we defend the thesis above. In short, our strategy is to develop and apply formal techniques to specify and verify security properties of a variety of computer systems. Such systems typically rely on access control for security; thus, through this exercise, we lay the foundations of access control for security in such systems. The systems include, in particular, operating systems, file systems, and other distributed systems, whose precise security properties are seldom articulated or enforced rigorously. The techniques build on a rich and mature literature on calculi, semantics, type systems, logics, and other foundations for program verification. Parts of this work appear in [Chaudhuri and Abadi, 2005, 2006b; Chaudhuri, 2006; Chaudhuri and Abadi, 2006a; Chaudhuri et al., 2008a; Blanchet and Chaudhuri, 2008; Chaudhuri et al., 2008c; Chaudhuri, 2008b].

1.1 Access control and secure storage

While secure communication is usually necessary for security in computer systems, it is seldom sufficient. Computer systems often rely on access control for security. For instance, access control plays a role in enforcing run-time security specifications in such systems (even if this role is not explicitly recognized as such). Unfortunately, while secure communication is fairly well understood, several aspects of access control—such as dynamic access control—are not. Clarifying those aspects for security in such systems is an important and challenging research problem.

The key idea behind access control is that accessing a secure object should require some privilege, that can be checked at run time. However, access control may not guarantee security *per se*. For example, it may be possible to circumvent access control. Less drastically:

(a) The implementation of access control may be incorrect. For instance, distributed implementations of access control often rely on cryptographic techniques, and the correctness of such implementations can be fairly tricky.

(b) Access control may not restrict information flow. For instance, a privileged user can inadvertently write confidential information to a publicly readable object, or trust information that is read from a publicly writable object.

Undoubtedly there are other, less important reasons. With some care, however, it should be possible to leverage access control to provide robust security guarantees.

(a) The implementation of access control can be considered correct if it preserves the security properties of some (obviously or provably) correct specification of access control. Information-flow properties like secrecy and integrity that assume the security properties of the specification can be carried over to the implementation.

(b) Information-flow properties can be guaranteed by combining access control with some static analysis. Access control can restrict any unprivileged code that may run in the environment; static analysis can restrict the remaining, privileged code. Their interplay can be exploited to prevent undesirable information flows under an arbitrary environment.

1.2 A research program

The observations above suggest a research program with two complementary directions: in direction (a), focus on the correctness of access controls in a variety of computer systems; in direction (b), show how to exploit such access controls in proofs of information-flow properties.

These directions of work are not necessarily orthogonal. Indeed, for some systems, it may be useful to work on these directions in tandem; for others, it may even be impossible to think of these directions in isolation. Still, these directions are driven by somewhat different concerns.

The motivation for direction (a) stems from the complexity of access-control implementations in contemporary file systems and operating systems. Such complexity is often justifiable in practice; there are various underlying assumptions and guarantees in these systems, and unusual improvisations may be required to meet them. Verifying the correctness of these implementations is typically not straightforward; in fact, formal verification helps understand the nuances of these implementations, uncover potential flaws, and articulate their precise properties.

But correct access control may not be enough for security. The motivation for direction (b) stems from the lack of understanding of the role of access control for security in computer systems. Indeed, without proper care, access control may turn out to be completely ineffective as a security mechanism. Showing how to achieve concrete information-flow properties through access control helps formalize the intended security guarantees of the access-control implementations in such systems.

Roughly, it is this research program that binds our work here. Before plotting an organized view of that work, let us present some highlights, that should give a taste of the systems and techniques involved.

1.3 Some highlights

In this section, we present an assortment of case studies, methodologies, and results that appear in this dissertation. We postpone their organization to Section 1.5.

1.3.1 Security enforcement on operating systems

Commercial operating systems are seldom designed to prevent information-flow attacks. Not surprisingly, such attacks are the source of many serious security problems in these systems. Microsoft's Windows Vista operating system implements an integrity model that can potentially prevent some of those attacks. In some ways, this model resembles classical models of multi-level integrity [Biba, 1977]: every process and object is tagged with an integrity label, the labels are ordered by levels of trust, and Vista enforces access control across trust boundaries. In other ways, it is radically different. While Vista's access control prevents low-integrity processes from writing to high-integrity objects, it does not prevent high-integrity processes from reading low-integrity objects. Further, Vista's integrity labels are dynamic—labels of processes and objects can change at run time. This model allows processes at different trust levels to communicate, and allows dynamic access control. At the same time, it admits various information-flow attacks. Fortunately, any such attack requires the participation of a trusted process; therefore, it is possible to eliminate such attacks by static analysis.

In Chapter 6, we provide a formalization of Vista's integrity model along these lines. (This work appears in [Chaudhuri et al., 2008a].) As a first step, we design a simple higher-order process calculus that emulates Vista's security environment. In this language, processes can fork new processes, create new objects, change the labels of processes and objects, and read, write, and execute objects in exactly the same ways as Vista allows. Then, we specify an information-flow property called *data-flow integrity* (DFI), and present a static type system to enforce DFI in this language.[1] Informally, DFI prevents the flow of data from untrusted code to objects whose contents are trusted; the formal definition requires a semantic technique to track precise sources of values. Our type system relies on Vista's run-time access checks for soundness. The key idea in the type system is to maintain a lower-bound label `S` for each object. While the dynamic label of an object can change at run time, the type system ensures that it never goes below `S`, and the object never contains a value that flows from a label lower than `S`. The

[1] [Castro et al., 2006] specifies and enforces a related data-flow integrity property, by statically extracting data-flow graphs from programs, and instrumenting the programs so that their run-time data flows do not violate those graphs.

label S is declared by the programmer. Typechecking requires no other annotations, and can be mechanized by an efficient algorithm. Further, we discover that while most of Vista's run-time access checks are required to enforce DFI, Vista's execution controls are (in some sense) redundant and can be optimized away.

1.3.2 Automatic analysis of security models

Of course, it is preferable to analyze security models during their design than after their implementation in computer systems. To that end, in Chapter 3 we present EON, a logic-programming language and tool that can be used to automatically analyze dynamic access control models. (This work appears in [Chaudhuri et al., 2008c].) Our language extends standard Datalog with some carefully designed constructs that allow the introduction and transformation of new relations. For example, these constructs can model the creation of processes and objects, and the modification of their security labels at run time. Security properties of such systems can be analyzed by asking queries in this language. We show that query evaluation in this language can be reduced to decidable query satisfiability in a fragment of Datalog, and further, under some restrictions, to efficient query evaluation in standard Datalog. We implement these reductions in our tool, and apply it to study the dynamic access control models of the Windows Vista and Asbestos [Efstathopoulos et al., 2005] operating systems. In particular, we automatically rediscover the attacks against integrity admitted by Vista's security model (which we eliminate by our type system above). We also automatically prove some secrecy properties for the security model of Asbestos, and verify the security of a webserver implemented on Asbestos [Efstathopoulos et al., 2005].

1.3.3 Automated security analysis of storage protocols

Over the years, protocols for secure communication have been studied in depth. In some cases, attacks have been found on old, seemingly robust protocols, and these protocols have been corrected [Denning and Sacco, 1981; Lowe, 1996; Wagner and Schneier, 1996]; in other cases, the security guarantees of those protocols have been found to be misunderstood, and they have been clarified and sometimes even formal-

ized and proved [Abadi and Gordon, 1999; Lowe, 1996; Paulson, 1998]. More generally, this line of work has underlined the difficulty of designing secure communication protocols, and the importance of verifying their precise security properties.

Unfortunately, protocols for secure storage have received far less attention. In Chapter 2, we show that protocols for secure storage are worth analyzing, and study an interesting example. Specifically, we analyze a state-of-the-art file-sharing protocol that exploits cryptographic techniques for secure storage on an untrusted server. (This work appears in [Blanchet and Chaudhuri, 2008].) The protocol is the basis for the file system Plutus [Kallahalla et al., 2003]. This setting is interesting for several reasons. First, compromise of storage servers is a reasonably common threat today, and it is prudent not to trust such servers for security. Next, the protocol we study has a very typical design for secure file sharing on untrusted storage, where data is stored encrypted and signed, and keys for encrypting, signing, verifying, and decrypting such data are managed by users. Several file systems follow this basic design. Finally, beyond the basic design, the protocol features some promising new schemes that facilitate dynamic access control with cryptographic techniques, but in turn complicate its security properties. These schemes are worthy of study in their own right.

Formal techniques play a significant role in our analysis. We model the protocol and verify its security properties in the automatic protocol verifier ProVerif [Blanchet, 2001a, 2002]. ProVerif is based on solid formal foundations that include theory for the applied pi calculus and proof theory for first-order logic. The formal language forces us to specify the protocol precisely, and prove or disprove precise security properties of the protocol. This level of rigor pays off in several ways. We find a new attack against integrity on the protocol, and show that it can have serious practical consequences. That this attack has eluded discovery for more than four years is testimony to the difficulty of finding such attacks "by hand". We propose a patch and prove that it corrects the protocol. Both the attack and the correction are relative to a formal specification of integrity that is not immediately apparent from the informal specification in [Kallahalla et al., 2003]. We also prove a weaker secrecy guarantee than the one claimed in [Kallahalla et al., 2003] (and show that their claim cannot be true). Further, we notice and clarify some ambiguities in [Kallahalla et al., 2003]; we also find some

new, simpler attacks where more complex ones were known. These discoveries vastly improve our understanding of the protocol's subtleties. More generally, they reconfirm that informal justifications (such as showing resistance to specific attacks) are not sufficient for protocols. As far as we know, this study is the first automated formal analysis of a secure storage protocol; we expect our approach to be fruitful for other protocols in this area.

1.3.4 Secure distributed sharing of services

The security architecture of Plutus generalizes quite naturally to an architecture for dynamic sharing of services in a distributed setting. In this architecture, access to services are protected by keys, which are generated and shared by administrators; these keys can be revoked by administrators to dynamically control access to those services. In Chapter 5, we show how to achieve information-flow properties in this setting. (A version of this work appears in [Chaudhuri, 2006]; we introduce some related concepts in Chapter 4, which are developed in more detail in [Chaudhuri and Abadi, 2006b].) As a first step, we develop a variant of Gordon and Hankin's concurrent object calculus [Gordon and Hankin, 1998] with support for flexible access control on methods. We then investigate safe administration and access of shared services in the resulting language. Specifically, we show a type system that guarantees safe manipulation of objects with respect to dynamic specifications, where such specifications are enforced via access changes on the underlying methods at run time. By labeling types with secrecy levels, we show that well-typed systems preserve their secrets amidst dynamic access control and untrusted environments.

1.3.5 Correctness of distributed access-control implementations

Distributed implementations of access control abound in distributed storage protocols. Such implementations are often accompanied by informal justifications of their correctness. However, in Chapter 7, we discover several subtleties in a standard implementation of access control with capabilities [Gobioff et al., 1997], that can undermine correctness under a simple specification of access control. (Some versions of this work

appear in [Chaudhuri and Abadi, 2005, 2006a; Chaudhuri, 2008b].)

We consider both "safety" and "security" for correctness; loosely, safety requires that an implementation does not introduce unspecified behaviors, and security requires that an implementation preserves the specified behavioral equivalences. We show that a secure implementation of a static access policy already requires some care in order to prevent unspecified leaks of information about the access policy. A dynamic access policy causes further problems. For instance, if accesses can be dynamically granted then the implementation does not remain completely secure—it leaks information about the access policy. If accesses can be dynamically revoked then the implementation does not even remain safe. We show that a safe implementation is possible if a clock is introduced in the implementation. A secure implementation is possible if the specification is accordingly generalized.

Our analysis details how formal criteria can guide the systematic design of a distributed implementation from a specification. We show how violations of those criteria can lead to attacks. We distill the key ideas behind those attacks and propose corrections in terms of useful design principles. We show how these principles can guide the derivation of secure distributed implementations of other stateful computations. This approach is reminiscent of secure program partitioning [Zdancewic et al., 2002], and deserves further investigation.

1.4 Ideas and techniques

In the studies above, we rely heavily on ideas and techniques that are founded in programming languages and logic. Let us review some of these influences up front.

1.4.1 Programming languages

The success of any security analysis ultimately depends on the soundness of the abstractions on which the analysis is based. Viewing the underlying system as a programming language can make these abstractions explicit. In particular, such a language pins down the power of the adversary. It also pins down the semantics of the environment in which that analysis is intended to apply. For example, we formalize Windows

Vista's security environment as a higher-order process calculus with references and labels (Chapter 6); the access controls enforced by Windows Vista are burnt into semantics of this language. Likewise, we formalize the Plutus protocol in an applied pi calculus (Chapter 2); the cryptographic and number theoretic algorithms that are used by the protocol are burnt into an equational theory that the language is equipped with. Here, the security of a program in the environment under study is defined in terms of the observable behaviours of that program under an arbitrary context in that environment. Such definitions provide strong, "worst-case" guarantees, since the context can be chosen by the adversary (modulo the language). Standard semantic concepts and proof techniques, such as substitution, reduction, bisimilarity, testing equivalence, and static equivalence, often play a crucial role in such definitions.

Sometimes, it is possible to guarantee the security of programs by static analysis. This approach is particularly attractive for security, since insecure programs can be eliminated at compile time, so that any program that is actually run on the system is guaranteed to be secure. Such an analysis can usually be formalized as a type system. In particular, the type of a program can specify the security invariants of that program. Typing rules combine these invariants with the semantics of the environment to derive other invariants. Finally, type preservation implies that the invariants hold at run time, showing that the analysis is sound. For example, we develop a type system that can enforce data-flow integrity on Windows Vista (Chapter 6). As another example, we develop a type system that can enforce secrecy in a file-system environment (Chapter 4). Here, dependent types are often required to specify security invariants that depend on program values, such as security labels (much as in first-order logic). Moreover, polymorphism is often required to reason about dynamic specifications (Chapter 5).

Going further, a security analysis in an abstract language can be proved sound under a more concrete semantics by showing a compilation that preserves the static semantics of the language. Such proofs allow the analysis to be applied soundly to the more concrete environment. Alternatively, such proofs justify the abstractions on which the analysis may be based. For example, we study the correctness of distributed implementations of access control by reducing them to simpler access control specifications (Chapter 7); analyses that assume such specifications then carry over to the

implementations "for free". As another example, we prove the correctness of a secrecy type system for a pi calculus extended with file-system constructs (Chapter 4) by translation to a sophisticated typed object calculus (Chapter 5); properties of the target type system then apply to the source language "for free". Here, the proofs of correctness are often guided by concepts such as type-preserving compilation, refinement, and full abstraction.

1.4.2 Logic

Some degree of automation is desirable in reasoning about the security of computer systems. For instance, it may be possible to automatically find high-level attacks in some systems, while leaving the discovery of other, low-level attacks to more refined analyses. Usually, the high-level attacks expose either serious design bugs or serious specification errors, so finding them early on can be extremely useful.

A logic can provide a fine basis for such automation. In particular, security models can be encoded as logic programs, and their properties can be studied by executing queries on those programs. For example, the tool ProVerif can automatically analyze cryptographic protocols following this approach; we apply ProVerif to study the security design of the Plutus file system, and discover various design bugs (Chapter 2). Going further, we develop a tool EON that can automatically analyze dynamic access control systems following this approach (Chapter 3); we apply EON to study the security designs of the Windows Vista and Asbestos operating systems, and discover various specification errors.

Many other, potentially useful ideas and techniques from logic are not explored in this dissertation. For instance, a security type system may be implemented in a logical framework, by interpreting types as formulae under the Curry-Howard isomorphism. Typechecking, and even type inference, may be reduced to logical satisfiability and mechanized by standard techniques. Conversely, a security type system may be guided by a security logic, exploiting the Curry-Howard isomorphism in the other direction. For instance, it may be possible to translate proofs in a logic of knowledge and belief to secure (well-typed) code in a language with cryptography. We leave the exploration of these ideas and techniques as future work.

1.5 Organization

We organize this dissertation roughly by the directions of work identified in our research program in Section 1.2. The presentation is divided into three parts.

- In Part I (Chapters 2–3), we focus on correctness of access control, following direction (a) of our program. More precisely, we consider automated techniques to specify and verify the correctness of various access-control implementations that appear in recent file and operating systems. Some of these implementations rely on cryptography, as outlined in Section 1.3.3; others rely on security labels, as outlined in Sections 1.3.1 and 1.3.2. The ideas and techniques in this part are related to logic (Section 1.4.2), and are summarized in Sections 1.3.3 and 1.3.2.

- In Part II (Chapters 4–6), we focus on security via access control, following direction (b) of our program. More precisely, we consider language-based techniques to enforce information-flow security on various computer systems that implement access control. Our analyses are fairly sophisticated; they not only rely on the underlying access controls for soundness, but also exploit them for precision. On the other hand, the targeted security properties are largely standard. The ideas and techniques in this part are related to programming languages (Section 1.4.1), and are summarized in Sections 1.3.1 and 1.3.4.

- Finally, in Part III (Chapter 7), we focus on preserving security by correctness, thereby illustrating how directions (a) and (b) can be tied. More precisely, we consider some powerful techniques to relate the security properties of access control implementations to their specifications; the implementations are correct only if they preserve the security properties of their specifications. Some of these implementations rely on cryptography and distribution, as outlined in Section 1.3.5. Their correctness makes it possible to reason about their security properties by analyzing those of their specifications, by the methods developed in Part II. The ideas and techniques in this part are strongly influenced by programming languages (Section 1.4.1), and are summarized in Section 1.3.5.

We outline related work and discuss our contributions in Chapter 8.

1.5.1 Dependencies

Despite their organization in parts, the chapters in this dissertation are mostly self-contained, so that they can be read in any order. Still, there are some indirect relationships between these chapters that may be helpful to see up front. We list some of the more obvious ones below; others appear in Section 1.5.2.

Chapter 4 ↔ Chapter 5 The type system of Chapter 5 is based partly on that of Chapter 4. Both type systems exploit access controls to guarantee secrecy. Further, some type constructs, such as secrecy groups, and the associated subtyping rules are shared by these type systems. Finally, the soundness proof for the type system of Chapter 4 is obtained by translation to that of Chapter 5.

Chapter 4 ↔ Chapter 7 The type system of Chapter 4 can be used to analyze specifications of the kind considered in Chapter 7. Moreover, since any security guarantees for such specifications carry over to correct implementations (such as those studied in Chapter 7), we can effectively use the above type system to analyze those implementations as well.

Chapter 3 ↔ Chapter 6 The type system of Chapter 6 is based partly on some insights gained in Chapter 3, from a logic-based analysis of the security design of Windows Vista with EON. In particular, that analysis discovers some high-level attacks, and develops a coarse discipline that can provably eliminate them. The type system of Chapter 6 further refines this discipline, to not only eliminate those attacks, but also do so more precisely.

Chapter 2 ↔ Chapter 5 The language of Chapter 5 can be viewed as describing a generalization of the setting of Chapter 2, following the outline in Section 1.3.4. Accordingly, it should be possible to apply the type system of Chapter 5 to analyze programs in this setting.

1.5.2 Common themes

Finally, some common themes run throughout this dissertation. Let us close this chapter by briefly discussing them; they should become apparent as we progress.

Dynamic effects A common observation in our studies is that access control can both complicate and improve security in non-trivial ways. Such sophistication seems to stem from the intrinsic dynamic effects of access control. Indeed, on the surface, access control is about dynamic checks; but more deeply, it is about dynamic constraints that influence those checks. For instance, access control can involve not only checking permissions at run time, but also revoking or granting those permissions at run time. Such flexibility has both pros and cons, and our analyses must be sensitive enough to exploit or avoid them.

Hybrid analyses Access controls can be viewed either as mechanisms or as policies. Some static analyses guarantee the success of access checks at run time, thereby allowing them to be optimized away; such analyses view access controls as policies, and aim to show conformance with those policies. Other analyses instead rely on the failure of access checks at run time; such analyses view access controls as mechanisms, and aim to show soundness of those mechanisms.

In this dissertation, we usually (but not always) adopt the latter view; we consider access controls as mechanisms to achieve information-flow properties such as secrecy and integrity. Our type systems for such properties rely on access controls for soundness, and exploit them for precision. This approach is similar in spirit to hybrid typechecking [Flanagan, 2006]—dynamic checks are used where possible or as required to complement static checks. Moreover, this approach is particularly relevant for the systems we study, since access controls are intended mainly as security mechanisms in these systems.

Decidability issues In any sufficiently expressive model of computation, most security questions of interest become undecidable. Thus, any general technique to answer such questions automatically are forced to choose between soundness and completeness. For example, ProVerif is sound, but incomplete—there are security questions for which it may not return decisive answers, or even terminate. Still, such tools can be quite successful, by relying on carefully chosen abstractions; for instance, they can be used to prove the absence of attacks, or warn about possible attacks (that may or may not correspond to real attacks). On the

other hand, sometimes it is possible to restrict the expressive power of the computational model so that security questions in that model become decidable. For instance, we design EON to be both sound and complete. Such tools either find real attacks or altogether prove their absence.

Somewhat similar choices arise in the design of type systems for security. The type systems are usually conservative; they always reject bad programs, and may sometimes reject good programs. Despite these abstractions, typechecking may still be undecidable. For instance, typechecking may involve "guessing" some types that cannot be infered automatically. Usually, it is possible to recover decidability of typechecking by requiring further annotations, or otherwise restricting the type system. We favor such type systems, because they allow automatic code certification (at least in principle).

Security properties Various security properties may be of interest in a particular system. These properties may range from simple secrecy and integrity properties, that consider only explicit (data) flows, to stronger "hyperproperties" [Clarkson and Schneider, 2008] such as noninterference [Goguen and Meseguer, 1982], that consider also implicit (control) flows.

Sometimes, it may be more reasonable to enforce weaker properties at the level of file and operating systems, while allowing stronger properties to be enforced at the level of specific applications, as necessary. For instance, the weaker properties may be less sensitive to modeling artifacts, and thus easier to preserve by translation. We follow this approach in several of our type systems. On the other hand, sometimes we find it useful to consider stronger properties, since counterexamples to such properties can expose unexpected information leaks in implementations.

We defer a more detailed discussion on these themes and their manifestations in our work to Chapter 8.

Part I

Correctness of Access Control

Overview

In this part, we focus on automated techniques for analyzing access control implementations in computer systems. Our main interests are specifying and verifying security properties of such implementations. Logic programming techniques seem to be particularly suitable for our purposes. We investigate and apply these techniques to study the security designs of some recent file and operating systems.

We begin by studying security properties of a state-of-the-art protocol for secure file sharing on untrusted storage, in the automatic protocol verifier ProVerif (Chapter 2). ProVerif translates the protocol into a Prolog-style program, and uses a resolution-based algorithm to prove or refute these properties. As far as we know, this is the first automated analysis of a secure storage protocol. The protocol itself, designed as the basis for the file system Plutus, features some interesting schemes for dynamic access control. These schemes complicate its security properties. Our analysis clarifies several ambiguities in the design and reveals some unknown attacks on the protocol. We propose corrections, and prove precise security guarantees for the corrected protocol.

While ProVerif is a powerful tool, it is necessarily incomplete—it is not guaranteed to produce definite results, or even terminate, on all inputs. For example, certain access control models that appear in operating systems seem difficult to analyze with ProVerif. To decidably analyze such models, next we develop a specialized logic-programming language and tool called EON (Chapter 3). Our language extends Datalog with some carefully designed constructs that allow the introduction and transformation of new relations. For example, these constructs can model the creation of processes and objects, and the modification of their security labels at run time. Security properties of such systems can be analyzed by asking queries in this language. We show that query evaluation in EON can be reduced to decidable query satisfiability in a fragment of Datalog, and further, under some restrictions, to efficient query evaluation in Datalog. We implement these reductions in our tool, and demonstrate its scope through several examples. In particular, we study the dynamic access control models of the Vista and Asbestos operating systems. We also automatically verify the design of a secure webserver running on Asbestos.

Chapter 2

Cryptographic access control

Much research in recent years has focused on the security analysis of communication protocols. In some cases, attacks have been found on old, seemingly robust protocols, and these protocols have been corrected [Denning and Sacco, 1981; Lowe, 1996; Wagner and Schneier, 1996]; in other cases, the security guarantees of those protocols have been found to be misunderstood, and they have been clarified and sometimes even formalized and proved [Abadi and Gordon, 1999; Lowe, 1996; Paulson, 1998]. More generally, this line of work has underlined the difficulty of designing secure communication protocols, and the importance of verifying their precise security properties.

While protocols for secure communication have been studied in depth, protocols for secure storage have received far less attention. In this chapter, we show that such protocols are worth analyzing, and study an interesting example. Specifically, we analyze a state-of-the-art file-sharing protocol that exploits cryptographic techniques for secure storage on an untrusted server. The protocol is the basis for the file system Plutus [Kallahalla et al., 2003]. This setting is interesting for several reasons:

- First, compromise of storage servers is a reasonably common threat today, and it is prudent not to trust such servers for security [Mazières and Shasha, 2002].
- Next, the protocol we study has a very typical design for secure file sharing on untrusted storage, where data is stored encrypted and signed, and keys for encrypting, signing, verifying, and decrypting such data are managed by users.

Access control is enforced via suitably restricting the distribution of those keys. Several file systems follow this basic design, including SNAD [Miller et al., 2002], SiRiUS [Goh et al., 2003], and other cryptographic file systems dating back to the 1990s [Blaze, 1993].

- Finally, beyond the basic design, the protocol features some interesting schemes such as *lazy revocation* and *key rotation*, to improve the protocol's performance in the presence of dynamic access control (see Section 2.1). These features are worthy of study. For instance, our analysis reveals that lazy revocation allows more precise integrity guarantees than a more naïve scheme [Goh et al., 2003]. On a different note, the computational security of key rotation schemes has generated a lot of interest recently [Backes et al., 2005, 2006; Fu et al., 2006]. Our analysis reveals some new integrity vulnerabilities in the protocol that can be exploited even if the key rotation scheme is secure.

Formal techniques play a significant role in our analysis. We model the protocol and verify its security properties in the automatic protocol verifier ProVerif [Blanchet, 2001a, 2002, 2008]. ProVerif is based on solid formal foundations that include theory for the applied pi calculus and proof theory for first-order logic. The formal language forces us to specify the protocol precisely, and prove or disprove precise security properties of the protocol. This level of rigor pays off in several ways:

- We find a new integrity attack on the protocol, and show that it can have serious practical consequences. That this attack has eluded discovery for more than four years is testimony to the difficulty of finding such attacks "by hand".

- We propose a patch and prove that it corrects the protocol. Both the attack and the correction are relative to a formal specification of integrity that is not immediately apparent from the informal specification in [Kallahalla et al., 2003]. We also prove a weaker secrecy guarantee than the one claimed in [Kallahalla et al., 2003] (and show that their claim cannot be true).

- The formal exercise allows us to notice and clarify some ambiguities in [Kallahalla et al., 2003]; it also allows us to find some new, simpler attacks where more

complex ones were known. These discoveries vastly improve our understanding of the protocol.

- Finally, the use of an automatic verifier yields a much higher level of confidence in our proofs than manual techniques, which have been known to be error-prone.

The rest of the chapter is organized as follows. In Section 2.1, we outline the protocol behind Plutus. In Section 2.2, we give an overview of ProVerif, and present our model of Plutus in ProVerif. Finally, in Section 2.3, we specify and analyze secrecy and integrity properties of Plutus in ProVerif, and present our results and observations. We assume some familiarity with basic cryptographic functions, such as those for encrypting, hashing, and signing, in this chapter; see [Goldwasser and Bellare, 2001] for an introduction to these functions.

2.1 Plutus

The file system Plutus [Kallahalla et al., 2003] is based on a storage design that does not rely on storage servers to provide strong secrecy and integrity guarantees. Instead, contents of files are cryptographically secured, and keys for writing and reading such contents are managed by the owners of those files. Special schemes are introduced to economize key distribution and cryptography in the presence of dynamic access control; those schemes complicate the protocol and its security properties.

In Plutus, principals are qualified as owners, writers, and readers. Every file belongs to a group[1], and all files in a group have the same writers and readers. The owner of a group generates and distributes keys for writing and reading contents for that group; those keys are shared by all files in that group.

Specifically, a *write key* is used to encrypt and sign contents, while a *read key* is used to verify and decrypt such contents. These keys can be revoked by the owner to dynamically control access to those files; a new write key and a new read key are then generated and distributed appropriately. However, the new write key is used only for

[1]There is a difference between the informal interpretation of a group in [Kallahalla et al., 2003], and the formal interpretation of a group in this chapter. In fact, the interpretation in [Kallahalla et al., 2003] is inconsistent; see Section 2.3.4 for a more detailed discussion of this issue.

subsequent writes: unlike SiRiUS [Goh et al., 2003], the files are not immediately secured with the new write key, so that the previous read key can be used to verify and decrypt the contents of those files until they are re-written. This scheme, called *lazy revocation*, avoids redundant cryptography, and is justified as follows:

- Encrypting the existing contents with the new write key would not guarantee secrecy of those contents from the previous readers, since those contents may have been cached by the previous readers.
- More subtly, since the existing contents come from the previous writers, signing those contents with the new write key would wrongly indicate that they come from the new writers. (With lazy revocation, if an untrusted writer is revoked, readers can distinguish contents that are written after the revocation from previous contents that may have been written by that writer; consequently, they can trust the former contents even if they do not trust the latter contents.)

Going further, a scheme called *key rotation* allows the new readers to derive the previous read key from the new read key, avoiding redundant key distribution. (Thus, the new readers do not need to maintain the previous read key for reading the existing contents.) In contrast, the new read key cannot be derived from the previous read key, so contents that are subsequently written with the new write key can only be read by the new readers.

Concretely, a write key is of the form (sk, lk), where sk is part of an asymmetric key pair (sk, vk), and lk is a symmetric encryption key; the complementary read key is (vk, lk). Here sk, vk, and lk are a *sign key*, a *verify key*, and a *lockbox key*. Contents are encrypted with lk[2] and signed with sk; those contents are verified with vk and decrypted with lk.

Plutus uses the RSA cryptosystem [Rivest et al., 1978], so we have $sk = (d, n)$ and $vk = (e, n)$, where the modulus n is the product of two large primes p and q, and the exponents d and e are inverses modulo $(p-1)(q-1)$, that is, $ed \equiv 1 \bmod (p-1)(q-1)$.

[2]More precisely, contents are divided into blocks, and each block is encrypted with a fresh key; these keys are in turn stored in a "lockbox" that is encrypted with lk. In this chapter, we consider for simplicity that the contents are directly encrypted with lk; we have checked that our results continue to hold with the details of the lockbox.

The pair (p, q) is called the RSA seed. Note that the functions $x \mapsto x^d \bmod n$ and $y \mapsto y^e \bmod n$ are inverses. Given a hash function hash, a message M is signed with sk by computing $S = \mathsf{hash}(M)^d \bmod n$, and S is verified with vk by checking that $S^e \bmod n = \mathsf{hash}(M)$.

In general, e may be chosen randomly, relatively prime to $(p-1)(q-1)$, and d may be computed from e, p, and q. However in Plutus, e is uniquely determined by n and lk as follows: given a pseudo-random sequence $\langle r_i \rangle$ generated with seed lk, e is the first prime number in the sequence $\langle r_i + \sqrt{n} \rangle$. We denote this algorithm by $\mathsf{genExp}(n, lk)$.

To sum up, a sign/verify key pair (sk, vk) is generated from a random RSA seed (p, q) and a lockbox key lk, by computing $n = pq$, $e = \mathsf{genExp}(n, lk)$, $vk = (e, n)$, and $sk = (d, n)$, where d is the inverse of e modulo $(p-1)(q-1)$.

The owner of a group distributes (sk, lk) to writers and lk to readers; users can further derive vk from n and lk using genExp. Note that n is already available to writers from sk. Further, the owner distributes a signed n to writers, which they attach whenever they write contents to the file system—so *any* user can obtain n from the file system and verify its authenticity. Thus writers can act for readers in Plutus, although in [Kallahalla et al., 2003] it is wrongly claimed that writers cannot derive vk (implying that read access is disjoint from writer access). It is already known that writers can act for readers in SiRiUS in a similar way [Goh et al., 2003; Naor et al., 2005].

Let (D, N) and (E, N) be the private key and the public key of the owner of a group. The initial and subsequent versions of keys for writers and readers of that group are generated as follows:

Version 0 The initial lockbox key lk_0 is random, and the initial sign/verify key pair (sk_0, vk_0) is generated from a random RSA seed (with modulus n_0) and lk_0.

Version v to version $v+1$ When keys for version v are revoked, a new lockbox key lk_{v+1} is generated by "winding" the previous lockbox key lk_v with the owner's private key, as $lk_{v+1} = lk_v^D \bmod N$. The previous lockbox key can be retrieved by "unwinding" the new lockbox key with the owner's public key, as $lk_v = lk_{v+1}^E \bmod N$. In particular, a reader with a lockbox key $lk_{v'}$ for any $v' \geq v$ can generate the verify key vk_v by obtaining the modulus n_v from the file system, re-

cursively unwinding $lk_{v'}$ to lk_v, and deriving vk_v from n_v and lk_v using genExp. The new sign/verify key pair (sk_{v+1}, vk_{v+1}) is generated from a random RSA seed (with modulus n_{v+1}) and lk_{v+1}.

While storage servers are not trusted to provide strong secrecy and integrity guarantees, there is still a degree of trust placed on servers to prevent unauthorized modification of the store by a scheme called *server-verified writes*. Specifically, the owner of a group generates a fresh *write token* for each version, and distributes that token to the writers of that version and to the storage server. The server allows a writer to modify the store only if the correct write token is presented to the server; in particular, revoked writers cannot revert the store to a previous state, or garbage the current state.

2.2 Formal model of Plutus

In order to study Plutus formally, we rely on the automatic protocol verification tool ProVerif. We briefly present this tool next, and then describe our model of Plutus.

2.2.1 Background on ProVerif

The tool ProVerif [Abadi and Blanchet, 2005; Blanchet, 2001a, 2002, 2008] is designed to verify security protocols. The protocol is specified in an extension of the pi calculus with cryptography, a dialect of the applied pi calculus [Abadi and Fournet, 2001]. The desired security properties can be specified, in particular, as correspondence assertions [Woo and Lam, 1993], which are properties of the form "if some event has been executed, then other events have been executed". (We illustrate this input language below.) Internally, the protocol is translated into a set of Horn clauses,[3] and the security properties are translated into derivability queries on these clauses: the properties are proved when certain facts are not derivable from the clauses. ProVerif uses a resolution-based algorithm to show this non-derivability.

ProVerif relies on the formal, so-called Dolev-Yao model of protocols [Dolev and Yao, 1983], in which messages are modeled as terms in an algebra. This rather abstract

[3]Informally, a Horn clause is a logical rule, possibly quantified over some variables, that allows the inference of a certain fact from some other facts.

model of cryptography makes it easier to automate proofs than the more concrete, computational model, in which messages are modeled as bitstrings. Consequently, ProVerif can handle a wide variety of cryptographic primitives specified by rewrite rules or equations over terms. Moreover:

- When ProVerif proves a property, the proof is valid for an unbounded number of sessions of the protocol and an unbounded message size.
- When the proof fails, ProVerif provides a derivation of a fact, and tries to reconstruct, from this derivation, a trace of the protocol that shows that the property is false [Allamigeon and Blanchet, 2005]. When trace reconstruction fails, ProVerif gives no definite answer. Such a situation is unavoidable due to the undecidability of the problem. Fortunately, in our study, whenever this situation happens, manual inspection of the derivation provided by ProVerif allows us to reconstruct an attack against the said property: the failure of the ProVerif proof always corresponds to an attack.

See [Blanchet, 2008] for detailed information on ProVerif and its foundations.

2.2.2 Plutus in ProVerif

We now present a model of Plutus in ProVerif; its security properties are specified and studied in Section 2.3.

2.2.2.1 Cryptographic primitives, lists, and integers

We abstract cryptographic primitives with function symbols, and specify their properties with rewrite rules and equations over terms. The term $\mathsf{enc}(M, K)$ denotes the result of encrypting message M with symmetric key K; and the rewrite rule

$$\mathsf{dec}(\mathsf{enc}(x, y), y) \to x$$

models the fact that any term of the form $\mathsf{enc}(M, K)$ can be decrypted with K to obtain M. (Here x and y are variables that can match any M and K.) The term $\mathsf{hash}(M)$ denotes the hash of message M. The term $\mathsf{exp}(M, (R, N))$ denotes the result of computing

$M^R \bmod N$. We abstract random RSA seeds as fresh names. The term $\mathsf{N}(s)$ denotes the modulus of seed s. The term $\mathsf{e}(s, K)$ denotes the unique exponent determined by the modulus $\mathsf{N}(s)$ and base K by the algorithm described in Section 2.1; this fact is modeled by the rewrite rule:

$$\mathsf{genExp}(\mathsf{N}(x), y) \rightarrow \mathsf{e}(x, y)$$

The term $\mathsf{d}(s, K)$ is the inverse exponent, as explained in Section 2.1. This fact is modeled by the equations:

$$\begin{aligned} \mathsf{exp}(\mathsf{exp}(z, (\mathsf{d}(x, y), \mathsf{N}(x))), (\mathsf{e}(x, y), \mathsf{N}(x))) &= z \\ \mathsf{exp}(\mathsf{exp}(z, (\mathsf{e}(x, y), \mathsf{N}(x))), (\mathsf{d}(x, y), \mathsf{N}(x))) &= z \end{aligned}$$

Finally, the rewrite rule

$$\mathsf{crack}(\mathsf{e}(x, y), \mathsf{d}(x, y), \mathsf{N}(x)) \rightarrow x$$

models the fact that a modulus $\mathsf{N}(s)$ can be efficiently "factored" to obtain the RSA seed s if both exponents $\mathsf{e}(s, K)$ and $\mathsf{d}(s, K)$ are known [Boneh, 1999].

We model sets of allowed writers and readers with lists: nil is the empty list, and $\mathsf{cons}(M, L)$ is the extension of the list L with M; we have $\mathsf{member}(N, L)$ if and only if N is a member of the list L. Likewise, we model version numbers with integers: zero is 0, and the integer $\mathsf{succ}(M)$ is the successor of the integer M; we have $\mathsf{geq}(N, M)$ if and only if the integer N is greater than or equal to the integer M. The following clauses define the predicates member and geq in ProVerif.

$$\begin{aligned} &\mathsf{member}(x, \mathsf{cons}(x, y)); \\ &\mathsf{member}(x, y) \;\Rightarrow\; \mathsf{member}(x, \mathsf{cons}(z, y)). \end{aligned}$$

$$\begin{aligned} &\mathsf{geq}(x, x); \\ &\mathsf{geq}(x, y) \;\Rightarrow\; \mathsf{geq}(\mathsf{succ}(x), y). \end{aligned}$$

For brevity, we write $0, 1, \ldots$ for $\mathsf{zero}, \mathsf{succ}(\mathsf{zero}), \ldots$; $M \geq N$ for $\mathsf{geq}(M, N)$; and $M \in L$ for $\mathsf{member}(M, L)$.

2.2.2.2 The protocol

We model principals as applied pi-calculus processes with events [Blanchet, 2008]. Informally:

- $\mathbf{out}(u, M); P$ sends the message M on a channel named u and continues as the process P; a special case is the process $\mathbf{out}(u, M)$, where there is no continuation.
- $\mathbf{in}(u, X); P$ receives a message M on a channel named u, matches M with the pattern X, and continues as the process P with variables in X bound to matching terms in M. Here X may be a variable x, which matches any message and stores it in x; a pattern $=N$, which matches only the message N; or even a more complex pattern like $(=N, x)$, which matches any pair whose first component is N and stores its second component in x.
- **new** $m; P$ creates a fresh name m and continues as the process P.
- **event** $e(M_1, \ldots, M_n); P$ executes the event $e(M_1, \ldots, M_n)$ and continues as the process P. A special case is the process **event** $e(M_1, \ldots, M_n)$, where there is no continuation. The execution of $e(M_1, \ldots, M_n)$ merely records that a certain program point has been reached for certain values of $M_1, \ldots, M_n$. Such events are used for specifying security properties, as explained in Section 2.3.1.
- **if** $M = M'$ **then** P **else** Q executes P if M evaluates to the same term as M'; otherwise it executes Q. A special case is the process **if** $M = M'$ **then** P, where there is no **else** continuation.
- **let** $X = M$ **in** P evaluates M, matches it with the pattern X and, when the matching succeeds, continues as P with the variables in X bound to matching terms in the value of M.
- $P \mid Q$ runs the processes P and Q in parallel.
- $!P$ runs an unbounded number of copies of the process P in parallel.

Below, we define processes that model the roles of owners, writers, and readers; the protocol is specified as the parallel composition of these processes. (The storage server

is assumed to be untrusted at this point, and therefore not modeled. We study server-verified writes and their properties later.) The network is modeled by a public channel net; as usual, we assume that the adversary controls the network. Likewise, the file system is modeled by a public channel fs. On the other hand, private (secure) channels are not available to the adversary. For instance, rprivchannel(r) and wprivchannel(w) are private channels on which an owner sends keys to reader r and writer w, respectively.

We limit the number of revocations that are possible in any group to $\max_{\text{rev}}$. (Thus the number of versions is bounded. At this level of detail, ProVerif does not terminate with an unbounded number of versions. We managed to obtain termination with an unbounded number of versions for a more abstract treatment of cryptography, thanks to an extension of ProVerif that takes advantage of the transitivity of geq in order to simplify the Horn clauses. However, we do not present that abstract model here because it misses some of the attacks that are found with the more detailed model below.)

First, we show the code for owners. An owner creates its private/public key pair (lines 2–5), and then creates groups on request (lines 7–9). For each group, the owner maintains some state on a private channel *currentstate*. (The current state is carried as a message on this channel, and the owner reads and writes the state by receiving and sending messages on this channel.) The state includes the current version number, the lists of allowed readers and writers, the lockbox key, and the sign key for that group. The owner creates the initial version of keys for the group (lines 12–14), generates at most $\max_{\text{rev}}$ subsequent versions on request (lines 17–21), and distributes those keys to the allowed readers and writers on request (lines 25–30 and 34–40). The generation and distribution of keys follow the outline in Section 2.1. Moreover, the owner signs the modulus of each version with its private key (line 38), sends the signed modulus to writers of that version (line 40), and sends its public key to readers so that they may verify that signature (line 30). Events model runtime assertions in the code: for instance, isreader(r, g, v) and iswriter(w, g, v) assert that r is a reader and w is a writer for group g at version v.

---

1 **let** *processOwr* =

2 **new** *seed1*; **new** *seed2*; *(* create owner's RSA key pair *)*

```
3   let ownerpubkey = (e(seed1, seed2), N(seed1)) in
4   let ownerprivkey = (d(seed1, seed2), N(seed1)) in
5   out(net, ownerpubkey);                                   (* publish owner's RSA public key *)
6   (
7   ! in(net, (= newgroup, initreaders, initwriters));       (* receive a new group creation request;
        initreaders and initwriters are the initial lists of allowed readers and writers, respectively *)
8    new g;                                                  (* create the new group g *)
9    out(net, g);                                            (* publish the group name g *)
10   new currentstate;                                       (* create a private channel for the current state for group g *)
11   (
12    ( new initlk;                                          (* create initial lk *)
13      new seed3; let initsk = (d(seed3, initlk), N(seed3)) in   (* generate initial sk *)
14      out(currentstate, (zero, initreaders, initwriters, initlk, initsk))
                                                            (* store state for version 0 on channel currentstate *)
15    )
16    |                                                     (* Next, we move from version 0 to version 1 *)
17    ( in(net, (= revoke, = g, newreaders, newwriters));   (* receive a revoke request for group g;
                                newreaders and newwriters are the new lists of allowed readers and writers *)
18      in(currentstate, (= zero, oldreaders, oldwriters, oldlk, oldsk));   (* read state for version 0 *)
19      let newlk = exp(oldlk, ownerprivkey) in             (* wind old lk to new lk *)
20      new seed3; let newsk = (d(seed3, newlk), N(seed3)) in   (* generate new sk *)
21      out(currentstate, (succ(zero), newreaders, newwriters, newlk, newsk))
                                                            (* store state for version 1 on channel currentstate *)
22    )
23    | ... |                                               (* Similarly, we move from version 1 to version 2, and so on *)
24    (
25    ! in(net, (= rkeyreq, r, = g));                       (* receive read key request for reader r and group g *)
26      in(currentstate, (v, readers, writers, lk, sk));    (* get the current state *)
27      out(currentstate, (v, readers, writers, lk, sk));
28      if member(r, readers) then                          (* check that the reader r is allowed *)
29       ( event isreader(r, g, v);                         (* assert that r is a reader for group g and version v *)
30         out(rprivchannel(r), (g, v, lk, ownerpubkey)) )  (* send lk and owner's public key to r *)
31    )
32    |
33    (
```

```
34      ! in(net, (= wkeyreq, w, = g));          (* receive write key request for writer w and group g *)
35        in(currentstate, (v, readers, writers, lk, sk));                    (* get the current state *)
36        out(currentstate, (v, readers, writers, lk, sk));
37        if member(w, writers) then                          (* check that the writer w is allowed *)
38         ( let (_, n) = sk in let sn = exp(hash(n), ownerprivkey) in          (* sign the modulus *)
39           event iswriter(w, g, v);          (* assert that w is a writer for group g and version v *)
40           out(wprivchannel(w), (g, v, lk, sk, sn)))          (* send lk, sk, and signed modulus to w *)
41      )
42     )
43   ).
```

Next, we show the code for writers. A writer for group g at version v obtains the lockbox key, the sign key, and the owner-signed modulus for v from the owner of g (lines 46–47). To write data, an honest writer encrypts that data with the lockbox key (line 50), signs the encryption with the sign key (line 51), and sends the signed encryption to the file system with a header that includes the owner-signed modulus (lines 52–54). The event $\mathsf{puts}(w, M, g, v)$ asserts that an honest writer w for group g sends data M to the file system using keys for version v. In contrast, a dishonest writer leaks the lockbox key, the sign key, and the owner-signed modulus (line 59); the adversary can use this information to act for that writer. The event $\mathsf{corrupt}(w, g, v)$ asserts that a writer w for group g is corrupt at version v.

```
44  let processWtr =
45   ! in(net, (w, g));                                  (* initiate a writer w for group g *)
46     out(net, (wkeyreq, w, g));                                  (* send write key request *)
47     in(wprivchannel(w), (= g, v, lk, sk, sn));                  (* obtain lk, sk, and signed modulus *)
48     (
49      ( new m;                                                   (* create data to write *)
50        let encx = enc(m, lk) in                                                  (* encrypt *)
51        let sencx = exp(hash(encx), sk) in                                          (* sign *)
52        event puts(w, m, g, v);  (* assert that data m has been written by w for group g at version v *)
53        let (dx, n) = sk in
```

```
54      out(fs, (g, v, n, sn, encx, sencx))                    (* send content to file system *)
55    )
56    |
57    ( in(net, = (corrupt, w));                                (* receive corrupt request for w *)
58      event corrupt(w, g, v);          (* assert that w has been corrupted for group g at version v *)
59      out(net, (lk, sk, sn))                                  (* leak lk, sk, and signed modulus *)
60    )
61  ).
```

Finally, we show the code for readers. A reader for group g at version v obtains the lockbox key for v from the owner of g (lines 64–65). To read data, an honest reader obtains content from the file system (line 67), and parses that content to obtain a signed encryption and a header that contains g, a version number vx, and a signed modulus. It verifies the signature of the modulus with the owner's public key (line 68); it then generates the verify key for vx from the modulus and the lockbox key (lines 69–71), verifies the signature of the encryption with the verify key (line 72), and decrypts the encryption with the lockbox key (line 73). The generation of the verify key for vx from the modulus for vx and the lockbox key for v follows the outline in Section 2.1: the lockbox key lk for vx is obtained from the lockbox key for v by unwinding it $v - vx$ times (line 70), after which genExp generates the required exponent (line 71). Below we detail only the case where $v = 1$ and $vx = 0$ (lines 69–75), in which case we unwind the lockbox key once (line 70); the ProVerif script includes a similar block of code for each $vx \leq v \leq \max_{\text{rev}}$, located at line 76 and omitted here. The event $\mathsf{gets}(r, x, g, vx)$ asserts that an honest reader r for group g receives data x from the file system using keys for version vx. In contrast, a dishonest reader leaks the lockbox key (line 81); the adversary can use this information to act for that reader. The event $\mathsf{corrupt}(r, g, v)$ asserts that a reader r in group g is corrupt at version v.

```
62  let processRdr =
63    ! in(net, (r, g));                                        (* initiate a reader r for group g *)
64      out(net, (rkeyreq, r, g));                              (* send read key request *)
```

```
65  in(rprivchannel(r), (= g, v, lk, ownerpubkey));          (* obtain lk and owner's public key *)
66  (
67   ( in(fs, (= g, vx, n, sn, encx, sencx));               (* obtain header and content from file system *)
68     if hash(n) = exp(sn, ownerpubkey) then               (* verify signature in header *)
69     ( if (v, vx) = (succ(zero), zero) then
70       ( let lk = exp(lk, ownerpubkey) in                 (* unwind lk *)
71         let vk = (genExp(n, lk), n) in                   (* derive vk *)
72         if hash(encx) = exp(sencx, vk) then              (* verify signature of encryption*)
73         let x = dec(encx, lk) in                         (* decrypt to obtain data *)
74         event gets(r, x, g, vx)      (* assert that reader r read data x for group g and version vx *)
75       )
76       ...
77     )
78  |
79   ( in(net, = (corrupt, r));                             (* receive corrupt request for r *)
80     event corrupt(r, g, v);          (* assert that r has been corrupted for group g at version v *)
81     out(net, lk)                                         (* leak lk *)
82   )
83  ).
```

2.3 Security results on Plutus

We now specify secrecy and integrity properties of Plutus in ProVerif, and verify those properties (showing proofs or attacks) using ProVerif. We propose corrections where attacks are possible, and clarify several security-relevant details of the design along the way.

2.3.1 Background on correspondences

Properties of the protocol are specified as correspondences [Woo and Lam, 1993]. The verifier ProVerif can prove such correspondences [Blanchet, 2008]. A simple example is the correspondence

$$e(M_1, \ldots, M_n) \rightsquigarrow e'(M'_1, \ldots, M'_{n'})$$

which means that in any trace of the protocol in the presence of an adversary, the event $e(M_1, \ldots, M_n)$ must not be executed unless the event $e'(M'_1, \ldots, M'_{n'})$ is executed. More generally, correspondences may include equality tests of the form $M = M'$, atoms of the form $pred(M_1, \ldots, M_n)$ that rely on user-defined predicates *pred* (such as geq and member), and atoms of the form $\mathsf{attacker}(M)$, which mean that the attacker knows the term M.

Definition 2.3.1 (Correspondences)**.** *Let $\mathcal{T}$ range over traces, σ over substitutions, and ϕ over formulas of the form* $\mathsf{attacker}(M)$*, $e(M_1, \ldots, M_n)$, $pred(M_1, \ldots, M_n)$, $M = M'$, $\phi_1 \wedge \phi_2$, or $\phi_1 \vee \phi_2$.*

- *$\mathcal{T}$ satisfies* $\mathsf{attacker}(M)$ *if the message M has been sent on a public channel in $\mathcal{T}$.*
- *$\mathcal{T}$ satisfies $e(M_1, \ldots, M_n)$ if the event $e(M_1, \ldots, M_n)$ has been executed in $\mathcal{T}$.*
- *$\mathcal{T}$ satisfies $M = M'$ if $M = M'$ modulo the equations that define the function symbols.*
- *$\mathcal{T}$ satisfies $pred(M_1, \ldots, M_n)$ if the atom $pred(M_1, \ldots, M_n)$ is true.*
- *$\mathcal{T}$ satisfies $\phi_1 \wedge \phi_2$ if $\mathcal{T}$ satisfies both ϕ_1 and ϕ_2.*
- *$\mathcal{T}$ satisfies $\phi_1 \vee \phi_2$ if $\mathcal{T}$ satisfies ϕ_1 or $\mathcal{T}$ satisfies ϕ_2.*

Let an Init-adversary be an adversary whose initial knowledge is Init. A process P satisfies the correspondence $\phi \rightsquigarrow \phi'$ against Init-adversaries if and only if, for any trace $\mathcal{T}$ of P in the presence of an Init-adversary, for any substitution σ, if $\mathcal{T}$ satisfies $\sigma\phi$, then there exists a substitution σ' such that $\sigma'\phi = \sigma\phi$ and $\mathcal{T}$ satisfies $\sigma'\phi'$ as well.

In a correspondence $\phi \rightsquigarrow \phi'$, the variables of ϕ are universally quantified (because σ is universally quantified), and the variables of ϕ' that do not occur in ϕ are existentially quantified (because σ' is existentially quantified). ProVerif can prove correspondences $\phi \rightsquigarrow \phi'$ of a more restricted form, in which ϕ is of the form $\mathsf{attacker}(M)$ or $e(M_1, \ldots, M_n)$. This corresponds to the formal definition of correspondences proved by ProVerif given in [Blanchet, 2008, Definition 3], except for two extensions: we allow atoms of the form $\mathsf{attacker}(M)$, $M = M'$, and $pred(M_1, \ldots, M_n)$ to occur in ϕ' and we do not require that ϕ' be in disjunctive normal form.

In order to prove correspondences, ProVerif translates the process and the actions of the adversary into a set of Horn clauses $\mathcal{R}$. In these clauses, messages are represented by *pure terms*[4] p, which are terms in which names a have been replaced with functions $a[\ldots]$. Free names are replaced with constants $a[\,]$, while bound names created by restrictions are replaced with functions of the messages previously received and of session identifiers that take a different value at each execution of the restriction—so that different names are represented by different pure terms. The clauses use the following kinds of facts:

- $\mathsf{attacker}(p)$, which means that the adversary may have the message p;
- $\mathsf{message}(p, p')$, which means that the message p' may be sent on channel p;
- $\mathsf{event}(e(p_1, \ldots, p_n))$, which means that the event $e(p_1, \ldots, p_n)$ may have been executed;
- $\mathsf{m\text{-}event}(e(p_1, \ldots, p_n))$, which means that the event $e(p_1, \ldots, p_n)$ must have been executed;
- the facts $\mathsf{geq}(p, p')$ and $\mathsf{member}(p, p')$, which are defined in Section 2.2.2.1.

The clauses that define geq and member are shown in Section 2.2.2.1. The other clauses in $\mathcal{R}$ are generated automatically by ProVerif from the process and from the definitions of the function symbols; see [Blanchet, 2008, Section 5.2] for details. ProVerif establishes security properties by proving that certain facts are derivable from these clauses only if certain hypotheses are satisfied. The derivability properties are determined by a resolution-based algorithm, described in [Blanchet, 2008, Section 6]. Specifically, ProVerif computes a function $\mathrm{solve}_{P,Init}(F)$ that takes as argument a process P, the initial knowledge of the adversary *Init*, and a fact F, and returns a set of Horn clauses that determines which instances of F are derivable. More precisely, let $\mathcal{F}_{\mathrm{me}}$ be any set of m-event facts, which are supposed to hold. An instance F_0 of F is derivable from $\mathcal{R} \cup \mathcal{F}_{\mathrm{me}}$ if and only if there exist a clause $H \Rightarrow C$ in $\mathrm{solve}_{P,Init}(F)$ and a substitution σ_0 such that $F_0 = \sigma_0 C$ and the facts in $\sigma_0 H$ are derivable from $\mathcal{R} \cup \mathcal{F}_{\mathrm{me}}$. In

[4]Note that such terms are called "patterns" in [Blanchet, 2008]. Here, we prefer to call them "pure terms" to avoid confusion with the patterns X in Section 2.2.2.2.

particular, if $\text{solve}_{P,Init}(F) = \emptyset$, then no instance of F is derivable from $\mathcal{R} \cup \mathcal{F}_{\text{me}}$ for any $\mathcal{F}_{\text{me}}$. Other values of $\text{solve}_{P,Init}(F)$ give information on which instances of F are derivable and under which conditions. In particular, the m-event facts in the hypotheses of clauses in $\text{solve}_{P,Init}(F)$ must be in $\mathcal{F}_{\text{me}}$ in order to derive an instance of F (since $\mathcal{R}$ contains no clause that concludes m-event facts), so the corresponding events must have been executed.

We can then prove the following theorem, which provides a technique for establishing correspondences.

Theorem 2.3.2 (Correspondences)**.** *Let P be a closed process. Let $\phi \rightsquigarrow \phi'$ be a correspondence, where ϕ is* $\mathsf{attacker}(M)$ *or* $e(M_1, \ldots, M_n)$*. Let $F =$* $\mathsf{attacker}(p)$ *if $\phi =$* $\mathsf{attacker}(M)$ *and $F =$* $\mathsf{event}(e(p_1, \ldots, p_n))$ *if $\phi = e(M_1, \ldots, M_n)$, where $p, p_1, \ldots, p_n$ are the pure terms obtained from the terms $M, M_1, \ldots, M_n$ respectively, by replacing names a with pure terms $a[\,]$. Let ψ' be the formula obtained from ϕ' by replacing names a with pure terms $a[\,]$.*

Suppose that, for all $H \Rightarrow C \in \text{solve}_{P,Init}(F)$, there exists a substitution σ such that $C = \sigma F$ and $H \vdash \sigma\psi'$, where

- *$H \vdash e(p_1, \ldots, p_n)$ if and only if* $\mathsf{m\text{-}event}(e(p_1, \ldots, p_n)) \in H$
- *$H \vdash p = p'$ if and only if $p = p'$ modulo the equations that define the function symbols.*
- *$H \vdash pred(p_1, \ldots, p_n)$ (where pred is a user-defined predicate or* $\mathsf{attacker}$*) if and only if $pred(p_1, \ldots, p_n)$ is derivable from the facts in H, the clauses that define user predicates, the clauses that express the initial knowledge of the adversary, and the clauses that express that the adversary can apply functions.*
- *$H \vdash \psi_1 \wedge \psi_2$ if and only if $H \vdash \psi_1$ and $H \vdash \psi_2$*
- *$H \vdash \psi_1 \vee \psi_2$ if and only if $H \vdash \psi_1$ or $H \vdash \psi_2$.*

Then P satisfies the correspondence $\phi \rightsquigarrow \phi'$ against Init-adversaries.

This theorem is an extension of [Blanchet, 2008, Theorem 4] to the case in which ϕ' may contain atoms $\mathsf{attacker}(M)$, $M = M'$, and $pred(M_1, \ldots, M_n)$, and ϕ' may not be in disjunctive normal form. Intuitively, if $\mathcal{T}$ satisfies $\sigma_M\phi$, then $\sigma_p F$ is derivable, where σ_p

is the substitution on pure terms that corresponds to the substitution on terms σ_M. So there exist a clause $H \Rightarrow C$ in $\mathrm{solve}_{P,Init}(F)$ and a substitution σ_0 such that $\sigma_p F = \sigma_0 C$ and the facts $\sigma_0 H$ are derivable. Since $H \vdash \sigma\psi'$, we also have $\sigma_0\sigma\psi'$. Moreover, $C = \sigma F$, so $\sigma_p F = \sigma_0\sigma F$. So, letting $\sigma'_p = \sigma_0\sigma$, we have $\sigma_p F = \sigma'_p F$ and $\sigma'_p\psi'$, so $\sigma_M\phi = \sigma'_M\phi$ and $\mathcal{T}$ satisfies $\sigma'_M\phi'$, where σ'_M is the substitution on terms that corresponds to the substitution σ'_p on pure terms. Hence the correspondence $\phi \rightsquigarrow \phi'$ is satisfied.

In this chapter, we use the more general language of correspondences of Definition 2.3.1, and show how to exploit the more limited queries that ProVerif can prove in order to prove the correspondences that we need.

2.3.2 Security properties of Plutus

We study secrecy and integrity properties of Plutus by specifying correspondences in ProVerif. Our security proofs with ProVerif assume $\max_{\mathrm{rev}} = 5$, that is, they apply to a model where at most five revocations are possible for any group. The attacks assume $\max_{\mathrm{rev}} = 1$, and remain *a fortiori* valid for any $\max_{\mathrm{rev}} \geq 1$. Running times of ProVerif appear later in the section. Recall that ProVerif does not terminate at this level of detail if the number of versions is unbounded. Nevertheless, we expect the results below to hold in that case as well.

2.3.2.1 Secrecy

We begin with secrecy. Specifically, we are interested in the secrecy of some fresh data m written by an honest writer for group g using keys for version v. We cannot expect m to be secret if a dishonest reader for g at v colludes with the adversary at v—but is it necessary that such a reader collude with the adversary in order to leak m? In order to determine that, we tentatively specify secrecy as follows: a secret m written by an honest writer for g at v is leaked only if a reader for g is corrupt at v, *i.e.*, the process modeling Plutus satisfies the correspondence

$$\mathsf{puts}(w, m, g, v) \ \wedge \ \mathsf{attacker}(m) \ \rightsquigarrow \\ \mathsf{corrupt}(r, g, v) \ \wedge \ \mathsf{isreader}(r, g, v)$$

Unfortunately, here writers can act for readers (see Section 2.1), so a corrupt writer at v leaks (at least) as much information as a corrupt reader at v. Note that on the contrary, it is intended in [Kallahalla et al., 2003] that read access be disjoint from write access. Moreover, since the read key for v can be obtained from the read key for any $v' \geq v$ by unwinding, even a corrupt reader (or writer) at such v' leaks as much information as a corrupt reader at v. Of course, if the set of readers does not increase, a reader at v' is already a reader at v, so this situation is not surprising. (Indeed, this is the case that motivates key rotation in [Kallahalla et al., 2003].) On the other hand, increasing the set of readers may result in unintended declassification of secrets. In light of these observations, we must weaken our specification of secrecy.

Definition 2.3.3 (Secrecy)**.** *Secrecy is preserved if, for all g and v, any secret m written by an honest writer for g using keys for v is leaked only if a reader or writer for g is corrupt at some $v' \geq v$,* i.e.*, the model satisfies the correspondence*

$$\begin{aligned} &\mathsf{puts}(w, m, g, v) \ \wedge \ \mathsf{attacker}(m) \ \rightsquigarrow \\ &\quad v' \geq v \ \wedge \ \mathsf{corrupt}(a, g, v') \\ &\quad \wedge \ (\mathsf{isreader}(a, g, v') \ \vee \ \mathsf{iswriter}(a, g, v')) \end{aligned}$$

This weaker property is proved as follows.

Theorem 2.3.4. *Secrecy is preserved by Plutus.*

Proof. Let $m[g = G, v = V]$ denote the name m created in line 49 when the variables g and v in lines 45 and 47 are bound to the terms G and V, respectively. (This notation can be used directly in ProVerif, exploiting ProVerif's internal representation of bound names by pure terms. It is detailed and justified in [Blanchet, 2008].) ProVerif automatically proves the following correspondence:

$$\begin{aligned} &\mathsf{attacker}(m[g = x_g, v = x_v]) \ \rightsquigarrow \\ &\quad v' \geq x_v \ \wedge \ \mathsf{corrupt}(a, x_g, v') \\ &\quad \wedge \ (\mathsf{isreader}(a, x_g, v') \ \vee \ \mathsf{iswriter}(a, x_g, v')) \end{aligned}$$

By the semantics of the language, for any terms W, M, G, and V, if $\mathsf{puts}(W, M, G, V)$ is executed, then $M = m[g = G, v = V]$. Thus, for all substitutions σ, if a trace $\mathcal{T}$

satisfies $\sigma\mathsf{puts}(w, x_m, x_g, x_v)$ and $\sigma\mathsf{attacker}(x_m)$, then $\sigma x_m = \sigma m[g = x_g, v = x_v]$; so $\mathcal{T}$ satisfies $\sigma\mathsf{attacker}(m[g = x_g, v = x_v])$; so by correspondence 2.1, $\mathcal{T}$ satisfies $\sigma'(v' \geq x_v \wedge \mathsf{corrupt}(a, x_g, v') \wedge (\mathsf{isreader}(a, x_g, v') \vee \mathsf{iswriter}(a, x_g, v')))$ for some substitution σ' such that $\sigma' x_g = \sigma x_g$ and $\sigma' x_v = \sigma x_v$. Hence, correspondence 2.1 is satisfied. ◀

2.3.2.2 Integrity

Next, we specify an integrity property. Specifically, we are interested in the integrity of some data x read by an honest reader r for group g using keys for version v. We expect x to come from the adversary if a dishonest writer for g at v colludes with the adversary at v; otherwise, we expect x to be written by an honest writer w for g using keys for version v. Moreover, such w must be a writer for g at v.

Definition 2.3.5 (Integrity)**.** *Integrity is preserved if for all g and v, any data x read by an honest reader for g using keys for v is written by an honest writer for g using keys for v unless a writer for g is corrupt at v,* i.e., *the model satisfies the correspondence*

$$\begin{aligned}&\mathsf{gets}(r, x, g, v) \rightsquigarrow \\ &\quad \mathsf{iswriter}(w, g, v) \\ &\quad \wedge (\mathsf{puts}(w, x, g, v) \vee \mathsf{corrupt}(w, g, v))\end{aligned}$$

Unfortunately, when we try to show that integrity is preserved by Plutus, ProVerif cannot prove the required correspondence for this model. Manual inspection of the derivation output by ProVerif reveals an attack, where the adversary is able to send data to an honest reader for group g at version 0 without corrupting a writer for g at 0.

Theorem 2.3.6. *Integrity is not preserved by Plutus.*

Proof. ProVerif cannot prove the correspondence in Definition 2.3.5; it outputs a derivation of $\mathsf{gets}(r, m, g, 0)$ from facts that do not include $\mathsf{puts}(w, m, g, 0)$ or $\mathsf{corrupt}(w, g, 0)$ for any w, and we manually check that this derivation corresponds to an attack. Briefly, a reader for g is corrupted at version 0 and a writer for g is corrupted at version 1; the adversary then constructs a bogus write key for version 0 and writes content that can be read by r using the read key for version 0. In more detail:

1. A reader for g is corrupted at version 0 to get the lockbox key lk_0 for version 0.

2. Next, a writer for g is corrupted at version 1 to get the lockbox key lk_1, the sign key $(\mathsf{d}(s_1, lk_1), \mathsf{N}(s_1))$, and the owner-signed modulus $sn_1 = \mathsf{exp}(\mathsf{hash}(\mathsf{N}(s_1)),$ *ownerprivkey*$)$ for version 1; here s_1 is the RSA seed for version 1 and *ownerprivkey* is the private key of the owner.

3. The exponent $\mathsf{e}(s_1, lk_1)$ is computed as $\mathsf{genExp}(\mathsf{N}(s_1), lk_1)$.

4. Next, the RSA seed s_1 is computed as $\mathsf{crack}(\mathsf{e}(s_1, lk_1), \mathsf{d}(s_1, lk_1), \mathsf{N}(s_1))$.

5. Now a bogus sign key sk' is constructed as $(\mathsf{d}(s_1, lk_0), \mathsf{N}(s_1))$.

6. Choosing some fresh data m, the following content is then sent to the file system, where $M = \mathsf{enc}(m, lk_0)$:

$$(g, 0, sn_1, \mathsf{N}(s_1), M, \mathsf{exp}(\mathsf{hash}(M), sk'))$$

7. An honest reader r for g reads m using keys for version 0, without detecting that the modulus in the sign key is in fact not the correct one!

Note that corrupting a reader for g at version 0 to obtain lk_0 is not a necessary step in the above attack; the adversary can instead compute lk_0 from lk_1 by unwinding. Orthogonally, the adversary can collude with a writer for a different group at version 0, instead of corrupting a writer for group g at version 1. In each case, a bogus sign key for the target group and version may be constructed from an unrelated modulus because the correct group and version of that modulus is not verified in this model. ◄

The above attack can have serious consequences, since it implies that *a writer for an arbitrary group can act as a legitimate writer for a target group simply by colluding with a reader for that group*. Here, we consider a model without server-verified writes, that is, we assume that the server is compromised and colludes with the adversary. As argued in [Mazières and Shasha, 2002; Goh et al., 2003], server compromise is a realistic possibility, so the above attack can be quite damaging. Worse, integrity is not preserved even in a model extended with server-verified writes. However with server-verified

writes, the consequences are less serious—in order to write data for a group, the adversary needs to obtain the current write token for that group, for which it needs to corrupt a current writer for that group. Still, the attack has the same undesirable effect as allowing rotation of write keys. Specifically, it allows a corrupt writer at a later version to modify data in such a way that readers date the modified data back to an earlier version; in other words, the modified data appears to be older than it actually is to readers. This situation can be dangerous. Suppose that a reader trusts all writers at version 0, but not some writer at version 1 (say because the corruption of that writer at version 1 has been detected and communicated to the reader). The reader may still trust data written at version 0. However, the above attack shows that such data cannot be trusted: that data may in fact come from a corrupt writer at version 1.

We propose a simple PATCH to correct the protocol: owners must sign each modulus with its correct group and version. More concretely, the term bound to *sn* at line 38 of the code for owners must be $\mathsf{exp}(\mathsf{hash}(n, g, v), \mathit{ownerprivkey})$, and conversely, line 68 of the code for readers must check that $\mathsf{hash}(n, g, v) = \mathsf{exp}(\mathit{sn}, \mathit{ownerpubkey})$. The corrected model preserves integrity as shown by Theorem 2.3.7 below. (Moreover, Theorem 2.3.4 continues to hold for the corrected model, with an unchanged proof.)

Theorem 2.3.7. *Integrity is preserved by Plutus with* PATCH.

Proof. ProVerif now automatically proves the correspondence in Definition 2.3.5. ◀

2.3.2.3 Strong integrity

While Definition 2.3.5 restricts the source of data read by honest readers, it still allows the adversary to replay stale data from a cache; in particular, content written by a writer at version v may be cached and replayed by the adversary at a later version v', when that writer is revoked. Unfortunately, in the model above we cannot associate contents that are read from the file system with the versions at which they are written to the file system. Such associations are possible only if the file system is (at least partially) trusted, as with server-verified writes.

Below we specify a stronger integrity property that we expect to hold in a model with server-verified writes; the property not only restricts the source of data read by

honest readers, but also requires that such data be fresh. The code for the extended model is included at the end of this chapter. Briefly, we define a process to model the storage server, and extend the code for owners so that for any group g, a new write token is created for each version v, communicated to the server, and distributed to writers for g at v. Corrupt writers leak their write tokens. A writer must send contents to the server with a token; the contents are written to the file system only if that token is verified by the server to be the write token for the current version. Honest readers securely obtain server-verified contents from the server. (Of course, those contents are also publicly available from the server.) To verify the stronger integrity property, we replace the event $\mathsf{gets}(r, x, g, vx)$ in the code for readers (line 74) with a more precise event $\mathsf{gets}(r, x, g, vx, v')$. The latter event subsumes the former, and further asserts that the relevant contents are written to the file system after server-verification at v'. We expect that $v' = vx$, where vx is the version of keys used to read those contents, unless a writer for g is corrupt at v'; in the latter case, the adversary is able to replay at v' data that is originally written using keys for vx, so we may have $v' \geq vx$.

Definition 2.3.8 (Strong integrity)**.** *Strong integrity is preserved if for all g and v, any data x read by an honest reader for g using keys for v is written by an honest writer for g using keys for v, unless a writer for g is corrupt at v; and further, such data is written either at v or at some version $v' \geq v$ at which a writer is corrupt,* i.e.*, the model satisfies the correspondence*

$$\begin{aligned}
&\mathsf{gets}(r, x, g, v, v') \rightsquigarrow \\
&\quad \mathsf{iswriter}(w, g, v) \\
&\quad \wedge (\mathsf{puts}(w, x, g, v) \vee \mathsf{corrupt}(w, g, v)) \\
&\quad \wedge (v' = v \vee (v' \geq v \wedge \mathsf{iswriter}(w', g, v') \wedge \mathsf{corrupt}(w', g, v')))
\end{aligned}$$

The corrected, extended model preserves strong integrity, as expected. Once again, the proof is automatic.

Theorem 2.3.9. *Strong integrity is preserved by Plutus with server-verified writes &* PATCH*.*

Proof. ProVerif now automatically proves the correspondence in Definition 2.3.8. ◀

Further, we show (using a correspondence omitted here) the correctness of server-verified writes: for any group g, only writers for g at the current version v can write

	Without PATCH	With PATCH				
$\max_{\text{rev}}$	1	1	2	3	4	5
Without server-verified writes	0:01	0:01	0:02	0:05	0:14	0:40
With server-verified writes	0:05	0:03	0:17	1:19	7:14	42:05

Figure 2.1: Running times of ProVerif

data for g at v. (Such writes must be authorized by the current write token for g, which is distributed only to the current writers for g.) Consequently, server-verified writes prevent at least two kinds of attacks:

- Unauthorized writers cannot destroy data by writing junk over such data.
- Revoked writers cannot roll back new data by writing data with old keys over such data.

2.3.2.4 Running times of ProVerif

Figure 2.1 presents the running times of ProVerif 1.14pl4 for the scripts above, in "minutes:seconds" format, on a 2.6 GHz AMD machine with 8 GB memory. We test models with or without PATCH, and with or without server-verified writes. We already find attacks assuming $\max_{\text{rev}} = 1$ for models without PATCH. On the other hand, models with PATCH are tested assuming $\max_{\text{rev}} \leq 5$, so our security proofs apply only to those models (although we expect them to hold with larger values of $\max_{\text{rev}}$ as well). Memory usage increases significantly with server-verified writes; for example, the script with $\max_{\text{rev}} = 5$, PATCH, and server-verified writes takes around 2.2 GB of memory. For $\max_{\text{rev}} = 6$, ProVerif runs out of memory on this 8 GB machine.

2.3.3 Analysis of some design details

Next, using ProVerif, we clarify some design details of Plutus.

2.3.3.1 Why should a new modulus be created for each version?

The following explanation is offered by [Kallahalla et al., 2003]:

. . . the reason for changing the modulus after every revocation is to thwart a collusion attack. . . a revoked writer can collude with a reader to become a valid writer. . .

We formalize this attack as a violation of integrity by Plutus: if the modulus for version 1 is the same as that for version 0, the adversary is able to send data to an honest reader for group g at version 1 without corrupting a writer for g at 1. We manually reconstruct the attack.

1. A writer for g is corrupted at version 0, and a reader for g is corrupted at version 1. Thus the adversary obtains the lockbox key lk_0 and sign key (d_0, n) for version 0, and the lockbox key lk_1 for version 1. We may assume that the writer corrupted at 0 is revoked at 1. Let there be another writer for g at version 1 that publishes some content, so that the adversary also knows the owner-signed header sn_1 for version 1.

2. The adversary computes the exponent $e_0 = \mathsf{genExp}(n, lk_0)$, the RSA seed $s = \mathsf{crack}(e_0, d_0, n)$, and the sign key $sk_1 = (\mathsf{d}(s, lk_1), \mathsf{N}(s))$ for version 1. (Since the modulus n is unchanged, the RSA seed s is the same for versions 0 and 1.) Finally, choosing some fresh data m the adversary sends the following content to the file system, where $M = \mathsf{enc}(m, lk_1)$:

$$(g, 1, sn_1, n, M, \mathsf{exp}(\mathsf{hash}(M), sk_1))$$

3. An honest reader for g reads m using keys for version 1.

However, we have two comments on this attack:

- With server-verified writes, the sentence of [Kallahalla et al., 2003] quoted above is not quite true: in order to become a valid writer, one additionally needs to obtain a write token at some version $v \geq 1$, which can be done only by corrupting a writer at some version $v \geq 1$.

- But by corrupting a writer at version $v \geq 1$, the adversary can mount a much simpler attack: the adversary can compute the RSA seed s and all keys for version

1 from the keys for such v, without corrupting a writer at version 0 or a reader at version 1! We reconstruct a simple attack along these lines by modifying the ProVerif script so that the modulus is not changed between versions and inspecting the derivation output by ProVerif. Here the adversary is able to send data to an honest reader for group g at version 0 without corrupting a writer for g at 0.

1. A writer for g is corrupted at version 1. Thus the adversary obtains the lockbox key lk_1, and the sign key (d_1, n) for version 1. Let there be another writer for g at version 0 that publishes some content, so that the adversary also knows the owner-signed header sn_0 for version 0.
2. The adversary computes the lockbox key lk_0 by unwinding lk_1; further, it computes the exponent $e_1 = \mathsf{genExp}(n, lk_1)$, the RSA seed $s = \mathsf{crack}(e_1, d_1, n)$, and the sign key $sk_0 = (\mathsf{d}(s, lk_0), \mathsf{N}(s))$ for version 0. Finally, choosing some fresh data m the adversary sends the following content to the file system, where $M = \mathsf{enc}(m, lk_0)$:

$$(g, 0, sn_0, n, M, \mathsf{exp}(\mathsf{hash}(M), sk_0))$$

3. An honest reader for g reads m using keys for version 0.

ProVerif does not exhibit the former attack mentioned in [Kallahalla et al., 2003] because it stops with this simpler attack.

2.3.3.2 With server-verified writes, why should a new write token be created for each version?

Suppose that a writer w, allowed at version 0, is revoked without changing the write token. Then the server accepts writes from w even after its revocation (at version 1), since the token obtained by w at version 0 remains valid. In particular, w may destroy files by overwriting them with unreadable junk after its revocation. This attack violates the correctness of server-verified writes. Furthermore, w may write valid contents after its revocation (at version 1) using keys that it obtained at version 0, and readers can read such data using keys for version 0, trusting that they were written at version 0. This attack violates strong integrity.

Accordingly, neither the correctness of server-verified writes nor strong integrity can be proved by ProVerif for a model where write tokens are not changed. We manually reconstruct the corresponding attacks from the derivations output by ProVerif. The more basic integrity property continues to hold in this case, however.

2.3.4 Additional remarks

Below we list some more observations on the original paper [Kallahalla et al., 2003]:

- The following sentence appears in [Kallahalla et al., 2003, Section 3.1]:

  > *All files with identical sharing attributes are grouped in the same filegroup...*

  Under this interpretation, each group is tied to a particular set of sharing attributes (writers and readers). So, if two files happen to have the same sharing attributes after some changes of sharing attributes, then these two files should join the same filegroup even if they initially belonged to different filegroups. Such a join actually does not happen in Plutus.

- The following sentence appears in [Kallahalla et al., 2003, Section 3.4]:

  > *A revoked reader...[can] never...read data updated since...[its] revocation.*

  We clarify that if a reader that is revoked at version v colludes with a corrupt reader or writer at any $v' > v$, or is itself a reader or writer at such v', it is able to read data updated in the interval $v + 1, \ldots, v'$.

- The following sentence appears in [Kallahalla et al., 2003, Section 3.5.2]:

  > *If the writers have no read access, then they never get the...[lockbox key], and so it is hard for them to determine the file-verify key from the file-sign key.*

  The claim here is wrong. Writers always get the lockbox key (to encrypt data), so they can always construct the verify key (just as well as readers can).

- The following sentence appears in [Kallahalla et al., 2003, Section 3.2]:

In order to ensure the integrity of the contents of the files, a cryptographic hash of the file contents is signed...

We clarify that contents should be signed *after* being encrypted for stronger security in the computational model of cryptography. Indeed, signing encrypted contents allows one to use a weaker encryption scheme: the encryption scheme needs to be only IND-CPA (indistinguishable under chosen plaintext attacks), with the signature providing integrity of the ciphertext. Signing contents in the clear instead requires a stronger security assumption for the encryption scheme, that allows the adversary to call the decryption oracle. This point is similar to the fact that when the encryption is IND-CPA and the MAC is UF-CMA (unforgeable under chosen message attacks), encrypt-then-MAC (in which the MAC is applied to the ciphertext) guarantees the secrecy of the plaintext, while encrypt-and-MAC (in which the MAC is applied to the plaintext) does not [Bellare and Namprempre, 2000]. Here, the signature plays the role of the MAC.

- As noted in [Fu et al., 2006, Section 3], the correctness of the key rotation scheme in [Kallahalla et al., 2003] is not provable in the computational model of cryptography under reasonable assumptions (one-wayness of RSA and IND-CPA symmetric encryption), because a key obtained by unwinding is not indistinguishable from a random key when one has access to other winded versions of this key. This problem is out of scope of our verification since we work in the Dolev-Yao model of cryptography. Recently several other rotation schemes have been proposed, and their cryptographic security properties have been formally studied [Backes et al., 2005; Fu et al., 2006; Backes et al., 2006]. One can note that the attacks discussed in this section do not depend on the specific scheme for generating, winding, and unwinding lockbox keys. Our results continue to hold if we change the rotation scheme to a hash-chaining scheme [Fu et al., 2006, Section 5.1], for instance. They also continue to hold if lockbox keys are hashed before they are used for encryption, as proposed in [Fu et al., 2006, Section 5.3] and [Backes et al., 2006, Section 4.2] to correct the key rotation scheme in [Kallahalla et al., 2003].

The scripts used in this chapter are available at:

`http://www.soe.ucsc.edu/~avik/projects/plutus/`

Chapter 3

Access control with labels

Most modern operating systems implement access control models that try to strike a reasonable balance between security and practice. Unfortunately, finding such a balance can be quite delicate: security concerns often lead to inflexible restrictions, which do not always seem practical. To mitigate this conflict, these systems typically admit various ways of controlling access at runtime.

This chapter is about verifying such access control systems automatically. We focus on systems in which processes and objects are labeled with security levels, and processes are prevented from accessing objects based on their labels. Such access control systems represent the state of the art in both the commercial world and the academic world, exemplified by Windows Vista and Asbestos [Efstathopoulos et al., 2005]. They are typically weaker than the pioneering models of this approach [Bell and LaPadula, 1975; Biba, 1977], which have strong secrecy and integrity properties, but turn out to be too restrictive in practice. In particular, some facility to control labels at runtime often seems to be necessary in these systems.

We illustrate this point with an example. Consider a model in which objects downloaded from the Internet are labeled `Low`, and `High` processes are prevented from executing `Low` objects. In this model, suppose that a `High` process needs to run an executable f downloaded from the Internet (say, to install a new application), and the integrity of f can be established (say, by verifying a digital certificate). Then, the `High` process should be able to run f by upgrading it to `High`. On the other hand, if the

integrity of f cannot be established, the `High` process should still be able to run f by downgrading itself to `Low` (following the principle of least privilege [Lampson, 1974]).

Windows Vista implements an access control model along these lines. In particular, Windows Vista's access control model aims to prevent privilege escalation, data tampering, and code tampering by viruses by enforcing a system-wide integrity policy based on labels. However, anticipating scenarios such as the one above, the model allows labels to be lowered or raised at runtime. Not surprisingly, this requires explicit authorization by the user. But while an informed user may be able to decide whether such authorization is safe, there is a real danger that an uninformed user may inadvertently authorize unsafe information flows. For instance, a `High` process can run a `Low` executable f, as above, by downgrading itself to `Low`. As such, running f cannot do much damage—in particular, f cannot write `High` objects, since `Low` processes are prevented from writing `High` objects in the model. However, another `High` process may upgrade f to `High` and run it, without verifying its integrity. Unfortunately, f may be a virus that can then write `High` objects.

The Asbestos operating system implements a related access control model. In this model, process labels are dynamically tainted on communication with other processes, and such taints are propagated to isolate processes based on the secrets they carry. The model aims to prevent leaking of those secrets. However, such dynamic taint-propagation mechanisms notoriously suffer from the "label-creep" problem—very soon, processes become so tainted that they are unable to communicate any further. To address this problem, the model allows a form of declassification that admits some information-flow vulnerabilities.

Although Windows Vista and Asbestos differ in their details and their goals, both systems implement dynamic access control models, based on labels, that try to balance concerns of security and practice. The information-flow properties of these systems have not been fully studied. In this chapter, we develop a technique to model and analyze such systems, and to automatically find information-flow attacks in those systems, or conversely prove their security.

At the heart of our technique is a new logic-programming language called EON, that extends Datalog with dynamic operators for creating and modifying simple ob-

jects. We show how we can code information-flow violations as queries in this language, and use query evaluation to find possible attacks. EON has some carefully designed restrictions—new names can be introduced only through unary relations, only unary relations can be transformed, and some monotonicity conditions must be satisfied. These restrictions are obeyed naturally by our specifications of Windows Vista and Asbestos. We show that with these restrictions, query evaluation for EON is decidable. Our crucial insight is that with these restrictions, it is possible to reduce query evaluation in EON to query satisfiability in a fragment of Datalog. Then, we adapt an existing algorithm [Halevy et al., 2001] to decide this satisfiability problem (with minor corrections). Further, if the EON program does not have negations over derived relations, we show a simpler reduction to query evaluation in Datalog, which allows us to solve the program and generate attacks or proofs very efficiently.

We implement these reductions in our tool, and evaluate the security designs of Windows Vista and Asbestos with EON. Our experiments highlight EON's programmability. For instance, we study the impact of various design choices, by making small, local changes in specific models and observing their influence on the attacks or proofs generated. We also model specific usage disciplines, and prove that some attacks are not possible if those disciplines are enforced (either statically or at runtime). Further, our experiments always have definite results, thanks to the decidability of query evaluation in EON. In sum, EON seems to be an effective tool to specify, understand, and verify access control models. We expect that this approach can be used to study other dynamic systems just as well.

The rest of the chapter is organized as follows. Sections 3.1 and 3.2 are devoted to theory. Sections 3.3 and 3.4 are devoted to applications. In Section 3.1, we describe the syntax and semantics of the EON language. In Section 3.2, we show how query evaluation in EON can be reduced to query satisfiability in a fragment of Datalog. (A satisfiability algorithm for this fragment is reviewed in the appendix.) We then show how query evaluation in a fragment of EON can be reduced to efficient query evaluation in Datalog. Finally, in Sections 3.3 and 3.4, we show applications of our technique through several experiments with the EON tool.

3.1 EON

In this section, we introduce the EON language, and describe its syntax and semantics. We begin by providing a brief review of Datalog. We then extend Datalog with some carefully designed dynamic operators (Section 3.1.1), and present the semantics of these operators (Section 3.1.2). Finally, we define the syntax and semantics of queries in the language (Section 3.1.3).

Datalog is a convenient logic-programming language to express relational access control models [Sarna-Starosta and Stoller, 2004; Naldurg et al., 2006; Dougherty et al., 2006; Becker et al., 2007]. In Datalog, a *positive literal* $\mathcal{S}$ is of the form $R(t_1, \ldots, t_m)$, where R is a relation, $m \geq 0$, and each t_i is a variable or a constant. A *negative literal* is of the form $!\mathcal{S}$ (where ! means "not"). A *clause* is of the form

$$\mathcal{S} \ :- \ \mathcal{L}_1, \ldots, \mathcal{L}_n.$$

where each $\mathcal{L}_i$ is a positive or negative literal.[1] We refer to the left hand side of $:-$ as the *head*, and the right hand side of $:-$ as the *body*. A Datalog program is a collection of clauses.

A clause without a body is a *fact*. A clause is *safe* if every variable in the clause appears in some positive literal in the body. A program is safe if all clauses in the program are safe.

A relation *depends* on another if there is a clause in the program that has the former relation in the head and the latter in the body; the dependency is *negative* if the literal that contains the latter relation is negative. A *base relation* does not depend on any other relation. A *base fact* is a fact on a base relation. A program is *stratified* if there is no negative dependency in any dependency cycle between relations in the program.

In a safe stratified program, a clause "$\mathcal{S} \ :- \ \mathcal{L}_1, \ldots, \mathcal{L}_n.$" with variables $\overrightarrow{x}$ is interpreted as the first-order logic formula $\forall \overrightarrow{x}.\ \mathcal{L}_1 \wedge \ldots \wedge \mathcal{L}_n \implies \mathcal{S}$. A program is interpreted as the conjunction of the interpretations of its clauses.

A *database* is a set of base facts. Given a program $\mathbb{F}$ and a database $\mathbb{DB}$, let $\mathcal{I}(\mathbb{F}, \mathbb{DB})$

[1] Some versions of Datalog do not allow negations, partly because unrestricted negations can lead to semantic inconsistencies. However, it is well-known that such inconsistencies can be eliminated with appropriate syntactic restrictions.

be the set of facts that are implied by the interpretation of $\mathbb{F} \cup \mathbb{DB}$. This set can be computed efficiently [Ullman, 1989].

3.1.1 Syntax

In EON, we extend Datalog with two dynamic operators: `new` and `next`. Before we formally describe their syntax and semantics, we present a simple example to illustrate the language. (More examples appear in Sections 3.3 and 3.4.) This example describes a dynamic system where new administrators and users can be added to the system over time, any user x can be promoted to an administrator by any administrator y, and any administrator can control the system. More precisely, the sets `Admin`, `User`, and `Control` contain constants that identify administrators, users, and principals that can control the system. The `new` operator models the creation of fresh constants, and the `next` operator models the transformation of relations over those constants:

```
new Admin.
new User.
next Admin(x) :- User(x), Admin(y).
Control(x) :- Admin(x).
```

The following query asks if a user that is not an administrator can control the system.

```
? User(x), !Admin(x), Control(x).
```

This query evaluates to false according to the operational semantics, described in Sections 3.1.2 and 3.1.3. Intuitively, the program does not reach a state where `User`(x) and `Control`(x) are both true but `Admin`(x) is not. In contrast, the following query asks if a user that is not an administrator can *eventually* control the system:

```
? User(x), !Admin(x) ; Control(x).
```

Here `;` denotes sequencing of queries. This query evaluates to true; intuitively, the program can reach a state where `User`(x) is true but `Admin`(x) is not, then reach a state where `Control`(x) is true. (In the latter state, we expect that `Admin`(x) is also true.)

Formally, an EON program is a set of EON clauses, as defined by the grammar below. Let $\mathcal{E}$ be a set of unary base relations, called *dynamic relations*, and $\mathcal{B}$ range over

subsets of $\mathcal{E}$. Intuitively, dynamic relations are the only relations that can be introduced or transformed by the dynamic operators. (For example, we do not allow binary relations to be introduced or transformed, so that query evaluation remains decidable. See Appendix B.2 for a formal justification.)

$\mathcal{C}$ ::=	EON clause
$\mathcal{S}$:– $\mathcal{L}_1, \ldots, \mathcal{L}_n$.	clause
new $\mathcal{B}$:– R.	create object
next $\mathcal{B}(x), !\mathcal{B}'(x)$:– $R(x)$.	modify object

In Sections 3.3 and 3.4, we present several examples that illustrate how access control systems are modeled in this language. Roughly, principals such as processes, objects, and so on are modeled as constants; and security-relevant metadata associated with those principals, such as labels, roles, and so on, are modeled as unary base relations (or sets) over those constants. The new operator creates fresh principals and initializes the security-relevant metadata associated with those principals; and the next operator modifies such security-relevant metadata, under constraints.

For our convenience, we require that the body of a new or next clause contains exactly one positive literal. In examples, we sometimes omit that literal, or write several literals instead; the required literal can be equivalently defined by a Datalog clause.

The Datalog fragment of an EON program $\mathbb{P}$ is written as $\widehat{\mathbb{P}}$. We say that $\mathbb{P}$ is safe if $\widehat{\mathbb{P}}$ is safe, and $\mathbb{P}$ is stratified if $\widehat{\mathbb{P}}$ is stratified. In the sequel, we consider only safe stratified programs.

3.1.2 Semantics

We now give an operational semantics for EON programs. Specifically, we describe the reduction of an EON program $\mathbb{P}$ by a binary relation $\xrightarrow{\mathbb{P}}$ over databases; an EON program defines a (possibly nondeterministic) transition system over databases.

Any transition involves the application of some dynamic clause in the program, that is *enabled* in the current database.

We first describe the semantics of the new operator. The clause "new $\mathcal{B}$:– R." is enabled if R evaluates to true in the current database. Execution of the clause creates a

fresh constant c and adds $B(c)$ to the database, for every B in $\mathcal{B}$.

$$\frac{\begin{array}{c}\mathsf{new}\ \mathcal{B}\ \mathrel{:-}\ R.\ \in \mathbb{P} \\ R \in \mathcal{I}(\widehat{\mathbb{P}}, \mathbb{DB}) \qquad c \text{ is a fresh constant} \qquad \mathbb{DB}^{+} = \{B(c) \mid B \in \mathcal{B}\}\end{array}}{\mathbb{DB} \xrightarrow{\mathbb{P}} \mathbb{DB} \cup \mathbb{DB}^{+}}$$

Next, we describe the semantics of the next operator. The clause "$\mathsf{next}\ \mathcal{B}(x), !\mathcal{B}'(x)$ $\mathrel{:-} R(x)$." is enabled if there is some constant c such that $R(c)$ evaluates to true in the current database. Execution of the clause modifies the interpretation of some relations in $\mathcal{E}$ for c. Specifically, it adds $B(c)$ to the database for every B in $\mathcal{B}$ and removes $B(c)$ from the database for every B in $\mathcal{B}'$. Note that if there are several constants c such that $R(c)$ evaluates to true in the current database, then execution of the clause non-deterministically chooses *one* such c for the update.

$$\frac{\begin{array}{c}\mathsf{next}\ \mathcal{B}(x), !\mathcal{B}'(x)\ \mathrel{:-}\ R(x).\ \in \mathbb{P} \\ R(c) \in \mathcal{I}(\widehat{\mathbb{P}}, \mathbb{DB}) \qquad \mathbb{DB}^{+} = \{B(c) \mid B \in \mathcal{B}\} \qquad \mathbb{DB}^{-} = \{B(c) \mid B \in \mathcal{B}'\}\end{array}}{\mathbb{DB} \xrightarrow{\mathbb{P}} \mathbb{DB} \cup \mathbb{DB}^{+} \setminus \mathbb{DB}^{-}}$$

The reflexive transitive closure of $\xrightarrow{\mathbb{P}}$ is written as $\xrightarrow{\mathbb{P}}{}^{\star}$.

3.1.3 Queries

Queries in EON can include basic (Datalog-style) queries; they can further use the operator ⨟ to sequence such queries.

$\mathcal{Q} ::=$	EON query
$\quad \mathcal{S}$	basic query
$\quad \mathcal{S} \fatsemi \mathcal{Q}$	sequencing

As usual, for our convenience we require that a basic query contains exactly one positive literal; elsewhere, we often write several literals instead.

The semantics of queries relies on the operational semantics above. Let σ range over substitutions of variables by constants. The judgment $\mathbb{DB}, \mathbb{DB}', \sigma \vdash_{\mathbb{P}} \mathcal{Q}$ means that:

"Starting from a database $\mathbb{DB}$, the program $\mathbb{P}$ eventually reaches a database $\mathbb{DB}'$, satisfying the query $\mathcal{Q}$ with substitution σ".

We first describe the semantics of basic queries. If the initial database $\mathbb{DB}$ evolves to a database $\mathbb{DB}'$ such that $\mathcal{S}\sigma$ evaluates to true in $\mathbb{DB}'$, then the program satisfies the basic query $\mathcal{S}$ with substitution σ.

$$\frac{\mathbb{DB} \xrightarrow{\mathbb{P}}{}^{\star} \mathbb{DB}' \qquad \mathcal{S}\sigma \in \mathcal{I}(\widehat{\mathbb{P}}, \mathbb{DB}')}{\mathbb{DB}, \mathbb{DB}', \sigma \vdash_{\mathbb{P}} \mathcal{S}}$$

Next, we describe the semantics of sequencing. If the initial database $\mathbb{DB}$ evolves to a database $\mathbb{DB}'$ such that the basic query $\mathcal{S}$ is satisfied with substitution σ, and $\mathbb{DB}'$ evolves to a database $\mathbb{DB}''$ such that the query $\mathcal{Q}$ is satisfied with substitution σ, then the program satisfies the query $\mathcal{S} \fatsemi \mathcal{Q}$ with substitution σ.

$$\frac{\mathbb{DB}, \mathbb{DB}', \sigma \vdash_{\mathbb{P}} \mathcal{S} \qquad \mathbb{DB}', \mathbb{DB}'', \sigma \vdash_{\mathbb{P}} \mathcal{Q}}{\mathbb{DB}, \mathbb{DB}'', \sigma \vdash_{\mathbb{P}} \mathcal{S} \fatsemi \mathcal{Q}}$$

3.2 Query evaluation

We now explain how EON queries can be evaluated. Formally, the query evaluation problem for EON is:

> Given an EON program $\mathbb{P}$ and an EON query $\mathcal{Q}$, are there some database $\mathbb{DB}$ and substitution σ such that $\varnothing, \mathbb{DB}, \sigma \vdash_{\mathbb{P}} \mathcal{Q}$?

We show that this problem is decidable under some suitable assumptions of monotonicity (see below). The essence of our algorithm is to reduce the EON query evaluation problem to a decidable satisfiability problem over Datalog.

Given a Datalog program $\mathbb{F}$ and a database $\mathbb{DB}$, recall that $\mathcal{I}(\mathbb{F}, \mathbb{DB})$ denotes the result of evaluating $\mathbb{F}$ over $\mathbb{DB}$. Given a positive literal $\mathcal{S}$, we use the notation $\mathbb{DB} \vdash_{\mathbb{F}} \mathcal{S}$ to indicate that there is some substitution σ such that $\mathcal{I}(\mathbb{F}, \mathbb{DB})$ contains $\mathcal{S}\sigma$. Now, $\mathcal{S}$ is *satisfiable* in $\mathbb{F}$ if there exists a database $\mathbb{DB}$ such that $\mathbb{DB} \vdash_{\mathbb{F}} \mathcal{S}$. The following satisfiability problem over Datalog is decidable.

Theorem 3.2.1 (A decidable fragment of Datalog [Halevy et al., 2001])**.** *Satisfiability is decidable for safe stratified Datalog programs with unary base relations.*

Recall that a database is a set of base facts. Given an EON program $\mathbb{P}$, we say that a database is *reachable* in $\mathbb{P}$ if it can be reached from the initial database $\varnothing$ by a sequence of transitions defined by $\mathbb{P}$. Now, the only base facts in any reachable database are over relations in $\mathcal{E}$. In the sequel, we focus on such databases. In particular, we view a database $\mathbb{DB}$ as a pair (U, I), where U is a set of constants and $I : \mathcal{E} \to 2^U$.

Given a database $\mathbb{DB} = (U, I)$ and a subset of constants $X \subseteq U$, we define the restriction of $\mathbb{DB}$ to X, denoted $\mathbb{DB}|_X$, to be (X, I_X), where $I_X(B) \triangleq I(B) \cap X$. We say that $\mathbb{DB}_1 \leq \mathbb{DB}_2$ if there exists an X such that $\mathbb{DB}_1 = \mathbb{DB}_2|_X$.

Now, a positive literal S is *monotonic* in $\mathbb{P}$ if for all $\mathbb{DB}_1$ and $\mathbb{DB}_2$, if $\mathbb{DB}_1 \vdash_{\widehat{\mathbb{P}}} S$ and $\mathbb{DB}_1 \leq \mathbb{DB}_2$, then $\mathbb{DB}_2 \vdash_{\widehat{\mathbb{P}}} S$.

3.2.1 Basic queries, unguarded transitions

Suppose that we are given a basic query $\mathcal{S}$ to evaluate on an EON program $\mathbb{P}$. We assume that $\mathcal{S}$ is monotonic in $\mathbb{P}$. Further, suppose that all dynamic clauses in $\mathbb{P}$ are *unguarded*. A new clause is unguarded if its body is a fact (*e.g.*, "`True.`") in the program. A next clause is unguarded if the relation in its body is a *faithful relation*. The concept of a faithful relation is defined inductively as follows: a (unary) relation R is faithful if either $R \in \mathcal{E}$, or every clause in the program with R in its head is of the form "$R(x)$:- $\mathcal{L}_1, \ldots, \mathcal{L}_n$.", where each $\mathcal{L}_i$ is either $R_i(x)$ or $!R_i(x)$ for some faithful R_i.

Note that an unguarded new clause is always enabled. Whether an unguarded next clause is enabled for a constant c depends only on the value of the relations in $\mathcal{E}$ for c.

Now, we evaluate $\mathcal{S}$ on $\mathbb{P}$ by translating $\mathbb{P}$ to a Datalog program $\lfloor \mathbb{P} \rfloor$, and deciding if there exists a database $\mathbb{DB}$ such that $\mathbb{DB} \vdash_{\lfloor \mathbb{P} \rfloor} \mathcal{S}$ and $\mathbb{DB}$ is reachable in $\mathbb{P}$. The latter problem is reduced to a basic satisfiability problem of the form $\mathbb{DB} \vdash_{\lfloor \mathbb{P} \rfloor} \lfloor \mathcal{S} \rfloor$, by encoding the reachability condition into $\lfloor \mathbb{P} \rfloor$ and defining $\lfloor \mathcal{S} \rfloor$ to be $\mathcal{S}$ augmented with the reachability condition.

Given a constant c that belongs to a database $\mathbb{DB} = (U, I)$, we define its *atomic state* to be the set $\{B \in \mathcal{E} \mid c \in I(B)\}$. We say that an atomic state $X \subseteq \mathcal{E}$ is reachable if there

exists a reachable database $\mathbb{DB}$ that contains a constant whose atomic state is X.

Lemma 3.2.2. *For an* EON *program* $\mathbb{P}$ *in which all dynamic clauses are unguarded, a database* $\mathbb{DB}$ *is reachable if and only if all constants in the database have a reachable atomic state.*

3.2.1.1 From EON to Datalog

We now show how reachable atomic states can be encoded in Datalog. Specifically, given a EON program $\mathbb{P}$, we define a set of Datalog clauses $\mathcal{T}(\mathbb{P})$ for a unary relation `Reachable`, such that every constant in `Reachable` has a reachable atomic state, and every constant that has a reachable atomic state is in `Reachable`. Some of these clauses are not safe. Later, we present a clause transformation that uniformly transforms all clauses to ensure safety.

We begin by defining some auxiliary relations. Let $\mathcal{E} = \{B_1, \ldots, B_k\}$. For each B_i ($i \in \{1, \ldots, k\}$), we include the following Datalog clauses, that check whether a pair of constants have the same value at B_i:

$$\texttt{Same}B_i(x, y) \ :- \ B_i(x),\ B_i(y).$$
$$\texttt{Same}B_i(x, y) \ :- \ !B_i(x),\ !B_i(y).$$

Now, consider an unguarded `new` clause of the form:

$$\texttt{new}\ \ B_{i_1}, \ldots, B_{i_m}.$$

Let $\{B_{j_1}, \ldots, B_{j_n}\} = \mathcal{E} \backslash \{B_{i_1}, \ldots, B_{i_m}\}$. We replace this clause with the following reachability clause in Datalog:

$$\begin{aligned}\texttt{Reachable}(x) \ :- \quad & B_{i_1}(x), \ldots, B_{i_m}(x), \\ & !B_{j_1}(x), \ldots, !B_{j_n}(x).\end{aligned}$$

This clause may be read as follows: a satisfying database for the transformed Datalog program may contain a constant x whose atomic state is $\{B_{i_1}, \ldots, B_{i_m}\}$. Intuitively, new constants in EON are represented by existentially quantified variables in Datalog.

Now, consider an unguarded `next` clause of the form:

$$\begin{aligned}\texttt{next}\ \ & B_{i_1}(x), \ldots, B_{i_m}(x), \\ & !B_{j_1}(x), \ldots, !B_{j_n}(x) \ :- \ R(x).\end{aligned}$$

Let $\{B_{k_1}, \ldots, B_{k_r}\} = \mathcal{E} \setminus \{B_{i_1}, \ldots, B_{i_m}, B_{j_1}, \ldots, B_{j_n}\}$. R is faithful; so we replace this clause with the following reachability clause in Datalog:

$$\begin{aligned}
&\texttt{Reachable}(x) \ :- \\
&\qquad \texttt{Reachable}(y),\ R(y), \\
&\qquad\ B_{i_1}(x), \ldots, B_{i_m}(x), \\
&\qquad\ !B_{j_1}(x), \ldots, !B_{j_n}(x), \\
&\qquad\ \texttt{Same}B_{k_1}(x,y), \ldots, \texttt{Same}B_{k_r}(x,y).
\end{aligned}$$

This clause may be read as follows: a satisfying database for the transformed Datalog program may contain a constant x whose atomic state is $\mathcal{B} \cup \{B_{i_1}, \ldots, B_{i_m}\} \setminus \{B_{j_1}, \ldots, B_{j_n}\}$, if that database also contains a constant y that satisfies $R(y)$ and has some atomic state $\mathcal{B}$. Intuitively, the Datalog variables x and y represent the same EON constant, in possibly different "states", one of which can be reached from the other.

Finally, the following clause checks whether there is any constant in a satisfying database for the transformed Datalog program whose atomic state is unreachable:

$$\texttt{BadState} \ :- \ !\texttt{Reachable}(x).$$

The set of clauses $\mathcal{T}(\mathbb{P})$ contains all of the clauses above. Now, let $\texttt{U} \in \mathcal{E}$ be a fresh relation, which models the range of substitutions. For any clause $\mathcal{C} \in \mathcal{T}(\mathbb{P})$, we obtain a transformed clause $\lfloor \mathcal{C} \rfloor$ by augmenting the body of $\mathcal{C}$ with an additional condition $\texttt{U}(x)$ for every variable x in $\mathcal{C}$. The clause $\lfloor \mathcal{C} \rfloor$ is guaranteed to be safe.

Now, let $\lfloor \mathbb{P} \rfloor = \{\lfloor \mathcal{C} \rfloor \mid \mathcal{C} \in \widehat{\mathbb{P}} \cup \mathcal{T}(\mathbb{P})\}$. Let $\lfloor \mathcal{S} \rfloor$ be the query $\mathcal{S}, !\,\texttt{BadState}$ augmented with an additional condition $\texttt{U}(x)$ for every variable x in $\mathcal{S}$. We then have the following result.

Theorem 3.2.3. *Given an* EON *program* $\mathbb{P}$ *in which all dynamic clauses are unguarded, a monotonic basic query* $\mathcal{S}$ *is true in* $\mathbb{P}$ *if and only if the query* $\lfloor \mathcal{S} \rfloor$ *is satisfiable in the Datalog program* $\lfloor \mathbb{P} \rfloor$.

3.2.1.2 A heuristic

The use of (double) negation to define the transformed query $\lfloor \mathcal{S} \rfloor$ can lead to potential inefficiencies in the satisfiability algorithm (described in [Chaudhuri et al., 2008b]).

We can eliminate the use of this negation by transforming every Datalog clause $\mathcal{C}$ in the given program $\mathbb{P}$ as follows: we augment the body of the clause with the condition $\texttt{Reachable}(x)$ for every variable x in the body. (It is possible to further optimize this transformation, by adding the condition only for variables that are not already in the head of the clause, as long as we add a similar condition for all variables in $\mathcal{S}$.)

3.2.2 Basic queries, guarded transitions

Guarded dynamic clauses do not significantly complicate the transformation. The reachability clause generated for a guarded dynamic clause now includes the guard (*i.e.*, the literal in the body of the dynamic clause) in the body of the reachability clause. The correctness proofs require the guards to be monotonic. Specifically, a generalization of Lemma 3.2.2 holds true even for programs with guarded dynamic clauses, as long as the guards are monotonic.

Recall that in the case of unguarded dynamic clauses, the `Reachable` relation depends only on the relations in $\mathcal{E}$, the auxiliary relations $\texttt{Same}B_i$, and itself. However, the encoding of guards in reachability clauses makes the `Reachable` relation dependent on other relations mentioned in those guards. If we now do the heuristic of Section 6.4.8, which adds reachability conditions to the clauses of the given program, we may introduce cyclic dependences between `Reachable` and other relations. Thus, we must verify that the transformed program is stratified before checking satisfiability on the transformed program. Interestingly, it turns out that *the transformed program is stratified if and only if the guards are monotonic*! This result yields a simple method to test for the monotonicity of guards.

3.2.3 Queries with sequencing

Finally, we show how we can handle queries with sequencing. We assume that every basic query in such queries is monotonic. Consider the query $\mathcal{S} \fatsemi \mathcal{Q}$. We first assume that $\mathcal{S}$ and $\mathcal{Q}$ share exactly one variable x. Let $\texttt{Done} \in \mathcal{E}$ be a fresh relation, and $\mathcal{Q}$ be of the form $\mathcal{S}_1 \fatsemi \ldots \fatsemi \mathcal{S}_n$, for some $n \geq 1$. We augment the original EON program with the following dynamic clause:

$$\texttt{next Done}(x) \ :\!-\ \mathcal{S}.$$

We then evaluate the query $\texttt{Done}(x), \mathcal{S}_1 \ ; \ \ldots \ ; \ \mathcal{S}_n$ on the augmented EON program.

More generally, we add a next clause with a fresh Done relation for each variable shared by $\mathcal{S}$ and $\mathcal{Q}$, and augment $\mathcal{Q}$ accordingly to account for those variables. For instance, if $\mathcal{S}$ and $\mathcal{Q}$ share exactly two variables x and y, we add the clauses:

$$\texttt{next Done}(x) \ :\!-\ \mathcal{S}.$$
$$\texttt{next Done}'(y) \ :\!-\ \texttt{Done}(x), \mathcal{S}.$$

and evaluate the query $\texttt{Done}(x), \texttt{Done}'(y), \mathcal{S}_1 \ ; \ \ldots \ ; \ \mathcal{S}_n$.

On the other hand, if $\mathcal{S}$ and $\mathcal{Q}$ do not share any variable, we add a new clause with a fresh Done relation, and augment $\mathcal{Q}$ with $\texttt{Done}(z)$, where z is a fresh variable.

3.2.4 Efficient query evaluation under further assumptions

Under further assumptions, we now show that query evaluation in EON can be reduced to simple query evaluation in Datalog. This result is independent of what we present above. The main advantage of this transformation is efficiency—while checking satisfiability of Datalog programs may take exponential time in the worst case, evaluating Datalog programs takes only polynomial time.

The requirements for this transformation are as follows. There should be no (in)equality constraints over variables. In particular, variables cannot be repeated in the head of a clause. Next, there should be no negations on non-base (derived) relations, although there may be negations on base relations. These conditions turn out to be quite reasonable in practice. In particular, our models of of Windows Vista and Asbestos in Sections 3.3 and 3.4 satisfy these conditions, and most of our queries on these models satisfy these conditions as well.

We assume that sequencing is compiled away as in our original reduction, and consider only basic queries. Further, we assume that no constants appear in the EON program itself. (The transformation can be extended in a straightforward way to allow constants.) The intuition behind the transformation is as follows. Let $\mathcal{E} = \{B_1, \ldots, B_k\}$. We can represent the atomic state of a constant c as the vector $(v_1, \ldots, v_k)$ where v_i is

1 if $B_i(c)$ is true and 0 otherwise. We say that two constants c and c' are *similar* if they have the same atomic state. Now in our case, a Datalog program cannot distinguish between similar constants, *i.e.*, it is not possible to define a query $R(x)$ that is satisfied by c and not c'. (More generally, if c_i is similar to c'_i for $1 \leq i \leq r$, then $R(c_1, \ldots, c_r)$ is true iff $R(c'_1, \ldots, c'_r)$ is true in the program.) Thus we can define a query $\lfloor R \rfloor(x_1, \ldots, x_k)$ which is true iff $R(x)$ is true for any x with atomic state $(x_1, \ldots, x_k)$ that is generated by the EON program.

For every non-base relation R of arity r, we define a new relation $\lfloor R \rfloor$ of arity rk. Given any Datalog clause $\mathcal{C}$, we replace it with a transformed clause $\lfloor \mathcal{C} \rfloor$ as follows. For every variable x in the clause, we introduce k new variables $x_1, \ldots, x_k$. Then, every literal of the form $R(y_1, \ldots, y_r)$, where R is a non-base relation, is transformed into a corresponding literal $\lfloor R \rfloor(y_{11}, \ldots, y_{1k}, \ldots, y_{r1}, \ldots, y_{rk})$ by replacing every occurrence of a variable y_j by the corresponding vector of variables $y_{j1}, \ldots, y_{jk}$. Further, every literal of the form $B_i(x)$ is transformed into the literal $\texttt{True}(x_i)$ and every literal of the form $!B_i(x)$ is transformed into $\texttt{False}(x_i)$. (The auxiliary predicates `True` and `False` are defined by the facts $\texttt{True}(1)$ and $\texttt{False}(0)$.) Finally, for every variable x in the head of the clause, we add the condition $\texttt{Reachable}(x_1, \ldots, x_k)$ to the body of the transformed clause. (As an optimization, we may consider adding this reachability condition only if no non-base relation is applied to x in the body of the clause.) For example, the clause

$$R(x, y) \ :- \ R'(x), !B_1(x), B_2(y).$$

yields the transformed clause:

$$\lfloor R \rfloor(x_1, x_2, y_1, y_2) \ :- \\ \lfloor R' \rfloor(x_1, x_2), \texttt{False}(x_1), \texttt{True}(y_2), \texttt{Reachable}(y_1, y_2).$$

Now, every clause of the form "new $\mathcal{B}$:– R." is transformed to

$$\texttt{Reachable}(z_1, \ldots, z_k) \ :- \ \lfloor R \rfloor.$$

where z_i is 1 if $B_i \in \mathcal{B}$ and 0 otherwise.

Further, every clause of the form "next $\mathcal{B}(x), !\mathcal{B}'(x)$:– $R(x)$." is transformed to

$$\texttt{Reachable}(z_1, \ldots, z_k) \ :-$$

$$\lfloor R \rfloor(x_1, \ldots, x_k),$$
$$\texttt{Reachable}(x_1, \ldots, x_k),$$
$$\texttt{Update}(x_1, z_1), \ldots, \texttt{Update}(x_k, z_k).$$

where $\texttt{Update}(x_i, z_i)$ is $\texttt{True}(z_i)$ if x_i is in $\mathcal{B}$, $\texttt{False}(z_i)$ if x_i is in $\mathcal{B}'$, and $z_i = x_i$ otherwise. (The literal $z_i = x_i$ is implemented by replacing z_i with x_i in the clause.)

We then have the following result.

Theorem 3.2.4. *Given an* EON *program* $\mathbb{P}$ *with the above restrictions, a query* $\mathcal{Q}$ *is true in* $\mathbb{P}$ *iff the query* $\lfloor \mathcal{Q} \rfloor$ *is true in the Datalog program* $\lfloor \mathbb{P} \rfloor$.

Proof details for all the results above appear separately in [Chaudhuri et al., 2008b]; some of those details are reproduced in the appendix.

3.2.5 Tool support and experiments

The transformations described above are at most quadratic in time complexity, and are implemented in the EON tool [Chaudhuri et al., 2008b]. Further, the back end includes implementations of satisfiability and evaluation algorithms over Datalog, and the front end supports some syntax extensions over EON, such as *embedded scripts* for model generation [Chaudhuri et al., 2008b].

We carry out a series of experiments with the EON tool, that illustrate how it can be used to model and analyze dynamic access control systems. These experiments are presented below. We begin with Windows Vista's access control model (Section 3.3). We automatically find some integrity attacks in this model. Then, we automatically prove that these attacks can be eliminated by enforcing a certain usage discipline on the model—via static analysis or runtime monitoring. (Roughly, it follows that a user can be informed about potentially unsafe authorization decisions in the model.) Next, we consider Asbestos's access control model (Section 3.4). We automatically verify some conditional secrecy properties of that model. Finally, we model an implementation of the webserver OKWS on Asbestos (as described in [Efstathopoulos et al., 2005]), and automatically prove a data isolation guarantee for the webserver.

3.3 Windows Vista in EON

The goal of Windows Vista's access control model is to maintain boundaries around trusted objects, in order to protect them from less trusted processes. Trust levels are denoted by *integrity labels* (ILs), such as `High`, `Med`, and `Low`. Every object has an IL. Further, every process is itself an object, and has an IL. A process can spawn new processes, create new objects, and change their ILs, based on its own IL. In particular, a process with IL P_L can:[2]

- raise an object's IL to O_L only if $O_L \sqsubseteq P_L$ and the object is not a process;
- lower an object's IL from O_L only if $O_L \sqsubseteq P_L$;
- read an object;
- write an object with IL O_L only if $O_L \sqsubseteq P_L$;
- execute an object with IL O_L by lowering its own IL to $P_L \sqcap O_L$.

Below, we present an excerpt of a model of such a system in EON. (The full model appears in [Chaudhuri et al., 2008b].) The unary base relations in the model have the following informal meanings: `P` contains processes; `Obj` contains objects (including processes); and `Low`, `Med`, `High`, *etc.* contain processes and objects with those ILs.

With `new` and `next` clauses, we specify how an unbounded number of processes and objects, of various kinds, can be created.

```
new Obj,Low.
new Obj,Med.
new Obj,High.

next P(x) :- Obj(x).
...
```

Further, with `next` clauses, we specify how ILs of processes and objects can be changed. For instance, a `Med` process can raise the IL of an object from `Low` to `Med` if

[2]The capabilities of a process may be further constrained by Windows Vista's discretionary access control model. However, we ignore this model because it is rather weak; see [Chaudhuri et al., 2008b] for a detailed discussion.

that object is not a process; it can also lower the IL of an object from `Med` to `Low`. A `High` process can lower its own IL to `Med` (*e.g.*, to execute a `Med` object).

```
next Med(y),!Low(y) :- Low(y),!P(y),Med(x),P(x).
next Low(y),!Med(y) :- Med(y),Med(x),P(x).

next Med(x),!High(x) :- High(x),P(x).
...
```

The full model contains several other rules that are implemented by the system. Specifying these rules manually can be tedious and error-prone; instead, EON allows us to embed scripts in our model (as syntax extensions) that generate these rules automatically [Chaudhuri et al., 2008b]. For instance, we embed Perl scripts to generate these rules uniformly for all labels, subject to the ordering constraints mentioned earlier in this section.

Finally, with Datalog clauses, we specify how processes can `Read`, `Write`, and `Execute` objects. A process x can `Read` an object y without any constraints. In contrast, x can `Write` y only if the IL of x is `Geq` (greater than or equal to) the IL of y. Conversely, x can `Execute` y only if the IL of y is `Geq` the IL of x.

```
Read(x,y) :- P(x),Obj(y).

Write(x,y) :- P(x),Geq(x,y).

Execute(x,y) :- P(x),Geq(y,x).

Geq(x,y) :- Med(x),Med(y).
Geq(x,y) :- Med(x),Low(y).
Geq(x,y) :- Low(x),Low(y).
...
```

3.3.1 Attacks on integrity

We now ask some queries on the model above. For instance, can a `Med` object be read by a `Med` process after it is written by a `Low` process? Can an object that is written by a `Low` process be eventually executed by a `High` process after downgrading only to `Med`?

```
? Med(y); Low(x),Write(x,y); Med(z),Read(z,y).
? Low(x),Write(x,y); High(z); Med(z),Execute(z,y).
```

The former encodes a simple data-flow integrity violation; the latter encodes a simple privilege-escalation violation. (In the full model, we study more general integrity violations.) When we run these queries, we obtain several attacks. (Some of these attacks have been documented elsewhere; see, *e.g.*, [Chaudhuri et al., 2008a; Conover, 2007] for details.) For each attack, our tool shows a derivation tree; from that tree, we find a sequence of `new`, `next`, and other clauses that lead the system to an insecure state and derive the query. For instance, the former query is derived as follows: first, a `Med` object y is created; next, y is downgraded to `Low` by a `Med` process; next, y is written by a `Low` process x; finally, y is read by a `Med` process z. The latter query is derived as follows: first, a `Low` object y is created; next, y is written by a `Low` process x; next, y is upgraded to `Med` by a `Med` process; next, a `High` process z is downgraded to `Med`; finally, y is executed by z.

Thus, EON can be quite effective as a *debugging* tool—if there is a bug, EON is guaranteed to find it. But recall that if there are no bugs, EON is also guaranteed to terminate without finding any! That is, EON can be just as effective as a theorem-proving tool. In particular, we now prove that the attacks above are eliminated if suitable constraints are imposed on the model. In practice, these constraints may be implemented either by static analysis or by runtime monitoring on programs running in the system.

3.3.2 A usage discipline to recover integrity

Basically, we attach to each object a label `SHigh`, `SMed`, or `SLow`, which indicates a static lower bound on the integrity of the contents of that object; further, we attach to each process a label `DHigh`, `DMed`, or `DLow`, which indicates a dynamic lower bound on the integrity of the values known to that process. The semantics of these labels are maintained as invariants by the model. The labels are initialized as follows.

```
new Obj,Low,SLow.
new Obj,Med,SMed.
new Obj,High,SHigh.
```

```
next P(x),DHigh(x) :- Obj(x).
...
```

Now, whenever an object's IL is lowered, the IL should not fall below the static label of the object.

```
next Low(y),!Med(y) :- Med(y),SLow(y),Med(x),P(x).
...
```

A process's dynamic label may be lowered to reflect that it may know the contents of an object with a lower static label.

```
next DLow(x),!DHigh(x) :- DHigh(x),SLow(y).
...
```

Now, a process x can `Read` an object y only if the dynamic label of x is less than or equal to the static label of y, that is, $\texttt{DSLeq}(x, y)$. Conversely, x can `Write` y only if the dynamic label of x is greater than or equal to the static label of y, that is, $\texttt{DSGeq}(x, y)$. In contrast, x can `Execute` y only if its own IL is lowered to or below the static label of y. This condition, $\texttt{SGeq}(y, x)$, subsumes the earlier condition $\texttt{Geq}(y, x)$.

```
Read(x,y) :- P(x),Obj(y),DSLeq(x,y).

DSLeq(x,y) :- DLow(x),SLow(y).
DSLeq(x,y) :- DLow(x),SMed(y).
DSLeq(x,y) :- DMed(x),SMed(y).
...

Write(x,y) :- P(x),Obj(y),Geq(x,y),DSGeq(x,y).

DSGeq(x,y) :- DLow(x),SLow(y).
DSGeq(x,y) :- DMed(x),SMed(y).
DSGeq(x,y) :- DMed(x),SLow(y).
...

Execute(x,y) :- P(x),Obj(y),SGeq(y,x).

SGeq(y,x) :- SLow(y),Low(x).
```

```
SGeq(y,x) :- SMed(y),Low(x).
SGeq(y,x) :- SMed(y),Med(x).
...
```

Finally, recall the dynamic queries that we ask above. We reformulate the former query for this model—instead of constraining the IL of z, we now constrain its dynamic label, which is the *de facto* dictator of its future `Writes` in this constrained model.

```
? Med(y) ; Low(x),Write(x,y) ; DMed(z),Read(z,y).
```

This query evaluates to false, showing that the encoded data-flow integrity violation is eliminated. The latter query also evaluates to false, showing that the encoded privilege-escalation violation is eliminated. The full constrained model appears in [Chaudhuri et al., 2008b]. There, we show that more general integrity violations are also eliminated under these constraints.

Thus, with EON, we not only find vulnerabilities in Windows Vista's access control model, but also prove that they can be eliminated by imposing suitable constraints on the model. We conclude that these constraints encode a formal "discipline" that is required to safely exploit the flexibilities provided by the model. This analysis can be further refined, using language-based techniques, to improve precision. In Chapter 6, we develop a type system based on this analysis, to enforce a data-flow integrity property that implies the absence of the violations above. We manually prove the correctness of that type system.

3.4 Asbestos in EON

The goal of Asbestos's access control model is to dynamically isolate trusted processes that require protection from less trusted processes. This isolation is achieved by *taint propagation*. Specifically, in Asbestos each process P has two labels: a *send label* P_S, which is a lower bound on the secrecy of messages that are sent by P, and a receive label P_R, which is an upper bound on the secrecy of messages that can be received by P. Further, each communication port C has a *port label* C_L, which is an upper bound on the secrecy of messages that can be carried by c. Sending a message from process P to

process Q on port C requires that:

$$P_S \sqsubseteq Q_R \sqcap C_L$$

Further, on communication, Q is tainted by P:

$$Q_S \leftarrow Q_S \sqcup P_S$$

In fact, this situation is slightly more complicated in the implementation, with *declassification*. Specifically, a label is a record of *security levels*, drawn from $\{\star, 0\text{–}3\}$, with minimum $\star$ ("declassification privilege") and 0–3 ordered as usual. Labels form a lattice $(\sqsubseteq, \sqcup, \sqcap)$, as follows. (Here L, L' range over labels, and ℓ over label fields.)

$$\begin{aligned} L \sqsubseteq L' \quad &\text{iff} \quad \text{for each } \ell\text{: } L.\ell \leq L'.\ell \\ \text{for each } \ell\text{: } (L \sqcup L').\ell \quad &\triangleq \quad \max(L.\ell, L'.\ell) \\ \text{for each } \ell\text{: } (L \sqcap L').\ell \quad &\triangleq \quad \min(L.\ell, L'.\ell) \end{aligned}$$

Now, an operation $\_^\star$ is defined as follows.

$$L^\star.\ell = \begin{cases} \star & \text{if } L.\ell = \star \\ 3 & \text{otherwise} \end{cases}$$

On communication, Q is tainted by P only in fields that are not $\star$.

$$Q_S \leftarrow Q_S \sqcup (P_S \sqcap Q_S^\star)$$

3.4.1 Conditional secrecy

To understand some security consequences of this model, let us focus on a single field ℓ, and the security levels $\{\star, 1\text{–}3\}$; further, suppose that the involved ports are unrestricted (*i.e.*, all port labels C_L satisfy $C_L.\ell = 3$). Below, we present an excerpt of a model of such a system in EON. Let `STAR` denote $\star$, and `i`,`j` range over 1–3. The unary base relations in the model have the following informal meanings: `P` contains processes; `LRi` and `LSj` contain processes x such that $x_R.\ell =$ `i` and $x_S.\ell =$ `j`, respectively; `LSTAR` contains processes x such that $x_S.\ell = \star$ and $x_R.\ell = 3$; and `Mj` contains processes x that carry messages generated by processes y such that $y_R.\ell =$ `j`, respectively. We boot our system with the following clauses; these clauses create an unbounded number of processes of various kinds, and let them generate messages accordingly.

```
new P,LSTAR.
new P,LR1,LS1.
new P,LR2,LS1.
new P,LR3,LS1.

next M2(x),LS2(x),!LS1(x) :- LS1(x),LR2(x).
next M3(x),LS3(x),!LS1(x) :- LS1(x),LR3(x).
...
```

Next, we specify clauses for communication on unrestricted ports. The requirements and effects of such communication appear in the bodies and heads of these clauses, respectively. Note, in particular, how the relations `Mj` are augmented on such communication, reflecting the dynamic transfer of messages. (The full model contains several other, similar rules, generated automatically by scripts.)

```
next M2(x) :- P(x),LSTAR(y),M2(y).
next M3(x) :- P(x),LSTAR(y),M3(y).

next M2(x) :- LSTAR(x),P(y),M2(y).
next M3(x) :- LSTAR(x),P(y),M3(y).

next M2(x),LS2(x),!LS1(x) :- M2(y),LS2(y),LS1(x),LR2(x).
next M3(x),LS2(x),!LS1(x) :- M3(y),LS2(y),LS1(x),LR2(x).
...
```

Finally, we ask some queries. According to [Efstathopoulos et al., 2005], in Asbestos the default security level in any field of a receive label is 2. Thus, having 3 in some field of the receive label gives higher read privileges than default; processes with such labels should be able to share messages that default processes cannot know. On the other hand, having 1 in some field of the receive label gives lower read privileges than default; processes with such labels should not be able to know messages shared by default processes. Let `ReadWithout3` denote the existence of a process x for which `M3`(x) is true despite `LRi`(x) for some $i < 3$. On the other hand, let `ReadWith1` denote the existence of a process x for which `Mj`(x) is true for some $j > 1$ despite `LR1`(x). These queries encode secrecy violations.

```
ReadWithout3 :- M3(x),LR2(x).
ReadWithout3 :- M3(x),LR1(x).

ReadWith1 :- M2(x),LR1(x).
ReadWith1 :- M3(x),LR1(x).

? ReadWithout3.
? ReadWith1.
```

We find attacks for both queries with EON. Indeed, the attacks may be anticipated—messages can be declassified, that is, forwarded by processes z for which LSTAR(z) is true, without any constraints or effects. To be fair, we must account for the participation of such processes, which we call *declassifying processes*, in our queries.

Now, let BlameReadWithout3 denote the existence of a process z for which M3(z) and LSTAR(z) are true. On the other hand, let BlameReadWith1 denote the existence of a process z for which Mj(z) and LSTAR(z) are true for some $j > 1$. We now ask the following, revised queries that account for declassification. (These queries encode violations of *robust declassification* [Zdancewic and Myers, 2001].)

```
BlameReadWithout3 :- M3(y),LSTAR(y).
BadReadWithout3 :- ReadWithout3,!BlameReadWithout3.

BlameReadWith1 :- M2(y),LSTAR(y).
BlameReadWith1 :- M3(y),LSTAR(y).
BadReadWith1 :- ReadWith1,!BlameReadWith1.

? BadReadWithout3.
? BadReadWith1.
```

Now EON does not find attacks for either query. Note that the revised queries use negation on non-base relations, so we expect them to take a longer time to run. However, we can approximate these queries without using negation, simply by removing the following clauses and asking the same queries as before.

```
next M2(x) :- LSTAR(x),M2(y).
next M3(x) :- LSTAR(x),M3(y).
```

Once again, EON does not find attacks for either query; however, the queries now run much faster. Thus, we have the following conditional secrecy theorem, proved automatically by EON.

Theorem 3.4.1 (Conditional secrecy)**.** *Assume that X is either $\{P \mid P_R.\ell = 3\}$ or $\{P \mid P_R.\ell \neq 1\}$. If $Q \notin X$, then Q can never carry a message generated by a process in X, unless some declassifying process carries that message as well.*

3.4.2 Data isolation in a webserver running on Asbestos

We now present a significantly more ambitious example to demonstrate the scope of our techniques. Specifically, we apply EON to verify the design of a webserver running on Asbestos. This webserver is described in detail in [Efstathopoulos et al., 2005]; below, we briefly review its architecture. We then present an excerpt of a model of this webserver in EON, and study its key security guarantee. The full model appears in [Chaudhuri et al., 2008b].

The relevant principals include a net daemon, a database proxy, and the users of the webserver. When a user connects, the net daemon spawns a dedicated worker process for that user. The worker process can communicate back and forth with that user over the net; further, it can access a database that is common to all users. The webserver relies on sophisticated protocols for connection handling and database interaction; the aim of these protocols is to isolate processes that run on behalf of different users, so that no user can see a different user's data.

In our model, we focus on two users `u` and `v`; processes that run on behalf of these users are tagged as such on creation. We focus on label fields that are relevant for secrecy—these include `uc` and `ut` (used for communication and taint propagation by `u`), and `vc` and `vt` (used for communication and taint propagation by `v`). We model labels with unary base relations that specify the security levels in each field: *e.g.*, for processes x, $\texttt{LSuc1}(x)$ denotes $x_S.\texttt{uc} = 1$; $\texttt{LRut2}(x)$ denotes $x_R.\texttt{ut} = 2$; and $\texttt{LSvcSTAR}(x)$ denotes $x_S.\texttt{vc} = \texttt{STAR}$; similarly, *e.g.*, for communication ports y, $\texttt{Lvt2}(y)$ denotes $y_L.\texttt{vt} = 2$.

The other unary base relations in the model have the following informal meanings. `Useru` and `Userv` contain processes run by `u` and `v`, respectively; `NETdu` and `Netdv` con-

tain processes run by the net daemon to communicate with u and v, respectively; and Wu and Wv contain worker processes that are spawned by the net daemon for u and v, respectively. All of these processes participate in a connection handling protocol. Further, Ready contains any such process that is ready for communication, after that protocol is executed. Other processes are run by the database proxy. In particular, DBproxyRu and DBproxyRv contain processes that receive database records for u and v, respectively; and DBproxySu and DBproxySv contain processes that send database records for u and v, respectively.

The processes above communicate on well-defined ports. Portu and Portv contain ports on which data is sent over the net by processes running on behalf of u and v, respectively. PortDBu and PortDBv contain ports on which data is received by the database proxy from processes running on behalf of u and v, respectively. PortUnrestricted contains unrestricted ports that are used for other communication.

Finally, to verify secrecy, we let Mu and Mv contain processes that carry u's data and v's data, respectively. We require that no process that runs on behalf of v is eventually in Mu (and vice versa).

We now outline our model. We describe only clauses that involve u; the clauses that involve v are symmetrical. Most processes in the system are created with default send and receive labels. (Any security level in a default send label is 1, and any security level in a default receive label is 2.) For instance, user processes are created as follows.

```
new Useru,Ready,Mu,
      LSuc1,LSut1,LSvc1,LSvt1,
      LRuc2,LRut2,LRvc2,LRvt2.
...
```

Next, we model the connection handling protocol in [Efstathopoulos et al., 2005]. When a user u initiates a connection, the net daemon creates a new process, as follows.

```
new NETdu,
      LSuc1,LSut1,LSvc1,LSvt1,
      LRuc2,LRut2,LRvc2,LRvt2.
```

This process creates a new port on which data can be sent over the net to u. The

security level in the relevant communication field uc of the port's label is 0; thus, processes with default send labels cannot send messages on this port.

```
new Portu,
      Luc0,Lut2,Lvc2,Lvt2.
...
```

The net daemon now lowers the security level in the field uc of its send label to STAR, so that it can delegate the ability to send messages on the above port.

```
next LSucSTAR(x),!LSuc1(x) :-
      NETdu(x),LSuc1(x),Portu(y).
...
```

Next, the net daemon lowers the security level in the relevant taint propagation field ut of its send label to STAR, and becomes ready for communication.

```
next LSutSTAR(x),!LSut1(x),Ready(x) :-
      NETdu(x),LSut1(x),LSucSTAR(x).
...
```

Eventually, the net daemon can raise the security level in the field ut of its receive label to 3, to receive tainted data for u. It can similarly raise the security level in the field ut of the above port's label, to allow it to carry tainted data for u.

```
next LRut3(x),!LRut2(x) :-
      NETdu(x),LRut2(x),LSutSTAR(x).

next Lut3(x),!Lut2(x) :-
      Portu(x),Lut2(x),NETdu(y),LucSTAR(y).
...
```

Further, the net daemon can spawn a new worker process for u.

```
new Wu,
      LSuc1,LSut1,LSvc1,LSvt1,
      LRuc2,LRut2,LRvc2,LRvt2.
...
```

The security levels in the fields uc and ut of the worker process are lowered and raised to STAR and 3, respectively, before the worker process becomes ready for communication. The worker process can now send data for u on the above port, and any such data is tainted.

```
next LSucSTAR(x),LSut3(x),!LSuc1(x),!LSut1(x),Ready(x) :-
      Wu(x),LSuc1(x),LSut1(x),
      NETd(y),LSucSTAR(y),LSutSTAR(y).
...
```

Eventually, the worker can raise the security level in the field ut of its receive label to 3, to receive tainted data for u.

```
next LRut3(x),!LRut2(x) :-
      Wu(x),LRut2(x),LSutSTAR(x).
...
```

Elsewhere, the database proxy creates the following processes and ports for receiving and sending records for u. Intuitively, only processes that can send on u's network port can send such records to the database. Moreover, such records are tainted when they are sent back.

```
new DBproxyRu,Ready,
       LSuc1,LSutSTAR,LSvc1,LSvtSTAR,
       LRuc2,LRut3,LRvc2,LRvt3.

new PortDBu,Luc0,Lut3,Lvc2,Lvt2.

new DBproxySu,Ready,
       LSuc1,LSut3,LSvc1,LSvt1,
       LRuc2,LRut2,LRvc2,LRvt2.
...
```

Further, unrestricted ports can be created, as necessary.

```
new PortUnrestricted,Luc3,Lut3,Lvc3,Lvt3.
```

We model all valid communication links between the above processes, following the implementation described in [Efstathopoulos et al., 2005]. Specifically, let Send(x,z)

denote that process x may send a message to process z. This condition is constrained by the auxiliary conditions $\texttt{Link}(x, y, z)$ and $\texttt{Comm}(x, y, z)$ for some port y, as follows. $\texttt{Link}(x, y, z)$ requires that x and z are ready for communication, and y is actually available for communication between x and z (see below). $\texttt{Comm}(x, y, z)$ is an encoding of the requirement $x_S \sqsubseteq z_R \sqcap y_L$ for communication, as described in the beginning of Section 3.4; the rules are generated automatically by scripts. Note that some of the communication links that we model below are redundant at run time, because of taint propagation. (Taint propagation prevents communication that might be dangerous for secrecy.) The auxiliary relations `AnyProc` and `AnyPort` are the unions of process relations and port relations in the system, respectively.

```
Link(x,y,z) :- Useru(x),PortUnrestricted(y),NETdu(z).
Link(x,y,z) :- NETdu(x),PortUnrestricted(y),Wu(z).
Link(x,y,z) :- AnyProc(x),Portu(y),NETdu(z).
Link(x,y,z) :- NETdu(x),PortUnrestricted(y),Useru(z).
Link(x,y,z) :- Wu(x),AnyPort(y),AnyProc(z).
Link(x,y,z) :- AnyProc(x),PortDBu(y),DBproxyRu(z).
Link(x,y,z) :- DBproxyRu(x),PortUnrestricted(y),DBproxySu(z).
Link(x,y,z) :- DBproxySu(x),AnyPort(y),AnyProc(z).
...

Send(x,y,z) :- Ready(x),Ready(z),Link(x,y,z),Comm(x,y,z).
```

Finally, we model the effects of communication. Specifically, the clauses below encode the effects of sending a message from process x to process z, as described in the beginning of Section 3.4: the label z_S is transformed to $z_S \sqcup (x_S \sqcap z_S^\star)$. For any field ℓ, the security level $z_S.\ell$ does not need to be raised if $\min(z_S^\star.\ell, x_S.\ell) \leq z_S.\ell$, that is, if $z_S.\ell = \star$ or $x_S.\ell \leq z_S.\ell$. This condition is denoted by $\texttt{LeqSTAR}\ell(x, z)$. Further, the relation `Mu` is augmented on such communication. (The rules are generated automatically by scripts.)

```
next Mu(z) :-
             Send(x,z),Mu(x),
             LeqSTARut(x,z),LeqSTARvt(x,z).
             LeqSTARuc(x,z),LeqSTARvc(x,z).
next Mu(z),
```

```
     LSvt3(z),!LSvt1(z) :-
             Send(x,z),Mu(x),
             LeqSTARut(x,z),LSvt1(z),LSvt3(x).
             LeqSTARuc(x,z),LeqSTARvc(x,z).
next Mu(z),
     LSut3(z),!LSut1(z) :-
             Send(x,z),Mu(x),
             LSut1(z),LSut3(x),LeqSTARvt(x,z).
             LeqSTARuc(x,z),LeqSTARvc(x,z).
next Mu(z),
     LSvt3(z),!LSvt1(z),LSut3(z),!LSut1(z) :-
             Send(x,z),Mu(x),
             LSut1(z),LSut3(x),LSvt1(z),LSvt3(x).
             LeqSTARuc(x,z),LeqSTARvc(x,z).
...
```

We now ask the query `SecrecyViolation`, which denotes the existence of a process x that runs on behalf of `v`, *i.e.*, `Userv`(x) or `Wv`(x), but carries `u`'s data, *i.e.*, `Mu`(x).

```
SecrecyViolation :- Userv(x),Mu(x).
SecrecyViolation :- Wv(x),Mu(x).

? SecrecyViolation.
```

EON does not find any exploits for this query. In other words, we have the following theorem, automatically proved by EON.

Theorem 3.4.2 (Data isolation)**.** *A user* `u`*'s data is never leaked to any process running on behalf of a different user* `v`*.*

We conclude by mentioning some statistics that indicate the scale of this experiment. The whole specification of the webserver is around 250 lines of EON. The translated Datalog program contains 152 recursive clauses over a 46-ary `Reachable` relation (that is, over 46-bit atomic states). Our query takes around 90 minutes to evaluate on a Pentium IV 2.8GHz machine with 2 GB memory—in contrast, the queries for the other examples take a few seconds.

Scripts for all the examples in this section are available in [Chaudhuri et al., 2008b].

Part II

Security via Access Control

Overview

In this part, we focus on techniques for enforcing security in systems that implement access control. The techniques integrate access control and static analysis in special type systems with notions of secrecy or integrity.

We begin by exploring the interplay of static analysis and access control in the setting of a file system. For this purpose, we study a pi calculus with file-system constructs. The calculus supports both access checks and a form of static scoping that limits the knowledge of terms—including file names and contents—to groups of clients. We design a system with secrecy types for the calculus; using this system, we can prove secrecy properties by typing programs that are subject to file-system access checks.

A limitation of this type system is that it cannot exploit access control to enforce dynamic specifications; for example, it cannot reason about the secrecy of contents that are written after revoking public access to a file. To address this limitation, next we develop a variant of Gordon and Hankin's concurrent object calculus with support for dynamic access control on methods. We investigate safe administration and access of shared services in the resulting language. Specifically, we show a type system that guarantees safe manipulation of objects with respect to dynamic specifications, where such specifications are enforced via access control on the underlying methods at run time. By labeling types with secrecy groups, we show that well-typed systems preserve their secrets amidst dynamic access control and untrusted environments. Moreover, we show that this type system generalizes the type system above through a type-directed compilation.

Finally, we consider the model of multi-level integrity implemented by Windows Vista. We observe that in this model, trusted code must participate in any information-flow attack. Thus, it is possible to eliminate such attacks by statically restricting trusted code. We formalize this model by designing a type system that can efficiently enforce data-flow integrity on Vista. Typechecking guarantees that objects whose contents are statically trusted never contain untrusted values, regardless of what untrusted code runs in the environment. We show that while some of Vista's run-time access checks are necessary for soundness, others are redundant and can be optimized away.

Chapter 4

Access control and types for secrecy

Secrecy properties can be guaranteed through a combination of static and dynamic checks. The static checks may include the application of special type systems with notions of secrecy (*e.g.*, [Abadi, 1999; Abadi and Blanchet, 2003; Cardelli et al., 2005]). The dynamic checks can be of various kinds; in practice, the most important are access checks. In this chapter, we explore the interplay of such static and dynamic checks.

The setting of our study is a fairly standard file system. More specifically, we study a pi calculus with file-system constructs. The calculus supports both access checks and a form of static scoping that limits the knowledge of terms—including file names and contents—to groups of clients. We design a system with secrecy types for the calculus. In this system, any type can be associated with a group of clients, which we call the reach of the type. By typing, we can then statically check certain secrecy properties, for instance, that a term is not leaked beyond the reach of its declared type. While the typing is static, it applies to programs that may be subject to dynamic access checks.

For example, suppose that a client creates a secret that it does not intend to share with other clients; it then writes that secret to a publicly known file. Suppose that another client attempts to read this file. We can analyze such a system in our calculus—this particular system typechecks only if the latter client does not have read access to that file. Various examples indicate that our type system is fairly permissive. Conversely, a soundness theorem states that any process that compromises secrecy intentions fails to typecheck. Further, typing has other interesting consequences; we derive,

for instance, certain integrity properties.

Somewhat similar type systems exist for other calculi, including several pi calculi. The main novelty of our work is the investigation of file-system constructs, including access checks. This investigation requires some new concepts and technical elements. It also enables us to treat examples that appear to be outside the scope of previous systems. The resulting secrecy properties, on the other hand, are fortunately standard.

In our calculus we can express and analyze programs that can request basic file operations and control permissions for such operations. We hide the details of file-system implementations. Our intent is that many of those details should be addressed via translations (from high-level constructs to lower-level mechanisms) with security-preservation results (for instance, full abstraction results). We have taken some steps in this direction [Chaudhuri and Abadi, 2005]. The present chapter complements those steps, by providing a type discipline and proof principles that apply to a source language for those translations. Thus, the techniques developed in this chapter can serve for establishing high-level secrecy guarantees, and those guarantees should carry over to lower-level systems obtained by translation.

The rest of this chapter is organized as follows. The next section gives an overview of the file-system environment we study. Section 4.2 presents a pi calculus with file-system constructs and the system of secrecy types that we design for this calculus. Finally, Section 4.3 defines a notion of secrecy, states our main results, and studies some consequences of the typing method. The soundness of the type system is established via a type-directed translation to a more sophisticated type system, developed in Chapter 5; the compilation itself is detailed in the appendix.

4.1 A file-system environment

We consider a distributed environment with some clients that interact among themselves and with a common file system. The file system stores data and maintains an access policy that is enforced on the clients. Below, we describe this environment and specify secrecy, semi-formally. Later sections contain the relevant formal details.

4.1.1 The file system and its clients

We assume a lattice $\langle \sqsupseteq, \sqcap, \sqcup, \top, \bot \rangle$ with $\top \neq \bot$, and identify each client with some level L in this lattice. Such a client may request an operation $\varkappa$ (see below) on the distinguished channel $\beta_L.\varkappa$. (In an implementation, $\beta_L.\varkappa$ may be realized with an authenticated encryption key shared by clients at level L.)

In the file system, each file is owned by some level, and associated with some permissions. A new file is created by the operation new. We focus on two operations on stores, read and write, and an operation chmod to modify read and write permissions. Formally, the file system maintains an access policy F, which is a partial function from files to access controls: $F(f) = L_o(L_r, L_w)$ means that the file f is owned by the level L_o, and the pair of levels (L_r, L_w) are the read and write permissions for f. Access controls have the following meanings: any level that is at least as high as L_o can change the permissions for f; any level that is at least as high as L_r can read f; and any level that is at least as high as L_w can write f. The file system also maintains a store ρ, which is a partial function from files to contents, whose domain is included in the domain of F.

4.1.2 Groups

Intuitively, we can think of levels as *groups* of clients (so that $\sqsupseteq$ means "subset", $\sqcap$ means "union", and $\sqcup$ means "intersection".) Some of those groups are induced by an access policy, *e.g.*, the group of clients who have read access to a certain file. It is not true, however, that only those clients who have read access to a file may come to know its contents: a client who has access may read the contents, then share it with another client who is not allowed to read the file. While such sharing is often desirable, it is reasonable to try to limit its scope—we would want to know, for instance, if clients who have been granted access to sensitive files are leaking their contents, either intentionally or by mistake, to dishonest ones.

We use groups as a declarative means of specifying boundaries within which secrets may be shared. To make the definition of these groups more concrete, we draw a distinction between honest clients and potentially dishonest ones. Honest clients are those who play by the rules—they are disciplined in the way they interact with other

clients and the file system, and this conformance may be checked statically by inspecting their code (*viz.* by typechecking). We identify honest clients with groups $\sqsupset \bot$. The remaining clients, identified with the group $\bot$, are assumed to be dishonest; in general they may make up an unknown, arbitrary attacker.

A *secrecy intention* is declared by stating that a certain name belongs to some group. In our type system, this declaration is made by assuming a type for a name. In turn, a type can be associated with a group, called its *reach*. Informally, the reach of a type is the group within which the inhabitants of that type may be shared. Typing guarantees that secrecy intentions are never violated, *i.e.*, a name is never leaked outside the reach of its declared type.

4.2 A typed pi calculus with file-system constructs

We use a synchronous pi calculus for writing and verifying client code. In this section, we give the syntax of terms and processes, preview some examples, present our type system for this calculus, and finally revisit the examples.

4.2.1 Terms and processes

Let $\varkappa$ range over operations in $\{\mathsf{new}, \mathsf{read}, \mathsf{write}, \mathsf{chmod}\}$, and L range over levels (or groups). Further, let x range over names, which denote variables, files, and channels; the channels include a request channel $\beta_L.\varkappa$ for each L and $\varkappa$.

The source language is a standard typed pi calculus, with the following grammar for terms and processes.

$M, N ::=$	terms
x	name
L	level
(M, N)	pair
$P, Q ::=$	processes
$\overline{M}\langle N \rangle; P$	output
$M(x); P$	input

$(\nu x : T)P$	restriction
split M as $(x, y); P$	projection
0	nil
$P \mid Q$	composition
$!P$	replication

Terms include names, levels, and pairs. Processes have the usual semantics:

- $\overline{M}\langle N \rangle; P$ sends N on M and continues as P—or blocks if M is not a channel at run time.
- $M(x); P$ receives a term on M, binds the term to x, and continues as P—or blocks if M is not a channel at run time.
- split M as $(x, x'); P$ splits M as (N, N'), binds N to x and N' to x', and continues as P—or blocks if M is not a pair at run time.
- $(\nu x : T)P$ creates a fresh name x, and continues as P; the type T is the declared type of x, and has no run-time significance.
- 0 does nothing.
- $P \mid Q$ behaves as the parallel composition of P and Q.
- $!P$ behaves as the parallel composition of an unbounded number of copies of P.

Moreover, the calculus allows interactions with an underlying file system, in parallel. A state ζ of the file system is a pair of the form (F, ρ), where F is an access policy and ρ is a store.

Creating a file On receiving a request x on channel $\beta_L.\mathsf{new}$, the file system creates a new file, with a fresh name (say f) and sends f back on the channel x. The file's owner is L, read and write permissions are $\top$ and $\top$, and content is empty; that is, $F(f) = L(\top, \top)$ and $\rho(f)$ is undefined.

Reading a file On receiving a request (f, x) on channel $\beta_L.\mathsf{read}$, the file system checks that L is at least as high as the read permissions for the file f, gets the content of

f, and sends it on the channel x. That is, if $F(f) = \_(L_r, \_)$ and $\rho(f) = M$, then the file system checks that $L \sqsupseteq L_r$ and sends M back on c.

Writing a file On receiving a request (f, M) on channel $\beta_L.\mathsf{write}$, the file system checks that L is at least as high as the write permissions for the file f, and sets M as the content of f. That is, if $F(f) = \_(\_, L_w)$, then the file system checks that $L \sqsupseteq L_w$ and sets $\rho(f)$ to M.

Changing permissions for a file On receiving a request $(f, (L_r, L_w))$ on channel $\beta_L.\mathsf{chmod}$, the file system checks that L is at least as high as the level that owns the file f, and sets the read and write permissions of M to L_r and L_w. That is, if $F(f) = L_o(\_, \_)$, then the file system checks that $L \sqsupseteq L_o$ and sets $F(f)$ to $L_o(L_r, L_w)$.

4.2.2 Some examples (preview)

In the examples below, we assume that an arbitrary (unspecified) adversary runs in parallel with the specified code; the channel *net* and the request channels $\beta_\bot.\varkappa$ are known to the adversary, but the request channels $\beta_\top.\varkappa$ are not known to the adversary. We are concerned about the secrecy of a name m written to a file f; in particular, we wish to guarantee that the adversary cannot know m by reading f.

We begin with the example sketched in the introduction. We return to this example and the others, giving additional details, in Section 4.2.7.

1. Suppose that f is a file created by level $\top$ (with implicit owner $\top$, and implicit initial read and write permissions $\top$ and $\top$); further, suppose that in parallel, the name f is published on *net* and a fresh name m is written to f by level $\top$.

$$(\nu x)\, \overline{\beta_\top.\mathsf{new}}\langle x\rangle; x(f); (\overline{net}\langle f\rangle \mid (\nu m)\, \overline{\beta_\top.\mathsf{write}}\langle (f, m)\rangle)$$

   We claim that m remains secret in this case, since the only way the adversary can read f is by sending a request on $\beta_\bot.\varkappa$, and $\bot$ does not have read permission for f; further, the adversary cannot set this read permission, since it does not own f. Indeed, this example typechecks in our system (see Section 4.2.7).

2. Next, consider a variation of (1), in which the name f of the file to which m is written is not the f in scope; instead, f is retrieved from *net*.

$$(\nu x)\,\overline{\beta_\top.\mathsf{new}}\langle x\rangle; x(f); (\overline{net}\langle f\rangle \mid net(f); (\nu m)\,\overline{\beta_\top.\mathsf{write}}\langle (f,m)\rangle)$$

   But in this case, the name retrieved from *net* may not be the same as the f in scope. In particular, the adversary may run the following code in parallel:

$$(\nu z)\,\overline{\beta_\bot.\mathsf{new}}\langle z\rangle; z(f'); (\overline{net}\langle f'\rangle \mid \overline{\beta_\bot.\mathsf{chmod}}\langle f', (\bot,\top)\rangle \mid \overline{\beta_\bot.\mathsf{read}}\langle f', net\rangle; \ldots)$$

   That is, a file f' is created by $\bot$; further, in parallel, the name f' is published on *net*, the read permissions for f' are set to $\bot$ by $\bot$, and f' is read by $\bot$. Unfortunately, now m may be written on f', and thus be leaked. This example does not typecheck in our system (see Section 4.2.7).

3. More directly, m may be leaked if it is written to a file created by $\bot$.

$$(\nu x)\,\overline{\beta_\bot.\mathsf{new}}\langle x\rangle; x(f); (\overline{net}\langle f\rangle \mid (\nu m)\,\overline{\beta_\top.\mathsf{write}}\langle (f,m)\rangle)$$

   As expected, this example does not typecheck in our system (see Section 4.2.7).

4. Now, consider a variation of (1), in which the read permissions for f are set to $\bot$ by $\top$, in parallel.

$$(\nu x)\,\overline{\beta_\top.\mathsf{new}}\langle x\rangle; x(f); (\overline{net}\langle f\rangle \mid (\nu m)\,\overline{\beta_\top.\mathsf{write}}\langle (f,m)\rangle \mid \overline{\beta_\top.\mathsf{chmod}}\langle (\bot,\top)\rangle)$$

   Of course, the adversary can read f in this case, so m may be leaked. This example does not typecheck in our system (see Section 4.2.7).

5. However, what if the name f is not published on *net*?

$$(\nu x)\,\overline{\beta_\top.\mathsf{new}}\langle x\rangle; x(f); ((\nu m)\,\overline{\beta_\top.\mathsf{write}}\langle (f,m)\rangle \mid \overline{\beta_\top.\mathsf{chmod}}\langle (\bot,\top)\rangle)$$

   We claim that m remains secret in this case, since the adversary cannot even know the name f. Indeed, this example typechecks in our system (see Section 4.2.7).

4.2.3 Types

Types include, in particular, those of the standard form $L[T]$ for channels; other types include those for request channels, files, levels, and pairs (see below). Source programs can create new channels and declare their types. The reach $\|T\|$ of a type T is defined as the group within which the inhabitants of T should be secret.

$T ::=$	types
$L[T]$	channel (declared)
$\mathsf{Req}_L.\varkappa$	request channel
$L\{T\}\#L_o(L_r, L_w)$	file
L	level
(S, T)	pair

- The type $L[T]$ is given to a channel that carries terms of type T; further, the name of the channel is secret within L, that is, $\|L[T]\| = L$.

- The type $\mathsf{Req}_L.\varkappa$ is given to the request channel $\beta_L.\varkappa$; further, the name of the request channel is secret within L, that is, $\|\mathsf{Req}_L.\varkappa\| = L$.

- The type $L\{T\}\#L_o(L_r, L_w)$ is given to a file that contains terms of type T, that is owned by L_o, and whose read and write permissions are at least L_r and L_w; further, the name of the file is secret within L, that is, $\|L\{T\}\#L_o(L_r, L_w)\| = L$.

- The type L is given to a level that is at least as high as L. Further, the special type $\bot$ ("public") is given to a term that may be known to $\bot$ (the adversary); in particular, all levels may be known to $\bot$, that is, $\|L\| = \bot$.

- The type (S, T) is given to a pair (M, N), such that M is of type S and N is of type T; since pairs can be projected, we define $\|(S, T)\| = \|S\| \sqcup \|T\|$.

Type declarations indicate secrecy intentions. However, they do not affect run-time behaviors, and the same "untyped" process can be type-annotated in several different ways to verify various secrecy intentions.

4.2.4 Preliminaries on typechecking

Typechecking a system involves typechecking clients and the file system under the same assumptions. For clients, the typechecking applies to honest clients; these clients are restricted in their use of channels and file-system requests, by the typing rules shown in Section 4.2.5. Typechecking the access policy imposes restrictions on the permissions of dishonest clients; these restrictions are specified in Section 4.2.6. Typechecking the store enforces consistency between the types of files and their contents.

The partition between honest and dishonest clients plays a central role in typechecking the system. The code for the clients as well as the access policy impose typing constraints that finally determine whether the partition is valid, *i.e.*, whether all honest clients are well-typed, and whether the access policy is suitably restrictive for the remaining (possibly dishonest) ones. Arriving at the correct partition may be delicate: overestimating the set of honest clients does not help if one of those clients is ill-typed; underestimating this set imposes more constraints on the access policy. Once we do have a valid partition, however, we can prove that an honest client (or indeed a subset of honest clients) can protect secrets from all other (honest and dishonest) clients.

4.2.5 Typing judgments and rules

We now show typing rules. Let Γ be a sequence of type assumptions $x : T$. The rules judge well-formed assumptions $\Gamma \vdash \diamond$, well-formed types $\Gamma \vdash T$, well-typed terms $\Gamma \vdash M : T$, and well-typed processes $\Gamma \vdash P$. Further, we have rules that define a "subtyping" preorder over types $S \leq T$.

The typing rules are based on the following key observations.

- Knowing the name of a channel is sufficient to receive and send messages on that channel. Consequently, any message sent to a public channel must be public, and any message received from a public channel must be untrusted. This is the main idea behind previous type systems for secrecy in the pi calculus. For a channel of type $L[\_]$, we maintain the invariant that the name of the channel may be known only to levels $\sqsupseteq L$.

- In contrast, knowing the name of a file is not sufficient to read or write that file; having read or write permissions for the file is necessary. Consequently, contents written to a public file may be secret, if the adversary does not have read permission for that file; and contents read from a public file may be trusted, if the adversary does not have write permission for that file. For a file of type $L\{\_\}\#L_o(L_r, L_w)$, we maintain the invariant that the name of the file may be known only to levels $\sqsupseteq L$, and further, the only levels that may have read permssions for the file are $\sqsupseteq L_r$, and the only levels that may have write permissions for the file are $\sqsupseteq L_w$; finally, since the level L_o can control read and write permissions, we require that $L_o \sqsupseteq L_r$ and $L_o \sqsupseteq L_w$.

Typing rules $S \leq T$

$$\text{(SubR)}\quad S \leq S \qquad \text{(SubT)}\quad \frac{S \leq S' \qquad S' \leq T}{S \leq T} \qquad \text{(Sub Pair)}\quad \frac{S \leq S' \qquad T \leq T'}{(S,T) \leq (S',T')} \qquad \text{(Sub } L\text{)}\quad \frac{L \sqsupseteq L'}{L \leq L'} \qquad \text{(Sub } \bot\text{)}\quad \frac{\bot \sqsupseteq \|T\|}{T \leq \bot}$$

We begin by looking at the rules for $S \leq T$. Intuitively, if $S \leq T$, then any term of type S also has type T (see below). By (Sub L), if L is at least as high as L', then L is a subtype of L'—since any level that is at least as high as L is also at least as high as L'. By (Sub $\bot$), if the reach of a type T is $\bot$, then T is a subtype of $\bot$—since any term of type T may be known to $\bot$. The remaining subtyping rules are straightforward. For any type T, we say that T is public if $T \leq \bot$, and untrusted if $\bot \leq T$; by the above subtyping rules, clearly the only untrusted type is $\bot$.

Typing rules $\Gamma \vdash \diamond$

$$\text{(Hyp Emp)}\quad \varnothing \vdash \diamond \qquad \text{(Hyp Typ)}\quad \frac{\Gamma \vdash T \qquad x : \_ \notin \{\Gamma\} \qquad x \text{ is not of the form } \beta_L.\varkappa}{\Gamma, x : T \vdash \diamond}$$

Next, we look at the rules for $\Gamma \vdash \diamond$. By (Hyp Emp), an empty sequence of assumptions is well-formed. Further, by (Hyp Typ), a sequence of assumptions Γ remains well-formed when extended with an assumption $x : T$, if T is a well-formed type, an

assumption for x does not already appear in Γ, and x is not a request channel. (Request channels are typed directly, rather than from assumptions; see below.)

Typing rules $\Gamma \vdash T$

$$\text{(Typ } L\text{)} \quad \Gamma \vdash L \qquad \text{(Typ Req)} \quad \Gamma \vdash \mathsf{Req}_L.\varkappa$$

$$\text{(Typ Pair)} \quad \frac{\Gamma \vdash S \qquad \Gamma \vdash T}{\Gamma \vdash (S, T)} \qquad \text{(Typ Chan)} \quad \frac{\Gamma \vdash T \qquad \bot \sqsupseteq L \;\Rightarrow\; T \leq \bot,\ \bot \leq T}{\Gamma \vdash L[T]}$$

$$\text{(Typ File)} \quad \frac{\Gamma \vdash T \qquad L_o \sqsupseteq L_r \sqcup L_w \qquad \bot \sqsupseteq L \sqcup L_r \Rightarrow T \leq \bot \qquad \bot \sqsupseteq L \sqcup L_w \Rightarrow \bot \leq T}{\Gamma \vdash L\{T\}\#L_o(L_r, L_w)}$$

In the rules for $\Gamma \vdash T$, we implicitly require $\Gamma \vdash \diamond$ in the antecedent. By (Typ Chan), the type $L[T]$ of a channel is well-formed if T is well-formed; further, if L is $\bot$ then T is public and untrusted, since terms of type T may be received and sent on such a channel. By (Typ File), the type $L\{T\}\#L_o(L_r, L_w)$ of a file is well-formed if T is well-formed, and L_o is at least as high as L_r and L_w; further, if L and L_r are $\bot$ then T is public, since terms of type T may be read from that file; and if L and L_w are $\bot$ then T is untrusted, since terms of type T may be written to that file. The remaining rules for well-formed types are straightforward.

Typing rules $\Gamma \vdash M : T$

$$\text{(Term Env)} \quad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(Term Req)} \quad \Gamma \vdash \beta_L.\varkappa : \mathsf{Req}_L.\varkappa \qquad \text{(Term Lev)} \quad \Gamma \vdash L : L$$

$$\text{(Term Pair)} \quad \frac{\Gamma \vdash M : S \qquad \Gamma \vdash N : T}{\Gamma \vdash (M, N) : (S, T)} \qquad \text{(Term Sub)} \quad \frac{\Gamma \vdash M : S \qquad S \leq T}{\Gamma \vdash M : T}$$

In the rules for $\Gamma \vdash M : T$, we implicitly require $\Gamma \vdash T$ in the antecedent. By (Term Env), names can be typed from assumptions. By (Term Req), request channels are typed directly. By (Term Lev), every level gets its own type. By (Term Sub), a term of type S also has type T if S is a subtype of T. The remaining rules for well-typed terms are straightforward.

Typing rules $\Gamma \vdash P$

$$\frac{\Gamma \vdash M : L[T] \qquad \Gamma \vdash N : T \qquad \Gamma \vdash P}{\Gamma \vdash \overline{M}\langle N \rangle; P} \text{ (Proc out)} \qquad \frac{\Gamma \vdash M : L[T] \qquad \Gamma, x : T \vdash P}{\Gamma \vdash M(x); P} \text{ (Proc in)}$$

$$\frac{\Gamma \vdash M : \bot \qquad \Gamma \vdash N : \bot \qquad \Gamma \vdash P}{\Gamma \vdash \overline{M}\langle N \rangle; P} \text{ (Proc out } \bot\text{)} \qquad \frac{\Gamma \vdash M : \bot \qquad \Gamma, x : \bot \vdash P}{\Gamma \vdash M(x); P} \text{ (Proc in } \bot\text{)}$$

$$\frac{\Gamma, x : L[T] \vdash P}{\Gamma \vdash (\nu x : L[T])\, P} \text{ (Proc new chan)} \qquad \frac{\Gamma \vdash M : (S, T) \qquad \Gamma, x : S, y : T \vdash P}{\Gamma \vdash \text{split } M \text{ as } (x, y); P} \text{ (Proc proj)}$$

$$\Gamma \vdash 0 \text{ (Proc nil)} \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \text{ (Proc par)} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P} \text{ (Proc repl)}$$

$$\frac{\Gamma \vdash M : \mathsf{Req}_L.\mathsf{read} \qquad L \sqsupset \bot \qquad \Gamma \vdash N : (\_\{T\}\#\_(L_r, \_),\ \_[T']) \qquad \Gamma \vdash P \qquad L \sqsupseteq L_r \ \Rightarrow\ T \leq T'}{\Gamma \vdash \overline{M}\langle N \rangle; P} \text{ (Proc read)}$$

$$\frac{\Gamma \vdash M : \mathsf{Req}_L.\mathsf{write} \qquad L \sqsupset \bot \qquad \Gamma \vdash N : (\_\{T\}\#\_(\_, L_w),\ T') \qquad \Gamma \vdash P \qquad L \sqsupseteq L_w \ \Rightarrow\ T' \leq T}{\Gamma \vdash \overline{M}\langle N \rangle; P} \text{ (Proc write)}$$

$$\frac{\Gamma \vdash M : \mathsf{Req}_L.\mathsf{chmod} \qquad L \sqsupset \bot \qquad \Gamma \vdash N : (\_\{\_\}\#L_o(L_r, L_w),\ (L'_r, L'_w)) \qquad \Gamma \vdash P \qquad L \sqsupseteq L_o \ \Rightarrow\ (L'_r, L'_w) \leq (L_r, L_w)}{\Gamma \vdash \overline{M}\langle N \rangle; P} \text{ (Proc chmod)}$$

$$\frac{\Gamma \vdash M : \mathsf{Req}_L.\mathsf{new} \qquad L \sqsupset \bot \qquad \Gamma \vdash N : \_[\_\{\_\}\#L(\_, \_)] \qquad \Gamma \vdash P}{\Gamma \vdash \overline{M}\langle N \rangle; P} \text{ (Proc new)}$$

In the rules for $\Gamma \vdash P$, we implicitly require $\Gamma \vdash \diamond$ in the antecedent. The rules (Proc Out) and (Proc In) are complementary; for any channel of type $L[T]$, terms sent on that channel must have type T, and terms received on that channel may be assumed

to have type T. Further, by (PROC OUT $\bot$) and (PROC IN $\bot$), terms of type $\bot$ may be sent and received on a channel of type $\bot$. By (PROC NEW), the type declared for a new name must be a channel type, and this type is assumed for the name. We discuss the rules for file-system requests next; the remaining rules are straightforward.

By (PROC read), if an honest client at level L requests to read content of type T' from a file of type $\_\{T\}\#\_(L_r, \_)$, and L is at least as high as L_r (so that the request may succeed), then T must be a subtype of T'. Conversely, by (PROC read), if an honest client at level L requests to write content of type T' to a file of type $\_\{T\}\#\_(\_, L_w)$, and L is at least as high as L_w (so that the request may succeed), then T' must be a subtype of T. Further, by (PROC chmod), if an honest client at level L requests to set permissions (L'_r, L'_w) for a file of type $\_\{\_\}\#L_o(L_r, L_w)$, and L is at least as high as L_o (so that the request may succeed), then L'_r must be at least as high as L_r, and L'_w must be at least as high as L_w. Finally, by (PROC new), if an honest client at level L requests to create a new file of type $\_\{\_\}\#L_o(\_, \_)$, then L_o must be the same as L.

The following proposition says that any client code with public free names and no secrecy intentions can be typed. Since such code can be part of the adversary, this result is similar to ones that allow typing of untyped processes in related type systems (*e.g.*, [Cardelli et al., 2005]).

Proposition 4.2.1. *We say that a process is intention-free if all declared types in it have reach $\bot$. Suppose that P is some intention-free client code, and Γ is some type environment, such that for all $x \in \mathtt{fn}(P)$, there is some T such that $x : T \in \{\Gamma\}$ and $\|T\| = \bot$. Then $\Gamma \vdash P$.*

4.2.6 Type constraints on the file system

The file system is treated as a process, and is typed under the same assumptions that type the system of clients, following (PROC PAR). Recall that a state ζ of the file system is a pair of the form (F, ρ), where F is an access policy and ρ is a store.

Let Γ be a well-formed type environment such that $\mathtt{fn}(\zeta) \subseteq \mathtt{dom}(\Gamma)$. Further, suppose that $\mathtt{dom}(\rho) \subseteq \mathtt{dom}(F)$, and for each $M \in \mathtt{dom}(F)$, either of the following holds.

- $\Gamma \vdash M : L\{T\}\#L_o(L_r, L_w)$ for some L, T, L_o, L_r, L_w and
  - $F(M) = L_o(L'_r, L'_w)$ such that $L'_r \sqsupseteq L_r$ and $L'_w \sqsupseteq L_w$

– if $M \in \mathtt{dom}(\rho)$ then $\Gamma \vdash \rho(M) : T$.

- $\Gamma \nvdash M : L\{T\}\#L_o(L_r, L_w)$ for all L, T, L_o, L_r, L_w, but
  - $\Gamma \vdash M : \bot$
  - if $M \in \mathtt{dom}(\rho)$ then $\Gamma \vdash \rho(M) : \bot$.

Then $\Gamma \vdash \zeta$. Note that the typing constraints on states allow dishonest clients to access only those files whose contents are public. Indeed, say $F(f) = \_(L'_r, L'_w)$ for some file $f \in \mathtt{dom}(\rho)$, such that $L'_r \sqcap L'_w \sqsubseteq \bot$. Then either $\Gamma \vdash f : \bot$, or $\Gamma \vdash f : L\{T\}\#\_(L_r, L_w)$ such that $L_r \sqcap L_w \sqsubseteq \bot$. In the former case, we have $\Gamma \vdash \rho(f) : \bot$, as required; and in the latter case, we have $\Gamma \vdash \rho(f) : T$, and since dishonest clients cannot know f unless $L \sqsubseteq \bot$, it follows by (TYP FILE) that $T = \bot$, as required. Further, dishonest clients cannot set potentially dangerous permissions for themselves, since by (TYP FILE), if $L_o \sqsubseteq \bot$ then $L_r \sqcup L_w \sqsubseteq \bot$.

4.2.7 The examples, revisited

Let us now try to typecheck the examples in Section 4.2.2. In all of these examples, we assume a well-formed type environment Γ; further, we assume that $net : T_{net} \in \{\Gamma\}$ for some T_{net} such that $\|T_{net}\| \sqsubseteq \bot$. Let $\mathsf{Secret} \triangleq \top[\bot]$; we assume that the type of m is declared Secret.

1. Let $T_x \triangleq \top[\bot\{\mathsf{Secret}\}\#\top(\top, \top)]$. We type-annotate the code as follows:

   $$(\nu x : T_x)\, \overline{\beta_\top.\mathsf{new}}\langle x\rangle; x(f); (\overline{net}\langle f\rangle \mid (\nu m : \mathsf{Secret})\, \overline{\beta_\top.\mathsf{write}}\langle (f, m)\rangle)$$

   Let $T_f \triangleq \bot\{\mathsf{Secret}\}\#\top(\top, \top)$. Applying (PROC NEW), (PROC new), and (PROC IN), we are left with the type environment $\Gamma_1 \triangleq \Gamma, x : T_x, f : T_f$ and the code

   $$\overline{net}\langle f\rangle \mid (\nu m : \mathsf{Secret})\, \overline{\beta_\top.\mathsf{write}}\langle (f, m)\rangle$$

   By (SUB $\bot$), we have $T_f \leq \bot$ and $T_{net} \leq \bot$. So, by (PROC OUT), we have $\Gamma_1 \vdash \overline{net}\langle f\rangle$. The remaining obligations are discharged by applying (PROC PAR), (PROC NEW), and (PROC write).

2. We type-annotate the code as follows:

$$(\nu x : T_x)\, \overline{\beta_\top.\mathsf{new}}\langle x\rangle; x(f); (\overline{net}\langle f\rangle \mid net(f); (\nu m : \mathsf{Secret})\, \overline{\beta_\top.\mathsf{write}}\langle (f, m)\rangle)$$

Proceeding as in (1), we are finally left with the type environment Γ_1 and the following code (after renaming bound variable f to f').

$$net(f'); (\nu m : \mathsf{Secret})\, \overline{\beta_\top.\mathsf{write}}\langle (f', m)\rangle$$

At this point, we can apply either (PROC IN) or (PROC IN $\bot$). In either case, we are left with the type environment $\Gamma_2 \triangleq \Gamma_1, f' : \bot$; indeed, by (TYP CHAN), *net* cannot have type say $\bot[\bot\{\mathsf{Secret}\}\#\top(\top, \top)]$. Finally, applying (PROC NEW), we are left with an obligation that we cannot discharge, since neither (PROC write) nor (PROC OUT $\bot$) can be applied.

3. We type-annotate the code as follows, with T'_x unknown:

$$(\nu x : T'_x)\, \overline{\beta_\bot.\mathsf{new}}\langle x\rangle; x(f); (\overline{net}\langle f\rangle \mid (\nu m : \mathsf{Secret})\, \overline{\beta_\top.\mathsf{write}}\langle (f, m)\rangle)$$

Applying (PROC NEW) and (PROC OUT $\bot$), we are left with the type environment $\Gamma'_1 \triangleq \Gamma, x : T'_x$, the constraint $T'_x \leq \bot$, and the code

$$x(f); (\overline{net}\langle f\rangle \mid (\nu m : \mathsf{Secret})\, \overline{\beta_\top.\mathsf{write}}\langle (f, m)\rangle)$$

Next, applying either (PROC IN) or (PROC IN $\bot$), we are left with the type environment $\Gamma'_2 \triangleq \Gamma'_1, f : \bot$ and the code

$$\overline{net}\langle f\rangle \mid (\nu m : \mathsf{Secret})\, \overline{\beta_\top.\mathsf{write}}\langle (f, m)\rangle$$

Finally, applying (PROC PAR), (PROC OUT $\bot$), and (PROC NEW), we are left with an obligation that we cannot discharge, since neither (PROC write) nor (PROC OUT $\bot$) can be applied.

Suppose that, instead of specifying the creation of f by $\bot$, we leave it implicit as part of the adversary. Fortunately, the code still fails to typecheck. That is, even if we have $\Gamma'_2 \triangleq \Gamma'_1, f : T'_f$, such that say $T'_f \triangleq \bot\{\mathsf{Secret}\}\#\bot(\top, \top)]$, we have that T'_f is not well-formed by (TYP FILE).

4. We can proceed as in (1), with $T''_x \triangleq \top[\bot\{\mathsf{Secret}\}\#\top(\bot, L)]$ for some $L \in \{\top, \bot\}$. Finally, we are left with the type environment $\Gamma''_1 \triangleq \Gamma, x : T''_x, f : T''_f, m : \mathsf{Secret}$, such that $T''_f \triangleq \bot\{\mathsf{Secret}\}\#\top(\bot, \top)$, and the code

$$\overline{\beta_\top.\mathsf{write}}\langle(f, m)\rangle$$

But we cannot discharge this obligation—applying (PROC write), we are left with the constraint $\mathsf{Secret} \leq \bot$, which reduces by (SUB $\bot$) to $\top \sqsubseteq \bot$ (contradiction).

Moreover, no other definition for T''_x works, because f is published on *net* and the read permissions for f are set to $\bot$.

5. Let $T'''_x \triangleq \top[\top\{\mathsf{Secret}\}\#\top(\bot, \top)]$. We type-annotate the code as follows.

$$(\nu x : T'''_x)\, \overline{\beta_\top.\mathsf{new}}\langle x\rangle; x(f); ((\nu m : \mathsf{Secret})\, \overline{\beta_\top.\mathsf{write}}\langle(f, m)\rangle \mid \overline{\beta_\top.\mathsf{chmod}}\langle(\bot, \top)\rangle)$$

Proceeding as in (1), we discharge all obligations. In particular, applying (PROC write) does not require the constraint $\mathsf{Secret} \leq \bot$, since the type of the name f indicates that it is secret; further, having such a type is not problematic since f is not published on *net*.

4.3 Properties of well-typed systems

This section presents our main results for the type system, namely subject reduction and secrecy. It also explores some related topics: integrity guarantees and treatment of client collusions.

4.3.1 Type preservation

The principal property of a well-typed system is that each part of the system remains well-typed during system execution. More concretely, if a process and a file-system state are typed using the same type environment, then they remain well-typed after an arbitrary number of reductions of their parallel composition.

Proposition 4.3.1 (Subject reduction)**.** *Let* $\Gamma \vdash P \mid \zeta$ *and* $P \mid \zeta \longrightarrow^\star (\nu \overrightarrow{n} : \overrightarrow{T})\,(P' \mid \zeta')$. *Then* $\Gamma, \overrightarrow{n} : \overrightarrow{T} \vdash P' \mid \zeta'$.

Subject reduction has a number of consequences; the most important is a secrecy theorem for well-typed systems, which we discuss next.

4.3.2 Secrecy by typing and access control

We view an attacker as arbitrary code that interacts with the system via dishonest clients. An attacker is modeled by its knowledge, which is a set of names, and is an upper bound on the set of free names in its code (see [Abadi and Blanchet, 2003; Cardelli et al., 2005] for similar analyses). Let *Init* range over such sets of names.

Definition 4.3.2 (*Init*-adversary)**.** *A process E is an* Init*-adversary if E is intention-free* (i.e., *all declared types in it have reaches $\sqsubseteq \bot$) and* $\mathsf{fn}(E) \subseteq$ Init*.*

Next, we provide a definition of secrecy, using the usual notion of escape (similar to that in, *e.g.*, [Abadi and Blanchet, 2003; Cardelli et al., 2005]). A term is revealed if it may eventually be published on a channel known to the adversary. A term is a L-secret if its type suggests that it should not be leaked outside the group L.

Definition 4.3.3 (Secrecy)**.** *Let P be a process, ζ be a file-system state,* Init *be a set of names, and M be a term. Let $\overrightarrow{m} = \mathtt{fn}(M) \setminus (\mathtt{fn}(P) \cup \mathtt{fn}(\zeta) \cup \text{Init})$.*

1. *P reveals M, under the assumptions $\overrightarrow{m} : \overrightarrow{T}$, to* Init *via ζ if $P \mid \zeta \mid E \longrightarrow^{\star} \equiv Q \mid (\nu \overrightarrow{m} : \overrightarrow{T})\, c\langle M \rangle$ for some* Init*-adversary E, $c \in$ Init*, and process Q.*

2. *If $\Gamma \vdash M : T$ with $\|T\| \sqsupseteq L$, then M is a L-secret under Γ.*

Subject reduction yields the following theorem.

Theorem 4.3.4 (Secrecy by typing and access control)**.** *Suppose that $\Gamma \vdash P \mid \zeta$ and for each $c \in$ Init, $c : T \in \{\Gamma\}$ for some T such that $\|T\| = \bot$. Let $K \sqsupset \bot$. Then P does not reveal any K-secret, under any extension of Γ, to* Init *via ζ.*

Thus in a well-typed system, any secret meant to be shared only within a subset of honest clients is never revealed to the other clients. As a special case, let $n : \top[\bot]$ be a new name declared inside P. If $\Gamma \vdash P \mid \zeta$, then P does not reveal n to $\mathtt{dom}(\Gamma)$ via ζ.

4.3.3 Integrity consequences

While above we focus on secrecy properties, the type system also yields integrity properties. Such properties can be specified by declaring "expectations", and verified statically with our type system. More concretely, an expectation specifies that a certain term should have a certain type, but has no observable effect. The reach of the type can be used to reason about the source of the message. Let

$$\text{expect}(M : T); P \triangleq (\nu x : \|T\|[T])\ (\overline{x}\langle M\rangle \mid x(y); P)$$

where $x, y \notin \texttt{fn}(P)$. Such expectations can be verified statically, by typechecking, although the term M may, of course, contain variables instantiated at run time.

4.3.4 Reasoning under client collusions

The type system can be extended to reason under client collusions by parameterizing the typing judgments $\Gamma \vdash M : T$ and $\Gamma \vdash P$ by levels K. (See [Chaudhuri and Abadi, 2006b] for details.) Informally, in this system, typing under $\vdash_K$ requires K to be at least as high as the reach of any type used by the relation. For example, the code of a client at level L is typechecked under the relation $\vdash_L$. Terms that the client may know must belong to groups that include L. Going further, the code of a collusion of clients at levels $L_1, \ldots, L_n$ is typechecked under the relation $\vdash_{L_1 \sqcup \ldots \sqcup L_n}$, so that terms that the clients may know must belong to groups that include at least some L_i for $i \in 1..n$. The file system is typechecked under $\vdash_\top$, since it may know terms that belong to any group. Conversely, the adversary is typechecked under $\vdash_\bot$, since it may only know terms that belong to $\bot$.

The family of relations $\vdash$ is monotonic in the parameter K, *i.e.*, if $\Gamma \vdash P$ then $\Gamma \vdash P$ for any $K' \sqsupseteq K$. Further, a stronger version of the subject reduction property holds for this system—any process that is well-typed under $\vdash_K$ remains well-typed under $\vdash_K$, when composed in parallel with other well-typed processes and a well-typed file-system state.

Collusions may arise when a group of honest clients who share a secret want to protect the secret from the rest of the clients. These remaining clients are then assumed to

act adversarially by colluding. Specifically, when reasoning about L-secrets, we allow clients at levels $\sqsupseteq L$ and those at levels $\sqsubset L$ to form a pair of collusions and typecheck accordingly. Further, by monotonicity, well-typedness of processes is robust under arbitrary collusions with dishonest clients. Therefore, when reasoning under collusions that involve both honest and dishonest clients, it is sufficient to consider only the levels of the honest clients in the parameter to the typing relation.

Chapter 5

Dynamic access control and polymorphism

Systems that share services often exercise some control on access to those services at run time. Implicitly, access control is intended to be a means to enforce dynamic specifications for those services at run time. Unfortunately, such enforcement is not straightforward. For example, users who have access to a file with sensitive contents may share the contents, intentionally or by mistake, with those who do not have access; even if the privileged users are careful, dynamic access control may eventually allow other users to read those contents, or write over them.

A convenient view of access control in this setting results from its characterization in terms of capabilities: a service may be accessed if and only if a corresponding capability is shown for its access. This view is independent of higher level specifications on service usage (say, in terms of types, or identities of principals). It suffices to guarantee that the flow of a capability that protects a service respects the corresponding high-level intention on service usage. For example, in Plutus (Chapter 2) a file must be written or read with the correct write key or read key; it suffices to guarantee that those keys are distributed only to the intended sets of writers and readers.

This view in turn relies on a sound low-level implementation of access control in terms of capabilities. To a first approximation, a capability for a service can be identified with a link to that service. Exporting a direct link to a service, however, poses problems for dynamic access control, as discussed in [Redell, 1974]. Redell suggests a simple alternative that uses indirection: export a link to the direct link, and overwrite

the direct link to revoke access. Of course, it should be difficult to derive the direct link from the indirect link, for soundness. We revisit this idea in this chapter.

Our setting is a concurrent object language. Services are often built over other services; dependencies between services may entail dependencies on their access assumptions for end-to-end safety. For example, suppose that two users read the same file to obtain what they believe is a shared secret key, that they then use to encrypt secret messages between themselves; it does not help if a third user can write its own key on that file and then decrypt the "secret" messages. A natural way to capture such dependencies is to group the related services into objects. (In the example above, the object would be the file in question, and the services would be a `content` field that holds the key, and `read` and `write` methods that manipulate that field.)

Building on Redell's idea in this setting, we develop a variant of Gordon and Hankin's concurrent object calculus **conc**ς [Gordon and Hankin, 1998]. In **conc**ς, as in most previous object calculi (*e.g.*, [Abadi and Cardelli, 1995; Blasio and Fisher, 1996; Vasconcelos, 1994]), a method is accessed by providing the name of the host object and a label that identifies the method in that object. For example, for a timer object t with two methods, `set` and `tick`, knowing the name t is sufficient to call (or even redefine) both methods (t.`set`, t.`tick`) in **conc**ς. We may, however, want to restrict access to `set` to the owner of t, while allowing other users to access `tick`. Further, we may want to allow the owner of t to dynamically control access to `tick`. Such requirements are not directly supported by **conc**ς. In languages such as Java, there is limited support for access control via access modifiers—however, such modifiers are not flexible enough for our purposes.

Our calculus, **conc**$\widetilde{\imath}$, supports *method names* to facilitate access control on methods, and *indirections* to facilitate dynamic access control. More specifically, we let every method inside an object definition be linked with some method name v. The method is called by sending a message on the indirection $\widetilde{v}$ of v. We assume that the indirection is easy to compute but hard to invert, that is, the function $\widetilde{\cdot}$ is a one-way function. Access to the method is revoked simply by linking the method with a different name.

Crucially, calling a method does not require the name of the host object. Instead, the name of an object is required for redefining and controlling access to its methods.

Object and method names are meant to be shared between the owner and other administrators of objects; their indirections are meant to be made available to the users of methods. In analogy with Plutus, object and method names correspond to private keys and RSA seeds, which remain secret to the owners of files; their indirections correspond to public keys, lockbox keys, sign keys, and verify keys, which are secretly distributed to writers and readers of files.

Dependencies between the methods of an object often require their access control and redefinition to be atomic—therefore $\mathbf{conc}\tilde{\imath}$ replaces $\mathbf{conc}\varsigma$'s method update primitive with a more general object update primitive. In analogy with Plutus, new contents must be written and read with new keys following a revocation. These primitives can in turn encode the mutex primitives of $\mathbf{conc}\varsigma_{\mathbf{m}}$ (an extension of $\mathbf{conc}\varsigma$), which allow encodings of locks, channels, *etc*. in the language.

We show a type system for $\mathbf{conc}\tilde{\imath}$ that guarantees safe manipulation of objects with respect to dynamic specifications. The key idea behind the type system is the use of parametric polymorphism in object types. More precisely, we allow methods of an object to have different (dynamic) types at run time; each such type is derived from an instantiation of a polymorphic type for the host object. We then show that it is possible to dynamically enforce those types via access control. Roughly, our run-time invariant is that a method is always linked with a name of the correct type; as the type of the method varies, so must the name that the method is linked with. This invariant ensures that a method is always accessed in a type-safe manner at run time.

Our type system formalizes a common practice in various contexts: indeed, objects often dynamically implement various specifications at run time. For example, run-time access to a file may vary to dynamically reflect various secrecy assumptions for the contents of that file. By a combination of access control (provided by the language) and static discipline (provided by the type system) we can show that the intentions of the administrators of objects and users of methods are respected throughout such variations. In particular, by decorating types with secrecy groups, we show that well-typedness guarantees secrecy under dynamic access control, even in the presence of possibly untyped, active environments.

The rest of the chapter is organized as follows. In the next section we present a

concurrent object calculus, **conc** $\hat{\imath}$, with indirections for dynamic access control. We accompany the formal syntax and semantics of our language with commentary on the conceptual and technical differences with Gordon and Hankin's **conc**ς calculus. Finally, in Section 5.2, we present a type system for the language, show examples of well-typed programs, and state our main theorem, *viz.* typing guarantees secrecy under dynamically changing type views and even under untyped environments.

5.1 The untyped conc $\hat{\imath}$ calculus

In this section we present **conc** $\hat{\imath}$, a variant of the concurrent object calculus **conc**ς [Gordon and Hankin, 1998]. The novel aspects of **conc** $\hat{\imath}$ lie in the separation of roles for object update and method call; this separation is induced by introducing method names and indirections. The separation has a clear effect on the suitability of the resulting language as a core calculus for studying security properties of concurrent objects.

5.1.1 Syntax

We begin with the syntax of the language.

$u, v ::=$	values
x	name
$\hat{u}$	indirection
L	level

Values include names, indirections, and levels. The names may be variables, object names, or method names. The levels belong to a lattice $(\sqsupseteq, \sqcap, \sqcup, \top, \bot)$ with $\top \neq \bot$.

$d ::=$	denotations
$\overrightarrow{v_i}[\overrightarrow{\ell_i \Mapsto (y_i)b_i}]$	object

We use the notation $\overrightarrow{\varphi_i}$ to abbreviate a finite sequence $\ldots, \varphi_i, \ldots$ whose indices are drawn from some total order $\{\overrightarrow{i}\}$. An object $\overrightarrow{v_i}[\overrightarrow{\ell_i \Mapsto (y_i)b_i}]$ is defined by $|\{\overrightarrow{i}\}|$ methods; the i^{th} method, defined by the abstraction $(y_i)b_i$, is identified by the label ℓ_i, and is linked with the method name v_i. To simplify the presentation, we do not consider "self" abstractions (with the binder ς) as in **conc**ς [Gordon and Hankin, 1998].

$a, b ::=$	expressions
u	value
$u \mapsto d$	denomination
$(\nu x)\, a$	restriction
$a \Rsh b$	fork
let $x = a$ in b	evaluation
$\ell(u)$	internal method call
$\overrightarrow{\ell_i \Mapsto (y_i) b_i}$	internal object update
$v\langle u \rangle$	external method call
$u \longmapsfrom d$	external object update
if $u = v$ then a	equality check
if $L \sqsupseteq v$ then a	level subsumption check

There are separate "internal" and "external" primitives for method call and object update. The internal primitives $\ell(u)$ and $\overrightarrow{\ell_i \Mapsto (y_i) b_i}$ do not carry a reference to the host object, unlike [Gordon and Hankin, 1998]. In fact, labels by themselves have no meaning outside objects; hence the use of internal primitives is limited to within objects. The external primitives, on the other hand, can be used in any context. An external method call $v\langle u \rangle$ sends a message u on v, where v is the indirection of a method name. Crucially, calling a method does not require the name of the host object (*cf.* [Gordon and Hankin, 1998]). Instead, updating an object requires the name of the object. Specifically, the external object update $u \longmapsfrom d$ redefines some of the methods of the object named u, and relinks some of them with different method names. This primitive is a generalization of **conc**ς's method update primitive.

We also include two syntactic forms for dynamic checking (*viz.* equality and level subsumption). The rest of the syntax follows that of **conc**ς. Informally, expressions have the following meanings. (The formal semantics is shown later in the section.)

- u is a value that is returned by an expression.
- $u \mapsto d$ links the object d to the name u.
- $(\nu x)\, a$ creates a new name x that is bound in the expression a, and executes a.

- $a \Rsh b$ is the (non-commutative) parallel composition of the expressions a and b; it returns whatever value is returned by b, while executing a for side-effect. This form, introduced in [Ferreira et al., 1998], is largely responsible for the elegance of the syntax, since it provides an uniform way to write expressions that return values, and "processes" that exhibit behaviours. (Of course, expressions that return values can have side-effects.)

- let $x = a$ in b binds the value of the expression a to the variable x and then executes the expression b; here x is bound in b.

- $\ell(u)$ is a local method call inside an object; see external method call.

- $\overrightarrow{\ell_i \Mapsto (y_i)b_i}$ is a local object update; see external object update.

- $v\langle u \rangle$ is an external method call on the indirection v, with message u; the expression blocks until there is a denomination that contains a method $(y)b$ that is indirectly linked with v; in this case, the expression b is executed after substituting u for y, and *exporting* any internal primitives as external primitives (see below).

- $u \hookleftarrow d$ is an external object update; the expression blocks until there is a denomination of the form $u \mapsto d'$; in this case, the method names linked to d are overwritten with those linked to d', the methods defined by d are overwritten by those defined by d', and $\perp$ is returned.

- if $u = v$ then a executes a if u and v are the same value, and blocks otherwise.

- if $L \sqsupseteq v$ then a executes a if L is at least as high as level v, and blocks otherwise.

Example 5.1.1. Assume that integers and channels can be encoded in the language, and there is an indirection $\downarrow$ for decrementing positive integers and indirections $c!$ and $c?$ for sending and receiving messages on a channel.[1] Consider the following code.

[1] Both λ-calculus and π-calculus can be expressed in **conc** $\widehat{\imath}$, following standard translations of these languages to object languages such as **imp**ς and **conc**ς. The encodings of integers and channels can be built over these translations, although simpler encodings should suffice for this example.

$$
\begin{array}{rll}
\text{System} \stackrel{\text{def}}{=} & (\nu x, \mathbf{val}, \mathbf{set}, \mathbf{tick})\ (\text{Server}) \Rsh \text{Client} & \\
\text{Server} \stackrel{\text{def}}{=} & x \mapsto \mathbf{val}, \mathbf{set}, \mathbf{tick}[\ \texttt{val} \Rrightarrow (\_)\texttt{val}, & \text{\# timer linked to } x \text{, with} \\
& \quad \texttt{set} \Rrightarrow (y)\ \texttt{val} \Rrightarrow (\_)y, & \text{\# } \texttt{set} \text{ linked to } \mathbf{set} \\
& \quad \texttt{tick} \Rrightarrow (\_)\ \text{let } z = \texttt{val}(\bot) \text{ in} & \text{\# } \texttt{tick} \text{ linked to } \mathbf{tick} \\
& \qquad\qquad \text{let } z' = \downarrow\langle z\rangle \text{ in } \texttt{set}(z')\ ] \Rsh & \\
& \grave{\mathbf{set}}\langle N\rangle \Rsh c!\langle \grave{\mathbf{tick}}\rangle \Rsh \ldots \Rsh & \text{\# timer gets activated...} \\
& (\nu \mathbf{tick}')\ x \hookleftarrow \mathbf{tick}'[\ ] & \text{\# timer gets deactivated} \\
\text{Client} \stackrel{\text{def}}{=} & \text{let } z = c?\langle\bot\rangle \text{ in } (z\langle\bot\rangle \Rsh \ldots \Rsh z\langle\bot\rangle) & \text{\# timer ticks}
\end{array}
$$

A server creates a new timer object, links the `tick` and `set` methods of the timer to the names **tick** and **set**, sets the value of the timer to an integer N by calling $\grave{\mathbf{set}}$, and sends the indirection $\grave{\mathbf{tick}}$ on $c!$. A client repeatedly ticks the timer by calling $\grave{\mathbf{tick}}$. At some point, the server creates a fresh method name and relinks the `tick` method of the timer object to this name. Consequently, the client can no longer tick the timer.

5.1.2 Semantics

We show a chemical semantics for the language, much as in [Gordon and Hankin, 1998; Flanagan and Abadi, 1999], with the following grammar of evaluation contexts.

$\mathcal{E}$ ::=	evaluation contexts
$\bullet$	hole
$\text{let } x = \mathcal{E} \text{ in } b$	evaluation
$\mathcal{E} \Rsh b$	fork side
$a \Rsh \mathcal{E}$	fork main
$(\nu x)\ \mathcal{E}$	restriction

Informally, an evaluation context is an expression container with exactly one hole. By plugging an expression a into the hole of an evaluation context $\mathcal{E}$, we obtain the expression $\mathcal{E}[\![a]\!]$. (In general, plugging may not be capture-free with respect to names or variables.) We define structural congruence of expressions as usual.

Structural congruence $a \equiv b$ # `fn` (*resp.* `bn`) collects free (*resp.* bound) variables

$$\frac{x \notin \mathtt{fn}(\mathcal{E}) \cup \mathtt{bn}(\mathcal{E})}{(\nu x)\,\mathcal{E}[\![a]\!] \equiv \mathcal{E}[\![(\nu x)\,a]\!]}\ (\textsc{Struct Res}) \qquad \frac{\mathtt{fn}(a) \cap \mathtt{bn}(\mathcal{E}) = \varnothing}{a \Rsh \mathcal{E}[\![b]\!] \equiv \mathcal{E}[\![a \Rsh b]\!]}\ (\textsc{Struct Par})$$

(STRUCT EQV) $\equiv$ is an equivalence

Next, we define reduction of expressions. Not surprisingly, there are no reduction rules for internal primitives: we restrict the sites of action to the external primitives. The reductions for external method call and object update, (Red Call) and (Red Upd), have some important differences from the corresponding reductions in **conc**ς. First, when a method expression is executed on call reduction, the labels in the expression are eliminated by translating internal calls to external calls, and internal updates to external updates. (This translation, called *export*, is shown below.) Second, an object update can not only redefine some methods, but also relink some methods with different names. In general, the update can block or unblock some external method calls: thus it serves as an access control mechanism in the language.

In the following, let $\overrightarrow{\varphi'_j} \circ \overrightarrow{\varphi_i} = \overrightarrow{\varphi'_j} \cup \overrightarrow{\varphi_k}$, where $\{\,\overrightarrow{k}\,\} = \{\,\overrightarrow{i}\,\} \setminus \{\,\overrightarrow{j}\,\}$.

Structural reduction $a \longrightarrow b$

$$\frac{d = \overrightarrow{v_i}[\overrightarrow{\ell_i \Mapsto (y_i)b_i}]}{(u \mapsto d) \Rsh \widehat{v_i}\langle u' \rangle \longrightarrow (u \mapsto d) \Rsh b_i\{u'/y_i\} \downarrow_u^{\overrightarrow{v_i}}}\ (\textsc{Red Call})$$

$$\frac{d = \overrightarrow{v_i}[\overrightarrow{\ell_i \Mapsto (y_i)b_i}] \qquad d' = \overrightarrow{v'_j}[\overrightarrow{\ell_k \Mapsto (y'_k)b'_k}] \qquad \{\,\overrightarrow{j}\,\} \cup \{\,\overrightarrow{k}\,\} \subseteq \{\,\overrightarrow{i}\,\} \qquad d'' = \overrightarrow{v'_j} \circ \overrightarrow{v_i}[\overrightarrow{\ell_k \Mapsto (y'_k)b'_k} \circ \overrightarrow{\ell_i \Mapsto (y_i)b_i}]}{(u \mapsto d) \Rsh u \hookleftarrow d' \longrightarrow (u \mapsto d'') \Rsh \bot}\ (\textsc{Red Upd})$$

(RED EVAL) $\mathsf{let}\ x = u\ \mathsf{in}\ b \longrightarrow b\{u/x\}$ $\qquad \dfrac{L \sqsupseteq L'}{\mathsf{if}\ L \sqsupseteq L'\ \mathsf{then}\ a \longrightarrow a}$ $\qquad \mathsf{if}\ u = u\ \mathsf{then}\ a \longrightarrow a$

$$\frac{a \longrightarrow b}{\mathcal{E}[\![a]\!] \longrightarrow \mathcal{E}[\![b]\!]}\ (\textsc{Red Context}) \qquad \frac{a \equiv a' \qquad a' \longrightarrow b' \qquad b' \equiv b}{a \longrightarrow b}\ (\textsc{Red Struct})$$

In (RED CALL), the appropriate method is dispatched after eliminating its labels by the translation $\downarrow_u^{\overrightarrow{v_i}}$; here, u is the name of the host object and $\overrightarrow{v_i}$ contains the names

of the methods of that object, at the time of dispatch. Eliminating labels assigns a definite meaning to the method expression outside the syntactical scope of the host object. More importantly, eliminating labels ensures that the execution of the method expression is type safe amidst future object updates (Section 5.2) .

Export $a \downarrow_u^{\overrightarrow{v_i}}$

$$\ell_i(u') \downarrow_u^{\overrightarrow{v_i}} \stackrel{\text{def}}{=} \grave{v}_i\langle u' \rangle \qquad \overrightarrow{\ell_k \Mapsto (y_k) b_k} \downarrow_u^{\overrightarrow{v_i}} \stackrel{\text{def}}{=} u \leftarrowtail \overrightarrow{v_i}[\overrightarrow{\ell_k \Mapsto (y_k) b_k \downarrow_u^{\overrightarrow{v_i}}}]$$

$$(a \Rsh b) \downarrow_u^{\overrightarrow{v_i}} \stackrel{\text{def}}{=} a \downarrow_u^{\overrightarrow{v_i}} \Rsh b \downarrow_u^{\overrightarrow{v_i}} \qquad (\text{let } x = a \text{ in } b) \downarrow_u^{\overrightarrow{v_i}} \stackrel{\text{def}}{=} \text{let } x = a \downarrow_u^{\overrightarrow{v_i}} \text{ in } b \downarrow_u^{\overrightarrow{v_i}}$$

$$((\nu n)\, a) \downarrow_u^{\overrightarrow{v_i}} \stackrel{\text{def}}{=} (\nu n)\, a \downarrow_u^{\overrightarrow{v_i}} \qquad \frac{a = u',\ v\langle u' \rangle,\ u' \mapsto d, \text{ or } u' \leftarrowtail d}{a \downarrow_u^{\overrightarrow{v_i}} \stackrel{\text{def}}{=} a}$$

To illustrate the semantics, next we show some sample reductions for parts of the code of Example 5.1.1. Here, let $\overrightarrow{m} = \mathbf{val}, \mathbf{set}, \mathbf{tick}$.

$$\begin{aligned} x \mapsto \overrightarrow{m}[\ldots] \Rsh \grave{\mathbf{set}}\langle N \rangle &\longrightarrow x \mapsto \overrightarrow{m}[\ldots] \Rsh x \leftarrowtail \overrightarrow{m}[\mathtt{val} \Mapsto (\_)N] \\ &\longrightarrow x \mapsto \overrightarrow{m}[\mathtt{val} \Mapsto (\_)N, \ldots] \Rsh \bot \qquad \text{\# activate} \end{aligned}$$

$$\begin{aligned} x \mapsto \overrightarrow{m}[\ldots] \Rsh \grave{\mathbf{tick}}\ \langle \bot \rangle &\longrightarrow n \mapsto \overrightarrow{m}[\ldots] \Rsh \\ &\qquad \text{let } z = \grave{\mathbf{val}}\langle \bot \rangle \text{ in let } z' = \downarrow \langle z \rangle \text{ in } \grave{\mathbf{set}}\langle z' \rangle \\ &\longrightarrow^{\star} x \mapsto \overrightarrow{m}[\mathtt{val} \Mapsto (\_)N - 1, \ldots] \Rsh \bot \qquad \text{\# tick} \end{aligned}$$

$$x \mapsto \overrightarrow{m}[\ldots] \Rsh x \leftarrowtail \mathbf{tick}'[\,] \longrightarrow x \mapsto \mathbf{val}, \mathbf{set}, \mathbf{tick}'[\ldots] \Rsh \bot \qquad \text{\# deactivate}$$

5.2 A type system for enforcing dynamic specifications

In this section we show a type system that can enforce dynamic specifications in **conc**$\grave{\imath}$. Specifically, we allow a method to have various types at run time: the type of a method is dynamically related to the type of the name it is linked with. For example, suppose that the owner of a file wants to change the type of the content field from "public" to "secret". Clearly, the name linked to the content field must be changed:

while the indirection of the previous name could have been public, the indirection of the new name has to be secret. Further, if the file has read and write methods that depend on the content field, their types change accordingly: therefore the names linked to these methods must be changed as well. In analogy with Plutus, the write and read keys must be changed whenever the type of contents are changed.

Changing method names is, however, not enough for end-to-end secrecy. (This inadequacy is typical of access control mechanisms, as mentioned earlier in this chapter.) A user that reads the file by calling the new indirection may regard the content as secret (even if it is not). For example, the user may read some (previously public) content v by calling the new indirection, believe that v is secret, and thence set $\tilde{v}$ as an indirection to read some other secret: unfortunately, that "secret" can be publicly read by calling $\tilde{v}$. Indeed, it is almost always possible to exploit such "type interpretation" errors to leak secrets. (For example, interpreting secret content as public can be equally bad.) To prevent such errors, the content field must be overwritten to reflect its new type. In analogy with Plutus, the new write and read keys must be used to write and read new contents; using the new keys without overwriting the content field can to dangerous type interpretation errors.

Going further, by the same argument, it appears that the read and write methods need to be overwritten as well. We can however do better. Typically read and write have types that are parametric with respect to the type of the content: informally, whenever the content type is X (say, instantiated to "public" or "secret"), the read and write methods have types $(\bot)X$ and $(X)\bot$. Therefore, those methods reflect their new types as soon as the content field is overwritten.

We summarize these insights in the following general principles that govern the type system below. First, an object update is consistent only if the types of the new method names match up with the types of the method definitions. Second, type consistency forces some methods to be overwritten. Methods which are parametric with respect to the overwritten methods, however, need not be overwritten. This form of polymorphism is typically exhibited by higher-order (generic) functions, compositionally defined procedures, or (in the degenerate case) methods that have static types, *i.e.*, whose types do not change.

5.2.1 Polymorphic types, constraints, and subtyping

The primary goal of our type discipline is type safety, despite dynamic variations of types for methods. We rely on a combination of access control and polymorphic typing to enforce this safety property. More specifically, we qualify methods with signs $\delta \in \{+, -\}$. The $-$ methods *must* be overwritten whenever method names are changed. In contrast, the $+$ methods *may* be overwritten, and if they are, they must remain polymorphically typed, as indicated above.

As in Chapter 4, we interpret levels as (secrecy) groups and lift them to types. Further, every type is associated with a group, which we call its reach. Thus, the type declaration for a name specifies the group within which that name is intended to be confined. We use type safety to verify that each such intention is preserved at run time.

Let $\mathcal{X}$ range over sequences of type variables. We allow universal quantification of type variables in object types; such variables can be shared by the types of the methods of those objects. Further, we allow universal quantification of type variables in method types. Finally, we allow existential quantification of type variables in all types, and allow types to carry subtyping constraints over the type variables in scope.

More specifically, the syntax of types is as follows. (We use the notation $\varphi\ldots$ to abbreviate a finite string beginning in φ.)

$S, T ::=$	types
X	type variable (declared)
$U^{G\ldots}$	limit (declared)
G	group (declared)
$S \mid \mathcal{C}$	constraint
$(\exists \mathcal{X})T$	existential type
$U ::=$	type schemes
$\forall \mathcal{X}[\overrightarrow{\ell_i^{\delta_i} : \forall \mathcal{Y}_i(S_i)T_i}]$	object type scheme
$\forall \mathcal{Y}\langle S\rangle T$	method type scheme
$G ::=$	groups
X	group variable
L	level

	$G \sqcup G'$	supremum
	$G \sqcap G'$	infimum
	$\|X\|$	reach of type variable (uninterpreted)
$\mathcal{C}$ ::=		constraints
	$S \leq T$	type subsumption
	$\mathcal{C} \Rightarrow \mathcal{C}'$	implication

Typed processes declare types for new names (with $(\nu x : T)\ a$, instead of $(\nu x)\ a$ in Section 5.1). Such types are restricted to type variables, limit types, and group types; constraints and existential types cannot be declared for new names.

Informally, the type forms have the following meanings:

- The type variable X may denote a type, and in particular, a group.
- The object type scheme $\forall \mathcal{X}[\overrightarrow{\ell_i^{\delta_i} : \forall \mathcal{Y}_i(S_i)T_i}]$ assigns the type schemes $\forall \mathcal{Y}_i\langle S_i\rangle T_i$ and the signs δ_i to the methods ℓ_i of an object. The type variables $\mathcal{X}$ may be shared by those schemes. Intuitively, at run time, these variables can be substituted with different concrete types, resulting in different concrete types for the methods. We maintain, for each method ℓ_i with $\delta_i = +$, the following invariant: for any substitution of $\mathcal{X}$, if every method in the object is assumed to have its assigned type, then the expression for ℓ_i has its assigned type.
- The method type scheme $\forall \mathcal{Y}\langle S\rangle T$ assigns a polymorphic type to a method that takes a value of type S and returns a value of type T. The type variables $\mathcal{Y}$ may be shared by S and T.
- The limit type $U^{G\cdots}$ is given to a value that should be secret within the group G. Further, the indirection of that value is given the type $U^{\grave{G}\cdots}$, defined as follows:

  $$U^{\grave{G}} = U^G \qquad U^{G\grave{G}'\cdots} = U^{G'\cdots}$$

  That is, for $i \geq 1$, the $(i-1)^{th}$ successive indirection of a value of type $U^{G_1\cdots G_k}$ is given the type $U^{G_i\cdots G_k}$ if $i < k$, and U^{G_k} if $i \geq k$.

  For instance, the type $\forall \mathcal{X}[\overrightarrow{\ell_i^{\delta_i} : \forall \mathcal{Y}_i(S_i)T_i}]^G$ may be given to an object name that should be secret within G, the group of administrators of that object. The type

$(\forall \mathcal{Y}_i(S_i\sigma)T_i\sigma)^{GG_i}$ may be given to a method name linked to ℓ_i, where σ is some type substitution for the variables $\mathcal{X}$. While the method name itself should be secret within G, its indirection, which is given the type $(\forall \mathcal{Y}_i(S_i\sigma)T_i\sigma)^{G_i}$, should be secret within G_i, the group of users of that method.

- The group type G is given to a level that is at least as high as G; further, the unit type $\bot$ is given to a value that may be known to the public group $\bot$.
- The constraint type $S \mid \mathcal{C}$ is given to a value of type S under the constraint $\mathcal{C}$. The constraint can be any simple logical formula over subtyping assertions.
- The existential type $(\exists \mathcal{X})S$ is given to a value of type S under some substitution of the type variables $\mathcal{X}$.

For example, the name of a file object may be given the following type (eliding useless quantifiers):

$$\forall X[\texttt{content}^- : (\bot)X,\ \texttt{read}^+ : (\bot)X,\ \texttt{write}^+ : (X)\bot]^{\text{Owner}\bot}$$

The indirection of the object name may be distributed as the file path, with the type:

$$\forall X[\texttt{content}^- : (\bot)X,\ \texttt{read}^+ : (\bot)X,\ \texttt{write}^+ : (X)\bot]^{\bot}$$

If, say, the content is of type T, a method name linked to `write` may be given the type:

$$\langle T\rangle\bot^{\text{ Owner Writer}}$$

The indirection of the method name may be distributed as the `write` capability, with the type:

$$\langle T\rangle\bot^{\text{Writer}}$$

As another example, consider an authenticated encryption object, whose name is given the type:

$$\forall X[\texttt{key}^- : (\bot)X, \texttt{authencrypt}^+ : \forall Y(Y)(\langle X\rangle Y)^{\bot}]^{\text{Authority}}$$

The value returned by encryption may be known to the public group $\bot$. If the type of the key is T, then the `key` and `authencrypt` capabilities may be given the types

$$(\langle\bot\rangle T)^{\text{Reader}} \qquad\qquad \forall Y(\langle Y\rangle(\langle T\rangle Y)^{\bot})^{\text{Writer}}$$

Examples of constraint types and existential types appear in more sophisticated applications, such as in Appendix C, that require typing information to be propagated across contexts for assume/guarantee-style reasoning.

The relationship between types and groups is made explicit by a *reach* function, defined below. Informally, the reach of a type is the group within which the values of that type may be shared (but not without). All groups have reach $\bot$. The group at the head of a limit type is the reach of that type. Reaches are propagated trivially through constraint types and existential types. The reach of a type variable is left uninterpreted.

Type reach $\|T\|$

$$\|U^{G\cdots}\| = G \qquad \|G\| = \bot \qquad \|S \mid \mathcal{C}\| = \|S\| \qquad \|(\exists \mathcal{X})S\| = \|S\|$$

Let Γ be a sequence of type assumptions $x : T$. The typing rules judge well-formed assumptions $\Gamma \vdash \diamond$, well-formed types $\Gamma \vdash T$, valid inference of constraints $\Gamma \vdash \mathcal{C}$, and well-typed expressions $\Gamma \vdash a : T$. We show the core typing rules for well-typed expressions in Section 5.2.3; the other rules are presented below.

Typing rules $\Gamma \vdash \diamond$

(HYP $\varnothing$)
$$\varnothing \vdash \diamond$$

(HYP TYP)
$$\frac{\Gamma \vdash T \qquad u : \_ \notin \{\Gamma\}}{\Gamma, x : T \vdash \diamond}$$

(HYP VAR)
$$\frac{\Gamma \vdash \diamond \qquad X \notin \Gamma}{\Gamma, X \vdash \diamond}$$

(HYP $\leq$)
$$\frac{\Gamma \vdash S, T}{\Gamma, S \leq T \vdash \diamond}$$

(HYP $\Rightarrow$)
$$\frac{\Gamma, \mathcal{C}, \mathcal{C}' \vdash \diamond}{\Gamma, \mathcal{C} \Rightarrow \mathcal{C}' \vdash \diamond}$$

By (HYP TYP), well-formed environments can introduce well-formed type assumptions for fresh variables. Further, by (HYP VAR), they can introduce fresh type variables. Finally, by (HYP $\leq$) and (HYP $\Rightarrow$), they can introduce subtyping constraints over well-formed types.

Typing rules $\Gamma \vdash T$

(TYP HYP)
$$\frac{X \in \{\Gamma\}}{\Gamma \vdash X}$$

(TYP LEV)
$$\Gamma \vdash L$$

(TYP $\|\cdot\|$)
$$\frac{\Gamma \vdash X}{\Gamma \vdash \|X\|}$$

(TYP $\sqcap$)
$$\frac{\Gamma \vdash G, G'}{\Gamma \vdash G \sqcap G'}$$

(TYP $\sqcup$)
$$\frac{\Gamma \vdash G, G'}{\Gamma \vdash G \sqcup G'}$$

$$\text{(Typ Obj)}\quad \frac{\Gamma \vdash G \qquad \Gamma, \mathcal{X} \vdash \diamond \qquad \overrightarrow{\ell_i} \text{ distinct} \qquad \forall i.\, \Gamma, \mathcal{X}, \mathcal{Y}_i \vdash S_i, T_i}{\Gamma \vdash \forall \mathcal{X}[\overrightarrow{\ell_i^{\delta_i} : \forall \mathcal{Y}_i (S_i) T_i}]^G} \qquad \text{(Typ Meth)}\quad \frac{\Gamma \vdash G \qquad \Gamma, \mathcal{Y} \vdash S, T}{\Gamma \vdash (\forall \mathcal{Y} \langle S \rangle T)^G}$$

$$\text{(Typ Lim)}\quad \frac{\Gamma \vdash U^{G' \cdots} \qquad \Gamma \vdash G \leq G'}{\Gamma \vdash U^{GG' \cdots}} \qquad \text{(Typ Cons)}\quad \frac{\Gamma, \mathcal{C} \vdash S}{\Gamma \vdash S \mid \mathcal{C}} \qquad \text{(Typ } \exists\text{)}\quad \frac{\Gamma, \mathcal{X} \vdash T}{\Gamma \vdash (\exists \mathcal{X}) T}$$

In the rules for $\Gamma \vdash T$, we implicitly require $\Gamma \vdash \diamond$ in the antecedent. By (Typ Hyp), a type variable is well-formed if it is already introduced in the environment. Indeed, all type variables that may occur in well-formed types must be introduced explicitly through quantification. By (Typ Obj), an object type is well-formed if all method labels in it are distinct, and the type variables quantified in the scheme have the correct scopes. Similarly, by (Typ Meth) the type variables quantified in the scheme have the correct scopes.

By (Typ Lim), a type of the form $U^{GG' \cdots}$ is well-formed only if the type $U^{G' \cdots}$ is well-formed, and G is a subtype of G'. Indeed, for any value v, any group G that may know v may also know its indirection $\hat{v}$; in particular, if v has type $U^{GG' \cdots}$, then $\hat{v}$ has type $U^{G' \cdots}$, so G must be at least as high as G', the group that may know $\hat{v}$.

By (Typ Cons), a type of the form $S \mid \mathcal{C}$ is well-formed if S is well-formed in an environment that remains well-formed after introducing $\mathcal{C}$. The remaining rules for well-formed environments are straightforward.

Typing rules $\Gamma \vdash \mathcal{C}$, $\Gamma \vdash a : T$

$$\text{(Sub } \bot\text{)}\quad \frac{\Gamma \vdash \bot \leq G}{\Gamma \vdash U^G \leq \bot} \qquad \text{(Sub Latt)}\quad \frac{L \sqsupseteq L'}{\Gamma \vdash L \leq L'} \qquad \text{(Sub Latt)}\quad \frac{L' \sqsupset L}{\Gamma \vdash L \leq L' \Rightarrow \bot \leq \top}$$

$$\text{(Sub } \bot\text{)}\quad \Gamma \vdash G \leq \bot \qquad \text{(Sub } \top\text{)}\quad \Gamma \vdash \top \leq G$$

$$\text{(Sub } \sqcap\text{)}\quad \Gamma \vdash G \sqcap G' \leq G,\ G \sqcap G' \leq G' \qquad \text{(Sub } \sqcup\text{)}\quad \Gamma \vdash G \leq G \sqcup G',\ G' \leq G \sqcup G'$$

$$\text{(Sub Ref)}\quad \Gamma \vdash S \leq S \qquad \text{(Sub Tran)}\quad \frac{\Gamma \vdash S \leq S' \qquad \Gamma \vdash S' \leq T}{\Gamma \vdash S \leq T} \qquad \text{(Exp Sub)}\quad \frac{\Gamma \vdash a : S \qquad \Gamma \vdash S \leq T}{\Gamma \vdash a : T}$$

$$\frac{\mathcal{C} \in \{\Gamma\}}{\Gamma \vdash \mathcal{C}}\ (\textsc{Cons Hyp}) \qquad \frac{\Gamma, \mathcal{C} \vdash \mathcal{C}' \qquad \Gamma \vdash \mathcal{C}}{\Gamma \vdash \mathcal{C} \Rightarrow \mathcal{C}'}\ (\textsc{Cons} \Rightarrow \textsc{intro}) \qquad \frac{\Gamma \vdash \mathcal{C} \Rightarrow \mathcal{C}' \qquad \Gamma \vdash \mathcal{C}}{\Gamma \vdash \mathcal{C}'}\ (\textsc{Cons} \Rightarrow \textsc{elim})$$

$$\frac{\Gamma, \mathcal{C} \vdash a : T \qquad \Gamma, \mathcal{C} \Rightarrow \bot \leq \top \vdash a : T}{\Gamma \vdash a : T}\ (\textsc{Cons Case}) \qquad \frac{\Gamma \vdash \bot \leq \top}{\Gamma \vdash a : T}\ (\textsc{Exp Cons}) \qquad \frac{\Gamma \vdash S \leq T}{\Gamma \vdash \|S\| \leq \|T\|,\ \|T\| \leq \|S\|}\ (\textsc{Cons Sub})$$

In the rules for $\Gamma \vdash \mathcal{C}$, we implicitly require $\Gamma, \mathcal{C} \vdash \diamond$ in the antecedent. The subtyping rules generalize those in Chapter 4 to account for group variables. For example, we use $\Gamma \vdash \bot \leq G$ instead of $G \sqsubseteq \bot$, since G may be a group variable constrained by Γ. Similarly, we have the obvious rules (Sub $\bot$), (Sub $\top$), (Sub $\sqcup$), and (Sub $\sqcap$) to extend $\sqsupseteq$ to group variables. By (Sub $\bot$), if $\Gamma \vdash \bot \leq G$ then U^G is a public type, as expected. The rules (Cons Hyp), (Cons $\Rightarrow$ intro), (Cons $\Rightarrow$ elim), (Cons Case), and (Exp Cons) implement a classical proof system for simple logical formulae over subtyping constraints, where $\bot \leq \top$ is considered a contradiction. Finally, (Cons Sub) axiomatizes the condition that reaches are preserved by subtyping, so that subtyping constraints over types can derive subtyping constraints on groups. The remaining rules for valid inference of subtyping constraints are straightforward.

5.2.2 Static invariants

Before going any further, let us review some of the static invariants that are captured by types, and how those invariants are maintained.

Consider a name of type $\forall \mathcal{X}[\overrightarrow{\ell_i^{\delta_i} : \forall \mathcal{Y}_i \langle S_i \rangle T_i}]^{G \cdots}$. By well-formedness, we can assume that all quantified type variables in this type are distinct. Further, we can assume that this name is associated with an object of the form $\overrightarrow{v_i}[\overrightarrow{\ell_i \Mapsto (y_i) b_i}]$. For some type substitution σ of the type variables $\mathcal{X}$, we have that each v_i has type scheme $\forall \mathcal{Y}_i \langle S_i \sigma \rangle T_i \sigma$, and accordingly, each b_i has type $T_i\sigma$ assuming y_i is of type $S_i\sigma$. In fact, the invariant that we maintain is somewhat stronger, since we need to prepare for future updates of this object; let us consider such an update next.

Specifically, consider an update of this object with the denotation $\overrightarrow{v_j}[\overrightarrow{\ell_k \Mapsto (y_k) b_k}]$. We assume that $\overrightarrow{j}$ and $\overrightarrow{k}$ are both subsets of $\overrightarrow{i}$, but not necessarily equal to $\overrightarrow{i}$; we

need to the maintain the invariants above for the resulting object. At the very least, we require that for some type substitution σ' of the type variables $\mathcal{X}$, each v_j has type scheme $\forall \mathcal{Y}_j \langle S_j \sigma' \rangle T_j \sigma'$. But this is clearly not enough; for any $i \notin \{ \overrightarrow{j} \}$, recall that v_i is typed under σ, not σ'. It follows that for any such i, no type variable in $\mathcal{X}$ may appear in the type scheme $\forall \mathcal{Y}_i \langle S_i \rangle T_i$, that is, S_i and T_i must already be well-formed under $\mathcal{Y}_i$.

At this point, we can fix the type substitution σ'. Now, at the very least, we require that each b_k is of type $T_k \sigma'$ assuming y_k is of type S_k. But this is clearly not enough; for any $i \notin \{ \overrightarrow{k} \}$, recall that $(y_i) b_i$ is typed under σ, not σ'. Therefore, we need to strengthen our invariants.

Specifically, we assume that $\{ \overrightarrow{k} \}$ includes all i such that $\delta_i = -$. If $\delta_i = +$, we ensure that $(y_i) b_i$ remains typed under σ', and indeed, under any future type substitution, as follows. We simply require that such b_i have type T_i assuming y_i is of type S_i (without substituting the type variables $\mathcal{X}$ in S_i and T_i).

At this point, we are almost done. Note that if the indirection of a method name may be public, then—irrespective of the type scheme for the method—the adversary can call that method with any public value, and the result of executing that method should be public. For any j, the reach G_j of the type of $\tilde{v_j}$ may be $\bot$ iff $\bot \leq G_j$ does not introduce a contradiction; we require that any such j be in $\{ \overrightarrow{k} \}$, and b_j have type $\bot$ assuming y_j is of type $\bot$.

Formally, the condition $\textsc{Invariance}(\overrightarrow{i}, \overrightarrow{j}, \overrightarrow{k})$, parameterized by the indices $\overrightarrow{i}$, $\overrightarrow{j}$, and $\overrightarrow{k}$, collects all the above requirements for object update, and is derived by the following rule.

$$\frac{\begin{array}{c} \mathtt{dom}(\sigma) = \mathcal{X} \qquad \forall j.\, \Gamma \vdash v_j : (\forall \mathcal{Y}_j \langle S_j \sigma \rangle T_j \sigma)^{GG_j \cdots} \\ \{ i \mid \Gamma, \mathcal{Y}_i \not\vdash S_i, T_i \} \subseteq \{ \overrightarrow{j} \} \subseteq \{ \overrightarrow{i} \} \\ \{ i \mid \delta_i = - \} \cup \{ j \mid \Gamma, \bot \leq G_j \not\vdash \bot \leq \top \} \subseteq \{ \overrightarrow{k} \} \subseteq \{ \overrightarrow{i} \} \\ \forall k.\, \delta_k = - \;\Rightarrow\; \begin{cases} \Gamma, \overrightarrow{Z_i}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i \sigma \rangle T_i \sigma)^{GZ_i}}, \mathcal{Y}_k, y_k : S_k \sigma \vdash b_k \sigma \not\downarrow_u^{\overrightarrow{z_i}} : T_k \sigma \\ k \in \{ \overrightarrow{j} \} \Rightarrow \Gamma, \overrightarrow{Z_i}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i \sigma \rangle T_i \sigma)^{GZ_i}}, \mathcal{Y}_k, y_k : \bot, \bot \leq G_k \vdash b_k \sigma \not\downarrow_u^{\overrightarrow{z_i}} : \bot \end{cases} \\ \forall k.\, \delta_k = + \;\Rightarrow\; \begin{cases} \Gamma, \overrightarrow{Z_i}, \mathcal{X}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i \rangle T_i)^{GZ_i}}, \mathcal{Y}_k, y_k : S_k \vdash b_k \not\downarrow_u^{\overrightarrow{z_i}} : T_k \\ k \in \{ \overrightarrow{j} \} \Rightarrow \Gamma, \overrightarrow{Z_i}, \mathcal{X}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i \rangle T_i)^{GZ_i}}, \mathcal{Y}_k, y_k : \bot, \bot \leq G_k \vdash b_k \not\downarrow_u^{\overrightarrow{z_i}} : \bot \end{cases} \end{array}}{\textsc{Invariance}(\overrightarrow{i}, \overrightarrow{j}, \overrightarrow{k})}$$

The condition $\textsc{Invariance}(\overrightarrow{i}, \overrightarrow{i}, \overrightarrow{i})$ for object initialization is just a special case.

$$\frac{\begin{array}{l}\forall i.\ \delta_i = -\ \Rightarrow \begin{cases} \mathtt{dom}(\sigma) = \mathcal{X} \qquad \forall i.\ \Gamma \vdash v_i : (\forall \mathcal{Y}_i \langle S_i\sigma \rangle T_i\sigma)^{GG_i \cdots} \\ \Gamma, \overrightarrow{Z_i}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i\sigma \rangle T_i\sigma)^{GZ_i}}, \mathcal{Y}_i, y_i : S_i\sigma \vdash b_i\sigma \not\downarrow_u^{\overrightarrow{z_i}} : T_i\sigma \\ \Gamma, \overrightarrow{Z_i}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i\sigma \rangle T_i\sigma)^{GZ_i}}, \mathcal{Y}_i, y_i : \bot, \bot \leq G_i \vdash b_i\sigma \not\downarrow_u^{\overrightarrow{z_i}} : \bot \end{cases} \\ \forall i.\ \delta_i = +\ \Rightarrow \begin{cases} \Gamma, \overrightarrow{Z_i}, \mathcal{X}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i \rangle T_i)^{GZ_i}}, \mathcal{Y}_i, y_i : S_i \vdash b_i \not\downarrow_u^{\overrightarrow{z_i}} : T_i \\ \Gamma, \overrightarrow{Z_i}, \mathcal{X}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i \rangle T_i)^{GZ_i}}, \mathcal{Y}_i, y_i : \bot, \bot \leq G_i \vdash b_i \not\downarrow_u^{\overrightarrow{z_i}} : \bot \end{cases} \end{array}}{\textsc{Invariance}(\overrightarrow{i}, \overrightarrow{i}, \overrightarrow{i})}$$

5.2.3 Core typing rules

We now present our core typing rules for expressions.

Typing rules $\Gamma \vdash a : T$

$$\frac{x : T \in \{\Gamma\}}{\Gamma \vdash x : T}\ (\textsc{Exp Hyp}) \qquad \Gamma \vdash L : L\ (\textsc{Exp Lev}) \qquad \frac{\Gamma \vdash v : U^{G\cdots}}{\Gamma \vdash \widehat{v} : U^{\widehat{G}\cdots}}\ (\textsc{Exp Ind}) \qquad \frac{\Gamma \vdash v : \bot}{\Gamma \vdash \widehat{v} : \bot}\ (\textsc{Exp Ind}\ \bot)$$

$$\frac{\mathtt{dom}(\sigma) = \mathcal{X} \qquad \Gamma \vdash a : T\sigma}{\Gamma \vdash a : (\exists \mathcal{X})T}\ (\textsc{Exp}\ \exists\ \textsc{intro}) \qquad \frac{\Gamma, \mathcal{X}, x : S \vdash a : T}{\Gamma, x : (\exists \mathcal{X})S \vdash a : T}\ (\textsc{Exp}\ \exists\ \textsc{elim})$$

$$\frac{\Gamma \vdash a : T \qquad \Gamma \vdash \mathcal{C}}{\Gamma \vdash a : T \mid \mathcal{C}}\ (\textsc{Exp Cons intro}) \qquad \frac{\Gamma, x : S, \mathcal{C} \vdash a : T}{\Gamma, x : S \mid \mathcal{C} \vdash a : T}\ (\textsc{Exp Cons elim})$$

$$\frac{S \text{ declared} \qquad \Gamma, n : S \vdash a : T}{\Gamma \vdash (\nu n : S)\, a : T}\ (\textsc{Exp New}) \qquad \frac{\Gamma \vdash a : S \qquad \Gamma \vdash b : T}{\Gamma \vdash a \Rsh b : T}\ (\textsc{Exp Fork}) \qquad \frac{\Gamma \vdash a : S \qquad \Gamma, x : S \vdash b : T}{\Gamma \vdash \text{let } x = a \text{ in } b : T}\ (\textsc{Exp Eval})$$

$$\frac{\Gamma \vdash u : T \qquad \Gamma, L \leq T \vdash a : T}{\Gamma \vdash \text{if } L \sqsupseteq u \text{ then } a : T}\ (\textsc{Exp} \sqsupseteq) \qquad \frac{\Gamma \vdash u : S \qquad \Gamma \vdash v : S' \qquad \Gamma, X, X \leq S, X \leq S' \vdash a : T}{\Gamma \vdash \text{if } u = v \text{ then } a : T}\ (\textsc{Exp} =)$$

$$\frac{\Gamma \vdash v : (\forall \mathcal{Y} \langle S \rangle T)^{G\cdots} \qquad \mathtt{dom}(\sigma) = \mathcal{Y} \qquad \Gamma \vdash u : S\sigma}{\Gamma \vdash\ v\langle u \rangle : T\sigma}\ (\textsc{Exp Call}) \qquad \frac{\Gamma \vdash v : \bot \qquad \Gamma \vdash u : \bot}{\Gamma \vdash\ v\langle u \rangle : \bot}\ (\textsc{Exp Call}\ \bot)$$

(EXP DEN)
$$\frac{\Gamma \vdash u : \forall \mathcal{X}[\overrightarrow{\ell_i^{\delta_i} : \forall \mathcal{Y}_i (S_i) T_i}]^{G\cdots} \qquad \textsc{Invariance}(\overrightarrow{i}, \overrightarrow{i}, \overrightarrow{i})}{\Gamma \vdash u \mapsto \overrightarrow{v_i}[\overrightarrow{\ell_i \Mapsto (y_i) b_i}] : \bot}$$

(EXP DEN $\bot$)
$$\frac{\Gamma \vdash u : \bot \qquad \forall i.\, \Gamma \vdash v_i : \bot \qquad \forall i.\, \Gamma, \overrightarrow{z_i : \bot}, y_i : \bot \vdash b_i \not\downarrow_n^{\overrightarrow{z_i}} : \bot}{\Gamma \vdash u \mapsto \overrightarrow{v_i}[\overrightarrow{\ell_i \Mapsto (y_i) b_i}] : \bot}$$

(EXP UPD)
$$\frac{\Gamma \vdash u : \forall \mathcal{X}[\overrightarrow{\ell_i^{\delta_i} : \forall \mathcal{Y}_i \langle S_i \rangle T_i}]^{G\cdots} \qquad \textsc{Invariance}(\overrightarrow{i}, \overrightarrow{j}, \overrightarrow{k})}{\Gamma \vdash u \leftarrow \overrightarrow{v_j}[\overrightarrow{\ell_k \Mapsto (y_k) b_k}] : \bot}$$

(EXP UPD $\bot$)
$$\frac{\Gamma \vdash u : \bot \qquad \forall j.\, \Gamma \vdash v_j : \bot \qquad \forall k.\, \Gamma, \overrightarrow{z_i : \bot}, y_k : \bot \vdash b_k \not\downarrow_n^{\overrightarrow{z_i}} : \bot}{\Gamma \vdash u \leftarrow \overrightarrow{v_j}[\overrightarrow{\ell_k \Mapsto (y_k) b_k}] : \bot}$$

In the rules for $\Gamma \vdash a : T$, we implicitly require $\Gamma \vdash T$ in the antecedent. (EXP $\exists$ INTRO) and (EXP $\exists$ ELIM) are standard rules for introduction and elimination of existential type quantifiers. By (EXP CONS INTRO), an expression has type $T \mid \mathcal{C}$ if it has type T, and the constraint $\mathcal{C}$ can be derived. Conversely, an assumption of the form $x : S \mid \mathcal{C}$ can be split into the assumptions $x : S$ and $\mathcal{C}$.

Dynamic checks imply some subtyping constraints, and we type the continuations of such checks under those constraints. By (EXP $\sqsupseteq$), if u is of type T, then the run-time check $L \sqsupseteq u$ introduces the subtyping constraint $L \leq T$; indeed, if T is level L', then u is some level at least as high as L', so that $L \sqsupseteq u$ implies $L \leq L'$. By (EXP $=$), if u is of type S and v is of type S', then the run-time check $u = v$ introduces the constraints $X \leq S$ and $X \leq S'$ for some fresh type variable X; indeed, $u = v$ implies that S and S' have a common subtype that is the type of both u and v.

By (EXP CALL), if v is an indirection with type scheme $\forall \mathcal{Y} \langle S \rangle T$, then for any type substitution σ for $\mathcal{Y}$, v can be called with a value of type $S\sigma$, and result of the call is of type $T\sigma$. Further, by (EXP CALL $\bot$), a public indirection can be called with a public value, and the result of the call is public. The rules (EXP DEN) and (EXP UPD) respectively require the conditions $\textsc{Invariance}(\overrightarrow{i}, \overrightarrow{i}, \overrightarrow{i})$ and $\textsc{Invariance}(\overrightarrow{i}, \overrightarrow{j}, \overrightarrow{k})$, as discussed above. On the other hand, if the object name is public, then by (EXP DEN

$\bot$) and (EXP UPD $\bot$), the method names linked to the object must be public, and the method expressions must take public values and return public results.

The remaining rules for well-typed expressions are straightforward.

Example 5.2.1. Recall the example with authenticated encryption objects. Let the object name x and the indirections $\mathbf{k\grave{e}y}$ and $\mathbf{en\grave{c}}$ have the shown types, and let k be a value of type T. Then the following denomination is well-typed if Authority, $\|T\|$, and Reader are all higher than $\bot$. (Why do we require this condition?)

$$\begin{array}{l} x \mapsto \mathbf{key}, \mathbf{enc}[\,\mathtt{key} \Rrightarrow (\_)k,\ \mathtt{authencrypt} \Rrightarrow (y) \\ \qquad (\nu x' : [\mathtt{decrypt}^{+} : (T)Y]^{\mathrm{Authority}\bot})\ (\nu \mathbf{dec} : (\langle T \rangle Y)^{\mathrm{Authority}\bot}) \\ \qquad\quad x' \mapsto \mathbf{dec}[\,\mathtt{decrypt} \Rrightarrow (y')\ \mathsf{if}\ y' = \mathtt{key}(\bot)\ \mathsf{then}\ y\,]\ \Rsh\ \mathbf{d\grave{e}c}\,] \end{array}$$

A reader belonging to Reader can obtain the key k by calling $\mathbf{k\grave{e}y}$. A writer belonging to Writer can encrypt a term M of any type S by calling $\mathbf{en\grave{c}}$; further, it can make the encrypted term $\mathbf{d\grave{e}c}$ public. A reader can retrieve M by calling $\mathbf{d\grave{e}c}$ with k.

5.3 Properties of well-typed code

The main result for our type system is that well-typed code never leaks secrets beyond declared boundaries, even under arbitrary untrusted environments. The result relies on a standard but non-trivial preservation property: well-typed expressions preserve their types on execution.

Proposition 5.3.1 (Preservation)**.** *Let* $\Gamma \vdash a : T$*. If* $a \longrightarrow b$*, then* $\Gamma \vdash b : T$*.*

Additionally, the type system has two important properties. First, reaches are preserved by subtyping. Second, the type system can accommodate arbitrary expressions, as long as they do contain only public names. This property is important, since we cannot assume that attackers attempting to learn secrets would politely follow our typing discipline.

Proposition 5.3.2 (Typability)**.** *Let a be any expression without free labels. Suppose all declared types in a are* $\bot$*, and* $x : \bot \in \{\Gamma\}$ *for all free names x in a. Then* $\Gamma \vdash a : \bot$*.*

Finally, we present the main result. Let a be trusted code typed under environment Γ, and b be (perhaps partially) untrusted code typed under the same environment Γ. In general, b may be some trusted code composed with arbitrary untrusted code, and the trusted code in b may even share secret names with a. Then no declared secret x can ever be learnt by executing b in composition with a.

Theorem 5.3.3 (Secrecy)**.** *Let* $\Gamma \vdash a : S$ *and* $\Gamma \vdash b : \bot$. *If* $\Gamma, \bot \leq \|T\| \vdash \bot \leq \top$, *then* $a \Rsh b \not\longrightarrow^\star (\nu x : T)\ \_ \Rsh x$.

The proof is based on a simple argument: if x can be learnt, then by Proposition 5.3.1, T must be a subtype of $\bot$; so by (CONS HYP) the reach of T must be $\bot$ (contradiction). A weaker version of the theorem that deals with top-level secrets also holds: for all variables x such that $x : T \in \{\Gamma\}$ and $\Gamma, \bot \leq \|T\| \vdash \bot \leq \top$, it must be the case that $a \Rsh b \not\longrightarrow^\star \_ \Rsh x$.

A significant application of Theorem 5.3.3 appears in the appendix, where we describe a type-directed encoding of the secrecy type system of Chapter 4 in this setting. More precisely, we show that any code that is well-typed under that system can be compiled to well-typed code in **conc** $\widetilde{\imath}$. The soundness of that system then follows from Theorem 5.3.3, and some auxiliary lemmas that establish the adequacy of the compilation (*i.e.*, the preservation of behaviors by the compiler).

Chapter 6

Access control and types for integrity

Commercial operating systems are seldom designed to prevent information-flow attacks. Unfortunately, such attacks are the source of many serious security problems in these systems [Sabelfeld and Myers, 2003]. Microsoft's Windows Vista operating system implements an integrity model that can potentially prevent some of those attacks. In some ways, this model resembles other, classical models of multi-level integrity [Biba, 1977]—every process and object[1] is tagged with an integrity label, the labels are ordered by levels of trust, and access control is enforced across trust boundaries. In other ways, it is radically different. While Windows Vista's access control prevents low-integrity processes from writing to high-integrity objects, it does not prevent high-integrity processes from reading low-integrity objects. Further, Windows Vista's integrity labels are dynamic—labels of processes and objects can change at run time. This model allows processes at different trust levels to communicate, and allows dynamic access control. At the same time, it admits various information-flow attacks. Fortunately, it turns out that such attacks require the participation of trusted processes, and can be eliminated by code analysis.

In this chapter, we provide a formalization of Windows Vista's integrity model. In particular, we specify an information-flow property called *data-flow integrity* (DFI), and present a static type system that can enforce DFI on Windows Vista.[2] Roughly, DFI

[1] In this context, an object may be a file, a channel, a memory location, or indeed any reference to data or executable code.

[2] [Castro et al., 2006] specifies and enforces a different property by the same name; see Chapter 1.

prevents any flow of data from the environment to objects whose contents are trusted. Our type system relies on Windows Vista's run-time access checks for soundness. The key idea in the type system is to maintain a static lower-bound label S for each object. While the dynamic label of an object can change at run time, the type system ensures that it never goes below S, and the object never contains a value that flows from a label lower than S. The label S is declared by the programmer. Typechecking requires no other annotations, and can be mechanized by an efficient algorithm.

By design, DFI does not prevent implicit flows [Denning and Denning, 1977]. Thus DFI is weaker than noninterference [Goguen and Meseguer, 1982]. Unfortunately, it is difficult to enforce noninterference on a commercial operating system such as Windows Vista. Implicit flows abound in such systems. Such flows arise out of frequent, necessary interactions between trusted code and the environment. They also arise out of covert control channels which, given the scope of such systems, are impossible to model sufficiently. Instead, DFI focuses on explicit flows [Denning and Denning, 1977]. This focus buys a reasonable compromise—DFI prevents a definite class of attacks, and can be enforced efficiently on Windows Vista. Several successful tools for malware detection follow this approach [Castro et al., 2006; Yin et al., 2007; Suh et al., 2004; Vogt et al., 2007; Clause et al., 2007; Wall et al., 1996], and similar ideas guide the design of some recent operating systems [Efstathopoulos et al., 2005; Zeldovich et al., 2006].

Our definition of DFI is dual to standard definitions of secrecy based on explicit flows—while secrecy prevents sensitive values from flowing to the environment, DFI prevents the flow of values from the environment to sensitive objects. Since there is a rich literature on type-based and logic-based analysis for such definitions of secrecy [Cardelli et al., 2005; Abadi and Blanchet, 2005; Tse and Zdancewic, 2004; Chaudhuri, 2006], it makes sense to adapt this analysis for DFI. Such an adaptation works, but requires some care. Unlike secrecy, DFI cannot be enforced in practice without run-time checks. In particular, access checks play a crucial role by restricting untrusted processes that may run in the environment. Further, while secrecy prevents any flow of high-security information to the environment, DFI allows certain flows of low-security information from the environment. We need to introduce new technical devices for this purpose, including a technique based on *explicit substitution* [Abadi et al., 1990] to

track precise sources of values. This device is required not only to specify DFI precisely but also to prove that our type system enforces DFI.

We design a simple higher-order process calculus that models Windows Vista's security environment [Howard and LeBlanc, 2007; Conover, 2007; Russinovich, 2007]. (The design of this language is discussed in detail in Chapter 8.) In this language, processes can fork new processes, create new objects, change the labels of processes and objects, and read, write, and execute objects in exactly the same ways as Windows Vista allows. Our type system exploits Windows Vista's run-time access checks to enforce DFI, and can recognize many correct programs. In particular, it encodes a more precise discipline than the one studied in Chapter 3. At the same time, our type system subsumes Windows Vista's execution controls, allowing them to be optimized away.

To sum up, we make the following main contributions in this chapter:

- We propose DFI as a practical multi-level integrity property to enforce in the setting of Windows Vista, and formalize DFI using a semantic technique based on explicit substitution.
- We present a type system that can enforce DFI on Windows Vista. Typechecking is efficient, and guarantees DFI regardless of what untrusted code runs in the environment.
- We show that while most of Windows Vista's run-time access checks are required to enforce DFI, Windows Vista's execution controls are not necessary and can be optimized away.

The rest of this chapter is organized as follows. In Section 6.1, we introduce Windows Vista's security environment, and show how DFI may be violated in that environment. In Section 6.2, we design a calculus that models Windows Vista's security environment, equip the calculus with a semantics based on explicit substitution, and formalize DFI in the calculus. In Section 6.3, we present a system of integrity types and effects for this calculus. Finally, in Section 6.4, we prove soundness and other properties of typing. Supplementary material, including proof details and an efficient typechecking algorithm, appear in [Chaudhuri et al., 2007] available online at `http://arxiv.org/abs/0803.3230`.

6.1 Windows Vista's integrity model

In this section, we provide a brief overview of Windows Vista's integrity model.[3] In particular, we introduce Windows Vista's security environment [Howard and LeBlanc, 2007; Conover, 2007; Russinovich, 2007], and show how DFI may be violated in that environment. We observe that such violations require the participation of trusted processes. Intuitively, the responsibility of security lies with trusted users. Our type system provides a way for such users to manage this responsibility automatically.

6.1.1 Windows Vista's security environment

In Windows Vista, every process and object is tagged with a dynamic integrity label. We indicate such labels in brackets $(\_)$ below. Labels are related by a total order $\sqsubseteq$, meaning "at most as trusted as". Let a range over processes, ω over objects, and P, O over labels. Processes can fork new processes, create new objects, change the labels of processes and objects, and read, write, and execute objects. In particular, a process with label P can:

(i) fork a new process $a(\mathsf{P})$;

(ii) create a new object $\omega(\mathsf{P})$;

(iii) lower its own label;

(iv) change the label of an object $\omega(\mathsf{O})$ to O' iff $\mathsf{O} \sqcup \mathsf{O}' \sqsubseteq \mathsf{P}$;

(v) read an object $\omega(\mathsf{O})$;

(vi) write an object $\omega(\mathsf{O})$ iff $\mathsf{O} \sqsubseteq \mathsf{P}$;

(vii) execute an object $\omega(\mathsf{O})$ by lowering its own label to $\mathsf{P} \sqcap \mathsf{O}$.

Rules (i) and (ii) are straightforward. Rule (iii) is guided by the principle of least privilege [Saltzer and Schroeder, 1975; Lampson, 1974], and is used in Windows Vista to implement a feature called *user access control* (UAC) [Russinovich, 2007; Windows Vista

[3] This overview elaborates on the one in Chapter 3; as noted there, Windows Vista further implements a discretionary access control model, which we ignore in this chapter.

Tech Center]. This feature lets users execute commands with lower privileges when appropriate. For example, when a system administrator opens a new shell (typically with label High), a new process is forked with label Medium; the shell is then run by the new process. When an Internet browser is opened, it is always run by a new process whose label is lowered to Low; thus any code that gets run by the browser gets the label Low—by Rule (i)—and any file that is downloaded by the browser gets the label Low—by Rule (ii).

Rules (iv) and (v) facilitate dynamic access control and communication across trust boundaries, but can be dangerous if not used carefully. (We show some attacks to illustrate this point below.) In particular, Rule (iv) allows trusted processes to protect unprotected objects by raising their labels. (Users are required to confirm such protections via the user interface.) Moreover, Rule (v) allows processes to read objects at lower trust levels.

Rule (vi) protects objects from being written by processes at lower trust levels. Thus, for example, untrusted code forked by a browser cannot touch local user files. User code cannot modify registry keys protected by a system administrator. Rule (vii) is part of UAC; it prevents users from accidentally launching less trusted executables with higher privileges. For example, a virus downloaded from the Internet cannot run in a trusted user shell. Neither can system code dynamically link user libraries.

6.1.2 Some attacks

We now show some (unsurprising) attacks that remain possible in this environment. Basically, these attacks exploit Rules (iv) and (v) to bypass Rules (vi) and (vii).

(**Write and copy**) By Rule (vi), $a(\mathsf{P})$ cannot modify $\omega(\mathsf{O})$ if $\mathsf{P} \sqsubset \mathsf{O}$. However, $a(\mathsf{P})$ can modify some object $\omega'(\mathsf{P})$, and then some process $b(\mathsf{O})$ can copy $\omega'(\mathsf{P})$'s content to $\omega(\mathsf{O})$. Thus, Rule (iv) can be exploited to bypass Rule (vi).

(**Copy and execute**) By Rule (vii), $a(\mathsf{P})$ cannot execute $\omega(\mathsf{O})$ at P if $\mathsf{O} \sqsubset \mathsf{P}$. However, $a(\mathsf{P})$ can copy $\omega(\mathsf{O})$'s content to some object $\omega'(\mathsf{P})$ and then execute $\omega'(\mathsf{P})$. Thus, Rule (iv) can be exploited to bypass Rule (vii).

(**Unprotect, write, and protect**) By Rule (vi), $a(\mathsf{P})$ cannot modify $\omega(\mathsf{O})$ if $\mathsf{P} \sqsubset \mathsf{O}$. However, some process $b(\mathsf{O})$ can unprotect $\omega(\mathsf{O})$ to $\omega(\mathsf{P})$, then $a(\mathsf{P})$ can modify $\omega(\mathsf{P})$, and then $b(\mathsf{O})$ can protect $\omega(\mathsf{P})$ back to $\omega(\mathsf{O})$. Thus, Rule (v) can be exploited to bypass Rule (vi).

(**Copy, protect, and execute**) By Rule (vii), $a(\mathsf{P})$ cannot execute $\omega(\mathsf{O})$ at P if $\mathsf{O} \sqsubset \mathsf{P}$. However, some process $b(\mathsf{O})$ can copy $\omega(\mathsf{O})$'s content to an object $\omega'(\mathsf{O})$, and then $a(\mathsf{P})$ can protect $\omega'(\mathsf{O})$ to $\omega'(\mathsf{P})$ and execute $\omega'(\mathsf{P})$. Thus, Rules (iv) and (v) can be exploited to bypass Rule (vii).

All of these attacks can violate DFI; however, we observe that access control forces the participation of a trusted process (one with the higher label) in any such attack.

- In (**Write and copy**) or (**Unprotect, write, and protect**), suppose that the contents of $\omega(\mathsf{O})$ are trusted, and P is the label of untrusted code, with $\mathsf{P} \sqsubset \mathsf{O}$. Then data can flow from $a(\mathsf{P})$ to $\omega(\mathsf{O})$, violating DFI, as above. Fortunately, some process $b(\mathsf{O})$ can be blamed here.

- In (**Copy and execute**) or (**Copy, protect, and execute**), suppose that the contents of some object $\omega''(\mathsf{P})$ are trusted, and O is the label of untrusted code, with $\mathsf{O} \sqsubset \mathsf{P}$. Then data can flow from some process $b(\mathsf{O})$ to $\omega''(\mathsf{P})$, violating DFI, as follows: $b(\mathsf{O})$ packs code to modify $\omega''(\mathsf{P})$ and writes the code to $\omega(\mathsf{O})$, and $a(\mathsf{P})$ unpacks and executes that code at P, as above. Fortunately, $a(\mathsf{P})$ can be blamed here.

Our type system can eliminate such attacks by restricting trusted processes (Section 6.3). (The type system does not restrict untrusted code running in the environment.) Conceptually, this guarantee can be cast as Wadler and Findler's "*well-typed programs can't be blamed*" [Wadler and Findler, 2007]. We rely on the fact that a trusted process can be blamed for any violation of DFI; it follows that if all trusted processes are well-typed, there cannot be any violation of DFI.

6.2 A calculus for analyzing DFI on Windows Vista

To formalize our approach, we design a simple higher-order process calculus that models Windows Vista's security environment. We introduce the syntax and informal

semantics, and present some examples of programs and attacks in the language. We then present a formal semantics, guided by a precise characterization of explicit flows.

6.2.1 Syntax and informal semantics

Several simplifications appear in the syntax of the language. We describe processes by their code. We use variables as object names, and let objects contain packed code or names of other objects. We enforce a mild syntactic restriction on nested packing (see below), which makes typechecking significantly more efficient [Chaudhuri et al., 2007]. Finally, we elide conditionals—for our purposes, the code

$$\mathsf{if}\ \texttt{condition}\ \mathsf{then}\ a\ \mathsf{else}\ b$$

can be conservatively analyzed by composing a and b in parallel. (DFI is a *safety property* in the sense of [Alpern and Schneider, 1985], and the safety of the latter code implies that of the former. We discuss this point in more detail in Section 6.2.3.)

Values include variables, unit, and packed expressions.[4] Expressions include those for forking new processes, creating new objects, changing the labels of processes and objects, and reading, writing, and executing objects. They also include standard expressions for evaluation and returning results (see Gordon and Hankin's concurrent object calculus [Gordon and Hankin, 1998]).

$a, b ::=$	process
$\quad a \Rsh b$	$\quad$ fork
$\quad t$	$\quad$ action
$\quad \mathsf{let}\ x = a\ \mathsf{in}\ b$	$\quad$ evaluation
$\quad u$	$\quad$ value
$u, v ::=$	value
$\quad r$	$\quad$ result
$\quad \mathsf{pack}(f)$	$\quad$ packed expression

[4] Packed expressions may be viewed as "thunks" of executable code, that must be unpacked to allow further evaluation. In particular, packed expressions can be written to objects, and unpacked by executing those objects; such objects model "binaries" in the language.

$f, g ::=$	expression
$f \Rsh g$	fork
t	action
let $x = f$ in g	evaluation
r	result
$t ::=$	action
$\mathsf{new}(x \# \mathsf{S})$	create object
$[\mathsf{P}]\ a$	change process label
$\langle \mathsf{O} \rangle\ \omega$	change object label
$!\omega$	read object
$\omega := x$	write object
$\mathsf{exec}\ \omega$	execute object
$r ::=$	result
$x, y, z, \ldots, \omega$	variable
unit	unit

Syntactically, we distinguish between processes and expressions: while every expression is a process, not every process is an expression. For example, $\mathsf{pack}(f)$ is not an expression, while $[\mathsf{P}]\ \mathsf{pack}(f)$ is. Only expressions can be packed. In particular, a process cannot be of the form $\mathsf{pack}(\mathsf{pack}(\ldots))$. This distinction does not reduce expressivity, since such a process can be expressed in the language as let $x = \mathsf{pack}(\ldots)$ in $\mathsf{pack}(x)$. The benefits of this distinction become clear in Section 6.4, where we discuss an algorithm for typechecking. However, for the bulk of the chapter, the reader may overlook this distinction—neither the semantics nor the type system depend on it.

Processes have the following informal meanings.

- $a \Rsh b$ forks a new process a with the current process label and continues as b (see Rule (i)).
- $\mathsf{new}(x \# \mathsf{S})$ creates a new object ω with the current process label, initializes ω with x, and returns ω (see Rule (ii)); the annotation S is used by the type system (Section 6.3) and has no run-time significance.

- $[\mathsf{P}]\ a$ changes the current process label to P and continues as a; it blocks if the current process label is lower than P (see Rule (iii)).
- $\langle \mathsf{O} \rangle\ \omega$ changes ω's label to O and returns unit; it blocks if ω is not bound to an object at run time, or the current process label is lower than ω's label or O (see Rule (iv)).
- $!\omega$ returns the value stored in ω; it blocks if ω is not bound to an object at run time (see Rule (v)).
- $\omega := x$ writes the value x to ω and returns unit; it blocks if ω is not bound to an object at run time, or if the current process label is lower than ω's label (see Rule (vi)).
- $\mathsf{exec}\ \omega$ unpacks the value stored in ω to a process f, lowers the current process label with ω's label, and executes f; it blocks if ω is not bound to an object at run time or if the value stored in ω is not a packed expression (see Rule (vii)).
- $\mathsf{let}\ x = a\ \mathsf{in}\ b$ executes a, binds the value returned by a to x, and continues as b with x bound.
- u returns itself.

6.2.2 Programming examples

We now consider some programming examples in the language. We assume that Low, Medium, High, and $\top$ are labels, ordered in the obvious way. We assume that the top-level process always runs with $\top$, which is the most trusted label.

Example 6.2.1. Suppose that a Medium user opens an Internet browser `ie.exe` with Low privileges (recall UAC), and clicks on a `url` that contains `virus.exe`; the virus contains code to overwrite the command shell executable `cmd.exe`, which has label $\top$.

$$
\begin{aligned}
p_1 \triangleq\ & \mathsf{let}\ \texttt{cmd.exe} = \mathsf{new}(\ldots\ \#\ \top)\ \mathsf{in} \\
& \mathsf{let}\ \texttt{url} = [\mathsf{Low}]\ \mathsf{new}(\ldots\ \#\ \mathsf{Low})\ \mathsf{in} \\
& \mathsf{let}\ \texttt{binIE} = \mathsf{pack}(\mathsf{let}\ x = !\texttt{url}\ \mathsf{in}\ \mathsf{exec}\ x)\ \mathsf{in}
\end{aligned}
$$

$$
\begin{array}{l}
\mathsf{let}\ \mathtt{ie.exe} = \mathsf{new}(\mathtt{binIE}\ \#\ \top)\ \mathsf{in} \\
[\mathsf{Medium}]\ (\ldots\ \Rsh\ [\mathsf{Low}]\ \mathsf{exec}\ \mathtt{ie.exe})\ \Rsh \\
[\mathsf{Low}]\ (\mathsf{let}\ \mathtt{binVirus} = \mathsf{pack}(\mathtt{cmd.exe} := \ldots)\ \mathsf{in} \\
\qquad \mathsf{let}\ \mathtt{virus.exe} = \mathsf{new}(\mathtt{binVirus}\ \#\ \mathsf{Low})\ \mathsf{in} \\
\qquad \mathtt{url} := \mathtt{virus.exe}\ \Rsh \\
\qquad \ldots)
\end{array}
$$

This code may eventually reduce to

$$
\begin{array}{rl}
q_1 \triangleq & [\mathsf{Medium}]\ (\ldots\ \Rsh\ [\mathsf{Low}]\ \mathtt{cmd.exe} := \ldots)\ \Rsh \\
& [\mathsf{Low}]\ (\ldots)
\end{array}
$$

However, at this point the write to `cmd.exe` blocks due to access control. (Recall that a process with label Low cannot write to an object with label $\top$.)

Example 6.2.2. Next, consider the following attack, based on the (**Copy, protect, and execute**) attack in Section 6.1.2. A Medium user downloads a virus from the Internet that contains code to erase the user's home directory (`home`), and saves it by default in `setup.exe`. A High administrator protects and executes `setup.exe`.

$$
\begin{array}{rl}
p_2 \triangleq & \mathsf{let}\ \mathtt{url} = [\mathsf{Low}]\ \mathsf{new}(\ldots\ \#\ \mathsf{Low})\ \mathsf{in} \\
& \mathsf{let}\ \mathtt{setup.exe} = [\mathsf{Low}]\ \mathsf{new}(\ldots\ \#\ \mathsf{Low})\ \mathsf{in} \\
& \mathsf{let}\ \mathtt{binIE} = \mathsf{pack}(\mathsf{let}\ z = \ !\mathtt{url}\ \mathsf{in} \\
& \qquad\qquad\qquad \mathsf{let}\ x = \ !z\ \mathsf{in}\ \mathtt{setup.exe} := \ x)\ \mathsf{in} \\
& \mathsf{let}\ \mathtt{ie.exe} = \mathsf{new}(\mathtt{binIE}\ \#\ \top)\ \mathsf{in} \\
& \mathsf{let}\ \mathtt{home} = [\mathsf{Medium}]\ \mathsf{new}(\ldots\ \#\ \mathsf{Medium})\ \mathsf{in} \\
& \mathsf{let}\ \mathtt{empty} = \mathsf{unit}\ \mathsf{in} \\
& [\mathsf{High}]\ (\ldots \Rsh \\
& \qquad \mathsf{let}\ \_ = \langle \mathsf{High} \rangle\ \mathtt{setup.exe}\ \mathsf{in} \\
& \qquad \mathsf{exec}\ \mathtt{setup.exe})\ \Rsh \\
& [\mathsf{Medium}]\ (\ldots\ \Rsh\ [\mathsf{Low}]\ \mathsf{exec}\ \mathtt{ie.exe})\ \Rsh
\end{array}
$$

$$
\begin{aligned}
&[\mathsf{Low}]\ (\text{let } \texttt{binVirus} = \mathsf{pack}(\texttt{home} := \texttt{empty}) \text{ in} \\
&\qquad \text{let } \texttt{virus.exe} = \mathsf{new}(\texttt{binVirus} \,\#\, \mathsf{Low}) \text{ in} \\
&\qquad \texttt{url} := \texttt{virus.exe} \ \Rsh \\
&\qquad \ldots)
\end{aligned}
$$

This code may eventually reduce to

$$
\begin{aligned}
q_2 \triangleq\ &[\mathsf{High}]\ (\ldots \ \Rsh\ \texttt{home} := \texttt{empty}) \ \Rsh \\
&[\mathsf{Medium}]\ (\ldots) \ \Rsh \\
&[\mathsf{Low}]\ (\ldots)
\end{aligned}
$$

The user's home directory may be erased at this point. (Recall that access control does not prevent a process with label High from writing to an object with label Medium.)

Here the administrator is required to confirm the protection of `setup.exe` via the user interface. Our type system can detect that this protection is dangerous, and warn the administrator.

6.2.3 An overview of DFI

Informally, DFI requires that objects whose contents are trusted at some label S never contain values that flow from labels lower than S. In Example 6.2.1, we trust the contents of `cmd.exe` at label $\top$, as declared by the static annotation $\top$. DFI is *not* violated in this example, since access control prevents the flow of data from Low to `cmd.exe`. On the other hand, in Example 6.2.2, we trust the contents of `home` at label Medium. DFI *is* violated in this example, since the value `empty` flows from Low to `home`.

By design, DFI is a safety property [Alpern and Schneider, 1985]—it can be defined as a set of behaviors such that for any behavior that is not in that set, there is some finite prefix of that behavior that is not in that set. Thus, DFI considers only *explicit* flows of data. Denning and Denning characterize explicit flows [Denning and Denning, 1977] roughly as follows: a flow of x is explicit if and only if the flow depends abstractly on x (that is, it depends on the existence of x, but not on the value x). Thus, for example, the violation of DFI in Example 6.2.2 does not depend on the value `empty`—*any* other value

causes the same violation. Conversely, `empty` is not dangerous in itself. Consider the reduced process q_2 in Example 6.2.2. Without any knowledge of execution history, we cannot conclude that DFI is violated in q_2. Indeed, it is perfectly legitimate for a High-process to execute the code `home` := `empty` intentionally, say as part of administration. However, in Example 6.2.2, we know that this code is executed by unpacking some code designed by a Low-process. The violation of DFI is *due to this history*.

It follows that in order to detect violations of DFI, we must distinguish between various instances of a value, and track the sources of those instances during execution. We maintain this execution history in the operational semantics (Section 6.2.4), by a technique based on explicit substitution [Abadi et al., 1990].

Before we move on, let us clarify the role of control constructs, such as conditionals, in DFI. In general, conditionals can cause implicit flows [Denning and Denning, 1977]; a flow of x can depend on the value x if x appears in the condition of some code that causes that flow. For example, the code

$$\mathsf{if}\ x = \texttt{zero}\ \mathsf{then}\ \omega := \texttt{zero}\ \mathsf{else}\ \omega := \texttt{one}$$

causes an implicit flow of x to ω that depends on the value x. DFI abstracts away this dependency by interpreting the code if `condition` then a else b as the parallel composition of a and b. Recall that DFI is a safety property. Following [Lamport, 1977], the safety of this parallel composition can be expressed by the logical formula $F \triangleq F_a \wedge F_b$, where F_a is the formula that expresses the safety of a, and F_b is the formula that expresses the safety of b. Likewise, the safety of if `condition` then a else b can be expressed by the formula $F' \triangleq (\texttt{condition} \Rightarrow F_a) \wedge (\neg\texttt{condition} \Rightarrow F_b)$. Clearly, we have $F \Rightarrow F'$, so that the code if `condition` then a else b is a refinement of the parallel composition of a and b. It is well-known that any safety property is preserved under refinement [Lamport, 1977], so our abstraction is correct.

But implicit flows are of serious concern in many applications; one may wonder whether focusing on explicit flows is at all desirable. Indeed, consider the code above; the implicit flow from x to ω violates noninterference, if x is an untrusted value and the contents of ω are meant to be trusted. In contrast, DFI is *not* violated in the code

$$\omega := \texttt{zero} \ \Rsh\ \omega := \texttt{one}$$

if `zero` and `one` are trusted values. Clearly, DFI ignores the implicit flow from x to ω. But this may be fine—DFI can still guarantee that "the contents of ω are trusted values (either `zero` or `one`)". This is certainly a non-trivial guarantee; for example, the code

$$\omega := x$$

does not maintain this invariant, since x may be an arbitrary value—and as expected, DFI is violated in this code.

6.2.4 An operational semantics that tracks explicit flows

We now present a chemical-style operational semantics for the language, that tracks explicit flows.[5] We begin by extending the syntax with some auxiliary forms.

$a, b ::=$	process
$\quad \cdots$	$\quad$ source process
$\quad \omega \overset{\mathsf{O}}{\mapsto} x$	$\quad$ store
$\quad (\nu x / \mu @ \mathsf{P})\, a$	$\quad$ explicit substitution
$\mu ::=$	substituted value
$\quad u$	$\quad$ value
$\quad \mathsf{new}(x \,\#\, \mathsf{S})$	$\quad$ object initialization

The process $\omega \overset{\mathsf{O}}{\mapsto} x$ asserts that the object ω contains x and is protected with label O. A key feature of the semantics is that objects store values "by instance"—only variables may appear in stores. We use explicit substitution to track and distinguish between the sources of various instances of a substituted value. Specifically, the process $(\nu x / \mu @ \mathsf{P})\, a$ creates a fresh variable x, records that x is bound to μ by a process with label P, and continues as a with x bound. Here x is an *instance* of μ and P is the *source* of x. If μ is a value, then this process is behaviorally equivalent to a with x substituted by μ. For example, in Example 6.2.2 the source of the instance of `empty` in `binVirus` is `Low`; this fact is described by rewriting the process q_2 as

$$(\nu x / \texttt{empty@Low})\, [\mathsf{High}]\, (\ldots \Rsh\ \texttt{home} := x) \Rsh \ldots$$

[5] This presentation is particularly convenient for defining and proving DFI; a concrete implementation of the language may rely on a lighter semantics that does not track explicit flows.

DFI prevents this particular instance (x) of `empty` from being written to `home`; but it allows other instances whose sources are at least as trusted as Medium. The rewriting follows a structural equivalence rule (STRUCT BIND), explained later in the section.

While explicit substitution has been previously used in language implementations, we seem to be the first to adapt this device to track data flow in a concurrent language. In particular, we use explicit substitution both to specify DFI (in Definitions 6.2.3 and 6.2.4) and to verify it statically (in proofs of Theorems 6.4.4 and 6.4.7). We defer a more detailed discussion on this technique to Section **??**.

We call sets of the form $\{x_1/\mu_1@\mathsf{P}_1, \ldots, x_k/\mu_k@\mathsf{P}_k\}$ *substitution environments*.

Definition 6.2.3 (Explicit flows)**.** *A variable x flows from a label P or lower in a substitution environment σ, written $x \overset{\sigma}{\blacktriangledown} \mathsf{P}$, if $x/\mu@\mathsf{P}' \in \sigma$ for some μ and P' such that either $\mathsf{P}' \sqsubseteq \mathsf{P}$, or μ is a variable and (inductively) $\mu \overset{\sigma}{\blacktriangledown} \mathsf{P}$.*

In other words, x flows from a label P or lower if x is an instance of a value substituted at P or lower. In Definition 6.2.4 below, we formalize DFI as a property of objects, as follows: *an object is protected from label L if it never contains instances that flow from L or lower.* We define $\sigma(x)$ to be the set of values in σ that x is an instance of: $x \in \sigma(x)$, and if (inductively) $y \in \sigma(x)$ and $y/u@\_ \in \sigma$ for some y and u, then $u \in \sigma(x)$. The operational semantics ensures that substitution environments accurately associate instances of values with their run-time sources.

We now present rules for local reduction, structural equivalence, and global reduction. Reductions are of the form $a \xrightarrow{\mathsf{P};\sigma} b$, meaning that "process a may reduce to process b with label P in substitution environment σ". Structural equivalences are of the form $a \equiv b$, meaning that "process a may be rewritten as process b". The notions of free and bound variables (`fv` and `bv`) are standard. We write $x \overset{\sigma}{=} y$ if $\sigma(x) \cap \sigma(y) \neq \varnothing$, that is, there is a value that both x and y are instances of.

Local reduction $a \xrightarrow{\mathsf{P};\sigma} b$

(REDUCT EVALUATE)
$\mathsf{let}\ x = u\ \mathsf{in}\ a \xrightarrow{\mathsf{P};\sigma} (\nu x/u@\mathsf{P})\ a$

(REDUCT NEW)
$\mathsf{new}(x\ \#\ \mathsf{S}) \xrightarrow{\mathsf{P};\sigma} (\nu\omega/\mathsf{new}(x\ \#\ \mathsf{S})@\mathsf{P})\ (\omega \overset{\mathsf{P}}{\mapsto} x \overset{\rightarrow}{\cdot} \omega)$

(REDUCT READ)

$$\frac{\omega \stackrel{\sigma}{=} \omega'}{\omega \overset{\mathsf{O}}{\mapsto} x \Rsh\ !\omega' \xrightarrow{\mathsf{P};\sigma} \omega \overset{\mathsf{O}}{\mapsto} x \Rsh\ x}$$

(REDUCT WRITE)

$$\frac{\omega \stackrel{\sigma}{=} \omega' \qquad \mathsf{O} \sqsubseteq \mathsf{P}}{\omega \overset{\mathsf{O}}{\mapsto} \_ \Rsh\ \omega' := x \xrightarrow{\mathsf{P};\sigma} \omega \overset{\mathsf{O}}{\mapsto} x \Rsh\ \mathsf{unit}}$$

(REDUCT EXECUTE)

$$\frac{\omega \stackrel{\sigma}{=} \omega' \qquad \mathsf{pack}(f) \in \sigma(x) \qquad \mathsf{P}' = \mathsf{P} \sqcap \mathsf{O}}{\omega \overset{\mathsf{O}}{\mapsto} x \Rsh\ \mathsf{exec}\ \omega' \xrightarrow{\mathsf{P};\sigma} \omega \overset{\mathsf{O}}{\mapsto} x \Rsh\ [\mathsf{P}']\ f}$$

(REDUCT UN/PROTECT)

$$\frac{\omega \stackrel{\sigma}{=} \omega' \qquad \mathsf{O} \sqcup \mathsf{O}' \sqsubseteq \mathsf{P}}{\omega \overset{\mathsf{O}}{\mapsto} x \Rsh\ \langle \mathsf{O}' \rangle\ \omega' \xrightarrow{\mathsf{P};\sigma} \omega \overset{\mathsf{O}'}{\mapsto} x \Rsh\ \mathsf{unit}}$$

We first look at the local reduction rules. In (REDUCT EVALUATE), a substitution binds x to the intermediate value u and associates x with its run-time source P. (REDUCT NEW) creates a new store denoted by a fresh variable ω, initializes the store, and returns ω; a substitution binds ω to the initialization of the new object and associates ω with its run-time source P. The value x and the trust annotation S in the initialization are used by the type system (Section 6.3). The remaining local reduction rules describe reactions with a store, following the informal semantics.

Next, we define evaluation contexts [Felleisen, 1988]. An evaluation context is of the form $\mathcal{E}_{\mathsf{P};\sigma}$, and contains a hole of the form $\bullet_{\mathsf{P}';\sigma'}$; the context yields a process that executes with label P in substitution environment σ, if the hole is plugged by a process that executes with label P' in substitution environment σ'.

$\mathcal{E}_{\mathsf{P};\sigma} ::=$	evaluation context
$\bullet_{\mathsf{P};\sigma}$	hole
let $x = \mathcal{E}_{\mathsf{P};\sigma}$ in b	sequential evaluation
$\mathcal{E}_{\mathsf{P};\sigma} \Rsh\ b$	fork left
$a \Rsh\ \mathcal{E}_{\mathsf{P};\sigma}$	fork right
$(\nu x/\mu@\mathsf{P}')\ \mathcal{E}_{\mathsf{P};\{x/\mu@\mathsf{P}'\}\cup\sigma}$	explicit substitution
$[\mathsf{P}']\ \mathcal{E}_{\mathsf{P}';\sigma} \quad (\mathsf{P}' \sqsubseteq \mathsf{P})$	lowering of process label

Evaluation can proceed sequentially inside let processes, and in parallel under forks [Gordon and Hankin, 1998]; it can also proceed under explicit substitutions and lowering of process labels. In particular, note how evaluation contexts build substitution environments from explicit substitutions, and labels from changes of process labels. We denote by $\mathcal{E}_{\mathsf{P};\sigma}[\![a]\!]_{\mathsf{P}';\sigma'}$ the process obtained by plugging the hole $\bullet_{\mathsf{P}';\sigma'}$ in $\mathcal{E}_{\mathsf{P};\sigma}$ with a.

Structural equivalence $a \equiv b$

(STRUCT BIND)
$$\mathcal{E}_{\mathsf{P};\sigma}[\![a\{x/u\}]\!]_{\mathsf{P}',\sigma'} \equiv \mathcal{E}_{\mathsf{P};\sigma}[\![(\nu x/u@\mathsf{P}')\ a]\!]_{\mathsf{P}',\sigma'}$$

(STRUCT RESULT)
$$[\mathsf{P}]\ u \equiv (\nu x/u@\mathsf{P})\ x$$

(STRUCT SUBSTITUTION)
$$\frac{x \notin \mathtt{fv}(\mathcal{E}_{\mathsf{P},\sigma}) \cup \mathtt{bv}(\mathcal{E}_{\mathsf{P},\sigma}) \qquad \mathtt{fv}(\mu) \cap \mathtt{bv}(\mathcal{E}_{\mathsf{P},\sigma}) = \varnothing}{\mathcal{E}_{\mathsf{P};\sigma}[\![(\nu x/\mu@\mathsf{P}'')\ a]\!]_{\mathsf{P}',\sigma'} \equiv (\nu x/\mu@\mathsf{P}'')\ \mathcal{E}_{\mathsf{P},\{x/\mu@\mathsf{P}''\}\cup\sigma}[\![a]\!]_{\mathsf{P}',\sigma'}}$$

(STRUCT FORK)
$$\frac{\mathtt{fv}(a) \cap \mathtt{bv}(\mathcal{E}_{\mathsf{P},\sigma}) = \varnothing}{\mathcal{E}_{\mathsf{P};\sigma}[\![a \Rsh b]\!]_{\mathsf{P},\sigma'} \equiv a \Rsh \mathcal{E}_{\mathsf{P};\sigma}[\![b]\!]_{\mathsf{P},\sigma'}}$$

(STRUCT STORE)
$$[\mathsf{P}]\ (\omega \overset{\mathsf{O}}{\mapsto} x \Rsh a) \equiv \omega \overset{\mathsf{O}}{\mapsto} x \Rsh [\mathsf{P}]\ a$$

(STRUCT EQUIV)
$\equiv$ is an equivalence

Global reduction $a \xrightarrow{\mathsf{P};\sigma} b$

(REDUCT CONTEXT)
$$\frac{a \xrightarrow{\mathsf{P}';\sigma'} b}{\mathcal{E}_{\mathsf{P};\sigma}[\![a]\!]_{\mathsf{P}';\sigma'} \xrightarrow{\mathsf{P};\sigma} \mathcal{E}_{\mathsf{P};\sigma}[\![b]\!]_{\mathsf{P}';\sigma'}}$$

(REDUCT CONGRUENCE)
$$\frac{a \equiv a' \qquad a' \xrightarrow{\mathsf{P};\sigma} b' \qquad b' \equiv b}{a \xrightarrow{\mathsf{P};\sigma} b}$$

Next, we look at the structural equivalence and global reduction rules. In (STRUCT BIND), $a\{x/u\}$ is the process obtained from a by the usual capture-avoiding substitution of x by u. The rule states that explicit substitution may *invert* usual substitution to create instances as required. In particular, variables that appear in packed code can be associated with the label of the process that packs that code, even though those variables may be bound later—by (REDUCT EVALUATE)—when that code is eventually unpacked at some other label. For example, the instance of `empty` in `binVirus` may be correctly associated with Low (the label at which it is packed) instead of High (the label at which it is unpacked). In combination, the rules (REDUCT EVALUATE) and (STRUCT BIND) track precise sources of values by explicit substitution. By (STRUCT RESULT), the process label of a result can be captured in an explicit substitution and eliminated.

By (STRUCT SUBSTITUTION), substitutions can float across contexts under standard scoping restrictions. By (STRUCT FORK), forked processes can float across contexts [Gordon and Hankin, 1998], but must remain under the same process label. By (STRUCT STORE), stores can be shared across further contexts.

Reduction is extended with contexts and structural equivalence in the natural way.

Finally, we formalize DFI in our language, as promised.

Definition 6.2.4 (DFI)**.** *The object ω is protected from the label* L *by process a if there is no process b, substitution environment σ, and instance x such that $a \Rsh [\mathsf{L}]\ b \xrightarrow{\top,\varnothing}{}^{\star} \mathcal{E}_{\top,\varnothing}[\![\omega \mapsto x]\!]_{\top,\sigma}$ and $x \overset{\sigma}{\blacktriangledown} \mathsf{L}$.*

For example, `cmd.exe` is protected from Low by (the trusted part of) p_1; but `home` is not protected from Low by (the trusted part of) p_2, since $p_2 \xrightarrow{\top,\varnothing}{}^{\star} \mathcal{E}_{\top,\varnothing}[\![\texttt{home} \mapsto x]\!]_{\top,\sigma}$ and $x \overset{\sigma}{\blacktriangledown} \mathsf{Low}$ for $\sigma = \{x/\texttt{empty@Low}\}$ and a suitable $\mathcal{E}_{\top,\varnothing}$.

6.3 A type system to enforce DFI

We now show a type system to enforce DFI in the language. (The formal protection guarantee for well-typed code appears in Section 6.4.) We begin by introducing types and typing judgments. We then present typing rules and informally explain their properties. Finally, we consider some examples of typechecking. An efficient algorithm for typechecking is outlined in [Chaudhuri et al., 2007].

6.3.1 Types and effects

The core grammar of types is shown below. Here effects are simply labels; these labels belong to the same ordering $\sqsubseteq$ as in the operational semantics.

$\tau ::=$	type
$\quad \mathbf{Obj}(T)$	object
$\quad \nabla_{\mathsf{P}}.\ \mathbf{Bin}(T)$	packed code
$\quad \mathbf{Unit}$	unit
$T ::=$	static approximation
$\quad \tau^{\mathsf{E}}$	type and effect

- The type $\mathbf{Obj}(\tau^{\mathsf{S}})$ is given to an object that contains values of type τ. Such contents may not flow from labels lower than S; in other words, S indicates the trust on the contents of this object. DFI follows from the soundness of object types.
- The type $\nabla_{\mathsf{P}}.\ \mathbf{Bin}(\tau^{\mathsf{E}})$ is given to packed code that can be run with label P. Values returned by the code must be of type τ and may not flow from labels lower

than E. In fact, our type system admits a subtyping rule that allows such code to be run in a typesafe manner with any label that is at most P.

- The effect E is given to a value that does not flow from labels lower than E.

When creating an object, the programmer declares the trust on the contents of that object. Roughly, an object returned by $\mathsf{new}(\_ \,\#\, \mathsf{S})$ gets a type $\mathbf{Obj}(\_^{\mathsf{S}})$. For example, in Examples 6.2.1 and 6.2.2, we declare the trust $\top$ on the contents of `cmd.exe` and the trust Medium on the contents of `home`.

A typing environment Γ contains typing hypotheses of the form $x : T$. We assume that any variable has at most one typing hypothesis in Γ, and define $\mathtt{dom}(\Gamma)$ as the set of variables that have typing hypotheses in Γ. A typing judgment is of the form $\Gamma \vdash_{\mathsf{P}} a : T$, where P is the label of the process a, T is the type and effect of values returned by a, and $\mathtt{fv}(a) \subseteq \mathtt{dom}(\Gamma)$.

6.3.2 Core typing rules

We now present typing rules that enforce the core static discipline required for our protection guarantee. Some of these rules have side conditions that involve a predicate $*$ on labels. These conditions are ignored in our first reading of these rules. (The predicate $*$ is true everywhere in the absence of a special label $\bot$, introduced in Section 6.3.4.) One of the rules has a condition that involves a predicate $\Box$ on expressions; we introduce that predicate in the discussion below.

Core typing judgments $\Gamma \vdash_{\mathsf{P}} a : T$

(TYP UNIT)
$$\Gamma \vdash_{\mathsf{P}} \mathsf{unit} : \mathbf{Unit}^{\mathsf{P}}$$

(TYP VARIABLE)
$$\frac{x : \tau^{\mathsf{E}} \in \Gamma}{\Gamma \vdash_{\mathsf{P}} x : \tau^{\mathsf{E} \sqcap \mathsf{P}}}$$

(TYP PACK)
$$\frac{\Gamma \vdash_{\mathsf{P}'} f : T \qquad \Box f}{\Gamma \vdash_{\mathsf{P}} \mathsf{pack}(f) : \nabla_{\mathsf{P}'}.\, \mathbf{Bin}(T)^{\mathsf{P}}}$$

(TYP FORK)
$$\frac{\Gamma \vdash_{\mathsf{P}} a : \_ \qquad \Gamma \vdash_{\mathsf{P}} b : T}{\Gamma \vdash_{\mathsf{P}} a \mathbin{\vec{\cdot}} b : T}$$

(TYP LIMIT)
$$\frac{\Gamma \vdash_{\mathsf{P}'} a : T}{\Gamma \vdash_{\mathsf{P}} [\mathsf{P}']\, a : T}$$

(TYP EVALUATE)
$$\frac{\Gamma \vdash_{\mathsf{P}} a : T' \qquad \Gamma, x : T' \vdash_{\mathsf{P}} b : T}{\Gamma \vdash_{\mathsf{P}} \mathsf{let}\ x = a\ \mathsf{in}\ b : T}$$

(TYP SUBSTITUTE)
$$\frac{\Gamma \vdash_{\mathsf{P}'} \mu : T' \qquad \Gamma, x : T' \vdash_{\mathsf{P}} a : T}{\Gamma \vdash_{\mathsf{P}} (\nu x / \mu @ \mathsf{P}')\, a : T}$$

(TYP STORE)
$$\frac{\{\omega : \mathbf{Obj}(\tau^{\mathsf{S}})^{\text{-}}, x : \tau^{\mathsf{E}}\} \subseteq \Gamma \qquad \mathsf{S} \sqsubseteq \mathsf{O} \sqcap \mathsf{E}}{\Gamma \vdash_{\mathsf{P}} \omega \overset{\mathsf{O}}{\mapsto} x : \_^{\mathsf{P}}}$$

(TYP NEW)
$$\frac{\Gamma \vdash_{\mathsf{P}} x : \tau^{\mathsf{E}} \qquad \mathsf{S} \sqsubseteq \mathsf{E}}{\Gamma \vdash_{\mathsf{P}} \mathsf{new}(x \,\#\, \mathsf{S}) : \mathbf{Obj}(\tau^{\mathsf{S}})^{\mathsf{P}}}$$

(TYP UN/PROTECT)
$$\frac{\Gamma \vdash_{\mathsf{P}} \omega : \mathbf{Obj}(\_^{\mathsf{S}})^{\mathsf{E}} \qquad \mathsf{S} \sqsubseteq \mathsf{O}}{\Gamma \vdash_{\mathsf{P}} \langle \mathsf{O} \rangle\, \omega : \mathbf{Unit}^{\mathsf{P}}} \; *\mathsf{P} \Rightarrow *\mathsf{E}$$

(TYP WRITE)
$$\frac{\Gamma \vdash_{\mathsf{P}} \omega : \mathbf{Obj}(\tau^{\mathsf{S}})^{\mathsf{E}} \qquad \Gamma \vdash_{\mathsf{P}} x : \tau^{\mathsf{E}'} \qquad \mathsf{S} \sqsubseteq \mathsf{E}'}{\Gamma \vdash_{\mathsf{P}} \omega := x : \mathbf{Unit}^{\mathsf{P}}} \; *\mathsf{P} \Rightarrow *\mathsf{E}$$

(TYP READ)
$$\frac{\omega : \mathbf{Obj}(\tau^{\mathsf{S}})^{\mathsf{E}} \in \Gamma}{\Gamma \vdash_{\mathsf{P}} \; !\omega : \tau^{\mathsf{S} \sqcap \mathsf{P}}} \; *(\mathsf{P} \sqcap \mathsf{S}) \Rightarrow *\mathsf{E}$$

(TYP EXECUTE)
$$\frac{\omega : \mathbf{Obj}((\nabla_{\mathsf{P}'}.\, \mathbf{Bin}(\tau^{\mathsf{E}'}))^{\,\mathsf{S}})^{\mathsf{E}} \in \Gamma \qquad \mathsf{P} \sqsubseteq \mathsf{P}' \sqcap \mathsf{S}}{\Gamma \vdash_{\mathsf{P}} \mathsf{exec}\, \omega : \tau^{\mathsf{E}' \sqcap \mathsf{P}}} \; *\mathsf{P} \Rightarrow *\mathsf{E}$$

The typing rules preserve several invariants.

(1) Code that runs with a label P cannot return values that have effects higher than P.

(2) The contents of an object of type $\mathbf{Obj}(\_^{\mathsf{S}})$ cannot have effects lower than S.

(3) The dynamic label that protects an object of type $\mathbf{Obj}(\_^{\mathsf{S}})$ cannot be lower than S.

(4) An object of type $\mathbf{Obj}(\_^{\mathsf{S}})$ cannot be created at a label lower than S.

(5) Packed code of type $\nabla_{\mathsf{P}}.\, \mathbf{Bin}(\_)$ must remain well-typed when unpacked at any label lower than P.

Invariant (1) follows from our interpretation of effects. To preserve this invariant in (TYP VARIABLE), for example, the effect of x at P is obtained by lowering x's effect in the typing environment with P.

In (TYP STORE), typechecking is independent of the process label, that is, a store is well-typed if and only if it is so at any process label; recall that by (STRUCT STORE)

stores can float across contexts, and typing must be preserved by structural equivalence. Further, (TYP STORE) introduces Invariants (2) and (3). Invariant (2) follows from our interpretation of static trust annotations. To preserve this invariant we require Invariant (3), which ensures that access control prevents code running with labels less trusted than S from writing to objects whose contents are trusted at S.

By (TYP NEW), the effect E of the initial content of a new object cannot be lower than S. Recall that by (REDUCT NEW), the new object is protected with the process label P; since $\mathsf{P} \sqsupseteq \mathsf{E}$ by Invariant (1), we have $\mathsf{P} \sqsupseteq \mathsf{S}$, so that both Invariants (2) and (3) are preserved. Conversely, if $\mathsf{P} \sqsubset \mathsf{S}$ then the process does not typecheck; Invariant (4) follows.

Let us now look carefully at the other rules relevant to Invariants (2) and (3); these rules—combined with access control—are the crux of enforcing DFI. (TYP WRITE) preserves Invariant (2), restricting trusted code from writing values to ω that may flow from labels lower than S. (Such code may not be restricted by access control.) Conversely, access control prevents code with labels lower than S from writing to ω, since by Invariant (3), ω's label is at least as trusted as S. (TYP UN/PROTECT) preserves Invariant (3), allowing ω's label to be either raised or lowered without falling below S. In (TYP READ), the effect of a value read from ω at P is approximated by S—the least trusted label from which ω's contents may flow—and further lowered with P to preserve Invariant (1).

In (TYP PACK), packing code requires work akin to proof-carrying code [Necula, 1997]. Type safety for the code is proved and "carried" in its type $\nabla_{\mathsf{P}'}.\ \mathbf{Bin}(T)$, independently of the current process label. Specifically, it is proved that when the packed code is unpacked by a process with label P', the value of executing that code has type and effect T. In Section 6.4, we show that such a proof in fact allows the packed code to be unpacked by any process with label $\mathsf{P} \sqsubseteq \mathsf{P}'$, and the type and effect of the value of executing that code can be related to T (Invariant (5)). This invariant is key to decidable and efficient typechecking [Chaudhuri et al., 2007]. Of course, code may be packed to run only at specific process labels, by requiring the appropriate label changes.

Preserving Invariant (5) entails, in particular, preserving Invariant (4) at all labels $\mathsf{P} \sqsubseteq \mathsf{P}'$. Since a new expression that is not guarded by a change of the process label

may be run with any label P, that expression must place the least possible trust on the contents of the object it creates. This condition is enforced by predicate $\Box$:

$$\begin{aligned}
\Box\mathsf{new}(x \,\#\, \mathsf{S}) &\triangleq \forall \mathsf{P}.\, \mathsf{S} \sqsubseteq \mathsf{P} \\
\Box(f \Rsh g) &\triangleq \Box f \wedge \Box g \\
\Box(\text{let } x = f \text{ in } g) &\triangleq \Box f \wedge \Box g \\
\Box(\ldots) &\triangleq \texttt{true}
\end{aligned}$$

(TYP EXECUTE) relies on Invariant (5); further, it checks that the label at which the code is unpacked (P) is at most as trusted as the label at which the code may have been packed (approximated by S). This check prevents privilege escalation—code that would perhaps block if run with a lower label cannot be packed to run with a higher label. For example, recall that in Example 6.2.2, the code `binVirus` is packed at Low and then copied into `setup.exe`. While a High-process can legitimately execute `home` := `empty` (so that the code is typed and is not blocked by access control), it should not run that code by unpacking `binVirus` from `setup.exe`. The type system prevents this violation. Let `setup.exe` be of type $\mathbf{Obj}((\nabla\_.\ \mathbf{Bin}(\_))^{\mathsf{S}})$. Then (TYP STORE) requires that $\mathsf{S} \sqsubseteq \mathsf{Low}$, and (TYP EXECUTE) requires that $\mathsf{High} \sqsubseteq \mathsf{S}$ (contradiction).

Because we do not maintain an upper bound on the dynamic label of an executable, we cannot rely on the lowering of the process label in (REDUCT EXECUTE) to prevent privilege escalation. (While it is possible to extend our type system to maintain such upper bounds, such an extension does not let us typecheck any more correct programs than we already do.) In Section 6.4, we show that the lowering of the process label can in fact be safely eliminated.

In (TYP EVALUATE), typing proceeds sequentially, propagating the type and effect of the intermediate process to the continuation. (TYP SUBSTITUTION) is similar, except that the substituted value is typed under the process label recorded in the substitution, rather than under the current process label. In (TYP LIMIT), the continuation is typed under the changed process label. In (TYP FORK), the forked process is typed under the current process label.

6.3.3 Typing rules for stuck code

While the rules above rely on access control for soundness, they do not *exploit* run-time protection provided by access control to typecheck more programs. For example, the reduced process q_1 in Example 6.2.1 cannot yet be typed, although we have checked that DFI is not violated in q_1. Below, we introduce *stuck typing* to identify processes that provably block by access control at run time. Stuck typing allows us to soundly type more programs by composition. (The general principle that is followed here is that narrowing the set of possible execution paths improves the precision of the analysis.) This powerful technique of combining static typing and dynamic access control for run-time protection is quite close to hybrid typechecking [Flanagan, 2006]. We defer a more detailed discussion of this technique to Section **??**.

We introduce the static approximation **Stuck** for processes that do not return values, but may have side effects.

$T ::=$	static approximation
$\cdots$	code
Stuck	stuck process

We now present rules for stuck-typing. As before, in our first reading of these rules we ignore the side conditions (which involve the predicate $*$).

Stuck typing judgments $\Gamma \vdash_{\mathsf{P}} a : \mathbf{Stuck}$

$$\frac{\mathsf{P} \sqsubset \mathsf{P}'}{\Gamma \vdash_{\mathsf{P}} [\mathsf{P}']\, a : \mathbf{Stuck}}\ (\textsc{Typ escalate stuck}) \qquad \frac{\omega : \mathbf{Obj}(\_^{\mathsf{S}})^{\mathsf{E}} \in \Gamma \qquad \mathsf{P} \sqsubset \mathsf{S}}{\Gamma \vdash_{\mathsf{P}} \omega := x : \mathbf{Stuck}}\ (\textsc{Typ write stuck})\ *\mathsf{E}$$

$$\frac{\omega : \mathbf{Obj}(\_^{\mathsf{S}})^{\mathsf{E}} \in \Gamma \qquad \mathsf{P} \sqsubset \mathsf{S} \sqcup \mathsf{O}}{\Gamma \vdash_{\mathsf{P}} \langle \mathsf{O} \rangle\, \omega : \mathbf{Stuck}}\ (\textsc{Typ un/protect stuck})\ *\mathsf{E}$$

$$\frac{\_ : \mathbf{Stuck} \in \Gamma}{\Gamma \vdash_{\mathsf{P}} a : \mathbf{Stuck}}\ (\textsc{Typ subsumption stuck-I}) \qquad \frac{\Gamma \vdash_{\mathsf{P}} a : \mathbf{Stuck}}{\Gamma \vdash_{\mathsf{P}} a : T}\ (\textsc{Typ subsumption stuck-II})$$

(TYP WRITE STUCK) identifies code that tries to write to an object whose static trust annotation S is higher than the current process label P. By Invariant (3), the label O that protects the object must be at least as high as S; thus $\mathsf{P} \sqsubset \mathsf{O}$ and the code must block at run time due to access control. For example, let `cmd.exe` be of type $\mathbf{Obj}(\_^{\top})$ in Example 6.2.1. By (TYP WRITE STUCK), the code q_1 is well-typed since $\mathsf{Low} \sqsubset \top$. (TYP UN/PROTECT STUCK) is similar to (TYP WRITE STUCK); it further identifies code that tries to raise the label of an object beyond the current process label. (TYP ESCALATE STUCK) identifies code that tries to raise the current process label. All such processes block at run time due to access control.

By (TYP SUBSUMPTION STUCK-I), processes that are typed under stuck hypotheses are considered stuck as well. For example, this rule combines with (TYP EVALUATE) to trivially type a continuation b if the intermediate process a is identified as stuck. Finally, by (TYP SUBSUMPTION STUCK-II), stuck processes can have any type and effect, since they cannot return values.

6.3.4 Typing rules for untrusted code

Typing must guarantee protection in arbitrary environments. Since the protection guarantee is derived via a type preservation theorem, arbitrary untrusted code needs to be accommodated by the type system. We assume that untrusted code runs with a special label $\bot$, introduced into the total order by assuming $\bot \sqsubseteq \mathsf{L}$ for all L. We now present rules that allow arbitrary interpretation of types at $\bot$.

Typing rules for untrusted code

(TYP SUBSUMPTION $\bot$-I)	(TYP SUBSUMPTION $\bot$-II)
$\dfrac{\Gamma, \omega : \mathbf{Obj}(\_^{\bot})^{\mathsf{E}} \vdash_{\mathsf{P}} a : T}{\Gamma, \omega : \mathbf{Obj}(\tau^{\bot})^{\mathsf{E}} \vdash_{\mathsf{P}} a : T}$	$\dfrac{\Gamma, x : \_^{\bot} \vdash_{\mathsf{P}} a : T}{\Gamma, x : \tau^{\bot} \vdash_{\mathsf{P}} a : T}$

By (TYP SUBSUMPTION $\bot$-I), placing the static trust $\bot$ on the contents of an object amounts to assuming any type for those contents as required. By (TYP SUBSUMPTION $\bot$-II), a value that has effect $\bot$ may be assumed to have any type as required. These rules provide the necessary flexibility for typing any untrusted code using the other

typing rules. On the other hand, arbitrary subtyping with objects can in general be unsound—we now need to be careful when typing trusted code. For example, consider the code

$$\omega_2 \overset{\mathsf{High}}{\mapsto} x \;\Rsh\; \omega_1 \overset{\mathsf{Low}}{\mapsto} \omega_2 \;\Rsh\; [\mathsf{High}]\ \mathrm{let}\ z = !\omega_1\ \mathrm{in}\ z := u$$

A High-process reads the name of an object (ω_2) from a Low-object (ω_1), and then writes u to that object (ω_2). DFI is violated if ω_2 has type $\mathbf{Obj}(\_^{\mathsf{High}})$ and u flows from Low. Unfortunately, it turns out that this code can be typed under process label $\top$ and typing hypotheses

$$\omega_2 : \mathbf{Obj}(\tau_2^{\mathsf{High}})^{\top},\ \omega_1 : \mathbf{Obj}(\mathbf{Obj}(\tau_2^{\mathsf{High}})^{\bot})^{\top},\ x : \tau_2^{\mathsf{High}},\ u : \tau_1^{\mathsf{Low}}$$

Specifically, the intermediate judgment

$$z : \mathbf{Obj}(\tau_2^{\mathsf{High}})^{\bot}, \ldots, u : \tau_1^{\mathsf{Low}} \ \vdash_{\mathsf{High}}\ z := u : \_$$

can be derived by adjusting the type of z in the typing environment to $\mathbf{Obj}(\tau_1^{\mathsf{Low}})$ with (TYP SUBSUMPTION $\bot$-II).

This source of unsoundness is eliminated if some of the effects in our typing rules are required to be trusted, that is, to be higher than $\bot$. Accordingly we introduce the predicate $*$, such that for any label L, $*\mathsf{L}$ simply means $\mathsf{L} \sqsupset \bot$. We now revisit the typing rules earlier in the section and focus on the side conditions in shaded boxes (which involve $*$). In some of those conditions, we care about trusted effects only if the process label is itself trusted. With these conditions, (TYP WRITE) prevents type-checking the offending write above, since the effect of z in the typing environment is untrusted.

6.3.5 Compromise

The label $\bot$ introduced above is an artificial construct to tolerate a degree of "anarchy" in the type system. We may want to specify that a certain label (such as Low) acts like $\bot$, *i.e.*, is *compromised*. The typing judgment $\Gamma \vdash_{\mathsf{P}} a : T$ despite C allows us to type arbitrary code a running at a compromised label C by assuming that C is the same as $\bot$, *i.e.*, by extending the total order with $\mathsf{C} \sqsubseteq \bot$ (so that all labels that are at most as trusted

as C collapse to $\bot$). We do not consider labels compromised at run time (as in Gordon and Jeffrey's type system for conditional secrecy [Gordon and Jeffrey, 2005]); however we do not anticipate any technical difficulty in including run-time compromise in our type system.

6.3.6 Typechecking examples

We now show some examples of typechecking.

We begin with the program p_2 in Example 6.2.2. Recall that DFI is violated in p_2. Suppose that we try to derive the typing judgment

$$\ldots \vdash_\top p_2 : \_ \text{ despite } \mathsf{Low}$$

This amounts to deriving $\ldots \vdash_\top p_2 : \_$ by assuming $\mathsf{Low} \sqsubseteq \bot$.

As a first step, we apply (TYP NEW), (TYP READ), (TYP WRITE), (TYP PACK), and (TYP EVALUATE), directed by syntax, until we have the following typing environment.

$$\begin{aligned}
\Gamma \;=\; & \ldots, \\
& \texttt{url} : \mathbf{Obj}(_^{\mathsf{Low}})^\top, \\
& \texttt{setup.exe} : \mathbf{Obj}(_^{\mathsf{Low}})^\top, \\
& \texttt{binIE} : (\nabla_{\mathsf{Low}}.\, \mathbf{Bin}(\mathbf{Unit}))^\top, \\
& \texttt{ie.exe} : \mathbf{Obj}((\nabla_{\mathsf{Low}}.\, \mathbf{Bin}(\mathbf{Unit}))^\top)^\top, \\
& \texttt{home} : \mathbf{Obj}(_^{\mathsf{Medium}})^\top \\
& \texttt{empty} : \mathbf{Unit}^\top
\end{aligned}$$

The only complication that may arise is in this step is in deriving an intermediate judgment

$$\ldots, z : \_^{\mathsf{Low}} \vdash_\top !z : \_$$

Here, we can apply (TYP SUBSUMPTION $\bot$-II) to adjust the typing hypothesis of z to $\mathbf{Obj}(\_)^\bot$, so that (TYP READ) may apply.

After this step, we need to derive a judgment of the form:

$$\Gamma \vdash_\top [\mathsf{High}]\ (\ldots)\ \Rsh\ [\mathsf{Medium}]\ (\ldots)\ \Rsh\ [\mathsf{Low}]\ (\ldots)$$

Now, we apply (TYP FORK). We first check that the code [Low] (...) is well-typed. (In fact, untrusted code is always well-typed, as we show in Section 6.4.) The judgment

$$\Gamma \vdash_{\mathsf{Low}} \texttt{home} := \texttt{empty} : \mathbf{Unit}$$

typechecks by (TYP WRITE STUCK). Thus, by (TYP PACK) and (TYP EVALUATE), we add the following hypothesis to the typing environment.

$$\texttt{binVirus} : (\nabla_{\mathsf{Low}}.\ \mathbf{Bin}(\mathbf{Unit}))^{\mathsf{Low}}$$

Let $T_{\texttt{binVirus}} = (\nabla_{\mathsf{Low}}.\ \mathbf{Bin}(\mathbf{Unit}))^{\mathsf{Low}}$. Next, by (TYP NEW) and (TYP EVALUATE), we add the following hypothesis to the typing environment.

$$\texttt{virus.exe} : \mathbf{Obj}(T_{\texttt{binVirus}})^{\mathsf{Low}}$$

Finally, the judgment

$$\Gamma, \ldots, \texttt{virus.exe} : \mathbf{Obj}(T_{\texttt{binVirus}})^{\mathsf{Low}} \vdash_{\mathsf{Low}} \texttt{url} := \texttt{virus.exe}$$

can be derived by (TYP WRITE), after massaging the typing hypothesis for virus.exe to the required $\_^{\mathsf{Low}}$ by (TYP SUBSUMPTION $\bot$-II).

On the other hand, the process [High] (...) does not typecheck; as seen above, an intermediate judgment

$$\Gamma \vdash_{\mathsf{High}} \mathsf{exec}\ \texttt{setup.exe} : \_ \tag{6.1}$$

cannot be derived, since (TYP EXECUTE) does not apply.

To understand this situation further, let us consider some variations where (TYP EXECUTE) does apply. Suppose that the code exec z is forked in a new process whose label is lowered to Low. Then p_2 typechecks. In particular, the following judgment can be derived by applying (TYP EXECUTE).

$$\Gamma \vdash_{\mathsf{High}} [\mathsf{Low}]\ \mathsf{exec}\ \texttt{setup.exe} : \_ \tag{6.2}$$

Fortunately, the erasure of home now blocks by access control at run time, so DFI is not violated.

Next, suppose that the static annotation for `setup.exe` is High instead of Low, and `setup.exe` is initialized by a process with label High instead of Low. Then p_2 typechecks. In particular, the type of `setup.exe` in Γ becomes $\mathbf{Obj}(\_^{\mathsf{High}})$. We need to derive an intermediate judgment

$$\Gamma, \ldots, x : \_ \vdash_{\mathsf{Low}} \texttt{setup.exe} := x : \mathbf{Unit} \tag{6.3}$$

This judgment can be derived by applying (TYP WRITE STUCK) instead of (TYP WRITE). Fortunately, the overwrite of `setup.exe` now blocks by access control at run time, so DFI is not violated.

Finally, we sketch how typechecking fails for the violations of DFI described in Section 6.1.2.

(**Write and copy**) Let the type of ω be $\mathbf{Obj}(\_^{\mathsf{S}})$, where $\mathsf{O} \sqsupseteq \mathsf{S} \sqsupset \mathsf{P}$. Then the write to $\omega(\mathsf{O})$ does not typecheck, since the value to be written is read from $\omega'(\mathsf{P})$ and thus has some effect E such that $\mathsf{E} \sqsubseteq \mathsf{P}$, so that $\mathsf{E} \sqsubset \mathsf{S}$.

(**Copy and execute**) Let the type of ω' be $\mathbf{Obj}(\_^{\mathsf{S}'})$. If $\mathsf{S}' \sqsubseteq \mathsf{O}$ then the execution of $\omega'(\mathsf{P})$ by $a(\mathsf{P})$ does not typecheck, since $\mathsf{S}' \sqsubset \mathsf{P}$. If $\mathsf{S}' \sqsupset \mathsf{O}$ then the write to $\omega'(\mathsf{P})$ does not typecheck, since the value to be written is read from $\omega(\mathsf{O})$ and thus has some effect E such that $\mathsf{E} \sqsubseteq \mathsf{O}$, so that $\mathsf{E} \sqsubset \mathsf{S}'$.

(**Unprotect, write, and protect**) Let the type of ω be $\mathbf{Obj}(\_^{\mathsf{S}})$, where $\mathsf{O} \sqsupseteq \mathsf{S} \sqsupset \mathsf{P}$. Then the unprotection of $\omega(\mathsf{O})$ does not typecheck, since $\mathsf{P} \sqsubset \mathsf{S}$.

(**Copy, protect, and execute**) Let the type of ω' be $\mathbf{Obj}(\_^{\mathsf{S}'})$, where $\mathsf{S}' \sqsubseteq \mathsf{O}$. Then the execution of $\omega'(\mathsf{P})$ does not typecheck, since $\mathsf{S}' \sqsubset \mathsf{P}$.

6.4 Properties of typing

In this section we show several properties of typing, and prove that DFI is preserved by well-typed code under arbitrary untrusted environments. All proof details appear in [Chaudhuri et al., 2007].

We begin with the proposition that untrusted code can always be accommodated by the type system.

Definition 6.4.1 (Adversary)**.** *A* C*-adversary is any process of the form* $[\mathsf{C}]\ \_$ *that does not contain stores, explicit substitutions, and static trust annotations that are higher than* C*.*

Proposition 6.4.2 (Adversary completeness)**.** *Let* Γ *be any typing environment and* c *be any* C*-adversary such that* $\mathtt{fv}(c) \subseteq \mathsf{dom}(\Gamma)$*. Then* $\Gamma \vdash_\top c : \_$ despite C*.*

Proposition 6.4.2 provides a simple way to quantify over arbitrary environments. By (TYP FORK) the composition of a well-typed process with any such environment remains well-typed, and thus enjoys all the properties of typing.

Next, we present a monotonicity property of typing that is key to decidable and efficient typechecking [Chaudhuri et al., 2007].

Proposition 6.4.3 (Monotonicity)**.** *The following inference rule is admissible.*

$$\frac{\Gamma \vdash_{\mathsf{P}'} f : \tau^{\mathsf{E}} \qquad \Box f \qquad \mathsf{P} \sqsubseteq \mathsf{P}'}{\Gamma \vdash_{\mathsf{P}} f : \tau^{\mathsf{E} \sqcap \mathsf{P}}}$$

This rule formalizes Invariant (5), and allows inference of "most general" types for packed code [Chaudhuri et al., 2007]. Further, it implies an intuitive proof principle—code that is proved safe to run with higher privileges remains safe to run with lower privileges, and conversely, code that is proved safe against a more powerful adversary remains safe against a less powerful adversary.

The key property of typing is that it is preserved by structural equivalence and reduction. Preservation depends delicately on the design of the typing rules, relying on the systematic maintenance of typing invariants. We write $\Gamma \vdash \sigma$, meaning that "the substitution environment σ is consistent with the typing environment Γ", if for all $x/\mu@\,\mathsf{P} \in \sigma$ there exists T such that $x : T \in \Gamma$ and $\Gamma \vdash_{\mathsf{P}} \mu : T$.

Theorem 6.4.4 (Preservation of typability)**.** *Suppose that* $\Gamma \vdash \sigma$ *and* $\Gamma \vdash_{\mathsf{P}} a : \_$*. Then*

- *if* $a \equiv b$ *then* $\Gamma \vdash_{\mathsf{P}} b : \_$*;*
- *if* $a \xrightarrow{\mathsf{P};\sigma} b$ *then* $\Gamma \vdash_{\mathsf{P}} b : \_$*.*

We now present our formal protection guarantee for well-typed code. We begin by generalizing the definition of DFI in Section 6.2. In particular, we assume that part of

the adversary is known and part of it is unknown. This assumption allows the analysis to exploit any sound typing information that may be obtained from the known part of the adversary. (As a special case, the adversary may be entirely unknown, of course. In this case, we recover Definition 6.2.4; see below.) Let Ω be the set of objects that require protection from labels L or lower. We let the unknown part of the adversary execute with some process label C ($\sqsubseteq \mathsf{L}$). We say that Ω is protected if no such adversary can write any instance that flows from L or lower, to any object in Ω.

Definition 6.4.5 (Generalized DFI)**.** *A set of objects Ω is protected by code a from label L despite C ($\sqsubseteq \mathsf{L}$) if there is no $\omega \in \Omega$, C-adversary c, substitution environment σ, and instance x such that $a \Rsh c \xrightarrow{\top,\varnothing}{}^{\star} \mathcal{E}_{\top,\varnothing}[\![\omega \mathrel{\dot{\mapsto}} x]\!]_{\top,\sigma}$ and $x \overset{\sigma}{\blacktriangledown} \mathsf{L}$.*

For example, we may want to prove that some code protects a set of High-objects from Medium despite (the compromised label) Low; then we need to show that no instance may flow from Medium or lower to any of those High-objects under any Low-adversary.

We pick objects that require protection based on their types and effects in the typing environment.

Definition 6.4.6 (Trusted objects)**.** *The set of objects whose contents are trusted beyond the label L in the typing environment Γ is $\{\omega \mid \omega : \mathbf{Obj}(\_^{\mathsf{S}})^{\mathsf{E}} \in \Gamma$ and $\mathsf{S} \sqcap \mathsf{E} \sqsupset \mathsf{L}\}$.*

Suppose that in some typing environment, Ω is the set of objects whose contents are trusted beyond label L, and C ($\sqsubseteq \mathsf{L}$) is compromised; we guarantee that Ω is protected by any well-typed code from L despite C.

Theorem 6.4.7 (Enforcement of strong DFI)**.** *Let Ω be the set of objects whose contents are trusted beyond L in Γ. Suppose that $\Gamma \vdash_{\top} a : \_$ despite C, where $\mathsf{C} \sqsubseteq \mathsf{L}$. Then a protects Ω from L despite C.*

In the special case where the adversary is entirely unknown, we simply consider L and C to be the same label.

The type system further enforces DFI for new objects, as can be verified by applying Theorem 6.4.4, (TYP SUBSTITUTE), and Theorem 6.4.7.

Finally, the type system suggests a sound run-time optimization: whenever a well-typed process executes packed code in a trusted context, the current process label is already appropriately lowered for execution.

Theorem 6.4.8 (Redundancy of execution control)**.** *Suppose that* $\Gamma \vdash_\top a : \_ \,\mathsf{despite}\, \mathsf{C}$ *and* $a \xrightarrow{\top;\varnothing} \mathcal{E}_{\top;\varnothing}[\![\omega \overset{\mathsf{O}}{\mapsto} \_ \Rsh \mathsf{exec}\, \omega']\!]_{\mathsf{P};\sigma}$ *such that* $\omega \overset{\sigma}{=} \omega'$ *and* $\mathsf{P} \sqsupset \mathsf{C}$*. Then* $\mathsf{P} \sqsubseteq \mathsf{O}$*.*

It follows that the rule (REDUCT EXECUTE) can be safely optimized as follows.

$$\frac{\omega \overset{\sigma}{=} \omega' \qquad \mathsf{pack}(f) \in \sigma(x)}{\omega \overset{\mathsf{O}}{\mapsto} x \Rsh \mathsf{exec}\, \omega' \xrightarrow{\mathsf{P};\sigma} \omega \overset{\mathsf{O}}{\mapsto} x \Rsh f}$$

This optimization should not be surprising. Lowering the process label for execution aims to prevent trusted code from executing untrusted code in trusted contexts; our core static discipline on trusted code effectively subsumes this run-time control. On the other hand, write-access control cannot be eliminated by any discipline on trusted code, since that control is required to restrict untrusted code.

Lastly, typechecking can be efficiently mechanized thanks to Proposition 6.4.3 and our syntactic restriction on nested packing.

Theorem 6.4.9 (Typechecking)**.** *Given a typing environment* Γ *and code a with* $\mathbb{L}$ *distinct labels, the problem of whether there exists T such that* $\Gamma \vdash_\top a : T$*, is decidable in time* $\mathcal{O}(\mathbb{L}|a|)$*, where* $|a|$ *is the size of a.*

A typechecking algorithm is outlined in [Chaudhuri et al., 2007]. As usual, the algorithm builds constraints and then checks whether those constraints are satisfiable. The only complication is due to pack processes, which require "most general" types.

Briefly, the grammar of types is extended with type variables, and a distinguished label ? is introduced to denote an "unknown" label. Let a *typechecking environment* Δ be a typing environment augmented by simple type constraints, and a *label constraint* (a boolean formula with propositions of the form $\mathsf{L}_1 \sqsubseteq \mathsf{L}_2$). The following typechecking judgments are defined, with mutually recursive rules:

- $\Delta \vdash_\mathsf{P} a : T \rhd \Delta'$, where the label constraint in Δ' is true.
- $\Delta \vdash f : T \rhd \Delta'$, where Δ' contains a label constraint over ?.

The rules for $\Delta \vdash_{\mathsf{P}} a : T \rhd \Delta'$ build simple type constraints in Δ', following the original typing rules. To derive a judgment of the form $\Delta \vdash_{\mathsf{P}} \mathsf{pack}(f) : \_ \rhd \_$, we need to derive a judgment of the form $\Delta \vdash f : \_ \rhd \_$. The rules for $\Delta \vdash f : T \rhd \Delta'$ build label constraints from conditions on labels in the original typing rules; here, the implicit (unknown) process label is taken to be ?. To derive a judgment of the form $\Delta \vdash [\mathsf{P}]\ a : \_ \rhd \_$, we need to derive a judgment of the form $\Delta \vdash_{\mathsf{P}} a : \_ \rhd \_$. On the other hand, the syntactic restriction on expressions ensures that we do not need to consider judgments of the form $\Delta \vdash \mathsf{pack}(f) : \_ \rhd \_$.

Solving the simple type constraints built by a judgment of the form $\Delta \vdash_{\mathsf{P}} a : \_ \rhd \_$ takes time $\mathcal{O}(|a|)$; solving the label constraint built by a judgment of the form $\Delta \vdash f : \_ \rhd \_$ takes time $\mathcal{O}(\mathbb{L}|f|)$. The running time of the typechecking algorithm follows by a straightforward inductive argument.

Part III

Preserving Security by Correctness

Overview

In this final part, we focus on advanced techniques for security analysis of computer systems. Specifically, we consider techniques to specify and verify security of computer systems through correctness of their implementations. These techniques roughly serve to bridge directions (a) and (b) of our research program. In particular, we apply these techniques to specify and verify the security of networked storage systems through the correctness of their implementations of access control.

Indeed, distributed implementations of access control abound in such networked storage protocols. While such implementations are often accompanied by informal justifications of their correctness, our formal analysis reveals that their correctness can be tricky. In particular, we discover several subtleties in a state-of-the-art implementation based on capabilities, that can undermine correctness under a simple specification of access control.

We consider both "safety" and "security" for correctness; loosely, safety requires that an implementation does not introduce unspecified behaviors, and security requires that an implementation preserves the specified behavioral equivalences. We show that a secure implementation of a static access policy already requires some care in order to prevent unspecified leaks of information about the access policy. A dynamic access policy causes further problems. For instance, if accesses can be dynamically granted then the implementation does not remain secure—it leaks information about the access policy. If accesses can be dynamically revoked then the implementation does not even remain safe. We show that a safe implementation is possible if a clock is introduced in the implementation. A secure implementation is possible if the specification is accordingly generalized.

Our analysis details how a distributed implementation can be systematically designed from a specification, guided by precise formal goals. While our results are based on formal criteria, we show how violations of each of those criteria can lead to real attacks. We distill the key ideas behind those attacks and propose corrections in terms of useful design principles. Other stateful computations can be distributed just as well using those principles.

Chapter 7

Distributed access control

Most file systems rely on access control for protection. Usually, the access checks are local—the file system maintains an access policy that specifies which principals may access which files, and any access to a file is guarded by a local check that enforces the policy for that file. In recent file systems, however, the access checks are distributed, and access control is implemented via cryptographic techniques.

In this chapter, we reason about the extent to which such access control implementations preserve the character of local access checks.[1] In particular, we consider implementations based on *capabilities* that appear in protocols for networked storage, such as the Network-Attached Secure Disks (NASD) and Object-based Storage Devices (OSD) [Gobioff et al., 1997; Halevi et al., 2005] protocols. Such protocols distribute access checks to improve performance. Specifically, when a user requests access to a file, an access-control server certifies the access decision for that file by providing the user with an unforgeable capability. Subsequently, the user accesses the file at a storage server by presenting that capability as proof of access; the storage server verifies that the capability is authentic before allowing access to the file.

We study the correctness of access control in this setting, under a simple specification of local access control. Implementing static access policies already requires some care; dynamic access policies cause further problems that require considerable analysis to iron out. We study these cases separately, in detail, in Sections 7.1 and 7.2.

[1] Since local access checks assume that the underlying file system is trusted, protocols for untrusted storage such as Plutus (Chapter 2) are outside the scope of this chapter.

We consider both "safety" and "security" for correctness; loosely, safety requires that an implementation does not introduce unspecified behaviors, and security requires that an implementation preserves the specified behavioral equivalences. We introduce these concepts in Section 7.1.

We formalize our results in the applied pi calculus [Abadi and Fournet, 2001]. Basically, our correctness theorems imply that safety and security properties that are proved in the specification carry over "for free" in the implementation. Our correctness proofs are built modularly by showing simulations; we develop the necessary definitions and proof techniques in Section 7.3, and outline the proofs in Section 7.4.

Our analysis details how a distributed implementation can be systematically designed from a specification, guided by precise formal goals. While our results are based on formal criteria, we show how violations of each of those criteria can lead to real attacks (Sections 7.1 and 7.2). We distill the key ideas behind those attacks and propose corrections in terms of useful design principles (Sections 7.1 and 7.2). Other stateful computations can be distributed just as well using those principles, as shown in [Chaudhuri, 2008a].

7.1 Implementing static access policies

To warm up, let us focus on implementing access policies that are *static*. In this case, a secure implementation appears in [Chaudhuri and Abadi, 2005]. Below, we systematically reconstruct that implementation, focusing on a detailed analysis of its correctness. This analysis allows us to distill some basic design principles, marked with bold **R**, in preparation for later sections, where we consider the more difficult problem of implementing dynamic access policies.

Consider the following protocol, NS^s, for networked storage.[2] This protocol captures the essence of the NASD and OSD protocols [Gobioff et al., 1997; Halevi et al., 2005]; as we move along, we present more complicated variants of this protocol. Prin-

[2] In protocol names throughout this chapter, we use the superscript ^{s} or ^{d} to indicate whether the access policy in the underlying protocol is "static" or "dynamic"; sometimes, we also use the superscript $^+$ or $^-$ to indicate whether the underlying protocol is derived by "extending" or "restricting" some other protocol.

cipals include users U, V, W, and so on, an access-control server A, and a storage server S. We assume that A maintains a static access policy F and S maintains a store ρ. Access decisions under F follow an arbitrary relation $F \vdash_U op$ over users U and operations op. Execution of an operation op under ρ follows an arbitrary relation $\rho[\![op]\!] \Downarrow \rho'[\![r]\!]$ over next stores ρ' and results r. Let K_{AS} be a secret key shared by A and S, and **mac** be a function over messages and keys that produces unforgeable message authentication codes (MACs) [Goldwasser and Bellare, 2001]. We assume that MACs can be decoded to retrieve their messages. (Usually MACs are explicitly paired with their messages, so that the decoding is trivial.)

(1)	U	$\rightarrow$	A	:	op	
(2)	A	$\rightarrow$	U	:	$\mathbf{mac}(op, K_{AS})$	if $F \vdash_U op$
(2′)	A	$\rightarrow$	U	:	`error`	otherwise
(3)	V	$\rightarrow$	S	:	κ	
(4)	S	$\rightarrow$	V	:	r	if $\kappa = \mathbf{mac}(op, K_{AS})$ and $\rho[\![op]\!] \Downarrow \rho'[\![r]\!]$
(4′)	S	$\rightarrow$	V	:	`error`	otherwise

Here a user U requests A for access to an operation op, and A returns a capability for op only if F specifies that U may access op. Elsewhere, a user V requests S to execute an operation by sending a capability κ, and S executes the operation only if κ authorizes access to that operation.

What does "safety" or "security" mean in this setting? A reasonable specification of correctness is the following trivial protocol, IS^s, for ideal storage. Here principals include users U, V, W, and so on, and a server D. The access policy F and the store ρ are both maintained by D; the access and execution relations remain as above. There is no cryptography.

(i)	V	$\rightarrow$	D	:	op	
(ii)	D	$\rightarrow$	V	:	r	if $F \vdash_V op$ and $\rho[\![op]\!] \Downarrow \rho'[\![r]\!]$
(ii′)	D	$\rightarrow$	V	:	`error`	otherwise

Here a user V requests D to execute an operation op, and V executes op only if F specifies that V may access op. This trivial protocol is correct "by definition"; so if NS^s

implements this protocol, it is correct as well.

What correctness criteria are appropriate here? A basic criterion is that of safety (by refinement) [Abadi and Lamport, 1991].

Definition 7.1.1 (Safety)**.** *Under any context (adversary), the behaviors of a safe implementation are included in the behaviors of the specification.*

In practice, the requirement of strict inclusion is often inconvenient, and a suitable alternative may need to be crafted to accommodate specific implementation behaviors by design (such as those due to messages (1), (2), and $(2')$ in NS^s). Typically, those behaviors can be eliminated by a specific context (called a "wrapper"), and safety may be defined modulo that context as long as other, interesting behaviors are not eliminated.

Still, safety only implies the preservation of certain trace properties. A more powerful criterion is derived from the programming languages concept of semantics preservation, otherwise known as *full abstraction* [Milner, 1977; Abadi, 1998].

Definition 7.1.2 (Security)**.** *A secure implementation preserves behavioral equivalences of the specification.*

In this chapter, we tie security to an appropriate may-testing equivalence [Nicola and Hennessy, 1984]. We consider a protocol instance to include the file system and some code run by "honest" users, and assume that an arbitrary, unspecified context colludes with the remaining "dishonest" users. From any NS^s instance, we derive its IS^s instance by an appropriate refinement map [Abadi and Lamport, 1991] (roughly, a map from implementation states to specification states). Then NS^s is a secure implementation of IS^s if and only if for all NS^s instances Q_1 and Q_2, whenever Q_1 and Q_2 can be distinguished, so can be their IS^s instances.

Breaking safety usually suffices to break security. For example, we are in trouble if operations that cannot be executed in IS^s can be executed in NS^s by manipulating capabilities. Suppose that $F \nvdash_V op$ for all dishonest V. Then no such V can execute op in IS^s. Now suppose that some such V requests execution of op in NS^s. Of course, op is executed only if V shows a capability κ for op. Since κ cannot be forged, it must be obtained from A by some honest U that satisfies $F \vdash_U op$. Therefore:

R1 Capabilities obtained by honest users must not be shared with dishonest users.

(However U can still share such κ with honest users, and any execution request with κ can then be reproduced in the specification as an execution request by U.)

While (R1) prevents *explicit* leaking of capabilities, we in fact require that capabilities do not leak *any* information that is not available to IS^s contexts. Information may also be leaked implicitly (by observable effects). Therefore:

R2 Capabilities obtained by honest users should not be examined (say, with destructors) or compared (say, with equality checks), *i.e.*, they must remain abstract.

Both (R1) and (R2) may be enforced by typechecking the code run by honest users.

Finally, we require that information is not leaked via capabilities obtained by dishonest users. (Recall that such capabilities are already available to the adversary.) Unfortunately, a capability for an operation *op* is provided *only* to those users who have access to *op* under F; in other words, A leaks information on F whenever it returns a capability! (If we do not care about this leak, then we must allow the same leak in the specification.) This leak breaks security. Why? Consider implementation instances Q_1 and Q_2 with *op* as the only operation, whose execution returns `error` and may be observed only by honest users; suppose that a dishonest user has access to *op* in Q_1 but not in Q_2. Then Q_1 and Q_2 can be distinguished by a context that requests a capability for *op*—a capability will be returned in Q_1 but not in Q_2—but their specification instances cannot be distinguished by any context.

Why does this leak concern us? After all, we expect that executing an operation *should* eventually leak some information about access to that operation, since otherwise, controlling access to that operation makes no sense. However, the leak here is premature; it allows a dishonest user to obtain information about its access to *op* in an undetectable way, *without* having to request execution of *op*.

To prevent this leak:

R3 "Fake" capabilities for *op* (rather than `error`) must be returned to users who do not have access to *op*.

The point is that it should not be possible to distinguish the fake capabilities from the real ones prematurely. Let $\overline{K}_{AS}$ be another secret key shared by A and S. As a

preliminary fix, let us modify the following message in NS^s.

$$(2') \quad A \rightarrow U : \mathbf{mac}(op, \overline{K}_{AS}) \quad \text{if } F \not\vdash_U op$$

Unfortunately, this modification is not enough, since the adversary can still compare capabilities that are obtained by different users for a particular operation op, to know if their accesses to op are the same under F. To prevent this leak:

R4 Capabilities for different users must be different.

For example, a capability can mention the user whose access it authenticates. Making the meaning of a message explicit in its content is a prudent practice for security [Abadi and Needham, 1996], and we use it on several occasions in this chapter. Accordingly, we modify the following messages in NS^s.

$$(2) \quad A \rightarrow U : \mathbf{mac}(\langle U, op\rangle, K_{AS}) \quad \text{if } F \vdash_U op$$
$$(2') \quad A \rightarrow U : \mathbf{mac}(\langle U, op\rangle, \overline{K}_{AS}) \quad \text{otherwise}$$

$$(4) \quad S \rightarrow V : r \quad \text{if } \kappa = \mathbf{mac}(\langle \_, op\rangle, K_{AS}) \text{ and } \rho[\![op]\!] \Downarrow \rho'[\![r]\!]$$

(On receiving a capability κ from V, S still does not care whether V is the user to which κ is issued, even if that information is now explicit in κ.)

The following result can then be proved [Chaudhuri and Abadi, 2005]; see Section 7.3 for a formal statement of this result.

Theorem 7.1.3. *NS^s is a secure implementation of IS^s.*

Recall that in this case, the access policy is forced to be static. It follows that if a capability correctly certifies an access decision, that decision is always correct. This restriction simplifies the implementation. However, in general, the access decision certified by a capability may not be correct in the future. This fact is a major source of difficulties, and we study those difficulties in the next section.

7.2 Implementing dynamic access policies

We now consider the general problem of implementing dynamic access policies. Let F be dynamic; the following protocol, NS^d, is obtained by adding administration mes-

sages to NS^s. Execution of an administrative operation θ under F follows an arbitrary relation $F[\![\theta]\!] \Downarrow F'[\![r]\!]$ over next policies F' and results r.

$$
\begin{array}{lllll}
(5) & W & \rightarrow & A & : \ \theta \\
(6) & A & \rightarrow & W & : \ r \qquad \text{if } F \vdash_W \theta \text{ and } F[\![\theta]\!] \Downarrow F'[\![r]\!] \\
(6') & A & \rightarrow & W & : \ \texttt{error} \qquad \text{otherwise}
\end{array}
$$

Here a user W requests A to execute an administrative operation θ, and A executes θ (perhaps modifying F) if F specifies that W controls θ. The following protocol, IS^d, is obtained by adding similar messages to IS^s.

$$
\begin{array}{lllll}
(\text{iii}) & W & \rightarrow & D & : \ \theta \\
(\text{iv}) & D & \rightarrow & W & : \ r \qquad \text{if } F \vdash_W \theta \text{ and } F[\![\theta]\!] \Downarrow F'[\![r]\!] \\
(\text{iv}') & D & \rightarrow & W & : \ \texttt{error} \qquad \text{otherwise}
\end{array}
$$

Unfortunately, NS^d does not remain a secure implementation of IS^d. Consider the NS^d pseudo-code below. Here κ is a capability for an operation *op* and θ modifies access to *op*. Informally,

- `acquire` κ means "obtain capability κ"—by sending *op* in message (1), receiving a capability in message (2) or (2′), and binding the capability to κ;
- `use` κ means "request execution with κ"—by sending κ in message (3);
- `success` means "detect successful use of a capability"—by receiving a result in message (4) or (4′) and examining the result;
- `chmod` θ means "request access modification θ"—by sending θ in message (5).

T1 `acquire` κ; `chmod` θ; `use` κ; `success`

T2 `chmod` θ; `acquire` κ; `use` κ; `success`

Now (T1) and (T2) map to the same IS^d pseudo-code `chmod` θ; `exec` *op*; `success`, where informally,

- `exec` *op* means "request execution of *op*"—by sending *op* in message (i).

(Requesting execution with κ in NS^d amounts to requesting execution of *op* in IS^d, so the refinement map from NS^d pseudo-code to IS^d pseudo-code erases occurrences of `acquire` and replaces occurrences of `use` with the appropriate occurrences of `exec`.)

Now, suppose that initially no user has access to *op*, and θ specifies that all users may access *op*. Then (T1) and (T2) can be distinguished by testing the event `success`. In (T1), κ cannot authorize access to *op*, so `success` must be false; but in (T2), κ may authorize access to *op*, so `success` may be true.

Worse, if revocation is possible, NS^d does not even remain a safe implementation of IS^d! Why? Let θ specify that access to *op* is revoked for some user U, and `revoked` be the event that θ is executed (thus modifying the access policy). In IS^d, U cannot execute *op* after `revoked`. But in NS^d, U can execute *op* after `revoked` by using a capability that it acquires before `revoked`.

7.2.1 Safety in a special case

One obvious way of eliminating the counterexample above is to assume that:

A1 Accesses cannot be dynamically revoked.

This assumption may be reasonable enough for particular applications; crucially, it does not restrict the access policy from dynamically accommodating new users. On the other hand, it suggests that any access should be granted only with sufficient care, because that access cannot be subsequently denied. While this situation is not ideal, it suffices, *e.g.*, for storing short-term secrets. Further, it allows us to prove the following new result, without complicating capabilities at all (see Section 7.4).

Theorem 7.2.1. *NS^d is a safe implementation of IS^d assuming* (A1).[3]

The key observation is that with (A1), since a user cannot access an operation until it can always access that operation, the user gains no advantage by acquiring capabilities early.

Of course, we must still find a way to recover safety (and security) with revocation. It is generally recognized that revocation is problematic for distributed implementations of access control, where authorization is certified by capabilities or keys. At the

[3]Some implementation details, such as (R3), are not required for safety.

very least, we expect that capabilities need to be more sophisticated. Below, we show how to recover safety by introducing *time*.

7.2.2 Safety in the general case

Let A and S share a counter, and let a similar counter appear in D. We use these counters as (logical) clocks, and refer to their values as time. We require that:

R5 Any capability that is produced at time Clk expires at time $\mathsf{Clk} + 1$.

R6 Any administrative operation requested at time Clk is executed at the next clock tick (to time $\mathsf{Clk} + 1$), so that policies in NS^d and IS^d may change only at clock ticks (and not between).

We call this scheme *midnight shifting*, since the underlying idea is roughly that of "changing permissions for the day while users are sleeping". Implementing this scheme is straightforward. To implement (R5), capabilities carry timestamps. To implement (R6), an auxiliary variable Ξ is introduced to shadow Γ—administrative operations are executed on Ξ instead of Γ, and at every clock tick, Γ is updated to Ξ. Accordingly, we modify the following messages in NS^d to obtain the protocol NS^{d+}.

$$(2) \quad A \rightarrow U\colon \mathbf{mac}(\langle U, op, \mathsf{Clk}\rangle, K_{AS}) \quad \text{if } F \vdash_U op$$
$$(2') \quad A \rightarrow U\colon \mathbf{mac}(\langle U, op, \mathsf{Clk}\rangle, \overline{K}_{AS}) \quad \text{otherwise}$$

$$(4) \quad S \rightarrow V\colon r \qquad \text{if } \kappa = \mathbf{mac}(\langle \_, op, \mathsf{Clk}\rangle, K_{AS}) \text{ and } \rho[\![op]\!] \Downarrow \rho'[\![r]\!]$$

$$(6) \quad A \rightarrow W\colon r \qquad \text{if } F \vdash_W \theta \text{ and } \Xi[\![\theta]\!] \Downarrow \Xi'[\![r]\!]$$

Likewise, we modify the following message in IS^d to obtain the protocol IS^{d+}.

$$(\text{iv}) \quad D \rightarrow W : r \quad \text{if } F \vdash_W \theta \text{ and } \Xi[\![\theta]\!] \Downarrow \Xi'[\![r]\!]$$

Now a capability that carries Clk as its timestamp certifies a particular access decision at the instant Clk: the meaning is made explicit in the content, following prudent practice. However, recall that MACs can be decoded to retrieve their messages. In

particular, users can tell the time in NS^{d+} by decoding capabilities. Clearly we require that:

R7 If it is possible for users to tell the time in NS^{d+}, it must also be possible for users to do so in IS^{d+}.

So we must make it possible for users to tell the time in IS^{d+}. (The alternative is to make it impossible for users to tell the time in NS^{d+}. We can do this by encrypting the timestamps carried by capabilities—recall that the notion of time here is purely logical. We consider this alternative later in the section.) Accordingly we add the following messages to IS^{d+}.

$$\begin{array}{llllll} (\mathrm{v}) & U & \rightarrow & D & : & () \\ (\mathrm{vi}) & D & \rightarrow & U & : & \mathsf{Clk} \end{array}$$

The following result can then be proved. A version of this result already appears in [Chaudhuri and Abadi, 2006a], but the definition of safety there is rather ad hoc; in Section 7.4, we prove this result again, for a stronger definition of safety.

Theorem 7.2.2. *NS^{d+} is a safe implementation of IS^{d+}.*

Unfortunately, beyond this result, [Chaudhuri and Abadi, 2006a] does not consider security. In the rest of this section, we analyze the difficulties that arise for security, and present further results that appear in [Chaudhuri, 2008b].

7.2.3 Obstacles to security

It turns out that there are several recipes to break security, and expiry of capabilities is a common ingredient. Clearly, using an expired capability has no counterpart in IS^{d+}. So:

R8 Any use of an expired capability must block (without any observable effect).

Indeed, security breaks without (R8). Consider the NS^{d+} pseudo-code below. Here κ is a capability for operation *op*. Informally,

- `stale` means "detect any use of an expired capability"—by receiving a result in message (4) or (4′) and examining the result.

T3 `acquire` κ`;use` κ`;stale`

Without (R8), (T3) can be distinguished from a `false` event by testing the event `stale`. But consider implementation instances Q_1 and Q_2 with *op* as the only operation, whose execution has no observable effect on the store; let Q_1 run (T3) and Q_2 run `false`. Since `stale` cannot be reproduced in the specification, it must map to `false`. So the specification instances of Q_1 and Q_2 run `exec` *op*`;false` and `false`. These instances cannot be distinguished.

Before we move on, let us carefully understand what (R8) implies. The soundness of this condition hinges on the fact that blocking is not observable by may-testing [Nicola and Hennessy, 1984]. However, under some reasonable fairness assumptions, blocking becomes observable. Then, the only way out is to allow a similar observation in the specification, say by letting an execution request block nondeterministically. We consider such a solution in more detail below; but first, let us explore how far we can go with (R8).

Expiry of a capability yields the information that time has elapsed between the acquisition and use of that capability. We may expect that leaking this information is harmless; after all, the elapse of time can be trivially detected by inspecting timestamps. Why should we care about such a leak? If the adversary knows that the clock has ticked at least once, it also knows that any pending administrative operations have been executed, possibly modifying the access policy. If this information is leaked in a way that cannot be reproduced in the specification, we are in trouble. Any such way allows the adversary to *implicitly* control the expiry of a capability before its use. (Explicit controls, such as comparison of timestamps, are not problematic, since they can be reproduced in the specification.)

For example, consider the NS^{d+} pseudo-code below. Here κ and κ' are capabilities for operations *op* and *op*′, and θ modifies access to *op*.

T4 `acquire` κ'`;chmod` θ`;acquire` κ`;use` κ`;success;use` κ'`;success`

T5 `chmod` θ`;acquire` κ`;use` κ`;success;acquire` κ'`;use` κ'`;success`

Both (T4) and (T5) map to the same IS^{d+} pseudo-code

$$\texttt{chmod}\ \theta; \texttt{exec}\ op; \texttt{success}; \texttt{exec}\ op'; \texttt{success}$$

But suppose that initially no user has access to op and all users have access to op', and θ specifies that all users may access op. Now, the intermediate `success` event is true only if θ is executed; therefore it "forces" time to elapse for progress. It follows that (T4) and (T5) can be distinguished by testing the final `success` event. In (T4), κ' must be stale when used, so the event must be false; but in (T5), κ' may be fresh when used, so the event may be true. Therefore, security breaks.

7.2.4 Security in a special case

One way of plugging this leak is to consider that the elapse of time is altogether unobservable to users. (This prospect is not as shocking as it sounds, since time here is simply the value of a privately maintained counter.) Let us assume that:

A2 Accesses cannot be dynamically granted.

A3 Any unsuccessful use of a capability blocks (without any observable effect).

It turns out that with (A2) and (A3), there remains no way to detect the elapse of time, except by comparing timestamps. To prevent the latter, we assume that:

A4 Timestamps are encrypted.

Let E_{AS} be a secret encryption key shared by A and S. The encryption of a term M with E_{AS} under a random coin m is written as $\{m, M\}_{E_{AS}}$. Randomization takes care of (R4), so capabilities do not need to mention users. Now, we remove message $(4')$ and modify the following messages in NS^{d+} to obtain the protocol NS^{d-}.

$$\begin{array}{lllll} (2) & A & \rightarrow & U : \mathbf{mac}(\langle op, \{m, \mathsf{Clk}\}_{E_{AS}}\rangle, K_{AS}) & \text{if } \Gamma \vdash_U op \\ (2') & A & \rightarrow & U : \mathbf{mac}(\langle op, \{m, \mathsf{Clk}\}_{E_{AS}}\rangle, \overline{K}_{AS}) & \text{otherwise} \end{array}$$

$$(4) \quad S \rightarrow V : r \qquad \text{if } \kappa = \mathbf{mac}(\langle op, \{\_, \mathsf{Clk}\}_{E_{AS}}\rangle, K_{AS}) \text{ and } \rho[\![op]\!] \Downarrow \rho'[\![r]\!]$$

Likewise, we remove the messages (iv'), (v), and (vi) from IS^{d+} to obtain the protocol IS^{d-}. We can then prove the following new result (see Section 7.4):

Theorem 7.2.3. *NS^{d-} is a secure implementation of IS^{d-} assuming* (A2), (A3), *and* (A4).

The key observation is that with (A2), (A3), and (A4), the adversary cannot force time to elapse, so capabilities do not need to expire! In this model, any access revocation can be faked by indefinitely delaying the service of requests that require that access. Note that (A4) is perfectly reasonable as an implementation strategy. On the other hand, (A2) is a bit conservative; in particular, new users must be accommodated by some default access policy that is based (at least partially) on static information. Finally, (A3) is as problematic as (R8). Thus, this result is largely of theoretical interest. Its main purpose is to expose the limitations of a secure implementation under the current specification.

7.2.5 Security in the general case

More generally, we may consider some static analysis for plugging all problematic information leaks caused by expiry of capabilities. (Any such analysis must be incomplete because of the undecidability of the problem.) However, several complications arise in this effort.

- The adversary can control the elapse of time (and hence the expiry of capabilities) by interacting with honest users in subtle ways. Such interactions lead to counterexamples of the same flavor as the one with (T4) and (T5) above, but are difficult to prevent statically without severely restricting the code run by honest users. For example, even if the suspicious-looking pseudo-code `chmod` θ; `acquire` κ; `use` κ; `success` in (T4) and (T5) is replaced by an innocuous pair of signals on a public channel *net*, the adversary can still run that code in parallel and serialize it between this pair of signals.

- Even if we restrict the code run by honest users, such that every use of a capability immediately follows its acquisition (or can be serialized as such), the adversary can still delay the service of requests by interacting with the file system. Unless we have a way to constrain this elapse of time, we are in trouble. (This point can be better appreciated by looking at our proof details, in the appendix.)

For example, consider the NS^{d+} pseudo-code below. Here κ is a capability for operation *op*, and θ modifies access to *op*; further, $net()$ and $\overline{net}\langle\rangle$ denote input and output on a public channel *net*, and $\overline{done}\langle\rangle$ denotes output on a public channel *done*.

T6 $\texttt{acquire}\ \kappa; \texttt{use}\ \kappa; \texttt{chmod}\ \theta; \overline{net}\langle\rangle; net(); \texttt{success}; \overline{done}\langle\rangle$

T7 $\overline{net}\langle\rangle; net(); \overline{done}\langle\rangle$

Although $\texttt{use}\ \kappa$ immediately follows $\texttt{acquire}\ \kappa$ in (T6), the adversary can force time to elapse between $\texttt{use}\ \kappa$ and `success`. Suppose that initially no user has access to *op* or *op*′, θ specifies that a honest user U may access *op*, and θ' specifies that all users may access *op*′. Consider the following context. Here κ' is a capability for *op*′.

$$net(); \texttt{chmod}\ \theta'; \texttt{acquire}\ \kappa'; \texttt{use}\ \kappa'; \texttt{success}; \overline{net}\langle\rangle$$

This context forces time to elapse between a pair of signals on *net*—indeed, `success` is true only if θ' is executed. Therefore, this context can distinguish (T6) and (T7) by testing output on *done*: in (T6), κ does not authorize access to *op*, so `success` must be false and there is no output on *done*; on the other hand, in (T7), there is. Security breaks as a consequence. Why? Consider implementation instances Q_1 and Q_2 with U as the only honest user and *op* and *op*′ as the only operations, such that only U can detect execution of *op* and all users can detect execution of *op*′; let Q_1 run (T6) and Q_2 run (T7). Then the specification instances of Q_1 and Q_2 run $\texttt{exec}\ op; \texttt{chmod}\ \theta; \overline{net}\langle\rangle; net(); \texttt{success}; \overline{done}\langle\rangle$ and $\overline{net}\langle\rangle; net(); \overline{done}\langle\rangle$. These instances cannot be distinguished, since the execution of *op* can always be delayed until θ is executed, so that `success` is true and there is an output on w.

Intuitively, an execution request in NS^{d+} commits to a time bound (specified by the timestamp of the capability used for the request) within which that request must be serviced for progress; but operation requests in IS^{d+} make no such commitment. In the end, the only way out is to allow such a commitment in IS^{d+}. Therefore, we assume that:

A5 In IS^{d+}, a time bound is specified for every operation request, so that the request is dropped if it is not serviced within that time bound.

Implementing this assumption is safe. Indeed, it refines the current specification—any request with time bound T can be abstractly interpreted as an unrestricted request. Conversely, implementing (A5) is adequate; any unrestricted request can carry a time bound ∞. Further, (A5) obviates the need for the problematic (R8), since using an expired capability now has a counterpart in IS^{d+}. Accordingly, we modify the following messages in IS^{d+}.

$$\begin{array}{llllll} \text{(i)} & V & \rightarrow & D & : & (op, T) \\ \text{(ii)} & D & \rightarrow & V & : & r \qquad \text{if } \mathsf{Clk} \leq T, \Gamma \vdash_V op, \text{ and } \rho[\![op]\!] \Downarrow \rho'[\![r]\!] \end{array}$$

Now, if a capability for an operation op is produced at time T in NS^{d+}, then any use of that capability in NS^{d+} is mapped to an execution request for op in IS^{d+} with time bound T. We can then prove our main new result (see Section 7.4):

Theorem 7.2.4 (Main theorem)**.** *NS^{d+} is a secure implementation of IS^{d+} assuming* (A5)*.*

While this result is quite pleasant, we should be careful about its limitations.

- On the bright side, (A5) captures and removes the essence of the difficulties of achieving security for an implementation of dynamic access control with capabilities. Further, implementing (A5) makes a lot of sense in practice.
- On the dark side, it seems that (A5) is necessary to reduce security proofs over NS^{d+} to those over IS^{d+}. Thus, even in abstract proofs, we are forced to deal with expiry, which is an implementational artifact. (In contrast, we do not require (A5) to reduce safety proofs.)

7.2.6 Some alternatives

Let us now revisit the principles developed in Sections 7.1 and 7.2, and discuss some alternatives.

First, recall (R3), where we introduce fake capabilities to prevent premature leaks of information about the access policy Γ. What if we do not care about such leaks (and return, say, `error` in message $(2')$ in NS^s)? Then, we must allow those leaks in the specification. For example, we can make Γ public. More practically, we can add messages to IS^s that allow a user to know whether it has access to a particular operation.

Next, recall (R5) and (R6), where we introduce the midnight-shift scheme. This scheme can be relaxed to allow different capabilities to expire after different intervals; we only require that administrative operations that affect their correctness are not executed before those intervals elapse.

Finally, the implementation details in Sections 7.1 and 7.2 are far from unique. Guided by the same underlying principles, we can design capabilities in various other ways. For example, we may have an implementation in which any capability is of the form $\mathbf{mac}(\langle\langle U, op, \mathsf{Clk}\rangle, \{m, L\}_{E_{AS}}\rangle, K_{AS})$, where m is a fresh nonce and L is the access-decision predicate $\Gamma \vdash_U op$. In particular, the key $\overline{K}_{AS}$ is not required; the access decision for U and op under Γ is explicit in the content of any capability that certifies that decision, following prudent practice. What does this design buy us? Consider applications where the access decision is not a bit, but a predicate, a decision tree, or some other data structure. The design in NS^{d+} requires a different signing key for each value of the access decision. Since the number of such keys may be infinite, verification of capabilities becomes very inefficient. The design above is appropriate for such applications, and we discuss it further in [Chaudhuri, 2008a].

7.3 Definitions and proof techniques

Let us now develop formal definitions and proof techniques for security and safety; these serve as background for Section 7.4, where we outline formal proofs for security and safety of NS^{d+} under IS^{d+}.

We write a process P under a context φ as $\varphi[P]$. Contexts act as tests for behaviors; intuitively, those behaviors refute specific safety properties [Nicola and Hennessy, 1984]. Let $\preceq$ be a precongruence on processes and $\simeq$ be the associated congruence.[4] $P \preceq Q$ means that any test that is passed by P is passed by Q—in other words, "P satisfies any safety property that Q satisfies". In practice, Q is usually a process that trivially satisfies some safety property of interest; $P \preceq Q$ then implies that P satisfies that property as well.

[4] A precongruence is a preorder that is closed under arbitrary contexts. The associated congruence is the intersection of the precongruence and its inverse.

We describe an implementation as a binary relation $\mathcal{R}$ over processes, which relates specification instances to implementation instances. This relation conveniently generalizes a refinement map [Abadi and Lamport, 1991]. Next, we define full abstraction and security.

Definition 7.3.1 (Full abstraction and security (*cf.* Definition 2))**.** *An implementation $\mathcal{R}$ is fully abstract if it satisfies:*

$$\text{(PRESERVATION)} \quad \forall (P, Q) \in \mathcal{R}.\ \forall (P', Q') \in \mathcal{R}.\ P \preceq P' \ \Rightarrow\ Q \preceq Q'$$
$$\text{(REFLECTION)} \quad \forall (P, Q) \in \mathcal{R}.\ \forall (P', Q') \in \mathcal{R}.\ Q \preceq Q' \ \Rightarrow\ P \preceq P'$$

An implementation is secure if it satisfies (PRESERVATION).

(PRESERVATION) and (REFLECTION) are respectively soundness and completeness of the implementation under $\preceq$. Security only requires soundness. Intuitively, a secure implementation does not *introduce* any bad behaviors—if (P, Q) and (P', Q') are in a secure $\mathcal{R}$ and P satisfies any safety property that P' satisfies, then Q satisfies any safety property that Q' satisfies. A fully abstract implementation moreover does not *eliminate* any bad behaviors.

Any subset of a secure implementation is secure. Security implies preservation of $\simeq$. Finally, testing itself is trivially secure since $\preceq$ is a precongruence.

Proposition 7.3.2. *Let φ be any context. Then $\{(P, \varphi[P]) \mid P \in \mathcal{W}\}$ is secure for any set of processes $\mathcal{W}$.*

On the other hand, a context may eliminate some bad behaviors by acting as a test for those behaviors. A fully abstract context does not; it merely *translates* behaviors.

Definition 7.3.3 (Fully abstract context)**.** *A context φ is fully abstract for a set of processes $\mathcal{W}$ if the relation $\{(P, \varphi[P]) \mid P \in \mathcal{W}\}$ is fully abstract.*

A fully abstract context can be used as a wrapper to translate behaviors between the specification and the implementation. We define an implementation to be safe if it preserves safety properties of the specification under such a wrapper.

Definition 7.3.4 (Safety (*cf.* Definition 1))**.** *An implementation $\mathcal{R}$ is safe if there exists a fully abstract context ϕ for the set of specification instances such that $\mathcal{R}$ satisfies:*

$$(\text{INCLUSION}) \quad \forall (P, Q) \in \mathcal{R}.\ Q \preceq \phi[P]$$

Let us see why ϕ must be fully abstract in the definition. Suppose that it is not. Then for some P and P' we have $\phi[P] \preceq \phi[P']$ and $P \not\preceq P'$. Intuitively, ϕ "covers up" the behaviors of P that are not included in the behaviors of P'. Unfortunately, those behaviors may be unsafe. For example, suppose that P' is a pi calculus process [Milner, 1993] that does not contain public channels. Further, suppose that $\{P'\}$ is in fact the set of specification instances (so that any output on a public channel is unsafe). Let *net* be a public channel; suppose that $P = \overline{net}\langle\rangle; P'$ and $\phi = \bullet \mid !\,\overline{net}\langle\rangle$. Then $P \not\preceq P'$ and $\phi[P] \preceq \phi[P']$, as required. But clearly P is unsafe by our assumptions; yet $P \preceq \phi[P']$, so that by definition $\{(P', P)\}$ is safe! Thus, the definition of safety is too weak unless ϕ is required to be fully abstract.

We now present some handy proof techniques for security and safety. A direct proof of security requires mappings between subsets of $\preceq$. Those mappings may be difficult to define and manipulate. Instead, a security proof may be built modularly by showing simulations, as in a safety proof. Such a proof requires simpler mappings between processes.

Proposition 7.3.5 (Proof of security)**.** *Let ϕ and ψ be contexts such that for all $(P, Q) \in \mathcal{R}$, $Q \preceq \phi[P]$ and $P \preceq \psi[Q]$ and $\phi[\psi[Q]] \preceq Q$. Then $\mathcal{R}$ is secure.*

Proof. Let $(P, Q) \in \mathcal{R}$, $P \preceq P'$, and $(P', Q') \in \mathcal{R}$. Then $Q \preceq \phi[P] \preceq \phi[P']$ (by Proposition 7.3.2) $\preceq \phi[\psi[Q']]$ (by Proposition 7.3.2) $\preceq Q'$. ◀

Intuitively, $\mathcal{R}$ is secure if $\mathcal{R}$ and its inverse both satisfy (INCLUSION), and the witnessing contexts "cancel out". A simple technique for proving full abstraction for contexts follows as a corollary.

Corollary 7.3.6 (Proof of full abstraction for contexts)**.** *Let there be a context φ^{-1} such that for all $P \in \mathcal{W}$, $\varphi^{-1}[\varphi[P]] \simeq P$. Then φ is a fully abstract context for $\mathcal{W}$.*

Proof. Take $\phi = \varphi^{-1}$ and $\psi = \varphi$ in the proposition above to show that $\{(\varphi[P], P) \mid P \in \mathcal{W}\}$ is secure. The converse follows by Proposition 7.3.2. ◀

7.4 Formal analysis

We now outline models of NS^{d+} and IS^{d+} in the applied pi calculus [Abadi and Fournet, 2001], and present proofs of our results. We omit the treatment of other versions that appear in Sections 7.1 and 7.2; the details remain essentially the same.

We fix an equational theory Σ that includes a theory of natural numbers with symbols 0 (zero), $\_ + 1$ (successor), and $\_ \leq \_$ (less than or equal to); a theory of finite tuples with symbols $\langle \_, \_ \rangle$ (concatenate) and $\_.\_$ (project); and exactly one equation that involves the symbol **mac**, which is

$$\mathbf{msg}(\mathbf{mac}(x, y)) = x$$

Users U are identified by natural numbers k; we fix a finite subset $\mathcal{I}$ of $\mathbb{N}$ and assume that any user not identified in $\mathcal{I}$ is dishonest. File-system code and other processes are conveniently modeled by parameterized process expressions [Milner, 1993]; we define their semantics by extending the usual semantic relations of structural equivalence $\equiv$ and reduction $\rightarrow$ [Abadi and Fournet, 2001].[5] Some of those processes model functions [Milner, 1993], and we define their semantics directly as such. To distinguish parameterized processes from terms, we write all parameters in subscript.

7.4.1 Models

Below, we show applied pi calculus models for the file systems under study. Let $k \in \mathbb{N}$. Both file systems are parameterized by an access policy F, an auxiliary variable Ξ to shadow F, a time Clk, and a store ρ. For the networked file system $\mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho}$, the interface includes channels α_k, β_k, and γ_k for every k; a user identified by k may send authorization requests on α_k, execution requests on β_k, and administration requests on γ_k. For the ideal file system $\mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho}$, the interface includes channels α°_k, β°_k, and γ°_k for every k; a user identified by k may send time requests on α_k, operation requests on β°_k, and administration requests on γ°_k. Other parameterized processes, such as $\mathsf{CReq}_{k,op,M}$, TReq_M, $\mathsf{EReq}_{\kappa,M}$, $\mathsf{OReq}_{k,op,T,M}$, $\mathsf{EOk}_{L,op,M}$, and $\mathsf{AReq}_{k,\theta,M}$, denote various internal states

[5] These expressions can be readily expanded to standard applied pi-calculus processes, that implement the specified semantics up to observational equivalence.

that are reached by the file systems on receiving and processing requests. The adversary is an arbitrary evaluation context in the language [Abadi and Fournet, 2001].

We encode the relations $F \vdash_k op$, $\rho[\![op]\!] \Downarrow \rho'[\![r]\!]$, $F \vdash_k \theta$, and $\Xi[\![\theta]\!] \Downarrow \Xi'[\![r]\!]$ in the equational theory. In particular, $\mathbf{auth}(F,k,op) = \mathbf{ok}$ means that k may access op under F; $\mathbf{auth}(F,k,\theta) = \mathbf{ok}$ means that k controls θ under F; $\mathbf{exec}(L,op,\rho) = \langle N,\rho'\rangle$ means that executing op on store ρ under decision L returns N and store ρ'; and $\mathbf{exec}(L,\theta,\Xi) = \langle N,\Xi'\rangle$ means that executing θ on accumulator Ξ under decision L returns N and accumulator Ξ'. We define the following functions:

$$\mathtt{perm}_{F,k,O} = \begin{cases} \mathbf{true} & \text{if } \mathbf{auth}(F,k,O) = \mathbf{ok} \\ \mathbf{false} & \text{otherwise} \end{cases}$$

$$\mathtt{cert}_{F,k,op,\mathsf{Clk}} = \begin{cases} \mathbf{mac}(\langle k,op,\mathsf{Clk}\rangle, K_{AS}) & \text{if } \mathbf{auth}(F,k,op) = \mathbf{ok} \\ \mathbf{mac}(\langle k,op,\mathsf{Clk}\rangle, \overline{K}_{AS}) & \text{otherwise} \end{cases}$$

$$\mathtt{verif}_\kappa = \begin{cases} \mathbf{true} & \text{if } \kappa = \mathbf{mac}(\mathbf{msg}(\kappa), K_{AS}) \\ \mathbf{false} & \text{if } \kappa = \mathbf{mac}(\mathbf{msg}(\kappa), \overline{K}_{AS}) \end{cases}$$

Networked file system

(AUTHORIZATION REQUEST)
$$\mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho} \equiv \alpha_k(op,x); \mathsf{CReq}_{k,op,x} \mid \mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho}$$

(AUTHORIZATION)
$$\frac{\mathtt{cert}_{F,k,op,\mathsf{Clk}} = \kappa}{\mathsf{CReq}_{k,op,M} \mid \mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho} \to \overline{M}\langle\kappa\rangle \mid \mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho}}$$

(EXECUTION REQUEST)
$$\mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho} \equiv \beta_k(\kappa,x); \mathsf{EReq}_{\kappa,x} \mid \mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho}$$

(EXECUTION OK)
$$\frac{\mathtt{verif}_\kappa = L \qquad L \in \{\mathbf{true},\mathbf{false}\} \qquad \mathbf{msg}(\kappa) = \langle \_, op, \mathsf{Clk}\rangle}{\mathsf{EReq}_{\kappa,M} \mid \mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho} \to \mathsf{EOk}_{L,op,M} \mid \mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho}}$$

(EXECUTION)
$$\frac{\mathbf{exec}(L,op,\rho) = \langle N,\rho'\rangle}{\mathsf{EOk}_{L,op,M} \mid \mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho} \to \overline{M}\langle N\rangle \mid \mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho'}}$$

(ADMINISTRATION REQUEST)
$$\mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho} \equiv \gamma_k(\theta,x); \mathsf{AReq}_{k,\theta,x} \mid \mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho}$$

(ADMINISTRATION)
$$\frac{\mathtt{perm}_{F,k,\theta} = L \qquad \mathbf{exec}(L,\theta,\Xi) = \langle N,\Xi'\rangle}{\mathsf{AReq}_{k,\theta,M} \mid \mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho} \to \overline{M}\langle N\rangle \mid \mathsf{Nfs}_{F,\Xi',\mathsf{Clk},\rho}}$$

(TICK)

$$\mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho} \rightarrow \mathsf{Nfs}_{\Xi,\Xi,\mathsf{Clk}+1,\rho}$$

For the networked file system, the rules (AUTHORIZATION REQUEST) and (AUTHORIZATION) model behaviors in the course of receiving and processing authorization requests. The rules (EXECUTION REQUEST), (EXECUTION OK), and (EXECUTION) model behaviors in the course of receiving and processing execution requests. The rules (ADMINISTRATION REQUEST) and (ADMINISTRATION) model behaviors in the course of receiving and processing administration requests. Finally, the rule (TICK) models the internal ticking of the clock.

Ideal file system

(TIME REQUEST)

$$\mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho} \equiv \alpha_k^\circ(x); \mathsf{TReq}_x \mid \mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho}$$

(TIME)

$$\mathsf{TReq}_M \mid \mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho} \rightarrow \overline{M}\langle \mathsf{Clk} \rangle \mid \mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho}$$

(OPERATION REQUEST)

$$\mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho} \equiv \beta_k^\circ(op, T, x); \mathsf{OReq}_{k,op,T,x} \mid \mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho}$$

(EXECUTION OK)

$$\frac{\mathtt{perm}_{F,k,op} = L \qquad \mathsf{Clk} \leq T}{\mathsf{OReq}_{k,op,T,M} \mid \mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho} \rightarrow \mathsf{EOk}_{L,op,M} \mid \mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho}}$$

(EXECUTION)

$$\frac{\mathbf{exec}(L, op, \rho) = \langle N, \rho' \rangle}{\mathsf{EOk}_{L,op,M} \mid \mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho} \rightarrow \overline{M}\langle N \rangle \mid \mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho'}}$$

(ADMINISTRATION REQUEST)

$$\mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho} \equiv \gamma_k^\circ(\theta, x); \mathsf{AReq}_{k,\theta,x} \mid \mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho}$$

(ADMINISTRATION)

$$\frac{\mathtt{perm}_{F,k,\theta} = L \qquad \mathbf{exec}(L, \theta, \Xi) = \langle N, \Xi' \rangle}{\mathsf{AReq}_{k,\theta,M} \mid \mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho} \rightarrow \overline{M}\langle N \rangle \mid \mathsf{Ifs}_{F,\Xi',\mathsf{Clk},\rho}}$$

(TICK)

$$\mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho} \rightarrow \mathsf{Ifs}_{\Xi,\Xi,\mathsf{Clk}+1,\rho}$$

On the other hand, for the ideal file system, the rules (TIME REQUEST) and (TIME) model behaviors in the course of receiving and processing time requests. The rules (OPERATION REQUEST), (EXECUTION OK), and (EXECUTION) model behaviors in the

course of receiving and processing operation requests. The rules (ADMINISTRATION REQUEST) and (ADMINISTRATION) model behaviors in the course of receiving and processing administration requests. Finally, the rule (TICK) models the internal ticking of the clock.

Roughly, states of the networked file system can be related to states of the ideal file system; for example, $\mathsf{CReq}_{k,op,M}$ is related to TReq_M, $\mathsf{EReq}_{\kappa,M}$ is related to $\mathsf{OReq}_{k,op,T,M}$, and so on. Further, this relation can be lifted to code interacting with these file systems.

Formally, a networked storage system may be described as

$$NS^{d+}_{F,\rho}(C) \triangleq (\nu_{i\in\mathcal{I}}\alpha_i\beta_i\gamma_i)(C \mid (\nu K_{AS}\overline{K}_{AS})\,\mathsf{Nfs}_{F,F,0,\rho})$$

Here C is code run by honest users, F is an access policy, and ρ is a store; initially the auxiliary shadowing variable is F and the time is 0. On the other hand, an ideal storage system may be described as

$$IS^{d+}_{F,\rho}(C) \triangleq (\nu_{i\in\mathcal{I}}\alpha^\circ_i\beta^\circ_i\gamma^\circ_i)(C \mid \mathsf{Ifs}_{F,F,0,\rho})$$

Channels associated with honest users are hidden from the adversary (or context). The adversary itself is left implicit; in particular, channels associated with dishonest users may be available to the adversary.

7.4.2 Proofs

We take $\preceq$ to be the standard may-testing precongruence for applied pi calculus processes: $P \preceq Q$ if and only if for all evaluation contexts φ, whenever $\varphi[P]$ outputs on the distinguished channel *done*, so does $\varphi[Q]$. Let F and ρ range over terms that do not contain any channel or key used by the file systems under study. Let C range over code for honest users in NS^{d+}, and let $\lceil\_\rceil$ abstract such C in IS^{d+} (see below). We define

$$\mathbb{IMP} = \bigcup_{F,\rho,C} \{(IS^{d+}_{F,\rho}(\lceil C\rceil), NS^{d+}_{F,\rho}(C))\}$$

We describe $\lceil\_\rceil$ as a typed compilation $\lceil\_\rceil_\Gamma$ under an appropriate type environment Γ. Let $i \in \mathbb{N}$, and $\mathtt{Cert}(i, op)$ be the type of any capability obtained by user i for operation

op. We show a fragment of the compiler below.

(AUTHORIZATION REQUEST TO TIME REQUEST)

$$\frac{c \notin \mathtt{fn}(Q) \qquad \lceil Q \rceil_{\Gamma, x:\mathtt{Cert}(i,op)} = P \qquad \ldots}{\lceil (\nu c)\, \overline{\alpha_i}\langle op, c\rangle; c(x); Q \rceil_\Gamma = (\nu c)\, \overline{\alpha_i^\circ}\langle c\rangle; c(x); P}$$

(EXECUTION REQUEST TO OPERATION REQUEST)

$$\frac{\Gamma(x) = \mathtt{Cert}(i', op) \qquad \lceil Q \rceil_\Gamma = P \qquad \ldots}{\lceil \overline{\beta_i}\langle x, M\rangle; Q \rceil_\Gamma = \overline{\beta_{i'}^\circ}\langle op, x, M\rangle; P}$$

(ADMINISTRATION REQUEST TO ADMINISTRATION REQUEST)

$$\frac{\lceil P \rceil_\Gamma = Q \qquad \ldots}{\lceil \overline{\gamma_i}\langle adm, M\rangle; Q \rceil_\Gamma = \overline{\gamma_i^\circ}\langle adm, M\rangle; Q}$$

The omitted fragment may be built from any type system that guarantees strong secrecy of terms of type $\mathtt{Cert}(i, op)$ for any i and op [Chaudhuri, 2008a].

We then show evaluation contexts ϕ and ψ such that:

Lemma 7.4.1. *$NS^{d+}_{F,\rho}(C) \preceq \phi[IS^{d+}_{F,\rho}(\lceil C \rceil)]$, $IS^{d+}_{F,\rho}(\lceil C \rceil) \preceq \psi[NS^{d+}_{F,\rho}(C)]$, and $\phi[\psi[NS^{d+}_{F,\rho}(C)]] \preceq NS^{d+}_{F,\rho}(C)$ for any F, ρ, and C.*

More precisely, we define processes $\uparrow^{\mathrm{IS}}_{\mathrm{NS}}$ and $\uparrow^{\mathrm{NS}}_{\mathrm{IS}}$ (see below), that translate public requests from NS^{d+} to IS^{d+} and from IS^{d+} to NS^{d+}. Let $j \in \mathbb{N} \setminus \mathcal{I}$. We define

$$\begin{aligned} \phi &= (\nu_{j \in \mathbb{N}\setminus\mathcal{I}} \alpha_j^\circ \beta_j^\circ \gamma_j^\circ)\ (\bullet \mid (\nu K_?)\ \uparrow^{\mathrm{IS}}_{\mathrm{NS}}) \\ \psi &= (\nu_{j \in \mathbb{N}\setminus\mathcal{I}} \alpha_j \beta_j \gamma_j)\ (\bullet \mid\ \uparrow^{\mathrm{NS}}_{\mathrm{IS}}) \end{aligned}$$

The process $\uparrow^{\mathrm{IS}}_{\mathrm{NS}}$

(DUMMY AUTHORIZATION REQUEST)

$$\uparrow^{\mathrm{IS}}_{\mathrm{NS}} \equiv \alpha_j(op, x); (\nu m)\, \overline{\alpha_j^\circ}\langle m\rangle; m(\mathsf{Clk}); \overline{x}\langle \mathbf{mac}(\langle j, op, \mathsf{Clk}\rangle, K_?)\rangle \mid \uparrow^{\mathrm{IS}}_{\mathrm{NS}}$$

(DUMMY EXECUTION REQUEST)

$$\uparrow^{\mathrm{IS}}_{\mathrm{NS}} \equiv \beta_j(\kappa, x); \mathsf{DReq}_{\kappa,x} \mid \uparrow^{\mathrm{IS}}_{\mathrm{NS}}$$

(DUMMY OPERATION REQUEST)

$$\frac{\kappa = \mathbf{mac}(\mathbf{msg}(\kappa), K_?) \qquad \mathbf{msg}(\kappa) = \langle j, op, \mathsf{Clk}\rangle}{\mathsf{DReq}_{\kappa,M} \rightarrow \overline{\beta_j^\circ}\langle op, \mathsf{Clk}, M\rangle}$$

(DUMMY ADMINISTRATION REQUEST)

$$\uparrow^{\mathrm{IS}}_{\mathrm{NS}} \equiv \gamma_j(op, x); \overline{\gamma_j^\circ}\langle op, x\rangle \mid \uparrow^{\mathrm{IS}}_{\mathrm{NS}}$$

The process $\uparrow_{\mathrm{IS}}^{\mathrm{NS}}$

(DUMMY TIME REQ)

$$\uparrow_{\mathrm{IS}}^{\mathrm{NS}} \equiv \alpha_j^\circ(x); (\nu c)\, \overline{\alpha_j}\langle x, c\rangle; c(y); \overline{x}\langle \mathbf{msg}(y).3\rangle \mid \uparrow_{\mathrm{IS}}^{\mathrm{NS}}$$

(DUMMY OPERATION REQUEST)

$$\uparrow_{\mathrm{IS}}^{\mathrm{NS}} \equiv \beta_j^\circ(op, \tau, x); (\nu c)\, \overline{\alpha_j}\langle op, c\rangle; c(\kappa); \mathsf{DReq}_{\kappa,x} \mid \uparrow_{\mathrm{IS}}^{\mathrm{NS}}$$

(DUMMY EXECUTION REQUEST)

$$\frac{\mathbf{msg}(\kappa).3 \leq \tau \qquad j \in \mathbb{N} \setminus \mathcal{I}}{\mathsf{DReq}_{\kappa,M} \to \overline{\beta_j}\langle \kappa, M\rangle}$$

(DUMMY ADMINISTRATION REQUEST)

$$\uparrow_{\mathrm{IS}}^{\mathrm{NS}} \equiv \gamma_j^\circ(op, x); \overline{\gamma_j}\langle op, x\rangle \mid \uparrow_{\mathrm{IS}}^{\mathrm{NS}}$$

Intuitively, a networked storage system is simulated by an ideal storage system by forwarding public requests directed at Nfs to a hidden Ifs interface (via ϕ). Capabilities are simulated by terms that encode the same messages, but are signed with a dummy key $K_?$ that is secret to the wrapper. Conversely, an ideal storage system is simulated by a networked storage system by forwarding public requests directed at Ifs to a hidden Nfs interface (via ψ). Finally, a networked storage system simulates another networked storage system where requests directed at Nfs are filtered through a hidden Ifs interface before forwarding them to a hidden Nfs interface (via $\phi[\psi]$). This detour essentially forces capabilities to be acquired immediately before their use. The existence of these simulations implies that:

Theorem 7.4.2 (*cf.* Theorem 5)**.** $\mathbb{IMP}$ *is secure.*

Proof. By Lemma 7.4.1 and Proposition 7.3.5. ◀

Further, we show that:

Lemma 7.4.3. $\psi[\phi[IS_{F,\rho}^{d+}(\lceil C \rceil)]] \preceq IS_{F,\rho}^{d+}(\lceil C \rceil)$ *for any* F, ρ, *and* C.

So in fact, $\mathbb{IMP}$ is fully abstract. Finally:

Theorem 7.4.4 (*cf.* Theorem 3)**.** $\mathbb{IMP}$ *is safe.*

Proof. By Lemma 7.4.1, we already have $NS_{F,\rho}^{d+}(C) \preceq \phi[IS_{F,\rho}^{d+}(\lceil C \rceil)]$. Further, by Lemma 7.4.1, $IS_{F,\rho}^{d+}(\lceil C \rceil) \preceq \psi[NS_{F,\rho}^{d+}(C)] \preceq \psi[\phi[IS_{F,\rho}^{d+}(\lceil C \rceil)]]$. So by Lemma 7.4.3 and Corollary 7.3.6, ϕ is fully abstract (taking $\phi^{-1} = \psi$). ◀

7.4.3 Some examples of security

Let us now revisit the counterexamples in Section 7.2. We model them formally, and show that they are eliminated.

The NS^{d+} code below formalizes (T1) and (T2). Here κ is received on a fresh channel c, and later used to execute op. The result of execution is received on a fresh channel n; we assume that `success` is an appropriate predicate that can detect successful use of κ by inspecting the result.

T1 $(\nu c)\ \overline{\alpha_i}\langle op, c\rangle; c(\kappa); (\nu m)\ \overline{\gamma_i}\langle \theta, m\rangle; m(z);$
$(\nu n)\ \overline{\beta_i}\langle \kappa, n\rangle; n(x); [\texttt{success}(x)]\ \overline{w}\langle\rangle$

T2 $(\nu m)\ \overline{\gamma_i}\langle \theta, m\rangle; m(z); (\nu c)\ \overline{\alpha_i}\langle op, c\rangle; c(\kappa);$
$(\nu n)\ \overline{\beta_i}\langle \kappa, n\rangle; n(x); [\texttt{success}(x)]\ \overline{w}\langle\rangle$

This code is abstracted to the following IS^{d+} code.

T1′ $(\nu c)\ \overline{\alpha_i^\circ}\langle c\rangle; c(T); (\nu m)\ \overline{\gamma_i^\circ}\langle \theta, m\rangle; m(z);$
$(\nu n)\ \overline{\beta_i^\circ}\langle op, T, n\rangle; n(x); [\texttt{success}(x)]\ \overline{w}\langle\rangle$

T2′ $(\nu m)\ \overline{\gamma_i^\circ}\langle \theta, m\rangle; m(z); (\nu c)\ \overline{\alpha_i^\circ}\langle c\rangle; c(T);$
$(\nu n)\ \overline{\beta_i^\circ}\langle op, T, n\rangle; n(x); [\texttt{success}(x)]\ \overline{w}\langle\rangle$

Now whenever (T1) and (T2) can be distinguished, so can (T1′) and (T2′). Indeed the time bound T is the same as the timestamp in κ; so the operation request in (T1′) is dropped whenever the execution request in (T1) is dropped.

A similar argument refutes the counterexample with (T4) and (T5). Finally, recall (T6) and (T7). The following NS^{d+} code formalizes (T6).

T6 $(\nu m)\ \overline{\alpha_i}\langle op, m\rangle; m(\kappa); (\nu n)\ \overline{\beta_i}\langle \kappa, n\rangle;$
$c(); (\nu m)\ \overline{\gamma_i}\langle \theta, m\rangle; m(z); c(); n(x); [\texttt{success}(x)]\ \overline{w}\langle\rangle$

This code is abstracted to the following IS^{d+} code.

T6′ $(\nu m)\ \overline{\alpha_i^\circ}\langle m\rangle; m(T); (\nu n)\ \overline{\beta_i^\circ}\langle op, T, n\rangle;$
$c(); (\nu m)\ \overline{\gamma_i^\circ}\langle \theta, m\rangle; m(z); c(); n(x); [\texttt{success}(x)]\ \overline{w}\langle\rangle$

The following NS^{d+} context distinguishes (T6) and (T7):

$$\overline{c}\langle\rangle; \overline{\alpha_j}\langle op', m_0\rangle; m_0(\kappa_0'); \overline{\beta_j}\langle\kappa_0', n_0\rangle; n_0(x); [\texttt{failure}(x)]$$
$$\overline{\gamma_j}\langle\theta, p\rangle; \overline{\alpha_j}\langle op', m_1\rangle; m_1(\kappa_1');$$
$$\overline{\beta_j}\langle\kappa_1', n_1\rangle; n_1(x); [\texttt{success}(x)]\ \overline{c}\langle\rangle$$

But the following IS^{d+} context distinguishes (T6′) and (T7):

$$\overline{c}\langle\rangle; \overline{\alpha_j}\langle m_0\rangle; m_0(T_0'); \overline{\beta_j^\circ}\langle op', T_0', n_0\rangle; n_0(x); [\texttt{failure}(x)]$$
$$\overline{\gamma_j^\circ}\langle\theta, p\rangle; \overline{\alpha_j^\circ}\langle m_1\rangle; m_1(T_1');$$
$$\overline{\beta_j^\circ}\langle op', T_1', n_1\rangle; n_1(x); [\texttt{success}(x)]\ \overline{c}\langle\rangle$$

Chapter 8

Discussion

In Chapter 1, we point out the importance—and the unfortunate lack—of understanding the foundations of access control for security in computer systems. We propose the following thesis:

> A formal understanding of the foundations of access control for secure storage can significantly help in articulating, evaluating, and improving the security of computer systems.

We claim that through this dissertation, we successfully defend the thesis above. Indeed, we develop and apply formal techniques to specify and verify security properties of a variety of computer systems. Through this exercise, we lay the foundations of access control for security in such systems. Formal techniques play a significant role in articulating, evaluating, and improving the security of such systems. More concretely:

- We specify security properties of several file systems (such as Plutus and NASD/OSD) and operating systems (such as Windows Vista and Asbestos). These properties are typically not straightforward, since the designs of the underlying systems often balance conflicting concerns of security and practice.
- We develop new, specialized techniques to analyze these security properties—in some cases, automatically. These techniques build on a rich and mature literature on calculi, semantics, type systems, logics, and other foundations for program verification.

- Finally, applying these techniques, we discover various attacks, implementation issues, and other weaknesses in these systems, and invent methods to provably eliminate such weaknesses.

We structure our work along a research program with two complementary directions: in direction (a), we focus on the correctness of access controls in a variety of computer systems; in direction (b), we show how to exploit such access controls in proofs of information-flow properties.

The motivation for direction (a) stems from the complexity of access-control implementations in contemporary file systems and operating systems. Such complexity is often justifiable in practice; there are various underlying assumptions and guarantees in these systems, and unusual improvisations may be required to meet them. For instance, we study various cryptographic implementations of access control in the context of untrusted storage (Chapter 2) and distributed storage (Chapter 7). These implementations often combine cryptographic primitives in innovative ways, driven by practical concerns. Similarly, we study various implementations of access control with security labels in operating systems (Chapter 3). Again, these implementations relax standard models in unexpected ways, driven by practical concerns. Verifying the correctness of these implementations is typically not straightforward; in fact, formal verification helps understand the nuances of these implementations, uncover potential flaws, and articulate their precise properties.

But correct access control is seldom enough for security. The motivation for direction (b) stems from the inadequacy of formal understanding of the role of access control for security in computer systems. Indeed, without proper care, access control may turn out to be completely ineffective as a security mechanism. Showing how to achieve concrete information-flow properties through access control helps formalize the intended security guarantees of the access-control implementations in such systems. To that end, we develop special type systems that leverage access control to guarantee secrecy and integrity properties in various file system and operating system environments (Chapters 4,5, and 6). These environments correspond to the systems we study above.

Below, we outline related work and discuss our contributions in more detail.

In Chapter 2, we formally study an interesting, state-of-the-art protocol for secure file sharing on untrusted storage (in the file system Plutus), and analyze its security properties using the automatic verifier ProVerif. Our study demonstrates that protocols for secure storage are worth analyzing. Indeed, the analysis vastly improves our understanding of the above protocol; we formally specify and verify its security properties, find and patch some unknown attacks, and clarify some design details that may be relevant for other storage protocols.

Working in the Dolev-Yao model allows a deep analysis of the security consequences of some promising new features of the protocol. At the same time, some consequences remain beyond the scope of a Dolev-Yao analysis. It should be interesting to study those consequences in the computational model, perhaps using an automated verifier such as CryptoVerif [Blanchet, 2007b,a]. Unfortunately, our initial attempts at modeling the protocol in CryptoVerif indicate that the tool is presently not mature enough to prove the relevant properties. We therefore postpone that study to a point at which tools for proofs in the computational model are more developed.

Our techniques build on a huge body of work on formal methods for the verification of security protocols, *e.g.*, [Lowe, 1996; Abadi and Gordon, 1999; Abadi, 1999; Paulson, 1998; Gordon and Jeffrey, 2003a; Bodei et al., 2005; Backes et al., 2007]. We refer the reader to [Blanchet, 2008] for more information on this work, and we focus here on more closely related work on the design and verification of secure file systems.

In file systems based on the network-attached/object storage protocols (NASD, OSD) [Gobioff, 1999; Halevi et al., 2005], distributed access control is implemented on trusted storage via cryptographic capabilities. A semi-formal security analysis of this protocol appears in [Halevi et al., 2005], while we present formal models and manual security proofs for this protocol in the applied pi calculus [Chaudhuri and Abadi, 2005, 2006a; Chaudhuri, 2008b]. This material is covered in detail in Chapter 7.

Among other protocols for secure file sharing on untrusted storage, the closest to the one we study here are those behind the file systems Cepheus [Fu, 1999], SiRiUS [Goh et al., 2003], and SNAD [Miller et al., 2002]. Lazy revocation first appears in Cepheus; see [Kallahalla et al., 2007] for a summary of the origins of lazy revocation, and its limitations. Keys for reading and writing files in SiRiUS are the same as those in Plutus.

However, those keys are stored and distributed securely by the server ("in-band"), instead of being directly distributed by users ("out-of-band"). Moreover, revocation in SiRiUS is immediate, instead of lazy. In SNAD, keys for reading files are distributed in-band as in SiRiUS. However, unlike Plutus and SiRiUS, there are no keys for writing files—any user can write contents by signing those contents with its private key, and the storage server is trusted to control access to writes.

While the protocol we study partially trusts the storage server to prevent so-called rollback attacks (where contents received from the file system are not the most recent contents sent to the file system), the protocol behind the file system SUNDR [Mazières and Shasha, 2002] specifically provides a guarantee called *fork consistency*, that allows users to detect rollback attacks without trusting the storage server. The correctness of that protocol is formally proved in [Mazières and Shasha, 2002]. SUNDR does not focus on other secrecy and integrity guarantees.

Recently several schemes for key rotation have been proposed and manually proved in the computational model of security [Backes et al., 2005, 2006; Fu et al., 2006], and various alternative schemes for key distribution and signatures have been designed to eliminate public-key cryptography in this context [Naor et al., 2005]. Mechanically verifying these schemes should be interesting future work.

Finally, to guarantee stronger information-flow properties than the ones studied in this chapter (and the next), access control must be complemented by precise code analysis. Recently, several type systems have been designed for such purposes [Pistoia et al., 2007a; Zheng and Myers, 2004; Chaudhuri and Abadi, 2006b; Chaudhuri, 2006]. We cover some of these type systems in Chapters 4–6. The type system in Chapter 5 is particularly suitable for proving such properties in the presence of dynamic access control and untrusted storage.

Over the years, storage has assumed a pervasive role in modern computing, and understanding secure storage has become as important as understanding secure communication. The study of secure communication has taught us the importance of rigor in the design and analysis of protocols. This observation certainly applies to secure storage as well. As far as we know, we are the first to present an automated formal analysis of a secure storage protocol. Our approach should be fruitful for other secure

storage protocols, and we expect to see further work in this new area.

Similar ideas can be applied to the study of secure operating systems. Specifically, in Chapter 3, we present EON, a logic-programming language and tool that can be used to model and analyze dynamic access control systems. Security violations can be modeled as temporal queries in this language, and query evaluation can be used to find attacks. We show that query evaluation in EON can be reduced to decidable query satisfiability in a fragment of Datalog, and under further restrictions, to efficient query evaluation in Datalog.

We are certainly not the first to propose a dynamic logic-programming language. Related languages have been studied, for instance, in [Abadi and Manna, 1989] and [Orgun, 1996]. However, we seem to be the first to introduce a new operator to Datalog, and show that it can be reduced to existential quantification in Datalog. Such an operator allows us to express specifications that quantify over an unbounded number of processes and objects.

Our design of EON requires much care to keep query evaluation decidable. In particular, we require that any base relation that is introduced or transformed be unary—allowing dynamic binary base relations easily leads to undecidability (see the appendix). Moreover, for correctness, we require transitions to have monotonic guards, and queries to be monotonic.

These restrictions do not prevent us from modeling state-of-the-art access control systems, such as those implemented by Windows Vista and Asbestos. With unary base relations and new clauses, we can create and label processes and objects. Further, with next clauses, we can model runtime effects such as dynamic access control, communication, and taint propagation. Thus, EON turns out to be a good fit for modeling dynamic access control systems.

Further, we demonstrate that EON can verify various security properties of interest. Since our query evaluation strategy is both sound and complete, EON either finds bugs or decisively proves the absence of bugs. We expect that there are other classes of systems that can be modeled and analyzed using this approach.

Of course, it is well-known that the "safety" problem for access control models (*i.e.*, whether a given access is allowed by a given access control model) is undecidable

in general [Harrison et al., 1975; Denning, 1976]. Nevertheless, there are restricted classes of access control models for which this problem is decidable. Our work may be viewed as a step towards identifying such classes of models: we design an expressive language for dynamic access control systems, in which information-flow properties remain decidable. [Li et al., 2003] makes similar discoveries about security properties in the context of trust management languages.

Analyzing access control models with logic programs has a fairly long history. We focus here only on more closely related work. Recently [Dougherty et al., 2006] proposes a technique to study the security properties of access control policies under dynamic environments. There, a policy is specified in a fragment of Datalog without negation and recursion, and an environment is specified as a finite state machine. The composition of the policy and the environment is then analyzed by reduction to first-order logic formulae. While the authors identify some decidable problems in this framework, the lack of recursion and negation limits the expressivity of both models and queries, and it is not always possible to specify accurate finite state machines for environments. Indeed, none of the dynamic access control models studied in this paper can be analyzed in their framework.

In another line of work, [Sarna-Starosta and Stoller, 2004] studies the Security-Enhanced Linux (SELinux) system in Prolog. The SELinux system enforces access control policies written in SELinux's policy language. The authors describe a tool called PAL that translates such policies into logic programs, and analyzes them by query evaluation. [Naldurg et al., 2006] studies both SELinux and Windows XP configurations in Datalog in a tool called Netra. Unlike PAL, Netra is both sound and complete, since query evaluation is decidable in Datalog (while in Prolog is not). However, neither tool can find vulnerabilities that are exploited dynamically. Some of these concerns are addressed by later work on policy analysis for administrative role-based access control [Stoller et al., 2007], which is similar in spirit to ours.

Recently, [Becker et al., 2007] proposes a language called SecPAL that can express authorization policies and fine-grained delegation control in decentralized systems. Their specifications are compiled down to programs in Datalog, much as in our work. Since Datalog is a subset of EON, it follows that EON is at least as expressive as Sec-

PAL. On the other hand, it is not clear whether SecPAL is as expressive as EON; the former is tailored to express authorization and delegation policies, while the latter remains largely agnostic in that respect. An interesting aspect of SecPAL is that it allows negations within queries. While EON allows such negations, the fragment discussed in Section 3.2.4 does not. However, we have checked that this restriction can be lifted from that fragment without compromising correctness or efficiency. More recently, [Gurevich and Neeman, 2008] proposes a distributed-knowledge authorization language called DKAL, based on existential fixed-point logic, that is more expressive than SecPAL. We leave the comparison of DKAL and EON as future work.

Other relevant work includes, of course, ProVerif [Blanchet, 2001b], which we use to study Plutus in Chapter 2. ProVerif is sound but not complete; it may not terminate on queries, and it may also fail to prove or disprove queries. Indeed, while ProVerif can handle Windows Vista's access control model, it does not terminate on our model of Asbestos's webserver. In sum, EON is less expressive than ProVerif; but for models that satisfy our restrictions, EON guarantees sound and complete results.

Going further, security properties can be enforced by a combination of static and dynamic checks. In Chapter 4, we investigate the interplay of secrecy types with access-control checks in the setting of a fairly standard file system. Our goal is to enable the analysis of programs that use the file system; the details of the file-system implementation can then be refined while preserving secrecy properties. The main novelty of this work is a principled integration of static and dynamic checks for security, in the spirit of hybrid typechecking. This idea is further explored in Chapter 6.

Our type system extends previous ones so as to deal with access checks. It is particularly close to an intermediate type system developed in the study of group creation [Cardelli et al., 2005]. It goes beyond that type system by introducing secrecy types for file-system constructs, in such a way that dynamic access checks, together with static scoping, play a role in guaranteeing secrecy of file contents.

Mobility regions [Kirli, 2001] for distributed functional programs are similar to groups as presented here. Yet another calculus uses group creation to specify discretionary access policies [Bugliesi et al., 2004b]; the type system controls the flow of values according to those policies. Ideas similar to group creation also appear in a calculus

for role-based access control [Braghin et al., 2004]. However, it is not clear how to apply these approaches to our setting. For example, in [Bugliesi et al., 2004b] it is possible to specify access controls statically, and verify that those access controls are enforced at run time; instead, in our setting it is possible to declare secrecy intentions, and verify that those intentions are enforced via appropriate access controls at run time.

As in most access control systems, and as in the study of group creation, we do not define secrecy as the absence of certain flows of information (that is, as some sort of non-interference property). Rather, we define secrecy as the impossibility of certain communication events (such as sending a message that contains a particular sensitive value). One may however imagine many possible variants, dealing with other concepts of secrecy, and also with authenticity properties beyond the ones verifiable in our system (*e.g.*, [Gordon and Jeffrey, 2003b]). We leave the investigation of such variants for further work.

The recent literature also includes a few calculi with constructs for authorization. In particular, [Fournet et al., 2005] develops a spi calculus with authorization assertions; a type system for that calculus serves for checking generalized correspondence assertions, rather than secrecy properties.

Several other works emphasize distribution. In the language KLAIM [de Nicola et al., 1998], a type system checks that processes have been granted the necessary permissions to perform operations at specified localities [Nicola et al., 2000]. Another type system for a distributed pi-calculus ensures that agents cannot access the resources of a system without first being granted the capability to do so [Hennessy and Riely, 1998]. [Bugliesi et al., 2004a] explores access-control types for the calculus of boxed ambients with a typing relation similar in form to ours, but without dynamic access control—access control is specified in terms of static secrecy levels.

Yet another research direction addresses access control in languages such as Java. [Banerjee and Naumann, 2003] examines the use of access control for secure information flow in that setting. [Pottier et al., 2005] develops type systems that guarantee the success of access checks. In contrast, our type system does not guarantee the success of access checks; indeed, type soundness depends on the failure of some of those checks. This approach of combining access control with types for security is itself close to hy-

brid typechecking [Flanagan, 2006], where dynamic checks are used where possible to as required to complement static checks.

More generally, in Chapter 5, we show that access control can soundly enforce dynamic specifications, *i.e.*, specifications that can vary at run time. Such specifications are quite useful, since they can rely on accurate, run-time security assumptions, instead of "worst-case", static security assumptions. Not surprisingly, they allow finer analyses than static specifications. For example, they allow us to reason about the secrecy of file contents that are written after revocation of public access; such reasoning is not possible if the contents of the file are statically assumed to be either public or secret. The possibility of enforcing such dynamic specifications seems to capture the essence of access control.

In this context, we implement low-level dynamic access controls in an existing object language to make it suitable as a core calculus for studying security properties of concurrent, stateful services, such as those implemented by network objects [Birrell et al., 1993]. We then show a typing approach for verifying high-level intentions on service manipulation in the resulting language. The type system allows dynamic specifications for services, and crucially relies on corresponding low-level dynamic access controls provided by the language runtime to verify those specifications. This combination helps in developing precise security analyses for shared services that are used under varying assumptions over time.

Along these lines, one body of work studies the enforcement of policies specified as security automata [Schneider, 2000; Hamlen et al., 2006]. Yet another studies systems with declassification, *i.e.*, conservative relaxation of secrecy assumptions at run time [Myers et al., 2004]. There is also some recent work on compromised secrets [Gordon and Jeffrey, 2005; Haack and Jeffrey, 2005] in the context of network protocols. In comparison, our analyses apply more generally to varying assumptions at run time. Perhaps closest to our work are analyses developed for dynamic access control in languages with locality and migration [Hennessy et al., 2003; Gorla and Pugliese, 2003]. Similar ideas appear in a type system for noninterference that allows the use of dynamic security labels [Zheng and Myers, 2004].

Our approach of combining access control with types for security is also fruitful for formalizing the security designs of operating systems. Specifically, in Chapter 6, we formalize DFI—a multi-level integrity property based on explicit flows—and present a type system that can efficiently enforce DFI in a language that simulates Windows Vista's security environment.

By design, our analysis is control-insensitive—it does not track implicit flows. In many applications, implicit flows are of serious concern. It remains possible to extend our analysis to account for such flows, following the ideas of [Volpano et al., 1996; Zdancewic and Myers, 2001; Myers et al., 2004; Li and Zdancewic, 2005]. However, we believe that it is more practical to enforce a weaker property like DFI at the level of an operating system, and enforce stronger, control-sensitive properties like noninterference at the level of the application, with specific assumptions.

Our core security calculus is simplified, although we believe that we include all key aspects that require conceptual modeling for reasoning about DFI. In particular, we model threads, mutable references, binaries, and data and code pointers; other features of x86 binaries, such as recursion, control flow, and parameterized procedures, can be encoded in the core calculus. We also model all details of Windows Vista that are relevant for mandatory integrity control with dynamic labels. On the other hand, we do not model details such as discretionary access control, file virtualization, and secure authorization of privilege escalation [Howard and LeBlanc, 2007], which can improve the precision of our analysis. Building a typechecker that works at the level of x86 binaries and handles all details of Windows Vista requires much more work. At the same time, we believe that our analysis can be applied to more concrete programming models by translation.

Our work is closely related to [Tse and Zdancewic, 2004] and [Zheng and Myers, 2004] on noninterference in lambda calculi with dynamic security levels. While [Tse and Zdancewic, 2004] does not consider mutable references in their language, it is possible to encode the sequential fragment of our calculus in the language of [Zheng and Myers, 2004]; however, well-typed programs in that fragment that rely on access control for DFI do not remain well-typed via such an encoding. Specifically, any restrictive access check for integrity in the presence of dynamically changing labels seems to

let the adversary influence trusted computations in that system, violating noninterference [Zheng, 2007].

Noninterference is known to be problematic for concurrent languages. In this context, [Zdancewic and Myers, 2003] studies the notion of observational determinism; [Abadi, 1999; Hennessy and Riely, 2002] study information flow using testing equivalence; and [Boudol and Castellani, 2002; Honda and Yoshida, 2002] use stronger notions based on observational equivalence. Sophisticated techniques that involve linearity, race analysis, behavior types, and liveness analysis also appear in the literature [Honda and Yoshida, 2002; Zdancewic and Myers, 2003; Hennessy and Riely, 2002; Kobayashi, 2005]. While most of these techniques are developed in the setting of the pi calculus, other works consider distributed, multi-threaded, and higher-order settings to study mobile code [Hennessy et al., 2005; Russo and Sabelfeld, 2006; Barthe et al., 2007; Yoshida, 2004].

DFI being a safety property [Alpern and Schneider, 1985] gets around some of the difficulties posed by noninterference. A related approach guides the design of the operating systems Asbestos [Efstathopoulos et al., 2005] and HiStar [Zeldovich et al., 2006], and dates back to the Clark-Wilson approach to security in commercial computer systems [Clark and Wilson, 1987; Shankar et al., 2006]. In comparison with generic models of trace-based integrity that appear in protocol analysis, such as correspondence assertions [Gordon and Jeffrey, 2003b; Fournet et al., 2005], our integrity model is far more specialized; as a consequence, our type system requires far less annotations than type systems for proving correspondence assertions.

Our definition of DFI relies on an operational semantics based on explicit substitution. Explicit substitution, as introduced in [Abadi et al., 1990], has been primarily applied to study the correctness of abstract machines for programming languages (whose semantics rely on substitution as a rather inefficient meta-operation), and in proof environments. It also appears in the applied pi calculus [Abadi and Fournet, 2001] to facilitate an elegant formulation of indistinguishability for security analysis. However, we seem to be the first to use explicit substitutions to track explicit flows in a concurrent language. Previously, dependency analysis [Lévy, 1978; Abadi et al., 1996] has been applied to information-flow analysis [Abadi et al., 1999; Pottier and Conchon,

2000; Zdancewic and Myers, 2002]. These analyses track stronger dependencies than those induced by explicit flows; in particular, the dependencies are sensitive to control flows. In contrast, the use of explicit substitutions to track explicit flows seems rather obvious and appropriate in hindsight. We believe that this technique should be useful in other contexts as well.

Our analysis manifests a genuine interplay between static typing and dynamic access control for runtime protection. We seem to be the first to study this interaction in a concurrent system with dynamic labels for multi-level integrity. This approach of combining static and dynamic protection mechanisms is reflected in previous work on typing, *e.g.*, for noninterference in a Java-like language with stack inspection and other extensions [Banerjee and Naumann, 2003; Pistoia et al., 2007b], for noninterference in lambda calculi with runtime principals and dynamic labels [Tse and Zdancewic, 2004; Zheng and Myers, 2004], and for secrecy in concurrent storage calculi with discretionary access control mechanisms, as covered in Chapters 4 and 5. A verification technique based on this approach is developed by Flanagan [Flanagan, 2006] for a lambda calculus with arbitrary base refinement types. In these studies and ours, dynamic checks complement static analysis where possible or as required, so that safety violations that are not caught statically are always caught at runtime. Moreover, static typing sometimes subsumes certain dynamic checks (as in our analysis), suggesting sound runtime optimizations. This approach is reflected in previous work on static access control [Hennessy and Riely, 2002; Pottier et al., 2005; Hoshina et al., 2001].

In most real-world systems, striking the right balance between security and practice is a delicate task that is never far from controversy. It is reassuring to discover that perhaps, in the future, such a balance can be enforced formally in a contemporary operating system.

Finally, in Chapter 7, we present a comprehensive analysis of the problem of implementing distributed access control with capabilities. This culminates a line of work that we begin in [Chaudhuri and Abadi, 2005] and continue in [Chaudhuri and Abadi, 2006a]. In [Chaudhuri and Abadi, 2005], we show how to securely implement static access policies with capabilities; in [Chaudhuri and Abadi, 2006a], we present a safe (but not secure) implementation of dynamic access policies in that setting. In this chapter,

we explain those results in new light. In particular, we reveal the several pitfalls that any such design must care about for correctness, while discovering interesting special cases that allow simpler implementations. Further, we systematically analyze the difficulties that arise for security in the case of dynamic access policies. Our analysis leads us to develop variants of the implementation in [Chaudhuri and Abadi, 2006a] that we can prove secure with appropriate assumptions. Further, guided by our analysis of access control, in [Chaudhuri, 2008a] we show how to automatically derive secure distributed implementations of other stateful computations. This approach is reminiscent of secure program partitioning [Zdancewic et al., 2002], and deserves further investigation.

Access control for networked storage has been studied in lesser detail in [Gobioff et al., 1997] using belief logics, and in [Halevi et al., 2005] using universal composability [Canetti, 2001]. The techniques used in this chapter are similar to those used previously for secure implementation of channel abstractions [Abadi et al., 1998] and authentication primitives [Abadi et al., 2000], and for studying the equivalence of communication patterns in distributed query systems [Maffeis, 2006]. These techniques rely on programming languages concepts, including testing equivalence [Nicola and Hennessy, 1984] and full abstraction [Milner, 1977; Abadi, 1998]. A huge body of such techniques have been developed for formal specification and verification of systems.

We do not consider access control for untrusted storage [Kallahalla et al., 2003]; a detailed treatment already appears in Chapter 2. In file systems for untrusted storage, such as Plutus, files are cryptographically secured before storage, and their access keys are managed and shared by users. As such, untrusted storage is quite similar to public communication, and standard techniques for secure communication on public networks apply for secure storage in this setting. Related work in that area includes formal analysis of protocols for secure file sharing on untrusted storage [Mazières and Shasha, 2002; Blanchet and Chaudhuri, 2008] (some of which is covered in Chapter 2), as well as correctness proofs for the cryptographic techniques involved in such protocols [Backes et al., 2005; Fu et al., 2006; Backes et al., 2006].

Appendix

Appendix A

Extended models of Plutus

In this appendix, we list a more detailed model of Plutus in ProVerif, that takes into account server-verified writes (and PATCH).

```
1  free net, newgroup, revoke, rkeyreq, wkeyreq, corrupt.                    (* public channels *)

2  private fun rprivchannel/1.                                               (* private channels *)
3  private fun wprivchannel/1.
4  private fun tokenprivchannel/1.
5  private fun writefs/0.
6  private fun readfs/0.

7  let processOwr =
8    new seed1; new seed2;                                                   (* create owner's RSA key pair *)
9    let ownerpubkey = (e(seed1, seed2), N(seed1)) in
10   let ownerprivkey = (d(seed1, seed2), N(seed1)) in
11   out(net, ownerpubkey);                                                  (* publish owner's RSA public key *)
12   (
13   ! in(net, (= newgroup, initreaders, initwriters));                      (* receive a new group creation request;
          initreaders and initwriters are the initial lists of allowed readers and writers, respectively *)
14     new g;                                                                (* create the new group g *)
15     out(net, g);                                                          (* publish the group name g *)
16     new currentstate;                                                     (* create a private channel for the current state for group g *)
17     (
```

```
18    ( new initt;                                              (* create initial token *)
19      out(tokenprivchannel(g), (hash(initt), succ(zero)));
                                                  (* send initial token's hash to the server *)
20      event istoken(initt, g, zero);        (* assert that initt is the token for group g at version 0 *)
21      new initlk;                                             (* create initial lk *)
22      new seed3; let initsk = (d(seed3, initlk), N(seed3)) in          (* generate initial sk *)
23      out(currentstate, (zero, initreaders, initwriters, initlk, initsk, initt))
                                          (* store state for version 0 on channel currentstate *)
24    )
25    |                                        (* Next, we move from version 0 to version 1 *)
26    ( in(net, (= revoke, = g, newreaders, newwriters));     (* receive a revoke request for group g;
                        newreaders and newwriters are the new lists of allowed readers and writers *)
27      in(currentstate, (= zero, oldreaders, oldwriters, oldlk, oldsk, oldt));
                                                          (* read state for version 0 *)
28      new seed3;                                          (* choose new RSA seed *)
29      new newt;                                               (* create new token *)
30      in(tokenprivchannel(g), (hashx, = zero));
31      out(tokenprivchannel(g), (hash(newt), succ(zero)));
                                                  (* send new token's hash to the server *)
32      event istoken(newt, g, succ(zero));(* assert that newt is the token for group g at version 1 *)
33      let newlk = exp(oldlk, ownerprivkey) in                 (* wind old lk to new lk *)
34      let newsk = (d(seed3, newlk), N(seed3)) in              (* generate new sk *)
35      out(currentstate, (succ(zero), newreaders, newwriters, newlk, newsk, newt))
                                          (* store state for version 1 on channel currentstate *)
36    )
37    | ... |                          (* Similarly, we move from version 1 to version 2, and so on *)
38    (
39    ! in(net, (= rkeyreq, r, = g));             (* receive read key request for reader r and group g *)
40      in(currentstate, (v, readers, writers, lk, sk, t));          (* get the current state *)
41      out(currentstate, (v, readers, writers, lk, sk, t));
42      if member(r, readers) then                    (* check that the reader r is allowed *)
43       ( event isreader(r, g, v);              (* assert that r is a reader for group g and version v *)
44         out(rprivchannel(r), (g, v, lk, ownerpubkey)) )    (* send lk and owner's public key to r *)
45    )
46    |
```

```
47      (
48      ! in(net, (= wkeyreq, w, = g));          (* receive write key request for writer w and group g *)
49        in(currentstate, (v, readers, writers, lk, sk, t));                (* get the current state *)
50        out(currentstate, (v, readers, writers, lk, sk, t));
51        if member(w, writers) then                    (* check that the writer w is allowed *)
52         ( let (_, n) = sk in
53           let sn = exp(hash((n, g, v)), ownerprivkey) in                  (* sign the modulus *)
54           event iswriter(w, g, v);          (* assert that w is a writer for group g and version v *)
55           out(wprivchannel(w), (g, v, lk, sk, sn, t)))
                                                  (* send lk, sk, signed modulus, and token to w *)
56      )
57     )
58    ).

59  let processWtr =
60   ! in(net, (w, g));                                    (* initiate a writer w for group g *)
61     out(net, (wkeyreq, w, g));                                 (* send write key request *)
62     in(wprivchannel(w), (= g, v, lk, sk, sn, t));       (* obtain lk, sk, signed modulus, and token *)
63     (
64      ( new m;                                                   (* create data to write *)
65        let encx = enc(m, lk) in                                              (* encrypt *)
66        let sencx = exp(hash(encx), sk) in                                    (* sign *)
67        event puts(w, m, g, v); (* assert that data m has been written by w for group g at version v *)
68        let (_, n) = sk in
69        out(writefs, (t, (g, v, n, sn, encx, sencx)))          (* send content with token to the server *)
70      )
71      |
72      ( in(net, = (corrupt, w));                              (* receive corrupt request for w *)
73        event corrupt(w, g, v);          (* assert that w has been corrupted for group g at version v *)
74        out(net, (lk, sk, sn, t))                          (* leak lk, sk, signed modulus, and token *)
75      )
76     ).

77  let processAdvWtr =                    (* allow the adversary to send data to the server *)
78   ! in(net, (t, content));
79     out(writefs, (t, content)).
```

```
80 let processServer =
81  ! in(net, g);                                                    (* initiate a group g *)
82   (
83    (
84     ! in(tokenprivchannel(g), (hashx, vx));  (* receive a hash of the current token from g's owner *)
85       out(tokenprivchannel(g), (hashx, vx))          (* carry the hash of the current token for g *)
86    )
87   |
88    (
89     ! in(writefs, (t, content));                              (* receive content sent with token t *)
90       out(net, content);                                                  (* leak the content *)
91       in(tokenprivchannel(g), (hashx, vx));(* get the hash of the token at (the current version) vx *)
92       out(tokenprivchannel(g), (hashx, vx));
93       if hash(t) = hashx then          (* check that t hashes to the same string as the token at vx *)
94       event authwrite(g, vx, t);      (* assert that content sent with token t is verified for g at vx *)
95       ! out(readfs, (content, g, vx))                  (* write server-verified content for g at vx *)
96    )
97   ).

98 let processRdr =
99  ! in(net, (r, g));                                          (* initiate a reader r for group g *)
100   out(net, (rkeyreq, r, g));                                          (* send read key request *)
101   in(rprivchannel(r), (= g, v, lk, ownerpubkey));              (* obtain lk and owner's public key *)
102   (
103    ( in(readfs, ((= g, vx, n, sn, encx, sencx), = g, v′));
                                    (* obtain header and server-verified content from the server *)
104      if hash((n, g, v)) = exp(sn, ownerpubkey) then                (* verify signature in header *)
105      ( if (v, vx) = (succ(zero), zero) then
106        ( let lk = exp(lk, ownerpubkey) in                                    (* unwind lk *)
107          let vk = (genExp(n, lk), n) in                                      (* derive vk *)
108          if hash(encx) = exp(sencx, vk) then                  (* verify signature of encryption*)
109          let x = dec(encx, lk) in                                          (* decrypt to obtain data *)
110          event gets(r, x, g, vx, v′)
              (* assert that reader r read data x for group g and version vx, from content written at v′ *)
111        )
```

```
112         ...
113       )
114     |
115     ( in(net, = (corrupt, r));                    (* receive corrupt request for r *)
116       event corrupt(r, g, v);          (* assert that r has been corrupted for group g at version v *)
117       out(net, lk)                                                    (* leak lk *)
118     )
119   ).

120 process processOwr | processWtr | processAdvWtr | processServer | processRdr
                                                      (* put all processes together *)
```

Appendix B

Supplementary material on EON

In this appendix, we provide supplementary material for Chapter 3. Specifically, in Appendix B.1, we review an algorithm for deciding satisfiability in Datalog, on which we rely in Section 3. In Section B.2, we outline an undecidability proof for query evaluation in an extension of EON with dynamic binary base relations.

B.1 Satisfiability in Datalog

We review a decision procedure for satisfiability of safe stratified Datalog programs with unary base relations. This procedure is due to Halevy *et al.* [Halevy et al., 2001]. By translating EON into this particular Datalog subset (as shown in Section 3.2), we arrive at a decision procedure for queries on EON programs.

Intuitively, Halevy *et al.* show that safe stratified Datalog programs with unary base relations can be translated to equivalent first-order logic formulae over unary relations, whose satisfiability is decidable. In fact, we show that due to the pleasant structure of those formulae, their satisfiability can be further reduced to satisfiability of simple boolean logic formulae.

We begin by recalling some key data structures from [Halevy et al., 2001]. A *region* $\mathcal{R}(x)$ for a variable x is of the form

$$\mathcal{B}(x), !\mathcal{B}'(x)$$

$\mathcal{R}(x)$ is said to be unsatisfiable if $\mathcal{B} \cap \mathcal{B}' \neq \varnothing$. Two regions for x are said to

be equivalent if they are the same or are both unsatisfiable. Intuitively, a region $\mathcal{R}(x) = \mathcal{B}(x), !\mathcal{B}'(x)$ is a membership constraint on x in the set $\bigcap \mathcal{B} \setminus \bigcup \mathcal{B}'$.

Next, a *generalized tuple* $\mathcal{G}(\overrightarrow{x})$ for $\overrightarrow{x} = x_1, \ldots, x_n$ is of the form

$$\begin{aligned} \mathcal{R}_1(x_1), &\quad \ldots, &\quad \mathcal{R}_n(x_n), \\ \exists y.\mathcal{R}'_1(y), &\quad \ldots, &\quad \exists y.\mathcal{R}'_m(y), \\ \nexists z.\mathcal{R}''_1(z), &\quad \ldots, &\quad \nexists z.\mathcal{R}''_k(z) \end{aligned}$$

This generalized tuple is interpreted as the first-order logic formula

$$\begin{aligned} \mathcal{R}_1(x_1) \wedge &\quad \ldots \wedge &\quad \mathcal{R}_n(x_n) \wedge \\ \exists y.\mathcal{R}'_1(y) \wedge &\quad \ldots \wedge &\quad \exists y.\mathcal{R}'_m(y) \wedge \\ \nexists z.\mathcal{R}''_1(z) \wedge &\quad \ldots \wedge &\quad \nexists z.\mathcal{R}''_k(z) \end{aligned}$$

Two generalized tuples for $\overrightarrow{x}$ are said to be the same if they have equivalent regions for $\overrightarrow{x}$, and equivalent sets of regions for both the positive as well as the negative existential variables. Intuitively, a generalized tuple $\mathcal{G}(\overrightarrow{x})$ is a constraint that involves multiple variables $\overrightarrow{x}$, yet is expressed entirely via region constraints on individual variables. (In other words, variables do not constrain each other in $\mathcal{G}(\overrightarrow{x})$.) As shown below, every positive literal of the form $S(\overrightarrow{x})$ in a safe stratified program with unary base relations can be expressed as a set of generalized tuples for $\overrightarrow{x}$, called the *extension* of $S(\overrightarrow{x})$, so that $S(\overrightarrow{x})$ can be interpreted as the disjunction of the interpretations of those generalized tuples.

B.1.1 Computing extensions

We mention a few elementary operations that involve straightforward applications of boolean laws. The negation of a generalized tuple for $\overrightarrow{x}$ yields a set of generalized tuples for $\overrightarrow{x}$. The conjunction of a generalized tuple for $\overrightarrow{x}$ and a generalized tuple for $\overrightarrow{x}'$ yields a generalized tuple for $\overrightarrow{x} \cup \overrightarrow{x}'$. The conjunction of sets of generalized tuples is the cross product of those sets. Negation of a set of generalized tuples is the conjunction of their negations. The projection of a generalized tuple for $\overrightarrow{x}$ on $\overrightarrow{x}' \subseteq \overrightarrow{x}$ is a generalized tuple for $\overrightarrow{x}'$; projection is trivially generalized to a set of generalized tuples.

Extensions for each literal of the form $S(\overrightarrow{x})$, where S is a relation in the program, can be computed by topologically sorting the strongly connected components in the dependency graph on relations, and visiting these components from the bottom up. For each base relation B, let the extension of $B(x)$ be $\{\mathcal{G}_B(x)\}$, where $\mathcal{G}_B(x) = B(x)$. The extensions of all other positive literals in the program are initialized to $\varnothing$.

Let $\mathbb{S}$ be the component that is currently under visit. Suppose that $\mathcal{C}$ is a clause with $S(\overrightarrow{x})$ in its head for some $S \in \mathbb{S}$. For each literal in the body of $\mathcal{C}$, we compute the negation of its extension; we then take the union of the resulting sets of generalized tuples, and project the result to yield a set of generalized tuples for $\overrightarrow{x}$. Finally, we take the union of this set with the extension of $S(\overrightarrow{x})$. Clauses such as $\mathcal{C}$ are iteratively considered to compute the extension of each literal of the form $S(\overrightarrow{x})$ in $\mathbb{S}$, till fixpoint.

B.1.2 Satisfiability of generalized tuples

It is easy to see that satisfiability of generalized tuples is decidable. Indeed, the interpretation of $\mathcal{G}(\overrightarrow{x})$ is a FOL formula $F_{\mathcal{G}}(\overrightarrow{x})$ over unary relations, and satisfiability of such formulae is decidable. In this case, a simple procedure exists given the structure of generalized tuples. If $\mathcal{G}(\overrightarrow{x})$ is

$$\begin{aligned} \mathcal{R}_1(x_1), &\quad \ldots, &\quad \mathcal{R}_n(x_n), \\ \exists y.\mathcal{R}'_1(y), &\quad \ldots, &\quad \exists y.\mathcal{R}'_m(y), \\ \nexists z.\mathcal{R}''_1(z), &\quad \ldots, &\quad \nexists z.\mathcal{R}''_k(z) \end{aligned}$$

then $\exists \overrightarrow{x}.F_{\mathcal{G}}(\overrightarrow{x})$ is equivalent to

$$\begin{aligned} \exists x_1.\mathcal{R}_1(x_1) \wedge &\quad \ldots \wedge &\quad \exists x_n.\mathcal{R}_n(x_n) \wedge \\ \exists y.\mathcal{R}'_1(y) \wedge &\quad \ldots \wedge &\quad \exists y.\mathcal{R}'_m(y) \wedge \\ \nexists z.\mathcal{R}''_1(z) \wedge &\quad \ldots \wedge &\quad \nexists z.\mathcal{R}''_k(z) \end{aligned}$$

which is further equivalent to

$$\begin{aligned} \exists z.\ (\mathcal{R}_1(z) \quad &\wedge \quad !\mathcal{R}''_1(z) \wedge \ldots \wedge\ !\mathcal{R}''_k(z)) \wedge \\ &\ldots \end{aligned}$$

$$\begin{aligned}
\exists z.\,(\mathcal{R}_n(z) \quad &\wedge \quad !\mathcal{R}''_1(z) \wedge \ldots \wedge\ !\mathcal{R}''_k(z)) \wedge \\
\exists z.\,(\mathcal{R}'_1(z) \quad &\wedge \quad !\mathcal{R}''_1(z) \wedge \ldots \wedge\ !\mathcal{R}''_k(z)) \wedge \\
&\ldots \\
\exists z.\,(\mathcal{R}'_m(z) \quad &\wedge \quad !\mathcal{R}''_1(z) \wedge \ldots \wedge\ !\mathcal{R}''_k(z)) \wedge
\end{aligned}$$

We claim that these are $m + n$ satisfiability problems in boolean propositional logic. Indeed, we can interpret unary base relations as propositional variables, rewrite regions $\mathcal{R}(x) = \mathcal{B}(x), !\mathcal{B}(x)$ as formulae $F_{\mathcal{R}} = \bigwedge \mathcal{B} \wedge\ ! \bigvee \mathcal{B}'$, and rewrite

$$\exists z.\,(\mathcal{R}(z) \wedge\ !\mathcal{R}''_1(z) \wedge \ldots \wedge\ !\mathcal{R}''_k(z))$$

as

$$F_{\mathcal{R}} \wedge\ !F_{\mathcal{R}''_1} \wedge \ldots \wedge\ !F_{\mathcal{R}''_k}$$

Finally, a literal $S(\overrightarrow{x})$ is satisfiable if and only if its extension contains a satisfiable generalized tuple.

Note that the test mentioned in [Halevy et al., 2001] for satisfiability of generalized tuples is obviously incomplete; there, it is stated that a generalized tuple is unsatisfiable if and only if the region for a negative existential variable is the *same* as the region for a non-negative existential variable. However, a generalized tuple is unsatisfiable even if the union of the regions for several negative existential variables is a superset of the region for a non-negative existential variable.

B.2 Undecidable query evaluation in an extension of EON

Recall that in EON, we restrict the relations in the head of new and next clauses to be unary. We show that lifting this restriction leads to a language in which query evaluation is undecidable.

Specifically, consider an extension of EON with the following sort of clauses:

$$\mathsf{new}\ B(\_, y)\ \text{:-}\ R(y).$$

The semantics of the language is extended as follows.

$$\frac{\mathsf{new}\ B(\_, y)\ \text{:-}\ R(y).\ \in \mathbb{P} \qquad R(c') \in \mathcal{I}(\widehat{\mathbb{P}}, \mathbb{DB}) \qquad c \text{ is a fresh constant}}{\mathbb{DB} \xrightarrow{\mathbb{P}} \mathbb{DB} \cup \{B(c, c')\}}$$

Basically, we show that we can encode an arbitrary instance of Post's correspondence problem (PCP) in this language. Let $\bullet$ denote concatenation over bitstrings. The PCP problem is:

> Given two finite lists of bitstrings $a_1, \ldots, a_m$ and $b_1, \ldots, b_m$, is there a non-empty sequence of indices $i_1, \ldots, i_k$ ($1 \leq i_k \leq m$) such that $a_{i_1} \bullet \ldots \bullet a_{i_k} = b_{i_1} \bullet \ldots \bullet b_{i_k}$?

The PCP problem is a classic undecidable problem. Given an arbitrary instance $a_1, \ldots, a_m$ and $b_1, \ldots, b_m$ of the PCP problem, we now construct a program in the extension of EON that encodes that instance. We denote the condition $x = 0 \bullet y$ by `Zero`(x, y), and the condition $x = 1 \bullet y$ by `One`(x, y). (Note that `Zero` and `One` are binary base relations.)

```
new Empty.
new Zero(_,y) :- Bitstring(y).
new One(_,y) :- Bitstring(y).

Bitstring(x) :- Empty(x).
Bitstring(x) :- Zero(x,y).
Bitstring(x) :- One(x,y).
```

Next, we define the derived relations `Concat_aj` and `Concat_bj` for each a_j and b_j ($1 \leq j \leq m$), as follows. Say $a_j = 0100$. Then we include the following clause:

```
Concat_aj(x,y) :-
     Zero(x,y1),One(y1,y2),Zero(y2,y3),Zero(y3,y).
```

Intuitively, `Concat_aj`(x, y) denotes the condition $x = a_j \bullet y$. Finally, we define the relation `Gen` as follows, by including a clause for each a_j and b_j ($1 \leq j \leq m$):

```
Gen(x,y) :-
  Concat_aj(x,x1),Concat_bj(y,y1),Gen(x1,y1).
...

Gen(x,y) :- Empty(x), Empty(y).
```

Now $\texttt{Gen}(x, y)$ is true if and only if there is a sequence $i_1, \cdots, i_k$ $(1 \leq j \leq m)$ such that $x = a_{i_1} \bullet \ldots \bullet a_{i_k}$ and $y = b_{i_1} \bullet \ldots \bullet b_{i_k}$. Now, if query evaluation in this language is decidable, evaluating the query $\texttt{Gen}(x, y)$ solves the given PCP instance (contradiction).

Appendix C

Implementing a typed file system in conc$\grave{\imath}$

In this appendix, we implement a typed file system in **conc** $\grave{\imath}$. More precisely, we lift the type system of Chapter 4 to a type-directed compilation whose target language is **conc** $\grave{\imath}$. We prove that the compiled programs are well-typed and can simulate their sources. The soundness of the former type system follows from the soundness of the type system for **conc** $\grave{\imath}$.

C.1 Type-directed compilation

The compilation judgements are of the form $\Gamma \vdash P \rightsquigarrow a$. The rules for these judgments extend the ones for well-typed processes $\Gamma \vdash P$ in Chapter 4. The specified compiler is type-directed, in the sense that the compilation of processes is guided by their typing derivations. In the target language, pairs can be constructed by the syntax (u, v), and destructed by the syntax $\mathsf{split}\ p\ \mathsf{as}\ (x, y);\ a$ (see below).

Typing rules $\Gamma \vdash P \rightsquigarrow a$

(PROC OUT)

$$\frac{\Gamma \vdash M : L[T] \qquad \Gamma \vdash N : T \qquad \Gamma \vdash P \rightsquigarrow a}{\Gamma \vdash \overline{M}\langle N\rangle; P \rightsquigarrow \left(\begin{array}{l} \mathsf{split}\ M\ \mathsf{as}\ (\_, y); \\ \mathsf{let}\ \_ = y\langle N\rangle\ \mathsf{in} \\ a \end{array}\right)}$$

(PROC IN)

$$\frac{\Gamma \vdash M : L[T] \qquad \Gamma, x : T \vdash P \rightsquigarrow a}{\Gamma \vdash M(x); P \rightsquigarrow \left(\begin{array}{l} \mathsf{split}\ M\ \mathsf{as}\ (y, \_); \\ \mathsf{let}\ x = y\langle \bot\rangle\ \mathsf{in} \\ a \end{array}\right)}$$

(PROC OUT $\bot$)

$$\frac{\Gamma \vdash M : \bot \qquad \Gamma \vdash N : \bot \qquad \Gamma \vdash P \rightsquigarrow a}{\Gamma \vdash \overline{M}\langle N\rangle; P \rightsquigarrow \left(\begin{array}{l}\mathsf{split}\ M\ \mathsf{as}\ (\_, y);\\ \mathsf{let}\ \_ = y\langle N\rangle\ \mathsf{in}\\ a\end{array}\right) \Rsh \left(\begin{array}{l}\mathsf{let}\ \_ = \mathsf{net!}\langle N\rangle\ \mathsf{in}\\ a\end{array}\right)}$$

(PROC IN $\bot$)

$$\frac{\Gamma \vdash M : \bot \qquad \Gamma, x : \bot \vdash P \rightsquigarrow a}{\Gamma \vdash M(x); P \rightsquigarrow \left(\begin{array}{l}\mathsf{split}\ M\ \mathsf{as}\ (y, \_);\\ \mathsf{let}\ x = y\langle \bot\rangle\ \mathsf{in}\\ a\end{array}\right) \Rsh \left(\begin{array}{l}\mathsf{let}\ x = \mathsf{net?}\langle \bot\rangle\ \mathsf{in}\\ a\end{array}\right)}$$

(PROC NEW CHAN)

$$\frac{\Gamma, x : L[\![T]\!] \vdash P \rightsquigarrow a}{\Gamma \vdash (\nu x : L[T])\, P \rightsquigarrow \left(\begin{array}{l}(\nu n : [\mathtt{Receive}^+ : (\bot)\ [\![T]\!], \mathtt{Send}^+ : ([\![T]\!])\bot]^\top)\\ (\nu m? : (\langle\bot\rangle [\![T]\!])^{\top L})\\ (\nu m! : (\langle [\![T]\!]\rangle \bot)^{\top L})\\ n \mapsto m?m![\,\mathtt{Receive} \Mapsto\ (\_).\, \mathtt{Receive}(\bot),\\ \qquad\qquad \mathtt{Send} \Mapsto\ (M').\, \mathtt{Receive} \Mapsto (\_).\, M'] \Rsh\\ \mathsf{let}\ x = (\widehat{m?}, \widehat{m!})\ \mathsf{in}\\ a\end{array}\right)}$$

(PROC PAR)

$$\frac{\Gamma \vdash P \rightsquigarrow a \qquad \Gamma \vdash Q \rightsquigarrow b}{\Gamma \vdash P \mid Q \rightsquigarrow a \Rsh b \Rsh \bot}$$

(PROC NIL)

$$\Gamma \vdash 0 \rightsquigarrow \bot$$

(PROC REPL)

$$\frac{\Gamma \vdash P \rightsquigarrow a}{\Gamma \vdash !P \rightsquigarrow \left(\begin{array}{l}(\nu n : \bot)\\ (\nu m : \bot)\\ n \mapsto m[\ell \Mapsto (\_)\ a \Rsh \ell(\bot)] \Rsh\\ \widehat{m}\langle\_\rangle\ \Rsh\ \bot\end{array}\right)}$$

(PROC `read`)

$$\frac{\Gamma \vdash x : \mathsf{Req}_L.\mathtt{read} \qquad \Gamma \vdash y : (\_\{T\}\#\_(L_r, \_),\ \_[T']) \qquad \Gamma \vdash P \rightsquigarrow a \qquad L \sqsupset \bot \qquad L \sqsupseteq L_r\ \Rightarrow\ T \leq T'}{\Gamma \vdash \overline{x}\langle y\rangle; P \rightsquigarrow \mathsf{let}\ \_ = x\langle y\rangle\ \mathsf{in}\ a}$$

(PROC `write`)

$$\frac{\Gamma \vdash x : \mathsf{Req}_L.\mathtt{write} \qquad \Gamma \vdash y : (\_\{T\}\#\_(\_, L_w),\ T') \qquad \Gamma \vdash P \rightsquigarrow a \qquad L \sqsupset \bot \qquad L \sqsupseteq L_w\ \Rightarrow\ T' \leq T}{\Gamma \vdash \overline{x}\langle y\rangle; P \rightsquigarrow \mathsf{let}\ \_ = x\langle y\rangle\ \mathsf{in}\ a}$$

(PROC `chmod`)

$$\frac{\Gamma \vdash x : \mathsf{Req}_L.\texttt{chmod} \qquad \Gamma \vdash y : (\_\{\_\}\#L_o(L_r, L_w),\ (L'_r,\ L'_w)) \qquad \Gamma \vdash P \rightsquigarrow a \qquad L \sqsupset \bot \qquad L \sqsupseteq L_o \ \Rightarrow\ L'_r \sqsupseteq L_r, L'_w \sqsupseteq L_w}{\Gamma \vdash \overline{x}\langle y\rangle; P \rightsquigarrow \text{let } \_ = x\langle y\rangle \text{ in } a}$$

(PROC `new`)

$$\frac{\Gamma \vdash x : \mathsf{Req}_L.\texttt{new} \qquad \Gamma \vdash c : \_[\_\{\_\}\#L(\_,\_)] \qquad \Gamma \vdash P \rightsquigarrow a \qquad L \sqsupset \bot}{\Gamma \vdash \overline{x}\langle c\rangle; P \rightsquigarrow \text{let } \_ = x\langle c\rangle \text{ in } a}$$

For completeness, we show a type-directed encoding of pairs in **conc**$\grave{\imath}$. The derived typing rules for pair construction and destruction are standard, and we omit them.

A pair type is an object type with `left` and `right` fields.

$$(S, T) \stackrel{\text{def}}{=} [\texttt{left}^+ : (\bot)S, \texttt{right}^+ : (\bot)T]^{\|S\|\sqcup\|T\|}$$

A pair is constructed by creating an object with `left` and `right` fields, populating the fields with its left and right projections, and returning the name of the object.

$$\frac{\Gamma \vdash u : S \qquad \Gamma \vdash v : T}{\Gamma \vdash (u, v) \stackrel{\text{def}}{=} \left(\begin{array}{l} (\nu n : (S, T)) \\ (\nu m_l : (\langle\bot\rangle S)^{\|S\|\sqcup\|T\|\ \|S\|}) \\ (\nu m_r : (\langle\bot\rangle T)^{\|S\|\sqcup\|T\|\ \|T\|}) \\ n \mapsto m_l, m_r[\texttt{left}^+ : (\_)u, \texttt{right}^+ : (\_)v]\ \restriction \\ n \end{array}\right)}$$

A pair is destructed by binding new method names to the `left` and `right` fields of the underlying object, and reading the fields by calling those names.

$$\frac{\Gamma \vdash p : (S, T)}{\Gamma \vdash \mathsf{split}\ p\ \mathsf{as}\ (x, y);\ a \stackrel{\text{def}}{=} \left(\begin{array}{l} (\nu m_l : (\langle\bot\rangle S)^{\|S\|\ \|S\|\sqcup\|T\|}) \\ (\nu m_r : (\langle\bot\rangle T)^{\|T\|\ \|S\|\sqcup\|T\|}) \\ p \leftarrow\!\shortmid m_l, m_r[] \\ \text{let } x = \grave{m}_l\langle\bot\rangle \text{ in let } y = \grave{m}_r\langle\bot\rangle \text{ in } a \end{array}\right)}$$

$$\frac{\Gamma \vdash p : \bot}{\Gamma \vdash \mathsf{split}\ p\ \mathsf{as}\ (x, y);\ a \stackrel{\text{def}}{=} \left(\begin{array}{l} (\nu m_l : \bot) \\ (\nu m_r : \bot) \\ p \leftarrowtail m_l, m_r[] \\ \text{let } x = \tilde{m_l}\langle\bot\rangle \text{ in let } y = \tilde{m_r}\langle\bot\rangle \text{ in } a \end{array}\right)}$$

A pi-calculus channel is compiled to a pair of indirections for receiving and sending on that channel. In particular, a new channel is compiled by creating a new object with `Receive` and `Send` methods, and binding the channel name to a pair of indirections to those methods.

$$[\![L[T]]\!] \stackrel{\text{def}}{=} ((\langle\bot\rangle [\![T]\!])^L, (\langle [\![T]\!] \rangle \bot)^L)$$

Further, we assume that a public channel net is available in the context, and compile that channel similarly. In the sequel, $\bullet$ represents a hole in a context.

$$net\mathcal{E} \stackrel{\text{def}}{=} \left(\begin{array}{l} (\nu n : [\mathtt{Receive}^+ : (\bot)\ \bot, \mathtt{Send}^+ : (\bot)\bot]^\top) \\ (\nu m? : (\langle\bot\rangle\bot)^{\bot\top}) \\ (\nu m! : (\langle\bot\rangle\bot)^{\bot\top}) \\ n \mapsto m?m![\ \mathtt{Receive} \Rightarrow\ (\_).\ \mathtt{Receive}(\bot), \\ \qquad\qquad \mathtt{Send} \Rightarrow\ (y).\ \mathtt{Receive} \Rightarrow (\_).\ y \\ \qquad\qquad\qquad\qquad \Rsh\ \beta_\bot.\mathtt{Read}\langle y\rangle \\ \qquad\qquad\qquad\qquad \Rsh\ \beta_\bot.\mathtt{Write}\langle y\rangle \\ \qquad\qquad\qquad\qquad \Rsh\ \beta_\bot.\mathtt{Chmod}\langle y\rangle \\ \qquad\qquad\qquad\qquad \Rsh\ \beta_\bot.\mathtt{New}\langle y\rangle]\ \Rsh \\ \text{let } (\mathsf{net?}, \mathsf{net!}) = (\tilde{m?}, \tilde{m!}) \text{ in} \\ \bullet \end{array}\right)$$

The compiler forwards any message sent on a public channel to net; conversely, any message expected from a public channel can be received on net. Such channels include, *e.g.*, the channels $\beta_\bot.\varkappa$, which are not compiled as usual pi-calculus channels; instead, any message sent on net is internally forwarded to the compiled channels $\beta_\bot.\varkappa$.

A file is compiled to the indirection of a file object name. In particular, a new file is compiled by creating a new object with `owner`, `acl`, `read`, `write`, and `chmod` methods.

$$[\![L\{T\}\#L_o(L_r, L_w)]\!] \quad \stackrel{\text{def}}{=} \quad [\mathtt{owner}^+ : (\bot)\ L_o,$$

$$\begin{array}{l}
\texttt{acl}^+ : (\bot)\ (L_r, L_w), \\
\texttt{read}^+ : (\bot)\ [\![T]\!], \\
\texttt{write}^+ : ([\![T]\!])\ \bot, \\
\texttt{chmod}^+ : ((L_r, L_w))\ \bot \quad]^L
\end{array}$$

Further, the indirections for the methods of the file object are recorded in a system table (via sysSnd), indexed by the file.

$$\mathit{newfile}_{L,T,L_o,L_r,L_w} \stackrel{\text{def}}{=} \left(\begin{array}{l}
(\nu n : [\texttt{owner}^+ : (\bot)\ L_o, \\
\qquad \texttt{acl}^+ : (\bot)\ (L_r, L_w), \\
\qquad \texttt{read}^+ : (\bot)\ T, \\
\qquad \texttt{write}^+ : (T)\ \bot, \\
\qquad \texttt{chmod}^+ : ((L_r, L_w))\ \bot \quad]^{\top L}) \\
(\nu m_o : (\langle \bot \rangle\ L)^{\top \bot}) \\
(\nu m_a : (\langle \bot \rangle\ (L_r, L_w))^{\top \bot}) \\
(\nu m_r : (\langle \bot \rangle\ T)^{\top (L \sqcup L_r)}) \\
(\nu m_w : (\langle T \rangle\ \bot)^{\top (L \sqcup L_w)}) \\
(\nu m_c : (\langle L_r, L_w \rangle\ \bot)^{\top (L \sqcup L_o)}) \\
n \mapsto m_o, m_a, m_r, m_w, m_c[\\
\quad \texttt{owner} \Rrightarrow (_).\ L_o, \\
\quad \texttt{acl} \Rrightarrow (_).\ (\top, \top), \\
\quad \texttt{read} \Rrightarrow (_).\ \texttt{read}(\bot), \\
\quad \texttt{write} \Rrightarrow (y).\ \texttt{read} \Rrightarrow (_)y, \\
\quad \texttt{chmod} \Rrightarrow (y).\ \texttt{acl} \Rrightarrow (_)y \quad]\ \Gamma \\
\text{let } f = \tilde{n}\ \text{ in} \\
\quad \widetilde{\mathsf{sysSnd}}\langle (f, (m_o, m_a, m_r, m_w, m_c)) \rangle\ \Gamma \\
\quad f
\end{array}\right)$$

A request channel $\beta_L.\varkappa$ is compiled to an indirection that, when called, simulates the behavior of the file system on receiving a message on $\beta_L.\varkappa$. The compilation strategy for $L \neq \bot$ is slightly different from that for $L = \bot$. (Different methods must be called to account for differences in the required typing invariants. We defer a more detailed discussion of this problem.) We begin by assuming that $L \neq \bot$.

$$
\beta_L\mathcal{E} \stackrel{\text{def}}{=}
\left(
\begin{array}{l}
(\nu n : [\texttt{Read}^+ : \forall Y, Z, Y_o, Y_r, Y_w, X, Z' \\
\quad ((Y\{Z\}\#Y_o(Y_r, Y_w), X[Z']) \mid L \le Y_r \Rightarrow Z \le Z') \\
\quad\quad \bot, \\
\quad \texttt{Write}^+ : \forall Y, Z, Y_o, Y_r, Y_w, Z' \\
\quad ((Y\{Z\}\#Y_o(Y_r, Y_w), Z') \mid L \le Y_r \Rightarrow Z' \le Z) \\
\quad\quad \bot, \\
\quad \texttt{Chmod}^+ : \forall Y, Z, Y_o, Y_r, Y_w, Y_r', Y_w' \\
\quad ((Y\{Z\}\#Y_o(Y_r, Y_w), (Y_r', Y_w')) \mid L \le Y_o \ \Rightarrow\ Y_r' \le Y_r, Y_w' \le Y_w) \\
\quad\quad \bot, \\
\quad \texttt{New}^+ : \forall X, Y, Z, Y_r, Y_w \\
\quad (X[Y\{Z\}\#L(Y_r, Y_w)] \mid L \le Y_r \sqcup Y_w, \\
\quad\quad\quad\quad\quad\quad \bot \le Y \sqcup Y_r \Rightarrow Z \le \bot, \bot \le Y \sqcup Y_w \Rightarrow \bot \le Z) \\
\quad\quad Y\{Z\}\#L(Y_r, Y_w) \quad]^\top) \\
(\nu \mathsf{Read}_L : (\ldots)^{\top L})\ (\nu \mathsf{Write}_L : (\ldots)^{\top L})\ (\nu \mathsf{Chmod}_L : (\ldots)^{\top L})\ (\nu \mathsf{New}_L : (\ldots)^{\top L}) \\
n \mapsto \mathsf{Read}_L, \mathsf{Write}_L, \mathsf{Chmod}_L, \mathsf{New}_L[\\
\quad \texttt{Read} \Rightarrow (y)\ \text{let}\ (f, c) = y\ \text{in} \\
\quad\quad \text{split}\ \ \widehat{\mathsf{sysRcv}}\langle f\rangle\ \text{as}\ (\_, v_a, v_r, \_, \_);\ \text{let}\ (g_r, \_) = v_a\langle\bot\rangle\ \text{in} \\
\quad\quad \text{if}\ L \sqsupseteq g_r\ \text{then let}\ x = v_r\langle\_\rangle\ \text{in split}\ c\ \text{as}\ (\_, c!);\ c!\langle x\rangle, \\
\quad \texttt{Write} \Rightarrow (y)\ \text{let}\ (f, M') = y\ \text{in} \\
\quad\quad \text{split}\ \ \widehat{\mathsf{sysRcv}}\langle f\rangle\ \text{as}\ (\_, v_a, \_, v_w, \_);\ \text{let}\ (\_, g_w) = v_a\langle\bot\rangle\ \text{in} \\
\quad\quad \text{if}\ L \sqsupseteq g_w\ \text{then}\ v_w\langle M'\rangle, \\
\quad \texttt{Chmod} \Rightarrow (y)\ \text{let}\ (f, (g_r', g_w')) = y\ \text{in} \\
\quad\quad \text{split}\ \ \widehat{\mathsf{sysRcv}}\langle f\rangle\ \text{as}\ (v_o, \_, \_, \_, v_c);\ \text{let}\ g_o = v_o\langle\bot\rangle\ \text{in} \\
\quad\quad \text{if}\ L \sqsupseteq g_o\ \text{then}\ v_c\langle(g_r', g_w')\rangle, \\
\quad \texttt{New} \Rightarrow (c)\ \text{let}\ f = \mathit{newfile}_{Y,Z,L,Y_r,Y_w}\ \text{in} \\
\quad\quad \text{split}\ c\ \text{as}\ (\_, c!);\ c!\langle f\rangle \quad]\ \Rsh \\
\text{split}\ (\widehat{\mathsf{Read}}_L, \widehat{\mathsf{Write}}_L, \widehat{\mathsf{Chmod}}_L, \widehat{\mathsf{New}}_L)\ \text{as}\ (\beta_L.\texttt{read}, \beta_L.\texttt{write}, \beta_L.\texttt{chmod}, \beta_L.\texttt{new}); \\
\bullet
\end{array}
\right)
$$

For $\varkappa = \texttt{New}$, the file system creates a new file as above. For $\varkappa \in \{\texttt{Read}, \texttt{Write}, \texttt{Chmod}\}$, the file system proceeds as follows. First, it retrieves the indirections for the methods of relevant file object from the system table (via sysRcv). These indirections can be used to look up the owners, readers, or writers of the file, and to

`Read`, `Write`, or `Chmod` the file. Next, it compares L with the relevant levels to check access to the file. Finally, it calls the relevant method $\varkappa$ for the file.

The types for the compiled request channels are sophisticated. First, they are polymorphic in the types of files that may be created or accessed by those channels. Next, they carry type constraints that must be guaranteed when sending requests on those channels. Conversely, these constraints can be assumed when typing the service of those requests by the file system.

$$\begin{aligned}
\llbracket \mathsf{Req}_L.\mathtt{Read} \rrbracket &\stackrel{\text{def}}{=} (\forall Y, Z, Y_o, Y_r, Y_w, X, Z' \\
&\quad \langle (\llbracket Y\{Z\}\#Y_o(Y_r, Y_w) \rrbracket, \llbracket X[Z'] \rrbracket) \mid L \leq Y_r \Rightarrow Z \leq Z' \rangle \\
&\quad \bot \quad)^L \\
\llbracket \mathsf{Req}_L.\mathtt{Write} \rrbracket &\stackrel{\text{def}}{=} (\forall Y, Z, Y_o, Y_r, Y_w, Z' \\
&\quad \langle (\llbracket Y\{Z\}\#Y_o(Y_r, Y_w) \rrbracket, Z') \mid L \leq Y_r \Rightarrow Z' \leq Z \rangle \\
&\quad \bot \quad)^L \\
\llbracket \mathsf{Req}_L.\mathtt{Chmod} \rrbracket &\stackrel{\text{def}}{=} (\forall Y, Z, Y_o, Y_r, Y_w, Y_r', Y_w' \\
&\quad \langle (\llbracket Y\{Z\}\#Y_o(Y_r, Y_w) \rrbracket, (Y_r', Y_w')) \mid L \leq Y_o \;\Rightarrow\; Y_r' \leq Y_r, Y_w' \leq Y_w \rangle \\
&\quad \bot \quad)^L \\
\llbracket \mathsf{Req}_L.\mathtt{New} \rrbracket &\stackrel{\text{def}}{=} (\forall X, Y, Z, Y_r, Y_w \\
&\quad \langle \llbracket X[Y\{Z\}\#L(Y_r, Y_w)] \rrbracket \mid L \leq Y_r \sqcup Y_w, \\
&\qquad\qquad \bot \leq Y \sqcup Y_r \Rightarrow Z \leq \bot, \bot \leq Y \sqcup Y_w \Rightarrow \bot \leq Z \rangle \\
&\quad \bot \quad)^L
\end{aligned}$$

The compiled types respectively specify the following requirements:

- To read a file, if L may be a reader of the file, then the content type of that file must be a subtype of the message type of the channel on which the content of that file may be sent (*cf.* (PROC `Read`)).
- To write a file, if L may be a writer of the file, then the type of the sent content must be a subtype of the content type of that file (*cf.* (PROC `Write`)).
- To chmod a file, if L is an owner of the file, then the sent access-control list must

respect the bound on access-control lists of that file (*cf.* (PROC `Chmod`)).

- To create a new file, L must be the level of owners of the file, and the type of the file must be well-formed (*cf.* (TYP FILE)).

We now assume that $L = \bot$. Some differences in the compilation strategy arise in types, and in the manner of retrieving information from the system table (via $\mathsf{sysRcv}\bot$).

$$
\beta_\bot \mathcal{E} \stackrel{\text{def}}{=} \left(\begin{array}{l}
(\nu n : [\mathtt{Read}^+ : (\bot)\bot, \\
\qquad \mathtt{Write}^+ : (\bot)\bot, \\
\qquad \mathtt{Chmod}^+ : (\bot)\bot, \\
\qquad \mathtt{New}^+ : (\bot)\bot \quad]^\top) \\
(\nu \mathsf{Read}_\bot : (\ldots)^{\top\bot})\ (\nu \mathsf{Write}_\bot : (\ldots)^{\top\bot})\ (\nu \mathsf{Chmod}_\bot : (\ldots)^{\top\bot})\ (\nu \mathsf{New}_\bot : (\ldots)^{\top\bot}) \\
n \mapsto \mathsf{Read}_\bot, \mathsf{Write}_\bot, \mathsf{Chmod}_\bot, \mathsf{New}_\bot[\\
\quad \mathtt{Read} \Rightarrow (y)\ \mathsf{let}\ (f, c) = y\ \mathsf{in} \\
\qquad \mathsf{split}\quad \widehat{\mathsf{sysRcv}}\bot\langle f\rangle\ \mathsf{as}\ (\_, v_a, v_r, \_, \_);\ \mathsf{let}\ (g_r, \_) = v_a\langle\bot\rangle\ \mathsf{in} \\
\qquad \mathsf{if}\ \bot \sqsupseteq g_r\ \mathsf{then}\ \mathsf{let}\ x = v_r\langle\_\rangle\ \mathsf{in}\ \mathsf{split}\ c\ \mathsf{as}\ (\_, c!);\ c!\langle x\rangle, \\
\quad \mathtt{Write} \Rightarrow (y)\ \mathsf{let}\ (f, M') = y\ \mathsf{in} \\
\qquad \mathsf{split}\quad \widehat{\mathsf{sysRcv}}\bot\langle f\rangle\ \mathsf{as}\ (\_, v_a, \_, v_w, \_);\ \mathsf{let}\ (\_, g_w) = v_a\langle\bot\rangle\ \mathsf{in} \\
\qquad \mathsf{if}\ \bot \sqsupseteq g_w\ \mathsf{then}\ v_w\langle M'\rangle, \\
\quad \mathtt{Chmod} \Rightarrow (y)\ \mathsf{let}\ (f, (g'_r, g'_w)) = y\ \mathsf{in} \\
\qquad \mathsf{split}\quad \widehat{\mathsf{sysRcv}}\bot\langle f\rangle\ \mathsf{as}\ (v_o, \_, \_, \_, v_c);\ \mathsf{let}\ g_o = v_o\langle\bot\rangle\ \mathsf{in} \\
\qquad \mathsf{if}\ \bot \sqsupseteq g_o\ \mathsf{then}\ v_c\langle(g'_r, g'_w)\rangle, \\
\quad \mathtt{New} \Rightarrow (c)\ \mathsf{let}\ f = \mathit{newfile}_{\bot,\bot,\bot,\bot,\bot}\ \mathsf{in} \\
\qquad \mathsf{split}\ c\ \mathsf{as}\ (\_, c!);\ c!\langle f\rangle \quad]\ \Rsh \\
\mathsf{split}\ (\widehat{\mathsf{Read}}_\bot, \widehat{\mathsf{Write}}_\bot, \widehat{\mathsf{Chmod}}_\bot, \widehat{\mathsf{New}}_\bot)\ \mathsf{as}\ (\beta_\bot.\mathtt{read}, \beta_\bot.\mathtt{write}, \beta_\bot.\mathtt{chmod}, \beta_\bot.\mathtt{new}); \\
\bullet
\end{array}\right)
$$

$$
[\![\, \mathsf{Req}_\bot.\varkappa \,]\!] \stackrel{\text{def}}{=} \bot
$$

Finally, we code the system table as a file-indexed list of records.

$$
sys\mathcal{E} = \left(\begin{array}{l}
(\nu n : [\texttt{find}^+ : (\bot) \\
\qquad (\forall Y, Z, Y_o, Y_r, Y_w \\
\qquad\ \langle (Y\{Z\}\#Y_o(Y_r, Y_w),\ (\langle\bot\rangle(T_{Y,Z,Y_o,Y_r,Y_w} \mid \mathcal{C}_{Y,Z,Y_o,Y_r,Y_w}))^{\top\top})\rangle \\
\qquad\ \bot)^{\top}, \\
\quad \texttt{find}\bot^+ : (\bot) \\
\qquad\ (\langle (\bot,\ (\langle\bot\rangle(\exists Z, Y_o, Y_r, Y_w)(T_{\bot,Z,Y_o,Y_r,Y_w} \mid \mathcal{C}_{\bot,Z,Y_o,Y_r,Y_w}))^{\top\top})\rangle \\
\qquad\ \bot)^{\top}, \\
\quad \texttt{sysRcv}^+ : \forall Y, Z, Y_o, Y_r, Y_w \\
\qquad\ (Y\{Z\}\#Y_o(Y_r, Y_w)) \\
\qquad\ (T_{Y,Z,Y_o,Y_r,Y_w}) \mid \mathcal{C}_{Y,Z,Y_o,Y_r,Y_w}), \\
\quad \texttt{sysRcv}\bot^+ : (\bot) \\
\qquad\ (\exists Z, Y_o, Y_r, Y_w)\ (T_{\bot,Z,Y_o,Y_r,Y_w} \mid \mathcal{C}_{\bot,Z,Y_o,Y_r,Y_w}), \\
\quad \texttt{sysSnd}^+ : \forall Y, Z, Y_o, Y_r, Y_w \\
\qquad\ ((Y\{Z\}\#Y_o(Y_r, Y_w),\ T_{Y,Z,Y_o,Y_r,Y_w}) \mid \mathcal{C}_{Y,Z,Y_o,Y_r,Y_w}) \\
\qquad\ \bot\ \ ]) \\
(\nu \mathsf{find} : \ldots)\ (\nu \mathsf{find}\bot : \ldots)\ (\nu \mathsf{sysRcv} : \ldots)\ (\nu \mathsf{sysRcv}\bot : \ldots)\ (\nu \mathsf{sysSnd} : \ldots) \\
n \mapsto \mathsf{find}, \mathsf{find}\bot, \mathsf{sysRcv}, \mathsf{sysRcv}\bot, \mathsf{sysSnd}[\\
\quad \texttt{find} \Rrightarrow (\_)(\nu m : \ldots)\ \ \tilde{m}, \\
\quad \texttt{find}\bot \Rrightarrow (\_)(\nu m : \ldots)\ \ \tilde{m}, \\
\quad \texttt{sysRcv} \Rrightarrow (f)\ \mathsf{let}\ x = \texttt{find}()\ \mathsf{in} \\
\qquad\ (\nu c : (\langle\bot\rangle(T_{Y,Z,Y_o,Y_r,Y_w} \mid \mathcal{C}_{Y,Z,Y_o,Y_r,Y_w}))^{\top\top}) \\
\qquad\ x\langle (f, c)\rangle\ \Rsh\ \ \tilde{c}\langle 1\rangle, \\
\quad \texttt{sysRcv}\bot \Rrightarrow (f)\ \mathsf{let}\ x = \texttt{find}\bot()\ \mathsf{in} \\
\qquad\ (\nu c : (\langle\bot\rangle(\exists Z, Y_o, Y_r, Y_w)(T_{\bot,Z,Y_o,Y_r,Y_w} \mid \mathcal{C}_{\bot,Z,Y_o,Y_r,Y_w}))^{\top\top}) \\
\qquad\ x\langle f, c\rangle\ \Rsh\ \ \tilde{c}\langle 1\rangle, \\
\quad \texttt{sysSnd} \Rrightarrow (z)\ \mathsf{split}\ z\ \mathsf{as}\ (f, ms); \\
\qquad\ \mathsf{let}\ x = \texttt{find}()\ \mathsf{in} \\
\qquad\ (\nu p : \ldots)\ (\nu m : \ldots)\ p \mapsto m[\ell \Rrightarrow (z)\ \mathsf{split}\ z\ \mathsf{as}\ (y, c); \\
\qquad\quad (\mathsf{if}\ y = f\ \mathsf{then}\ (\nu q : \ldots)\ q \mapsto c[\ell \Rrightarrow (\_)ms])\ \Rsh\ x\langle (y, c)\rangle] \\
\qquad\ \Rsh\ \texttt{find} \Rrightarrow ()\ \ \tilde{m} \\
\qquad \Rsh\ \ \mathsf{let}\ x = \texttt{find}\bot()\ \mathsf{in} \\
\qquad\ (\nu p : \ldots)\ (\nu m : \ldots)\ p \mapsto m[\ell \Rrightarrow (z)\ \mathsf{split}\ z\ \mathsf{as}\ (y, c); \\
\qquad\quad (\mathsf{if}\ y = f\ \mathsf{then}\ (\nu q : \ldots)\ q \mapsto c[\ell \Rrightarrow (\_)ms])\ \Rsh\ x\langle (y, c)\rangle] \\
\qquad\ \Rsh\ \texttt{find}\bot \Rrightarrow ()\ \ \tilde{m}\ \ \ ]\ \Rsh \\
\bullet
\end{array}\right)
$$

In the code, we use the following abbreviations in the types of records, and the associated type constraints.

$$T_{Y,Z,Y_o,Y_r,Y_w} \stackrel{\text{def}}{=} ((\langle\bot\rangle Y_o)^{Y},\ (\langle\bot\rangle(Y_r, Y_w))^{Y},\ (\langle\bot\rangle Z)^{Y\sqcup Y_r},\ (\langle Z\rangle\bot)^{Y\sqcup Y_w},\ (\langle(Y_r, Y_w)\rangle\bot)^{Y\sqcup Y_o})$$
$$\mathcal{C}_{Y,Z,Y_o,Y_r,Y_w} \stackrel{\text{def}}{=} Y_o \leq Y_r \sqcup Y_w,\ \bot \leq Y \sqcup Y_r \Rightarrow Z \leq \bot,\ \bot \leq Y \sqcup Y_w \Rightarrow \bot \leq Z$$

The fields `find` and `find`$\bot$ contain similar values that have different types. Specifically, the fields contain indirections to methods that search lists of records with files as indices, and return the associated records at specified addresses. The methods `sysRcv` and `sysRcv`$\bot$, which call those indirections, have the same behaviors but have different types. The method `sysSnd` updates the fields `find` and `find`$\bot$ in parallel, to search for new records.

We assume the type constraint $\mathcal{C}_{Y,Z,Y_o,Y_r,Y_w}$ on the type $Y\{Z\}\#Y_o(Y_r, Y_w)$ of any file that has a record in the system table. The type of `sysSnd` specifies this assumption. Conversely, we guarantee that type contraint on the type T_{Y,Z,Y_o,Y_r,Y_w} of the record associated with such a file. The types of `sysRcv` and `sysRcv`$\bot$ specify this guarantee. This guarantee is necessary to type the compiled code for the request channels $\beta_L.\varkappa$.

Why do `sysRcv` and `sysRcv`$\bot$ have different types? Recall that the method `sysRcv` is called by compiled code for request channels $\beta_L.\varkappa$, where $L \neq \bot$. The compiled types for such channels are explicit about the types of the files that are passed to `sysRcv`; thus, the type contraints guaranteed by `sysRcv` apply to those types. On the other hand, the method `sysRcv`$\bot$ is called by compiled code for request channels $\beta_\bot.\varkappa$. The compiled types for such channels are not explicit about the types of the files that are passed to `sysRcv`$\bot$—those types are assumed to be $\bot$. Thus, the type contraints guaranteed by `sysRcv`$\bot$ apply to some unknown (existential) file types.

This difference in the required typing invariants for $L \neq \bot$ and $L = \bot$ forces the somewhat awkward dichotomy in the compilation strategies for $L \neq \bot$ and $L = \bot$. We should point out that this dichotomy can be eliminated if we allow a method to have multiple types.

Finally, if a well-typed process P compiles under $\rightsquigarrow$ to the **conc** $\tilde{\imath}$ program a, then

we define the output of the compiler for P to be

$$net\mathcal{E}[\, sys\mathcal{E}[\, \overrightarrow{\beta_L \mathcal{E}}[\, a \,]]]$$

where $\{\, \overrightarrow{L} \,\}$ is the security lattice.

C.2 Theorems

We prove the following theorems that, together with the soundness of the type system for $\mathbf{conc}\,\tilde{\imath}$, imply the soundness of the proposed type system in this section.

We begin with typability, which states that the compiled $\mathbf{conc}\,\tilde{\imath}$ program is well-typed if the source process is well-typed.

Theorem C.2.1 (Typability)**.** *Suppose that* $\mathsf{net} : \bot, \overrightarrow{\beta_L.\varkappa : \mathsf{Req}_L.\tilde{\varkappa}} \vdash P \rightsquigarrow a$*. Then*

$$\varnothing \vdash \mathrm{net}\mathcal{E}[\mathrm{sys}\mathcal{E}[\overrightarrow{\beta_L \mathcal{E}}[a]]] : \bot$$

Next, we prove simulatability, which states that the compiled $\mathbf{conc}\,\tilde{\imath}$ program can simulate the behaviours of the source process.

Theorem C.2.2 (Simulatability)**.** *Suppose that* $\mathsf{net} : \bot, \overrightarrow{\beta_L.\varkappa : \mathsf{Req}_L.\tilde{\varkappa}} \vdash P \rightsquigarrow a$ *and* $P \longrightarrow^{\star} (\nu y : L[T])\ (\_ \mid \overline{\mathsf{net}}\langle y \rangle; \_)$*. Then*

$$\mathrm{net}\mathcal{E}[\mathrm{sys}\mathcal{E}[\overrightarrow{\beta_L \mathcal{E}}[a]]] \longrightarrow^{\star} \left(\begin{array}{l} (\nu m? : (\langle \bot \rangle [\![T]\!])^{L^{\top}}) \\ (\nu m! : (\langle [\![T]\!] \rangle \bot)^{L^{\top}}) \\ (\nu n! : (\langle \bot \rangle \bot)^{\bot^{\top}}) \\ \_ \Rsh \ let\ \_ = \ \tilde{n}!\langle (\tilde{m}?, \tilde{m}!) \rangle\ in\ \_ \end{array} \right)$$

Appendix D

Proofs

In this appendix we provide proof details for various results that appear in this dissertation. We begin with the correctness of query evaluation in EON. We then consider the soundness of the type system for **conc**$\tilde{\iota}$.

D.1 Correctness of query evaluation in EON

Lemma D.1.1. *If* $\texttt{Reachable}(c) \in \mathcal{I}(\lfloor \mathbb{P} \rfloor, \mathbb{DB})$*, then the atomic state of c in* $\mathbb{DB}$ *is reachable. Conversely, if an atomic state is reachable, then there exists a database* $\mathbb{DB}$ *that contains a constant c with that atomic state, such that* $\texttt{Reachable}(c) \in \mathcal{I}(\lfloor \mathbb{P} \rfloor, \mathbb{DB})$.

Proof. Note that there is a one-to-one correspondence between a derivation of $\texttt{Reachable}(c)$ using the Datalog program $\lfloor \mathbb{P} \rfloor$, and a transition sequence using the EON program $\mathbb{P}$ that results in a database with a constant with the same atomic state as c. ◀

Lemma D.1.2. *If* $\mathbb{DB} \vdash_{\lfloor \mathbb{P} \rfloor} \texttt{!BadState}$ *then for every constant c,* $\texttt{Reachable}(c) \in \mathcal{I}(\lfloor \mathbb{P} \rfloor, \mathbb{DB})$ *iff* $\texttt{U}(c) \in \mathcal{I}(\lfloor \mathbb{P} \rfloor, \mathbb{DB})$.

Proof. Note that the clause transformation augments the body of every clause for $\texttt{Reachable}(x)$ with the literal $\texttt{U}(x)$. Hence, it trivially follows that $\texttt{Reachable}(c)$ is true only if $\texttt{U}(c)$ is true. The transformed clause for `BadState` is as follows:

$\texttt{BadState} \texttt{ :- } \texttt{U}(x), \texttt{!Reachable}(x).$

Thus, `BadState` is true if there exists some constant c such that $\mathtt{U}(c)$ is true but $\mathtt{Reachable}(c)$ is not. Consequently, in any database that satisfies `!BadState`, we have that $\mathtt{Reachable}(c)$ is true iff $\mathtt{U}(c)$ is true. ◀

Lemma D.1.3. *If* $\mathbb{DB} \vdash_{\lfloor \mathbb{P} \rfloor} \lfloor \mathcal{S} \rfloor$ *then* $\mathbb{DB}|_{\mathtt{U}} \vdash_{\lfloor \mathbb{P} \rfloor} \lfloor \mathcal{S} \rfloor$.

Proof. Note that the body of every clause in $\lfloor \mathbb{P} \rfloor$ contains a literal $\mathtt{U}(x)$ for every variable x occurring in the clause. As a consequence, no derivation of $\lfloor \mathcal{S} \rfloor$ can use a constant c for which $\mathtt{U}(c)$ is false. ◀

Lemma D.1.4. *If* $\mathbb{DB} \vdash_{\lfloor \mathbb{P} \rfloor} \lfloor \mathcal{S} \rfloor$ *then* $\mathbb{DB}|_{\mathtt{U}}$ *is a reachable database.*

Proof. It follows from Lemma D.1.3 that $\mathbb{DB}|_{\mathtt{U}}$ satisfies the transformed query $\lfloor \mathcal{S} \rfloor$ and, hence, it satisfies `!BadState` as well. It follows from Lemma D.1.2 and the definition of $\mathbb{DB}|_{\mathtt{U}}$ that $\mathtt{Reachable}(c)$ is true for every constant c in $\mathbb{DB}|_{\mathtt{U}}$. Lemma D.1.1 implies that all constants in $\mathbb{DB}|_{\mathtt{U}}$ have a reachable atomic state. It follows from Lemma 3.2.2 that $\mathbb{DB}|_{\mathtt{U}}$ is a reachable database. ◀

Theorem D.1.5. *If* $\mathbb{DB} \vdash_{\lfloor \mathbb{P} \rfloor} \lfloor \mathcal{S} \rfloor$*, then the query* $\mathcal{S}$ *is true in the* EON *program* $\mathbb{P}$.

Proof. Follows from Lemmas D.1.3 and D.1.4. ◀

Theorem D.1.6. *If the query* $\mathcal{S}$ *is true in the* EON *program* $\mathbb{P}$*, then there exists a database* $\mathbb{DB}$ *such that* $\mathbb{DB} \vdash_{\lfloor \mathbb{P} \rfloor} \lfloor \mathcal{S} \rfloor$.

Proof. Let $\mathbb{DB}_1 \xrightarrow{\mathbb{P}} \ldots \xrightarrow{\mathbb{P}} \mathbb{DB}_m$ and $\mathcal{S}\sigma \in \mathcal{I}(\hat{\mathbb{P}}, \mathbb{DB}_m)$ for some ground substitution σ. We define the database $\mathbb{DB}$ to be the disjoint sum of the databases $\mathbb{DB}_1$ through $\mathbb{DB}_m$. Specifically, let each $\mathbb{DB}_i = (U_i, I_i)$. We define $\mathbb{DB} = (U, I)$, where $U = \{(i, c) \mid 1 \leq i \leq m, c \in U_i\}$, and $I(B) = \{(i, c) \mid 1 \leq i \leq m, c \in I_i(B)\}$ for $B \in \mathcal{E}$, and $I(\mathtt{U}) = U$. It can be shown that $\mathtt{Reachable}(u)$ is true for every u in U. Further, the monotonicity of the query $\mathcal{S}$ guarantees that $\lfloor \mathcal{S} \rfloor$ is satisfiable in $\mathbb{DB}$. ◀

D.2 Soundness of the type system for conc$\tilde{\imath}$

Lemma D.2.1 (Subject congruence)**.** *Let* $a \equiv b$ *and* $\Gamma \vdash a : T$*. Then* $\Gamma \vdash b : T$.

Proof. By induction on $\equiv$. We omit the cases for equivalence.

Case (Struct Res).

Case $(\nu n)\ a \equiv (\nu n)\ a$. Trivial.

Case $(\nu n)\ \text{let } x = \mathcal{E}[\![a]\!] \text{ in } b \equiv \text{let } x = \mathcal{E}[\![(\nu n)\ a]\!] \text{ in } b$.

Then $\Gamma \vdash (\nu n)\ \text{let } x = \mathcal{E}[\![a]\!] \text{ in } b : T$

iff $\Gamma, n : U \vdash \text{let } x = \mathcal{E}[\![a]\!] \text{ in } b : T$

iff $\Gamma, n : U \vdash \mathcal{E}[\![a]\!] : S$ and $\Gamma, n : U, x : S \vdash b : T$

iff $\Gamma \vdash (\nu n)\ \mathcal{E}[\![a]\!] : S$ and $\Gamma, x : S \vdash b : T$

iff (by induction hypothesis) $\Gamma \vdash \mathcal{E}[\![(\nu n)\ a]\!] : S$ and $\Gamma, x : S \vdash b : T$

iff $\Gamma \vdash \text{let } x = \mathcal{E}[\![(\nu n)\ a]\!] \text{ in } b : T$.

Case $(\nu n)\ \mathcal{E}[\![a]\!] \Rsh b \equiv \mathcal{E}[\![(\nu n)\ a]\!] \Rsh b$.

Then $\Gamma \vdash (\nu n)\ \mathcal{E}[\![a]\!] \Rsh b : T$

iff $\Gamma, n : U \vdash \mathcal{E}[\![a]\!] \Rsh b : T$

iff $\Gamma, n : U \vdash \mathcal{E}[\![a]\!] : S$ and $\Gamma, n : U \vdash b : T$

iff $\Gamma \vdash (\nu n)\ \mathcal{E}[\![a]\!] : S$ and $\Gamma \vdash b : T$

iff (by induction hypothesis) $\Gamma \vdash \mathcal{E}[\![(\nu n)\ a]\!] : S$ and $\Gamma \vdash b : T$

iff $\Gamma \vdash \mathcal{E}[\![(\nu n)\ a]\!] \Rsh b : T$.

Case $(\nu n)\ a \Rsh \mathcal{E}[\![b]\!] \equiv a \Rsh \mathcal{E}[\![(\nu n)\ b]\!]$.

Then $\Gamma \vdash (\nu n)\ a \Rsh \mathcal{E}[\![b]\!] : T$

iff $\Gamma, n : U \vdash a \Rsh \mathcal{E}[\![b]\!] : T$

iff $\Gamma, n : U \vdash a : S$ and $\Gamma, n : U \vdash \mathcal{E}[\![b]\!] : T$

iff $\Gamma \vdash a : S$ and $\Gamma \vdash (\nu n)\ \mathcal{E}[\![a]\!] : T$

iff (by induction hypothesis) $\Gamma \vdash a : S$ and $\Gamma \vdash \mathcal{E}[\![(\nu n)\ a]\!] : T$

iff $\Gamma \vdash a \Rsh \mathcal{E}[\![(\nu n)\ b]\!] : T$.

Case $(\nu n)\ (\nu m)\ \mathcal{E}[\![a]\!] \equiv (\nu m)\ \mathcal{E}[\![(\nu n)\ a]\!]$.

Then $\Gamma \vdash (\nu n)\ (\nu m)\ \mathcal{E}[\![a]\!] : T$

iff $\Gamma, n : U \vdash (\nu m)\ \mathcal{E}[\![a]\!] : T$

iff $\Gamma, n : U, m : S \vdash \mathcal{E}[\![a]\!] : T$

iff $\Gamma, m : S \vdash (\nu n)\ \mathcal{E}[\![a]\!] : T$

iff (by induction hypothesis) $\Gamma, m : S \vdash \mathcal{E}[\![(\nu n)\ a]\!] : T$

iff $\Gamma \vdash (\nu m)\ \mathcal{E}[\![(\nu n)\ a]\!] : T$.

Case (Struct Par).

Case $a \Rsh b \equiv a \Rsh b$. Trivial.

Case $a \Rsh \text{let } x = \mathcal{E}[\![b]\!] \text{ in } b' \equiv \text{let } x = \mathcal{E}[\![a \Rsh b]\!] \text{ in } b'$.

Then $\Gamma \vdash a \Rsh \text{let } x = \mathcal{E}[\![b]\!] \text{ in } b' : T$

iff $\Gamma \vdash a : U$ and $\Gamma \vdash \text{let } x = \mathcal{E}[\![b]\!] \text{ in } b' : T$

iff $\Gamma \vdash a : U$ and $\Gamma \vdash \mathcal{E}[\![b]\!] : S$ and $\Gamma, x : S \vdash b' : T$

iff $\Gamma \vdash a \Rsh \mathcal{E}[\![b]\!] : S$ and $\Gamma, x : S \vdash b' : T$

iff (by induction hypothesis) $\Gamma \vdash \mathcal{E}[\![a \Rsh b]\!] : S$ and $\Gamma, x : S \vdash b' : T$

iff $\Gamma \vdash \text{let } x = \mathcal{E}[\![a \Rsh b]\!] \text{ in } b' : T$.

Case $a \Rsh (\mathcal{E}[\![b]\!] \Rsh b') \equiv \mathcal{E}[\![a \Rsh b]\!] \Rsh b'$.

Then $\Gamma \vdash a \Rsh (\mathcal{E}[\![b]\!] \Rsh b') : T$

iff $\Gamma \vdash a : U$ and $\Gamma \vdash \mathcal{E}[\![b]\!] \Rsh b' : T$

iff $\Gamma \vdash a : U$ and $\Gamma \vdash \mathcal{E}[\![b]\!] : S$ and $\Gamma \vdash b' : T$

iff $\Gamma \vdash a \Rsh \mathcal{E}[\![b]\!] : S$ and $\Gamma \vdash b' : T$

iff (by induction hypothesis) $\Gamma \vdash \mathcal{E}[\![a \Rsh b]\!] : S$ and $\Gamma \vdash b' : T$

iff iff $\Gamma \vdash \mathcal{E}[\![a \Rsh b]\!] \Rsh b' : T$.

Case $a \Rsh (a' \Rsh \mathcal{E}[\![b]\!]) \equiv a' \Rsh \mathcal{E}[\![a \Rsh b]\!]$.

Then $\Gamma \vdash a \Rsh (a' \Rsh \mathcal{E}[\![b]\!]) : T$

iff $\Gamma \vdash a : U$ and $\Gamma \vdash a' \Rsh \mathcal{E}[\![b]\!] : T$

iff $\Gamma \vdash a : U$ and $\Gamma \vdash a' : S$ and $\Gamma \vdash \mathcal{E}[\![b]\!] : T$

iff $\Gamma \vdash a' : S$ and $\Gamma \vdash a \Rsh \mathcal{E}[\![b]\!] : T$

iff (by induction hypothesis) $\Gamma \vdash a' : S$ and $\Gamma \vdash \mathcal{E}[\![a \Rsh b]\!] : T$

iff $\Gamma \vdash a' \Rsh \mathcal{E}[\![a \Rsh b]\!] : T$.

Case $a \Rsh (\nu n)\ \mathcal{E}[\![b]\!] \equiv (\nu n)\ \mathcal{E}[\![a \Rsh b]\!]$.

Then $\Gamma \vdash a \Rsh (\nu n)\ \mathcal{E}[\![b]\!] : T$

iff $\Gamma \vdash a : U$ and $\Gamma \vdash (\nu n)\ \mathcal{E}[\![b]\!] : T$

iff $\Gamma \vdash a : U$ and $\Gamma, n : S \vdash \mathcal{E}[\![b]\!] : T$

iff $\Gamma, n : S \vdash a : U$ and $\Gamma, n : S \vdash \mathcal{E}[\![b]\!] : T$
iff $\Gamma, n : S \vdash a \Rsh \mathcal{E}[\![b]\!] : T$
iff (by induction hypothesis) $\Gamma, n : S \vdash \mathcal{E}[\![a \Rsh b]\!] : T$
iff $\Gamma \vdash (\nu n)\, \mathcal{E}[\![a \Rsh b]\!] : T$. ◀

Lemma D.2.2 (Well-typed contexts)**.** *Suppose that* $\Gamma \vdash b : S$ *whenever* $\Gamma \vdash a : S$*. Then* $\Gamma, \Gamma' \vdash \mathcal{E}[\![b]\!] : T$ *whenever* $\Gamma, \Gamma' \vdash \mathcal{E}[\![a]\!] : T$*.*

Proof. By induction on $\mathcal{E}$.

Case $\Gamma \vdash a : T$. Trivial.

Case $\Gamma \vdash \text{let } x = \mathcal{E}[\![a]\!] \text{ in } b' : T$.
Thus $\Gamma \vdash \mathcal{E}[\![a]\!] : S$ and $\Gamma, x : U \vdash b' : T$
thus (by induction hypothesis) $\Gamma \vdash \mathcal{E}[\![b]\!] : S$ and $\Gamma, x : S \vdash b' : T$
thus $\Gamma \vdash \text{let } x = \mathcal{E}[\![b]\!] \text{ in } b' : T$.

Case $\Gamma \vdash \mathcal{E}[\![a]\!] \Rsh b' : T$.
Thus $\Gamma \vdash \mathcal{E}[\![a]\!] : S$ and $\Gamma \vdash b' : T$
thus (by induction hypothesis) $\Gamma \vdash \mathcal{E}[\![b]\!] : S$ and $\Gamma \vdash b' : T$
thus $\Gamma \vdash \mathcal{E}[\![b]\!] \Rsh b' : T$.

Case $\Gamma \vdash a' \Rsh \mathcal{E}[\![a]\!] : T$.
Thus $\Gamma \vdash a' : S$ and $\Gamma \vdash \mathcal{E}[\![a]\!] : T$
thus (by induction hypothesis) $\Gamma \vdash a' : S$ and $\Gamma \vdash \mathcal{E}[\![b]\!] : T$
thus $\Gamma \vdash a' \Rsh \mathcal{E}[\![b]\!] : T$.

Case $\Gamma \vdash (\nu n)\, \mathcal{E}[\![a]\!] : T$.
Thus $\Gamma, n : S \vdash \mathcal{E}[\![a]\!] : T$
thus (by induction hypothesis) $\Gamma, n : S \vdash \mathcal{E}[\![b]\!] : T$
thus $\Gamma \vdash (\nu n)\, \mathcal{E}[\![b]\!] : T$. ◀

Lemma D.2.3 (Substitution)**.** $\Gamma, x : T \vdash a : S$*. Then* $\Gamma \vdash a\{n/x\} : S$ *whenever* $\Gamma \vdash n : T$*.*

Proof. By induction on $\vdash$ derivation. ◀

Lemma D.2.4 (Polymorphic typing). *Let* $\Gamma, \mathcal{X}, \overrightarrow{u} : \overrightarrow{T} \vdash a : S$, *and* $\mathtt{dom}(\sigma) = \mathcal{X}$. *Then* $\Gamma, \overrightarrow{u} : \overrightarrow{T\sigma} \vdash a\sigma : S\sigma$.

Proof. By induction on $\vdash$ derivation. ◀

Proposition 5.3.1 (Subject reduction). *Let* $\Gamma \vdash a : T$. *If* $a \longrightarrow b$, *then* $\Gamma \vdash b : T$.

Proof. By induction on $\longrightarrow$.

Case (Red Eval).

$\Gamma \vdash \text{let } x = n \text{ in } a : T$.

Thus $\Gamma \vdash n : S$ and $\Gamma, x : S \vdash a : T$

thus (by Lemma D.2.3) $\Gamma \vdash a\{n/x\} : T$.

Case (Red Context).

$\Gamma \vdash \mathcal{E}[\![a]\!] : T$ and $\Gamma \vdash a : S$ and $a \longrightarrow b$.

Thus (by induction hypothesis) $\Gamma \vdash b : S$

thus (by Lemma D.2.2) $\Gamma \vdash \mathcal{E}[\![b]\!] : T$.

Case (Red Struct).

$\Gamma \vdash \mathcal{E}[\![a]\!] : T$ and $a \equiv a'$ and $a' \longrightarrow b'$ and $b' \equiv b$.

Thus (by Lemma D.2.1) $\Gamma \vdash a' : T$

thus (by induction hypothesis) $\Gamma \vdash b' : T$

thus (by Lemma D.2.1) $\Gamma \vdash b : T$.

Case (Red Call).

Let $d = \overrightarrow{v_i}[\overrightarrow{\ell_i \Rightarrow (y_i)b_i}]$.

Case $\Gamma \vdash (p \mapsto d) \Cap \widetilde{v_i}\langle u \rangle : T$

and thus $\Gamma \vdash p \mapsto d : \bot$ and $\Gamma \vdash \widetilde{v_i}\langle u \rangle : T$.

Thus $\Gamma \vdash v_i : (\forall \mathcal{Y} \langle S' \rangle T')^{G \cdots}$, $\Gamma \vdash u : S$, and $\Gamma, \mathcal{Y}, y_i : S' \vdash b_i \lightning_p^{\overrightarrow{v_i}} : T'$

for some σ such that $S'\sigma = S$ and $T'\sigma = T$.

Thus (by Lemma D.2.3) $\Gamma \vdash b_i \lightning_p^{\overrightarrow{v_i}} \{u/y_i\} : T$

thus $\Gamma \vdash (p \mapsto d) \Cap b_i \lightning_p^{\overrightarrow{v_i}} \{u/y_i\} : T$.

Case $\Gamma \vdash (p \mapsto d) \Rsh \tilde{v_i}\langle u \rangle : \bot$

and thus $\Gamma \vdash p \mapsto d : \bot$ and $\Gamma \vdash \tilde{v_i}\langle u \rangle : \bot$.

Thus $\Gamma \vdash v_i : \bot$, $\Gamma \vdash u : \bot$, and $\Gamma, \mathcal{Y}, y_i : \bot \vdash b_i \lightning_p^{\overrightarrow{v_i}} : \bot$.

Thus (by Lemma D.2.3) $\Gamma \vdash b_i \lightning_p^{\overrightarrow{v_i}} \{u/y_i\} : \bot$

thus $\Gamma \vdash (p \mapsto d) \Rsh b_i \lightning_p^{\overrightarrow{v_i}} \{u/y_i\} : \bot$.

Case (Red Upd).

Let $d = \overrightarrow{v_i}[\overrightarrow{\ell_i \Mapsto (y_i)b_i}]$, $d' = \overrightarrow{v_j}[\overrightarrow{\ell_k \Mapsto (y_k)b_k}]$,

and $d'' = \overrightarrow{v_j} \circ \overrightarrow{v_i}[\overrightarrow{\ell_k \Mapsto (y_k)b_k} \circ \overrightarrow{\ell_i \Mapsto (y_i)b_i}]$.

We have $\Gamma \vdash p \leftarrowtail d \Rsh p \leftarrowtail d' : \bot$

and thus $\Gamma \vdash p \leftarrowtail d : \bot$ and $\Gamma \vdash p \leftarrowtail d' : \bot$.

Thus $\Gamma \vdash p : \forall \mathcal{X}[\overrightarrow{\ell_i^{\delta_i} : \forall \mathcal{Y}_i(S_i)T_i}]^{G\cdots}$, $\textsc{Invariance}(\overrightarrow{i}, \overrightarrow{i}, \overrightarrow{i})$,

and $\textsc{Invariance}(\overrightarrow{i}, \overrightarrow{j}, \overrightarrow{k})$.

Thus for some σ such that $\mathtt{dom}(\sigma) = \mathcal{X}$:

$$\forall i.\ \Gamma \vdash v_i : (\forall \mathcal{Y}_i \langle S_i\sigma \rangle T_i\sigma)^{GG_i\cdots}$$

$$\forall i.\ \delta_i = - \ \Rightarrow \begin{cases} \Gamma, \overrightarrow{Z_i}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i\sigma \rangle T_i\sigma)^{GZ_i}}, \mathcal{Y}_i, y_i : S_i\sigma \vdash b_i\sigma \lightning_u^{\overrightarrow{z_i}} : T_i\sigma \\ \Gamma, \overrightarrow{Z_i}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i\sigma \rangle T_i\sigma)^{GZ_i}}, \mathcal{Y}_i, y_i : \bot, \bot \leq G_i \vdash b_i\sigma \lightning_u^{\overrightarrow{z_i}} : \bot \end{cases}$$

$$\forall i.\ \delta_i = + \ \Rightarrow \begin{cases} \Gamma, \overrightarrow{Z_i}, \mathcal{X}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i \rangle T_i)^{GZ_i}}, \mathcal{Y}_i, y_i : S_i \vdash b_i \lightning_u^{\overrightarrow{z_i}} : T_i \\ \Gamma, \overrightarrow{Z_i}, \mathcal{X}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i \rangle T_i)^{GZ_i}}, \mathcal{Y}_i, y_i : \bot, \bot \leq G_i \vdash b_i \lightning_u^{\overrightarrow{z_i}} : \bot \end{cases}$$

and for some σ' such that $\mathtt{dom}(\sigma') = \mathcal{X}$:

$$\forall j.\ \Gamma \vdash v_j : (\forall \mathcal{Y}_j \langle S_j\sigma' \rangle T_j\sigma')^{GG_j\cdots}$$

$$\{i \mid \Gamma, \mathcal{Y}_i \not\vdash S_i, T_i\} \subseteq \{\overrightarrow{j}\} \subseteq \{\overrightarrow{i}\}$$

$$\{i \mid \delta_i = -\} \cup \{j \mid \Gamma, \bot \leq G_j \not\vdash \bot \leq \top\} \subseteq \{\overrightarrow{k}\} \subseteq \{\overrightarrow{i}\}$$

$$\forall k.\ \delta_k = - \ \Rightarrow \begin{cases} \Gamma, \overrightarrow{Z_i}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i\sigma' \rangle T_i\sigma')^{GZ_i}}, \mathcal{Y}_k, y_k : S_k\sigma' \vdash b_k\sigma' \lightning_u^{\overrightarrow{z_i}} : T_k\sigma' \\ k \in \{\overrightarrow{j}\} \Rightarrow \\ \Gamma, \overrightarrow{Z_i}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i\sigma' \rangle T_i\sigma')^{GZ_i}}, \mathcal{Y}_k, y_k : \bot, \bot \leq G_k \vdash b_k\sigma' \lightning_u^{\overrightarrow{z_i}} : \bot \end{cases}$$

$$\forall k.\ \delta_k = + \ \Rightarrow \begin{cases} \Gamma, \overrightarrow{Z_i}, \mathcal{X}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i \rangle T_i)^{GZ_i}}, \mathcal{Y}_k, y_k : S_k \vdash b_k \lightning_u^{\overrightarrow{z_i}} : T_k \\ k \in \{\overrightarrow{j}\} \Rightarrow \\ \Gamma, \overrightarrow{Z_i}, \mathcal{X}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i \rangle T_i)^{GZ_i}}, \mathcal{Y}_k, y_k : \bot, \bot \leq G_k \vdash b_k \lightning_u^{\overrightarrow{z_i}} : \bot \end{cases}$$

For all $i \in \{\overrightarrow{i}\} \setminus \{\overrightarrow{j}\}$, we have $\Gamma, \mathcal{Y}_i \vdash S_i, T_i$, so that $S\sigma' = S$ and $T\sigma' = T$.

Thus for all $i \in \{\overrightarrow{i}\} \setminus \{\overrightarrow{j}\}$, we have $\Gamma \vdash v_i : (\forall \mathcal{Y}_i \langle S_i\sigma' \rangle T_i\sigma')^{GG_i \cdots}$.

Thus $\forall i.\ \Gamma \vdash v_i : (\forall \mathcal{Y}_i \langle S_i\sigma' \rangle T_i\sigma')^{GG_i \cdots}$.

For all $i \in \{\overrightarrow{i}\} \setminus \{\overrightarrow{k}\}$, we have $\delta_i = +$ and $i \in \{\overrightarrow{j}\} \Rightarrow \Gamma, \bot \leq G_i \vdash \bot \leq \top$.

So

$$\forall i.\ \delta_i = - \ \Rightarrow \begin{cases} \Gamma, \overrightarrow{Z_i}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i\sigma \rangle T_i\sigma')^{GZ_i}}, \mathcal{Y}_i, y_i : S_i\sigma \vdash b_i\sigma \lightning_u^{\overrightarrow{z_i}} : T_i\sigma \\ \Gamma, \overrightarrow{Z_i}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i\sigma \rangle T_i\sigma')^{GZ_i}}, \mathcal{Y}_i, y_i : \bot, \bot \leq G_i \vdash b_i\sigma \lightning_u^{\overrightarrow{z_i}} : \bot \end{cases}$$

$$\forall i.\ \delta_i = + \ \Rightarrow \begin{cases} \Gamma, \overrightarrow{Z_i}, \mathcal{X}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i \rangle T_i)^{GZ_i}}, \mathcal{Y}_i, y_i : S_i \vdash b_i \lightning_u^{\overrightarrow{z_i}} : T_i \\ \Gamma, \overrightarrow{Z_i}, \mathcal{X}, \overrightarrow{z_i} : \overrightarrow{(\forall \mathcal{Y}_i \langle S_i \rangle T_i)^{GZ_i}}, \mathcal{Y}_i, y_i : \bot, \bot \leq G_i \vdash b_i \lightning_u^{\overrightarrow{z_i}} : \bot \end{cases}$$

Thus $\Gamma \vdash p : \forall \mathcal{X} [\overrightarrow{\ell_i^{\delta_i} : \forall \mathcal{Y}_i (S_i) \overrightarrow{T_i}}]^{G \cdots}$ and $\textsc{Invariance}(\overrightarrow{i}, \overrightarrow{i}, \overrightarrow{i})$ for d''.

Thus $\Gamma \vdash p \mapsto d'' : \bot$.

Thus $\Gamma \vdash p \mapsto d'' \Rsh \bot : \bot$. ◀

D.3 Soundness of the type system for DFI on Windows Vista

In this section we outline proofs of the results in Section 6.4.

Proposition 6.4.2 (Adversary completeness)**.** *Let* Γ *be any typing environment and* e *be any* C*-adversary such that* $\mathtt{fv}(e) \subseteq \mathsf{dom}(\Gamma)$*. Then* $\Gamma \vdash_\top e : \_$ despite C*.*

Proof. We prove typability by induction on the structure of processes.

- $e \equiv x$ where u is a variable.
  Then $x \in \mathtt{dom}(\Gamma)$.
  By (Typ value) $\Gamma \vdash_\mathsf{C} x : \_$.

- $e \equiv \mathsf{new}(x \,\#\, \mathsf{S})$.
  By I.H. $\Gamma \vdash_\mathsf{C} x : \tau^\mathsf{E}$
  Then $\mathsf{S} \sqsubseteq \mathsf{C} \sqsubseteq \bot \sqsubseteq \mathsf{E}$.
  By (Typ new) $\Gamma \vdash_\mathsf{C} \mathsf{new}(x \,\#\, \mathsf{S}) : \_$.

- $e \equiv \langle \mathsf{O} \rangle\ \omega$.
  By I.H. $\Gamma \vdash_\mathsf{C} \omega : \_$.
  So by (Typ value) $\omega : \tau^\mathsf{E} \in \Gamma$.

Case $*\mathsf{E}$ and τ is not of the form $\mathbf{Obj}(\_)$.

By (TYP BOGUS STUCK-I) $\Gamma \vdash_{\mathsf{C}} \langle \mathsf{O} \rangle\ \omega : \_$.

Case $*\mathsf{E}$, $\tau = \mathbf{Obj}(\_^{\mathsf{S}})$, and $\mathsf{C} \sqsubset \mathsf{S} \sqcup \mathsf{O}$.

By (TYP UN/PROTECT STUCK) $\Gamma \vdash_{\mathsf{C}} \langle \mathsf{O} \rangle\ \omega : \_$.

Case $*\mathsf{E}$, $\tau = \mathbf{Obj}(\_^{\mathsf{S}})$, and $\bot \sqsubseteq \mathsf{S} \sqcup \mathsf{O} \sqsubseteq \mathsf{C} = \bot$.

Then $\mathsf{S} \sqsubseteq \mathsf{O}$.

By (TYP VALUE) and (TYP UN/PROTECT) $\Gamma \vdash_{\mathsf{C}} \langle \mathsf{O} \rangle\ \omega : \_$.

Case $\mathsf{E} = \bot$.

By (TYP SUBSUMPTION $\bot$-II) $\tau = \mathbf{Obj}(\_^{\mathsf{S}})$ such that $\mathsf{S} \sqsubseteq \mathsf{O}$.

By (TYP VALUE) and (TYP UN/PROTECT) $\Gamma \vdash_{\mathsf{C}} \langle \mathsf{O} \rangle\ \omega : \_$.

- $e \equiv\ !\omega$.

  By I.H. $\Gamma \vdash_{\mathsf{C}} \omega : \_$.

  So by (TYP VALUE) $\omega : \tau^{\mathsf{E}} \in \Gamma$.

  Case $*\mathsf{E}$ and τ is not of the form $\mathbf{Obj}(\_)$.

  By (TYP BOGUS STUCK-I) $\Gamma \vdash_{\mathsf{C}}\ !\omega : \_$.

  Case $*\mathsf{E}$ and $\tau = \mathbf{Obj}(\_)$.

  By (TYP READ) $\Gamma \vdash_{\mathsf{C}}\ !\omega : \_$.

  Case $\mathsf{E} = \bot$.

  By (TYP SUBSUMPTION $\bot$-II) $\tau = \mathbf{Obj}(\_)$.

  By (TYP READ) $\Gamma \vdash_{\mathsf{C}}\ !\omega : \_$.

- $e \equiv \omega := x$.

  By I.H. $\Gamma \vdash_{\mathsf{C}} \omega : \_$ and $\Gamma \vdash_{\mathsf{C}} x : \tau_1^{\mathsf{E}'}$.

  So by (TYP VALUE) $\omega : \tau^{\mathsf{E}} \in \Gamma$.

  Case $*\mathsf{E}$ and τ is not of the form $\mathbf{Obj}(\_)$.

  By (TYP BOGUS STUCK-I) $\Gamma \vdash_{\mathsf{C}} \omega := x : \_$.

  Case $*\mathsf{E}$, $\tau = \mathbf{Obj}(\_^{\mathsf{S}})$, and $\mathsf{C} \sqsubset \mathsf{S}$.

  By (TYP WRITE STUCK) $\Gamma \vdash_{\mathsf{C}} \omega := x : \_$.

Case $*\mathsf{E}$, $\tau = \mathbf{Obj}(\tau_1^{\mathsf{S}})$, and $\bot \sqsubseteq \mathsf{S} \sqsubseteq \mathsf{C} = \bot$.

Then $\mathsf{S} \sqsubseteq \mathsf{E}'$.

By (TYP VALUE) and (TYP WRITE) $\Gamma \vdash_{\mathsf{C}} \omega := x : \_$.

Case $\mathsf{E} = \bot$.

By (TYP SUBSUMPTION $\bot$-II) $\tau = \mathbf{Obj}(\tau_1^{\mathsf{S}})$ such that $\mathsf{S} \sqsubseteq \mathsf{E}'$.

By (TYP VALUE) and (TYP WRITE) $\Gamma \vdash_{\mathsf{C}} \omega := x : \_$.

- $e \equiv \mathsf{pack}(f)$.

  By I.H. $\Gamma \vdash_{\mathsf{C}} f : T$.

  By (TYP PACK) $\Gamma \vdash_{\mathsf{C}} \mathsf{pack}(f) : \_$.

- $e \equiv \mathsf{exec}\ \omega$.

  By I.H. $\Gamma \vdash_{\mathsf{C}} \omega : \_$, so by (TYP VALUE) $\omega : \tau^{\mathsf{E}} \in \Gamma$.

  Case $*\mathsf{E}$ and τ is not of the form $\mathbf{Obj}(\_)$.

  By (TYP BOGUS STUCK-I) $\Gamma \vdash_{\mathsf{C}} \mathsf{exec}\ \omega : \_$.

  Case $\tau = \mathbf{Obj}(\tau_1^{\mathsf{S}})$, $*\mathsf{E}$, and τ_1 is not of the form $\nabla\_.\ \mathbf{Bin}(\_)$.

  By (TYP BOGUS STUCK-II) $\Gamma \vdash_{\mathsf{C}} \mathsf{exec}\ \omega : \_$.

  Case $\tau = \mathbf{Obj}(\tau_1^{\mathsf{S}})$, $*\mathsf{E}$, and $\tau_1 = \nabla_{\mathsf{P}}.\ \mathbf{Bin}(\_)$.

  Then $\mathsf{C} = \bot \sqsubseteq \mathsf{P} \sqcap \mathsf{S}$.

  By (TYP EXECUTE) $\Gamma \vdash_{\mathsf{C}} \mathsf{exec}\ \omega : \_$.

  Case $\mathsf{E} = \bot$.

  By (TYP SUBSUMPTION $\bot$-II) $\tau = \mathbf{Obj}(\tau_1^{\mathsf{S}})$
  and $\tau_1 = \nabla_{\mathsf{P}}.\ \mathbf{Bin}(\_)$ such that $\mathsf{C} = \bot \sqsubseteq \mathsf{P} \sqcap \mathsf{S}$.

  By (TYP EXECUTE) $\Gamma \vdash_{\mathsf{C}} \mathsf{exec}\ \omega : \_$.

  Case $*\mathsf{E}$, $\tau = \mathbf{Obj}(\tau_1^{\mathsf{S}})$, and $\mathsf{S} = \bot$.

  By (TYP SUBSUMPTION $\bot$-I) $\tau_1 = \nabla_{\mathsf{P}}.\ \mathbf{Bin}(\_)$ such that $\mathsf{C} = \bot \sqsubseteq \mathsf{P} \sqcap \mathsf{S}$.

  By (TYP EXECUTE) $\Gamma \vdash_{\mathsf{C}} \mathsf{exec}\ \omega : \_$.

- $e \equiv [\mathsf{P}]\ a$.

  If $\mathsf{P} \sqsupset \mathsf{C}$ then by (TYP ESCALATE) $\Gamma \vdash_{\mathsf{C}} [\mathsf{P}]\ a : \_$.

  Otherwise by I.H. $\Gamma \vdash_{\mathsf{P}} a : \_$.

  By (TYP LIMIT) $\Gamma \vdash_{\mathsf{C}} [\mathsf{P}]\ a : \_$.

- $e \equiv \text{let } x = a \text{ in } b$.
  By I.H. $\Gamma \vdash_{\mathsf{C}} a : T$ and $\Gamma, x : T \vdash_{\mathsf{C}} b : T'$.
  By (TYP EVALUATE) $\Gamma \vdash_{\mathsf{C}} \text{let } x = a \text{ in } b : \_$.

- $e \equiv a \mathbin{↱} b$.
  By I.H. $\Gamma \vdash_{\mathsf{C}} a : \_$ and $\Gamma \vdash_{\mathsf{C}} b : T$.
  By (TYP FORK) $\Gamma \vdash_{\mathsf{C}} a \mathbin{↱} b : \_$. ◀

Proposition 6.4.3 (Monotonicity)**.** *The following typing rule is admissible.*

$$\frac{\Gamma \vdash_{\mathsf{P}'} f : \tau^{\mathsf{E}} \qquad \Box f \qquad \mathsf{P} \sqsubset \mathsf{P}'}{\Gamma \vdash_{\mathsf{P}} f : \tau^{\mathsf{E} \sqcap \mathsf{P}}}$$

Proof. We proceed by induction on the structure of derivations.

Suppose that $\mathsf{P}' \sqsubset \mathsf{P}$.

Case (Typ variable) By (TYP VALUE) $\Gamma \vdash_{\mathsf{P}} x : \tau^{\mathsf{E} \sqcap \mathsf{P}'}$.
Here $\mathsf{E} \sqcap \mathsf{P}' = \mathsf{E} \sqcap \mathsf{P} \sqcap \mathsf{P}'$.

Case (Typ new) By I.H. $\Gamma \vdash_{\mathsf{P}'} x : \tau^{\mathsf{E} \sqcap \mathsf{P}'}$
Then $\mathsf{S} \sqsubseteq \mathsf{E} \sqcap \mathsf{P}'$.
By (TYP NEW) $\Gamma \vdash_{\mathsf{P}'} \mathsf{new}(x \# \mathsf{S}) : \mathbf{Obj}(\tau^{\mathsf{S}})^{\mathsf{P}'}$.
Here $\mathsf{P}' = \mathsf{P} \sqcap \mathsf{P}'$.

Case (Typ fork) Let $T = \tau^{\mathsf{E}}$.
By I.H. $\Gamma \vdash_{\mathsf{P}'} a : \_$ and $\Gamma \vdash_{\mathsf{P}'} b : \tau^{\mathsf{E} \sqcap \mathsf{P}'}$.
By (TYP FORK) $\Gamma \vdash_{\mathsf{P}'} a \mathbin{↱} b : \tau^{\mathsf{E} \sqcap \mathsf{P}'}$.

Case (Typ store) By (TYP STORE) $\Gamma \vdash_{\mathsf{P}'} \omega \overset{\mathsf{O}}{\mapsto} x : \_^{\mathsf{P}'}$.
Here $\mathsf{P}' = \mathsf{P} \sqcap \mathsf{P}'$.

Case (Typ un/protect) By I.H. $\Gamma \vdash_{\mathsf{P}'} \omega : \mathbf{Obj}(\tau^{\mathsf{S}})^{\mathsf{E} \sqcap \mathsf{P}'}$
and if $*\mathsf{P}'$ then $*\mathsf{P}$, then $*\mathsf{E}$, and then $*(\mathsf{E} \sqcap \mathsf{P}')$.
By (TYP UN/PROTECT) $\Gamma \vdash_{\mathsf{P}'} \langle \mathsf{O} \rangle\, \omega : \mathbf{Unit}^{\mathsf{P}'}$.

Case (Typ write) By I.H. $\Gamma \vdash_{\mathsf{P}'} \omega : \mathbf{Obj}(\tau^{\mathsf{S}})^{\mathsf{E} \sqcap \mathsf{P}'}$ and $\Gamma \vdash_{\mathsf{P}'} x : \tau^{\mathsf{E}' \sqcap \mathsf{P}'}$

and if $*\mathsf{L}'_r$ then $*\mathsf{P}$, then $*\mathsf{E}$, and then $*(\mathsf{E} \sqcap \mathsf{P}')$

and $\mathsf{S} \sqcap \mathsf{P}' \sqsubseteq \mathsf{E}' \sqcap \mathsf{P}'$.

If $\mathsf{S} \sqsubseteq \mathsf{P}'$ then $\mathsf{S} \sqsubseteq \mathsf{E}' \sqcap \mathsf{P}'$.

By (TYP WRITE) $\Gamma \vdash_{\mathsf{P}'} \omega := x : \mathbf{Unit}^{\mathsf{P}'}$.

Otherwise $\mathsf{P}' \sqsubset \mathsf{S}$, so that $*\mathsf{S}$.

Because $\mathsf{S} \sqsubseteq \mathsf{E}' \sqsubseteq \mathsf{P}$, we have $*\mathsf{P}$ and thus $*\mathsf{E}$.

By (TYP VALUE) $\omega : \mathbf{Obj}(\tau^{\mathsf{S}})^{\mathsf{E}''} \in \Gamma$ and $\mathsf{E} \sqsubseteq \mathsf{E}''$.

Then $*\mathsf{E}''$.

By (TYP WRITE STUCK) $\Gamma \vdash_{\mathsf{P}} \omega := x : \mathbf{Stuck}$.

By (TYP SUBSUMPTION STUCK-II) $\Gamma \vdash_{\mathsf{P}} \omega := x : \mathbf{Unit}^{\mathsf{P}'}$.

Case (Typ execute) $\mathsf{P}' \sqsubset \mathsf{P} \sqsubseteq \mathsf{P}'' \sqcap \mathsf{S}$

and if $*\mathsf{L}'_r$ then $*\mathsf{P}$, and then $*\mathsf{E}$.

By (TYP EXECUTE) $\Gamma \vdash_{\mathsf{P}'} \mathsf{exec}\ \omega : \tau^{\mathsf{E}' \sqcap \mathsf{P}'}$.

Here $\mathsf{E}' \sqcap \mathsf{P}' = \mathsf{E}' \sqcap \mathsf{P} \sqcap \mathsf{P}'$.

Case (Typ read) If $*(\mathsf{S} \sqcap \mathsf{P}')$ then $*(\mathsf{S} \sqcap \mathsf{P})$, and then $*\mathsf{E}$.

By (TYP READ) $\Gamma \vdash_{\mathsf{P}'} !\omega : \tau^{\mathsf{S} \sqcap \mathsf{P}'}$.

Here $\mathsf{S} \sqcap \mathsf{P}' = \mathsf{S} \sqcap \mathsf{P} \sqcap \mathsf{P}'$.

Case (Typ limit) Let $T = \tau^{\mathsf{E}}$.

Then $\mathsf{E} \sqsubseteq \mathsf{P}''$.

If $\mathsf{P}'' \sqsubseteq \mathsf{P}'$ then

$\mathsf{E} \sqcap \mathsf{P}' = \mathsf{E}$.

By (TYP LIMIT) $\Gamma \vdash_{\mathsf{P}'} [\mathsf{P}''] \, a : \tau^{\mathsf{E} \sqcap \mathsf{P}'}$.

Otherwise $\mathsf{P}' \sqsubset \mathsf{P}''$.

By (TYP ESCALATE STUCK) $\Gamma \vdash_{\mathsf{P}'} [\mathsf{P}''] \, a : \mathbf{Stuck}$.

By (TYP SUBSUMPTION STUCK-II) $\Gamma \vdash_{\mathsf{P}'} [\mathsf{P}''] \, a : \tau^{\mathsf{E} \sqcap \mathsf{P}'}$.

Case (Typ evaluate) Let $T = \tau^{\mathsf{E}}$.

By I.H. $\Gamma \vdash_{\mathsf{P}'} a : T''$ and $\Gamma, x : T'' \vdash_{\mathsf{P}'} b : \tau^{\mathsf{E} \sqcap \mathsf{P}'}$.

By (TYP EVALUATE) $\Gamma \vdash_{\mathsf{P}'} \mathsf{let}\ x = a\ \mathsf{in}\ b : \tau^{\mathsf{E} \sqcap \mathsf{P}'}$.

Case (Typ substitute) Let $T = \tau^{\mathsf{E}}$.

By I.H. $\Gamma, x : T' \vdash_{\mathsf{P}'} a : \tau^{\mathsf{E} \sqcap \mathsf{P}'}$.

By (TYP SUBSTITUTE) $\Gamma \vdash_{\mathsf{P}'} (\nu x / \mu @ \mathsf{P}')\ a : \tau^{\mathsf{E} \sqcap \mathsf{P}'}$. ◀

Lemma D.3.1 (Bind)**.** *Suppose that* $a = a'\{x/y\}$. *Then* $\Gamma \vdash_{\mathsf{P}} a : \_$ *if and only if* $\Gamma \vdash_{\mathsf{P}} (\nu x / y @ \mathsf{P})\ a'$.

Proof. By induction on the structure of a'. ◀

Theorem 6.4.4 (Type preservation)**.** *Suppose that* $\Gamma \vdash_{\mathsf{P}} \sigma$ *and* $\Gamma \vdash_{\mathsf{P}} a : \_$. *Then*

1. *If* $a \equiv b$ *then* $\Gamma \vdash_{\mathsf{P}} b : \_$.
2. *If* $a \xrightarrow{\mathsf{P};\sigma} b$ *then* $\Gamma \vdash_{\mathsf{P}} b : \_$.

Proof of (1). Preservation under $\equiv$ by induction on the structure of derivations.

Case (Struct substitution)

Let $\sigma'' = \{x / \mu @ \mathsf{L}''\} \cup \sigma$.

- $(\nu x / \mu @ \mathsf{L}'')$ let $y = \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'}$ in $b' \equiv$ let $y = \mathcal{E}_{\mathsf{L};\sigma}[\![(\nu x / \mu @ \mathsf{L}'')\ a']\!]_{\mathsf{L}';\sigma'}$ in b'
  and $\Gamma' \vdash_{\mathsf{L}} (\nu x / \mu @ \mathsf{L}'')$ let $y = \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'}$ in $b' : T$.

  By (TYP SUBSTITUTE) and (TYP EVALUATE)

  $\Gamma' \vdash_{\mathsf{L}''} \mu : T''$

  and $\Gamma', x : T'' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} : T'''$

  and $\Gamma', x : T'', y : T''' \vdash_{\mathsf{L}} b' : T$.

  By (TYP SUBSTITUTE) and S.R.

  $\Gamma' \vdash_{\mathsf{L}} (\nu x / \mu @ \mathsf{L}'')\ \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} : T'''$

  and $\Gamma', y : T''' \vdash_{\mathsf{L}} b' : T$.

  By I.H. $\Gamma' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![(\nu x / \mu @ \mathsf{L}'')\ a']\!]_{\mathsf{L}';\sigma'} : T'''$.

  By (TYP EVALUATE)

  $\Gamma' \vdash_{\mathsf{L}}$ let $y = \mathcal{E}_{\mathsf{L};\sigma}[\![(\nu x / \mu @ \mathsf{L}'')\ a']\!]_{\mathsf{L}';\sigma'}$ in $b' : T$.

- $(\nu x / \mu @ \mathsf{L}'')\ \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} \Rsh b' \equiv \mathcal{E}_{\mathsf{L};\sigma}[\![(\nu x / \mu @ \mathsf{L}'')\ a']\!]_{\mathsf{L}';\sigma'} \Rsh b'$
  and $\Gamma' \vdash_{\mathsf{L}} (\nu x / \mu @ \mathsf{L}'')\ \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} \Rsh b' : T$.

  By (TYP SUBSTITUTE) and (TYP FORK)

$\Gamma' \vdash_{\mathsf{L}''} \mu : T''$

and $\Gamma', x : T'' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} : T'''$

and $\Gamma', x : T'' \vdash_{\mathsf{L}} b' : T$.

By (TYP SUBSTITUTE) and S.R.

$\Gamma' \vdash_{\mathsf{L}} (\nu x/\mu@\mathsf{L}'')\ \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} : T'''$

and $\Gamma' \vdash_{\mathsf{L}} b' : T$.

By I.H. $\Gamma' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![(\nu x/\mu@\mathsf{L}'')\ a']\!]_{\mathsf{L}';\sigma'} : T'''$.

By (TYP FORK)

$\Gamma' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![(\nu x/\mu@\mathsf{L}'')\ a']\!]_{\mathsf{L}';\sigma'} \Rsh b' : T$.

- $(\nu x/\mu@\mathsf{L}'')\ b' \Rsh \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} \equiv b' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![(\nu x/\mu@\mathsf{L}'')\ a']\!]_{\mathsf{L}';\sigma'}$

  and $\Gamma' \vdash_{\mathsf{L}} (\nu x/\mu@\mathsf{L}'')\ b' \Rsh \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} : T$.

  By (TYP SUBSTITUTE) and (TYP FORK)

  $\Gamma' \vdash_{\mathsf{L}''} \mu : T''$

  and $\Gamma', x : T'' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} : T$

  and $\Gamma', x : T'' \vdash_{\mathsf{L}} b' : T'''$.

  By (TYP SUBSTITUTE) and S.R.

  $\Gamma' \vdash_{\mathsf{L}} (\nu x/\mu@\mathsf{L}'')\ \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} : T$

  and $\Gamma' \vdash_{\mathsf{L}} b' : T'''$.

  By I.H. $\Gamma' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![(\nu x/\mu@\mathsf{L}'')\ a']\!]_{\mathsf{L}';\sigma'} : T$.

  By (TYP FORK)

  $\Gamma' \vdash_{\mathsf{L}} b' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![(\nu x/\mu@\mathsf{L}'')\ a']\!]_{\mathsf{L}';\sigma'} : T$.

- $(\nu x/\mu@\mathsf{L}'')\ (\nu y/\mu'@\mathsf{L}''')\ \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} \equiv (\nu y/\mu'@\mathsf{L}''')\ \mathcal{E}_{\mathsf{L};\sigma}[\![(\nu x/\mu@\mathsf{L}'')\ a']\!]_{\mathsf{L}';\sigma'}$

  and $\Gamma' \vdash_{\mathsf{L}} (\nu x/\mu@\mathsf{L}'')\ (\nu y/\mu'@\mathsf{L}''')\ \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} :\ T$.

  By (TYP SUBSTITUTE) and (TYP SUBSTITUTE)

  $\Gamma' \vdash_{\mathsf{L}''} \mu : T''$

  and $\Gamma', x : T'' \vdash_{\mathsf{L}'''} v : T'''$

  and $\Gamma', x : T'', y : T''' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} : T$.

  By (TYP SUBSTITUTE) and S.R.

  $\Gamma',, y : T''' \vdash_{\mathsf{L}''} u : T''$

  and $\Gamma' \vdash_{\mathsf{L}'''} \mu' : T'''$

and $\Gamma', y : T''', x : T'' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} : T$.

By (TYP SUBSTITUTE)

$\Gamma', y : T''' \vdash_{\mathsf{L}} (\nu x/\mu@\mathsf{L}'')\ \mathcal{E}_{\mathsf{L};\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} : T$.

By I.H. $\Gamma', y : T''' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![(\nu x/\mu@\mathsf{L}'')\ a']\!]_{\mathsf{L}';\sigma'} : T$.

By (TYP SUBSTITUTE)

$\Gamma' \vdash_{\mathsf{L}} (\nu y/\mu'@\mathsf{L}''')\ \mathcal{E}_{\mathsf{L};\sigma}[\![(\nu x/\mu@\mathsf{L}'')\ a']\!]_{\mathsf{L}';\sigma'} : T$.

- $(\nu x/\mu@\mathsf{L}'')\ [\mathsf{L}''']\ \mathcal{E}_{\mathsf{L}''';\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} \equiv [\mathsf{L}''']\ \mathcal{E}_{\mathsf{L}''';\sigma}[\![(\nu x/\mu@\mathsf{L}'')\ a']\!]_{\mathsf{L}';\sigma'}$

  and $\Gamma' \vdash_{\mathsf{L}} (\nu x/\mu@\mathsf{L}'')\ [\mathsf{L}''']\ \mathcal{E}_{\mathsf{L}''';\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} : T$.

  By (TYP SUBSTITUTE) and (TYP LIMIT)

  $\Gamma' \vdash_{\mathsf{L}''} \mu : T''$

  and $\Gamma', x : T'' \vdash_{\mathsf{L}'''} \mathcal{E}_{\mathsf{L}''';\sigma''}[\![a']\!]_{\mathsf{L}';\sigma'} : T$.

  By (TYP SUBSTITUTE)

  $\Gamma' \vdash_{\mathsf{L}'''} (\nu x/\mu@\mathsf{L}'')\ \mathcal{E}_{\mathsf{L}''';\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} : T$.

  By I.H. $\Gamma' \vdash_{\mathsf{L}'''} \mathcal{E}_{\mathsf{L}''';\sigma}[\![(\nu x/\mu@\mathsf{L}'')\ a']\!]_{\mathsf{L}';\sigma'} : T$.

  By (TYP LIMIT)

  $\Gamma' \vdash_{\mathsf{L}} [\mathsf{L}''']\ \mathcal{E}_{\mathsf{L}''';\sigma}[\![(\nu x/\mu@\mathsf{L}'')\ a']\!]_{\mathsf{L}';\sigma'} : T$.

Case (Struct fork)

- $a'' \Rsh \text{let } x = \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} \text{ in } b' \equiv \text{let } x = \mathcal{E}_{\mathsf{L};\sigma}[\![a'' \Rsh a']\!]_{\mathsf{L}} \text{ in } b'$

  and $\Gamma' \vdash_{\mathsf{L}} a'' \Rsh \text{let } x = \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} \text{ in } b' : T$.

  By (TYP FORK) and (TYP EVALUATE)

  $\Gamma' \vdash_{\mathsf{L}} a'' : T''$

  and $\Gamma' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} : T'''$

  and $\Gamma', x : T''' \vdash_{\mathsf{L}} b' : T$.

  By (TYP FORK)

  $\Gamma' \vdash_{\mathsf{L}} a'' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} : T'''$

  and $\Gamma', x : T''' \vdash_{\mathsf{L}} b' : T$.

  By I.H. $\Gamma' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a'' \Rsh a']\!]_{\mathsf{L}} : T'''$.

  By (TYP EVALUATE)

  $\Gamma' \vdash_{\mathsf{L}} \text{let } x = \mathcal{E}_{\mathsf{L};\sigma}[\![a'' \Rsh a']\!]_{\mathsf{L}} \text{ in } b' : T$.

- $a'' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} \Rsh b' \equiv \mathcal{E}_{\mathsf{L};\sigma}[\![a'' \Rsh a']\!]_{\mathsf{L}} \Rsh b'$
  and $\Gamma' \vdash_{\mathsf{L}} a'' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} \Rsh b' : T$.

  By (TYP FORK) and (TYP FORK)

  $\Gamma' \vdash_{\mathsf{L}} a'' : T''$

  and $\Gamma' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} : T'''$

  and $\Gamma' \vdash_{\mathsf{L}} b' : T$.

  By (TYP FORK)

  $\Gamma' \vdash_{\mathsf{L}} a'' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} : T'''$

  and $\Gamma' \vdash_{\mathsf{L}} b' : T$.

  By I.H. $\Gamma' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a'' \Rsh a']\!]_{\mathsf{L}} : T'''$.

  By (TYP FORK)

  $\Gamma' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a'' \Rsh a']\!]_{\mathsf{L}} \Rsh b' : T$.

- $a'' \Rsh b' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} \equiv b' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![a'' \Rsh a']\!]_{\mathsf{L}}$
  and $\Gamma' \vdash_{\mathsf{L}} a'' \Rsh b' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} : T$.

  By (TYP FORK) and (TYP FORK)

  $\Gamma' \vdash_{\mathsf{L}} a'' : T''$

  and $\Gamma' \vdash_{\mathsf{L}} b' : T'''$

  and $\Gamma' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} : T$.

  By (TYP FORK)

  $\Gamma' \vdash_{\mathsf{L}} b' : T'''$

  and $\Gamma' \vdash_{\mathsf{L}} a'' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} : T$.

  By I.H. $\Gamma' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a'' \Rsh a']\!]_{\mathsf{L}} : T$.

  By (TYP FORK)

  $\Gamma' \vdash_{\mathsf{L}} b' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![a'' \Rsh a']\!]_{\mathsf{L}} : T$.

- $a'' \Rsh (\nu x / \mu @ \mathsf{L}')\, \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} \equiv (\nu x / u @ \mathsf{L}')\, \mathcal{E}_{\mathsf{L};\sigma}[\![a'' \Rsh a']\!]_{\mathsf{L}}$
  and $\Gamma' \vdash_{\mathsf{L}} a'' \Rsh (\nu x / u @ \mathsf{L}')\, \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} : T$.

  By (TYP FORK) and (TYP SUBSTITUTE)

  $\Gamma' \vdash_{\mathsf{L}} a'' : T''$

  and $\Gamma' \vdash_{\mathsf{L}';\sigma'} \mu : T$

  and $\Gamma', x : T''' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} : T$.

By S.R. $\Gamma', x : T''' \vdash_{\mathsf{L}} a'' : T''$.

By (TYP FORK)

$\Gamma', x : T''' \vdash_{\mathsf{L}} a'' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}} : T$.

By I.H. $\Gamma', x : T''' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a'' \Rsh a']\!]_{\mathsf{L}} : T$.

By (TYP SUBSTITUTE)

$\Gamma' \vdash_{\mathsf{L}} (\nu x / u@\mathsf{L}')\ \mathcal{E}_{\mathsf{L};\sigma}[\![a'' \Rsh a']\!]_{\mathsf{L}} : T$.

Case (Struct store) $\omega \overset{\mathsf{L}''}{\mapsto} u \Rsh [\mathsf{L}']\ a' \equiv [\mathsf{L}']\ (\omega \overset{\mathsf{L}''}{\mapsto} u \Rsh a')$
and $\Gamma' \vdash_{\mathsf{L}} \omega \overset{\mathsf{L}''}{\mapsto} u \Rsh [\mathsf{L}']\ a' : T$.

By (TYP FORK)

$\Gamma' \vdash_{\mathsf{L}} \omega \overset{\mathsf{L}''}{\mapsto} u : \_$

and $\Gamma' \vdash_{\mathsf{L}} [\mathsf{L}']\ a' : T$.

By (TYP LIMIT)

$\Gamma' \vdash_{\mathsf{L}'} \omega \overset{\mathsf{L}''}{\mapsto} u : \_$

and $\Gamma' \vdash_{\mathsf{L}'} a' : T$.

By (TYP FORK) $\Gamma' \vdash_{\mathsf{L}'} \omega \overset{\mathsf{L}''}{\mapsto} u \Rsh a' : T$.

By (TYP LIMIT) $\Gamma' \vdash_{\mathsf{L}} [\mathsf{L}']\ \omega \overset{\mathsf{L}''}{\mapsto} u \Rsh a' : T$.

Case (Struct bind)

By Lemma D.3.1. ◀

Proof of (2). Preservation under $\longrightarrow$ by induction on the structure of derivations.

Case (Reduct evaluate) $\Gamma \vdash_{\mathsf{L}} \text{let } x = u \text{ in } a' : T$.

By (TYP EVALUATE)

$\Gamma \vdash_{\mathsf{L}} u : T''$

and $\Gamma, x : T'' \vdash_{\mathsf{L}} a' : T$.

By (TYP SUBSTITUTE) $\Gamma \vdash_{\mathsf{L}} (\nu x / u@\mathsf{L})\ a' : T$.

Case (Reduct new) $\Gamma \vdash_{\mathsf{P}} \mathsf{new}(x \# \mathsf{S}) : T$.

By (TYP NEW)

$\Gamma \vdash_{\mathsf{P}} x : \tau^{\mathsf{E}}$,

$\mathsf{S} \sqsubseteq \mathsf{E}$,

and $T = \mathbf{Obj}(\tau^{\mathsf{S}})^{\mathsf{P}}$.

By (TYP STORE) $\Gamma, \omega : T \vdash_{\mathsf{P}} \omega \overset{\mathsf{P}}{\mapsto} x : \_$.

By (TYP FORK) $\Gamma, \omega : T \vdash_{\mathsf{P}} \omega \overset{\mathsf{P}}{\mapsto} x \Rsh \omega : T$.

By (TYP SUBSTITUTE)

$\Gamma \vdash_{\mathsf{P}} (\nu\omega/\mathsf{new}(x \# \mathsf{S})@\mathsf{P})\ (\omega \overset{\mathsf{P}}{\mapsto} x \Rsh \omega) : T$.

Case (Reduct read) $\Gamma \vdash_{\mathsf{L}} \omega \overset{\mathsf{O}}{\mapsto} x \Rsh\ !\omega' : \tau^{\mathsf{E}}$.

By (TYP FORK)

$\Gamma \vdash_{\mathsf{L}} \omega \overset{\mathsf{O}}{\mapsto} x : \_$.

By (TYP STORE) $\Gamma \vdash_{\mathsf{L}} x : \_$.

By (TYP FORK) $\Gamma \vdash_{\mathsf{L}} \omega \overset{\mathsf{O}}{\mapsto} x \Rsh x : \_$.

Case (Reduct write) $\Gamma \vdash_{\mathsf{L}} \omega \overset{\mathsf{O}}{\mapsto} x \Rsh \omega' := x' : \mathbf{Unit}^{\mathsf{L}}$.

By (TYP FORK)

$\Gamma \vdash_{\mathsf{L}} \omega \overset{\mathsf{O}}{\mapsto} x : \_$

and $\Gamma \vdash_{\mathsf{L}} \omega' := x' : \mathbf{Unit}^{\mathsf{L}}$

and $\mathsf{O} \sqsubseteq \mathsf{L}$.

By (TYP STORE), (TYP WRITE), and $\Gamma \vdash \sigma$

$\omega : \mathbf{Obj}(\tau^{\mathsf{S}})^{\text{-}} \in \Gamma$,

$\mathsf{S} \sqsubseteq \mathsf{O}$,

$\Gamma \vdash_{\mathsf{L}} \omega' : \mathbf{Obj}(\tau^{\mathsf{S}})^{\mathsf{E}}$,

$\Gamma \vdash_{\mathsf{L}} x' : \tau^{\mathsf{E}'}$,

and $\mathsf{S} \sqsubseteq \mathsf{E}'$.

By (TYP STORE) $\Gamma \vdash_{\mathsf{L}} \omega \overset{\mathsf{O}}{\mapsto} x' : \_$.

By (TYP UNIT) $\Gamma \vdash_{\mathsf{L}} \omega \overset{\mathsf{O}}{\mapsto} \mathsf{unit} : \mathbf{Unit}^{\mathsf{L}}$.

By (TYP FORK) $\Gamma \vdash_{\mathsf{L}} \omega \overset{\mathsf{O}}{\mapsto} x' \Rsh \mathsf{unit} : \mathbf{Unit}^{\mathsf{L}}$.

Case (Reduct execute) $\Gamma \vdash_{\mathsf{L}} \omega \overset{\mathsf{O}}{\mapsto} x \Rsh \mathsf{exec}\ \omega' : \_$.

By (TYP FORK)

$\Gamma \vdash_{\mathsf{L}} \omega \overset{\mathsf{O}}{\mapsto} x : \_$

and $\Gamma \vdash_{\mathsf{L}} \mathsf{exec}\ \omega' : \_$.

By (TYP STORE), (TYP EXECUTE), and $\Gamma \vdash \sigma$

$\Gamma \vdash_{\mathsf{P}'} \mathsf{pack}(f) : \nabla_{\mathsf{P}}.\ \mathbf{Bin}(T)^{\mathsf{P}'}$ for some P',

$x : \nabla_{\mathsf{P}}.\ \mathbf{Bin}(T)^{\mathsf{E}} \in \Gamma$,

$\omega : \mathbf{Obj}(\nabla_{\mathsf{P}}.\ \mathbf{Bin}(T)^{\mathsf{S}})^{-} \in \Gamma$,

$\mathsf{S} \sqsubseteq \mathsf{O} \sqcap \mathsf{E}$,

and $\mathsf{L} \sqsubseteq \mathsf{P} \sqcap \mathsf{S}$.

By (TYP PACK) $\Gamma \vdash_{\mathsf{P}} f : \_$.

By (TYP SUBSUMPTION PROCESS LABEL) $\Gamma \vdash_{\mathsf{L}} f : \_$.

By (TYP FORK) $\Gamma \vdash_{\mathsf{L}} \omega \overset{\mathsf{O}}{\mapsto} x \Rsh f : \_$.

Case (Reduct un/protect) $\Gamma \vdash_{\mathsf{L}} \omega \overset{\mathsf{O}}{\mapsto} x \Rsh \langle \mathsf{L}' \rangle\ \omega' : \mathbf{Unit}^{\mathsf{L}}$.

By (TYP FORK)

$\Gamma \vdash_{\mathsf{L}} \omega \overset{\mathsf{O}}{\mapsto} x : \_$

and $\Gamma \vdash_{\mathsf{L}} \langle \mathsf{L}' \rangle\ \omega' : \mathbf{Unit}^{\mathsf{L}}$

$\mathsf{O} \sqcup \mathsf{L}' \sqsubseteq \mathsf{L}$.

By (TYP STORE), (TYP UN/PROTECT), and $\Gamma \vdash \sigma$,

$\omega : \mathbf{Obj}(\tau^{\mathsf{S}})^{-} \in \Gamma$,

$\mathsf{S} \sqsubseteq \mathsf{O}$,

$\Gamma \vdash_{\mathsf{L}} \omega' : \mathbf{Obj}(\tau^{\mathsf{S}})^{-}$,

and $\mathsf{S} \sqsubseteq \mathsf{L}'$.

By (TYP STORE) $\Gamma \vdash_{\mathsf{L}} \omega \overset{\mathsf{L}'}{\mapsto} x : \_$.

By (TYP UNIT) $\Gamma \vdash_{\mathsf{L}} \mathsf{unit} : \mathbf{Unit}^{\mathsf{L}}$.

By (TYP FORK) $\Gamma \vdash_{\mathsf{L}} \omega \overset{\mathsf{L}'}{\mapsto} x \Rsh \mathsf{unit} : \mathbf{Unit}^{\mathsf{L}}$.

Case (Reduct context)

- $\mathsf{let}\ x = \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}';\sigma'}\ \mathsf{in}\ b' \xrightarrow{\mathsf{L};\sigma} \mathsf{let}\ x = \mathcal{E}_{\mathsf{L};\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'}\ \mathsf{in}\ b'$,
  $a' \xrightarrow{\mathsf{L}';\sigma'} a''$,
  and $\Gamma \vdash_{\mathsf{L}} \mathsf{let}\ x = \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}';\sigma'}\ \mathsf{in}\ b' : T$.
  By (REDUCT CONTEXT) and (TYP EVALUATE)
  $\mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} \xrightarrow{\mathsf{L};\sigma} \mathcal{E}_{\mathsf{L}}[\![a'']\!]_{\mathsf{L}';\sigma'}$,

$\Gamma \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} : T''$,

and $\Gamma, x : T'' \vdash_{\mathsf{L}} b' : T$.

By I.H. $\Gamma \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'} : T''$.

By (TYP EVALUATE)

$\Gamma \vdash_{\mathsf{L}} \text{let } x = \mathcal{E}_{\mathsf{L};\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'} \text{ in } b' : T$.

- $\mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} \Rsh b' \xrightarrow{\mathsf{L};\sigma} \mathcal{E}_{\mathsf{L};\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'} \Rsh b'$,
  $a' \xrightarrow{\mathsf{L}';\sigma'} a''$,
  and $\Gamma \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} \Rsh b' : T$.

  By (REDUCT CONTEXT) and (TYP FORK)

  $\mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} \xrightarrow{\mathsf{L};\sigma} \mathcal{E}_{\mathsf{L};\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'}$,

  $\Gamma \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} : T''$,

  and $\Gamma \vdash_{\mathsf{L}} b' : T$.

  By I.H. $\Gamma \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'} : T''$.

  By (TYP FORK)

  $\Gamma \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'} \Rsh b' : T$.

- $b' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} \xrightarrow{\mathsf{L};\sigma} b' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'}$,
  $a' \xrightarrow{\mathsf{L}';\sigma'} a''$,
  and $\Gamma \vdash_{\mathsf{L}} b' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} : T$.

  By (REDUCT CONTEXT) and (TYP FORK)

  $\mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} \xrightarrow{\mathsf{L};\sigma} \mathcal{E}_{\mathsf{L};\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'}$,

  $\Gamma \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} : T$,

  and $\Gamma \vdash_{\mathsf{L}} b' : T''$.

  By I.H. $\Gamma \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'} : T$.

  By (TYP FORK)

  $\Gamma \vdash_{\mathsf{L}} b' \Rsh \mathcal{E}_{\mathsf{L};\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'} : T$.

- $(\nu x/u@\mathsf{L}'')\, \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} \xrightarrow{\mathsf{L};\sigma} (\nu x/u@\mathsf{L}'')\, \mathcal{E}_{\mathsf{L};\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'}$,
  $a' \xrightarrow{\mathsf{L}';\sigma'} a''$,
  and $\Gamma \vdash_{\mathsf{L}} (\nu x/u@\mathsf{L}'')\, \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} : T$.

  By (REDUCT CONTEXT) and (TYP SUBSTITUTE)

$\mathcal{E}_{\mathsf{L}}[\![a']\!]_{\mathsf{L}';\sigma'} \xrightarrow{\mathsf{L};\sigma} \mathcal{E}_{\mathsf{L};\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'}$,
and $\Gamma \vdash_{\mathsf{L}''} u : T''$,
and $\Gamma, x : T'' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} : T$.
By I.H. $\Gamma, x : T'' \vdash_{\mathsf{L}} \mathcal{E}_{\mathsf{L};\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'} : T$.
By (TYP SUBSTITUTE)
$\Gamma \vdash_{\mathsf{L}} (\nu x / u@\mathsf{L}'')\ \mathcal{E}_{\mathsf{L};\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'} : T$.

- $[\mathsf{L}'']\ \mathcal{E}_{\mathsf{L}'';\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} \xrightarrow{\mathsf{L};\sigma} [\mathsf{L}'']\ \mathcal{E}_{\mathsf{L}'';\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'}$,
  $a' \xrightarrow{\mathsf{L}';\sigma'} a''$,
  and $\Gamma \vdash_{\mathsf{L}} [\mathsf{L}'']\ \mathcal{E}_{\mathsf{L}'';\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} : T$.
  By (REDUCT CONTEXT) and (TYP LIMIT)
  $\mathcal{E}_{\mathsf{L}'';\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} \xrightarrow{\mathsf{L}'';\sigma} \mathcal{E}_{\mathsf{L}'';\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'}$
  and $\Gamma \vdash_{\mathsf{L}''} \mathcal{E}_{\mathsf{L}'';\sigma}[\![a']\!]_{\mathsf{L}';\sigma'} : T$.
  By I.H. $\Gamma \vdash_{\mathsf{L}''} \mathcal{E}_{\mathsf{L}'';\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'} : T$.
  By (TYP LIMIT)
  $\Gamma \vdash_{\mathsf{L}} [\mathsf{L}'']\ \mathcal{E}_{\mathsf{L}'';\sigma}[\![a'']\!]_{\mathsf{L}';\sigma'} : T$.

Case (Reduct congruence) $\Gamma \vdash_{\mathsf{L}} a : T$,
$a \equiv a'$,
$a' \xrightarrow{\mathsf{L};\sigma} b'$,
and $b' \equiv b$.
By Theorem 6.4.4(1) $\Gamma \vdash_{\mathsf{L}} a' : \_$.
By I.H. $\Gamma \vdash_{\mathsf{L}} b' : \_$.
So by Theorem 6.4.4(1) $\Gamma \vdash_{\mathsf{L}} b : \_$. ◀

Theorem 6.4.7 (Enforcement of strong DFI)**.** *Let Ω be the set of objects whose contents are trusted beyond L in Γ. Suppose that $\Gamma \vdash_{\top} a : \_$ despite C, where $\mathsf{C} \sqsubseteq \mathsf{L}$. Then a protects Ω from L despite C.*

Proof. Let e be any C-adversary $[\mathsf{C}]\ e'$.
By Proposition 6.4.2 $\Gamma \vdash_{\top} e : \_$.
By (TYP FORK) $\Gamma \vdash_{\top} a \Rsh e :_{\_} \_$.
Suppose that $\omega \in \Omega$. We need to prove that there are no σ and x such that $a \Rsh [\mathsf{C}]\ e' \xrightarrow{\top}$

* $\mathcal{E}_{\top;\varnothing}[\![\omega \mathrel{\dot{\mapsto}} x]\!]_{\top;\sigma}$ and $x \overset{\sigma}{\blacktriangledown} \mathsf{L}$. Assume otherwise.

By Theorem 6.4.4 there exists Γ' extending Γ such that

$\Gamma' \vdash \sigma$ and $\Gamma' \vdash_\top \omega \mathrel{\dot{\mapsto}} x : \_$.

By (TYP STORE) $\omega : \mathbf{Obj}(\tau^{\mathsf{S}})^- \in \Gamma'$ such that $\mathsf{S} \sqsubseteq \mathsf{E}$.

We proceed by induction on the derivation of $x \overset{\sigma}{\blacktriangledown} \mathsf{L}$.

Case $\mathsf{P} \sqsubseteq \mathsf{L}$.

For some τ and E, $\Gamma' \vdash_\mathsf{P} \mu : \tau^\mathsf{E}$.

Then $\mathsf{E} \sqsubseteq \mathsf{P}$ and by (TYP VALUE) $\Gamma' \vdash_\top x : \tau^\mathsf{E}$.

Then $\mathsf{E} \sqsubseteq \mathsf{L}$.

Then $\mathsf{S} \sqsubseteq \mathsf{L}$.

But by assumptions $\mathsf{S} \sqsupset \mathsf{L}$ (contradiction).

Case $\mu \equiv y$ for some y and $y \overset{\sigma}{\blacktriangledown} \mathsf{L}$.

By I.H. $\Gamma' \vdash_\top y : \tau^\mathsf{E}$ for some E such that $\mathsf{E} \sqsubseteq \mathsf{L}$.

Then $\mathsf{S} \sqsubseteq \mathsf{L}$.

But by assumptions $\mathsf{S} \sqsupset \mathsf{L}$ (contradiction). ◀

Theorem 6.4.8 (Redundancy of execution control)**.** *Suppose that* $\Gamma \vdash_\top a : \_\ \mathsf{despite}\ \mathsf{C}$ *and* $a \xrightarrow{\top;\varnothing}{}^\star \mathcal{E}_{\top;\varnothing}[\![\omega \overset{\mathsf{O}}{\mapsto} x \Rsh \mathsf{exec}\,\omega']\!]_{\mathsf{P};\sigma}$ *such that* $\omega \overset{\sigma}{=} \omega'$*, and* $\mathsf{P} \sqsupset \mathsf{C}$*. Then* $\mathsf{P} \sqsubseteq \mathsf{O}$.

Proof. The proof is by inspection of Case (Reduct execute) in the proof of Theorem 6.4.4. Recalling that case (where L is the process label): $\mathsf{L} \sqsubseteq \mathsf{S} \sqsubseteq \mathsf{O}$. ◀

D.4 Correctness of distributed access control implementations

We show that $\mathbb{IMP}$ is secure, safe, and fully abstract. Simulation relations for our proofs are shown below. All these relations are closed under $\equiv$. Here η_1 and η_2 rename the public interfaces of NS^d and IS^d and η_3 renames the private authentication keys K_{AS} and $\overline{K}_{AS}$.

$$\begin{aligned}
\eta_1 &\triangleq [\alpha_j \mapsto \alpha_{j?}, \beta_j \mapsto \beta_{j?}, \gamma_j \mapsto \gamma_{j?} \mid j \in \mathbb{N}\backslash\mathcal{I}] \\
\eta_2 &\triangleq [\alpha^\circ_j \mapsto \alpha^\circ_{j?}, \beta^\circ_j \mapsto \beta^\circ_{j?}, \gamma^\circ_j \mapsto \gamma^\circ_{j?} \mid j \in \mathbb{N}\backslash\mathcal{I}] \\
\eta_3 &\triangleq [a \mapsto K_? \mid a \in \{K_{AS}, \overline{K}_{AS}\}]
\end{aligned}$$

These renamings map to names in $\mathcal{A}$, a set of special names whose uses in well-formed code are either disciplined or forbidden.

$$\mathcal{A} \triangleq \{\alpha_i, \beta_i, \gamma_i \mid i \in \mathcal{I}\} \cup \{\alpha_{j?}, \beta_{j?}, \gamma_{j?}, \alpha^\circ_{j?}, \beta^\circ_{j?}, \gamma^\circ_{j?} \mid j \in \mathbb{N}\backslash\mathcal{I}\} \cup \{K_{AS}, \overline{K}_{AS}, K_?\}$$

The names in $\{\alpha_{j?}, \beta_{j?}, \gamma_{j?}, \alpha^\circ_{j?}, \beta^\circ_{j?}, \gamma^\circ_{j?} \mid j \in \mathbb{N}\backslash\mathcal{I}\} \cup \{K_?\}$ are invented to simplify proofs below. In particular, the purpose of η_1 and η_2 is to rename some public channels to fresh ones that can be hidden by restriction in ψ and ϕ. (A similar purpose is served by quantification in logic.) Hiding those names strengthens Lemmas 7.4.1.1–2 while not affecting their proofs; but more importantly, the restrictions are required to prove Lemma 7.4.1.3. Further the purpose of η_3 is to abstract terms that may be available to contexts. Such terms must be of type `Export` (see below); intuitively, K_{AS} and $\overline{K}_{AS}$ may appear only as authentication keys in capabilities issued to dishonest users.

A binary relation $\_, \_ \rightsquigarrow \_, \_$ ("leads-to") is defined over the product of access policies and clocks. Access policies may change at clock ticks (but not between).

$$F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk} \triangleq (\mathsf{Clk}' < \mathsf{Clk}) \vee (\mathsf{Clk}' = \mathsf{Clk} \wedge F' = F)$$

Let $\mathcal{F}$ range over functions from clocks to access policies.

$$\frac{\begin{array}{c} N = N'\sigma \qquad \{K_{AS}, \overline{K}_{AS}, K_?\} \cap \mathtt{fn}(N') = \varnothing \\ \forall L \in \mathtt{rng}(\sigma).\ \exists j \in \mathbb{N}\backslash\mathcal{I}, op, \mathsf{Clk}'. \\ op :_{\mathcal{F},F,\mathsf{Clk}} \mathtt{Export} \wedge (\mathcal{F}(\mathsf{Clk}'), \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk}) \wedge L = \mathtt{cert}(\mathcal{F}(\mathsf{Clk}'), j, op, \mathsf{Clk}') \end{array}}{N :_{\mathcal{F},F,\mathsf{Clk}} \mathtt{Export}}$$

We show that term abstraction preserves equivalence in the equational theory. This lemma is required to show static equivalence in proofs of soundness for the relations $\mathcal{S}$, $\mathcal{T}$, $\mathcal{U}$, and $\mathcal{V}$ below, which in turn lead to Lemmas 7.4.1 and 7.4.3.

Lemma D.4.1. *Suppose that* $M :_{\mathcal{F},F,\mathsf{Clk}} \mathtt{Export}$ *and* $N :_{\mathcal{F},F,\mathsf{Clk}} \mathtt{Export}$. *Tthen* $M = N$ *iff* $\eta_3(M) = \eta_3(N)$.

Simulation relation for Lemma 7.4.1.1 $\_ \preccurlyeq \phi[\_]$

$$\frac{\mathtt{fn}(\kappa, M) \cap \mathcal{A} = \varnothing \qquad F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk}}{\mathsf{Req}(\kappa, M)\ \ \mathcal{S}_1'^{F,\mathsf{Clk}}\ \ \mathsf{DReq}(\kappa, M)^{\eta_2}}$$

$$\frac{k \in \mathbb{N} \qquad \mathtt{fn}(op, M) \cap \mathcal{A} = \varnothing \qquad F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk}}{\mathsf{Req}(\mathtt{cert}(F', k, op, \mathsf{Clk}'), M) \;\; \mathcal{S}_1'^{F,\mathsf{Clk}} \;\; \mathsf{Req}_k(op, \mathsf{Clk}', M)}$$

$$\frac{\mathtt{fn}(L, op, M) \cap \mathcal{A} = \varnothing}{\mathsf{EOk}(L, op, M) \;\; \mathcal{S}_1'^{F,\mathsf{Clk}} \;\; \mathsf{EOk}(L, op, M)} \qquad \frac{k \in \mathbb{N} \qquad \mathtt{fn}(adm, M) \cap \mathcal{A} = \varnothing}{\mathsf{AReq}_k(adm, M) \;\; \mathcal{S}_1'^{F,\mathsf{Clk}} \;\; \mathsf{AReq}_k(adm, M)}$$

$$\frac{j \in \mathbb{N} \backslash \mathcal{I} \qquad \mathtt{fn}(op, M) \cap \mathcal{A} = \varnothing}{\mathsf{CReq}_j(op, M) \;\; \mathcal{S}_1'^{F,\mathsf{Clk}} \;\; (\nu m)\, \overline{\alpha^\circ_{j\,?}}\langle m\rangle; m(x); \overline{M}\langle \mathbf{mac}(\langle j, op, x\rangle, \mathrm{K}_?)\rangle}$$

(FILE SYSTEMS)

$$\frac{\forall r \in \mathcal{L}.\;\; P_r \;\; \mathcal{S}_1'^{F,\mathsf{Clk}} \;\; Q_r \qquad \mathtt{fn}(\Xi, \rho) \cap \mathcal{A} = \varnothing}{\mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho} \mid \Pi_{r\in\mathcal{L}} P_r \;\; \mathcal{S}_1^{F,\mathsf{Clk}} \;\; \mathsf{Ifs}^{\eta_2}_{F,\Xi,\mathsf{Clk},\rho} \mid \Pi_{r\in\mathcal{L}} Q_r}$$

(HONEST USERS)

$$\frac{\begin{array}{c}\mathtt{dom}(\sigma) = \mathtt{dom}(\sigma') = X \\ \forall x.\;\; x \in X \;\Rightarrow\; \exists F', \mathsf{Clk}', i \in \mathcal{I}, op.\;\; (F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk}) \;\wedge\; \sigma'(x) = \mathsf{Clk}' \\ \wedge\; \Gamma(x) = \mathtt{Cert}(i, op) \;\wedge\; \sigma(x) = \mathtt{cert}(F', i, op, \mathsf{Clk}')\end{array}}{C\sigma \;\; \mathcal{S}_2^{\Gamma,F,\mathsf{Clk}} \;\; \lceil C\rceil_\Gamma \sigma'}$$

$$\frac{i \in \mathcal{I} \qquad P \;\; \mathcal{S}_2^{\Gamma,F,\mathsf{Clk}} \;\; Q \qquad \Gamma(x) = \mathtt{Cert}(i, op)}{(\nu c)(c(x); P \mid \mathsf{CReq}_i(op, c)) \;\; \mathcal{S}_3^{F,\mathsf{Clk}} \;\; (\nu c)(c(x); Q \mid \mathsf{TReq}(c))}$$

(TRUSTED CODE)

$$\frac{P \;\; \mathcal{S}_1^{F,\mathsf{Clk}} \;\; Q \qquad P' \;\; \mathcal{S}_2^{\Gamma,F,\mathsf{Clk}} \;\; Q' \qquad \forall r \in \mathcal{L}.\;\; P_r \;\; \mathcal{S}_3^{F,\mathsf{Clk}} \;\; Q_r}{(\nu_{i\in\mathcal{I}} \alpha_i \beta_i \gamma_i)(P \mid P' \mid \Pi_{r\in\mathcal{L}} P_r) \;\; \mathcal{S}'^{F,\mathsf{Clk}} \;\; (\nu_{i\in\mathcal{I}} \alpha_i^\circ \beta_i^\circ \gamma_i^\circ)(Q \mid Q' \mid \Pi_{r\in\mathcal{L}} Q_r)}$$

(SYSTEM CODE)

$$\frac{P \;\; \mathcal{S}'^{F,\mathsf{Clk}} \;\; Q \qquad \forall x, N.\;\; (\exists \sigma'.\;\; \sigma \equiv \{^N/_x\} \mid \sigma') \;\Rightarrow\; N :_{\mathcal{F},F,\mathsf{Clk}} \mathtt{Export}}{(\nu \overrightarrow{n})(\nu K_{AS} \overline{K}_{AS})(\sigma \mid P) \;\; \mathcal{S}^{F,\mathsf{Clk}} \;\; (\nu \overrightarrow{n})(\nu K_?)(\eta_3(\sigma) \mid (\nu_{j\in\mathbb{N}\backslash\mathcal{I}} \alpha^\circ_{j\,?} \beta^\circ_{j\,?} \gamma^\circ_{j\,?})(Q \mid \uparrow^{\mathrm{IS}}_{\mathrm{NS}}))}$$

Simulation relation for Lemma 7.4.1.2 $\_ \preccurlyeq \psi[\_]$

$$\frac{i \in \mathcal{I} \qquad \mathtt{fn}(op, M) \cap \mathcal{A} = \varnothing \qquad F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk}}{\mathsf{Req}_i(op, \mathsf{Clk}', M) \;\; \mathcal{T}_1'^{F,\mathsf{Clk}} \;\; \mathsf{Req}(\mathtt{cert}(F', k, op, \mathsf{Clk}'), M)^{\eta_1}}$$

$$\frac{j \in \mathbb{N} \backslash \mathcal{I} \qquad \mathtt{fn}(op, \tau, M) \cap \mathcal{A} = \varnothing}{\mathsf{Req}_j(op, \tau, M) \;\; \mathcal{T}_1'^{F,\mathsf{Clk}} \;\; (\nu c)\, \overline{\alpha_{j?}}\langle op, c\rangle; c(\kappa); [\mathbf{msg}(\kappa).3 \le \tau]\, \overline{\beta_{j?}}\langle \kappa, M\rangle}$$

$$\frac{\mathtt{fn}(L, op, M) \cap \mathcal{A} = \varnothing}{\mathsf{EOk}(L, op, M) \;\; \mathcal{T}_1'^{F,\mathsf{Clk}} \;\; \mathsf{EOk}(L, op, M)^{\eta_1}}$$

$$\frac{k \in \mathbb{N} \qquad \mathtt{fn}(adm, M) \cap \mathcal{A} = \varnothing}{\mathsf{AReq}_k(adm, n) \;\; {\mathcal{T}_1'}^{F,\mathsf{Clk}} \;\; \mathsf{AReq}_k(adm, n)^{\eta_1}}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(M) \cap \mathcal{A} = \varnothing}{\mathsf{TReq}(M) \;\; {\mathcal{T}_1'}^{F,\mathsf{Clk}} \;\; (\nu c)\, \overline{\alpha_{j?}}\langle M, c\rangle; c(x); \overline{M}\langle \mathbf{msg}(x).3\rangle}$$

(FILE SYSTEMS)

$$\frac{\forall r \in \mathcal{L}.\;\; P_r \;\; {\mathcal{T}_1'}^{F,\mathsf{Clk}} \;\; Q_r \qquad \mathtt{fn}(\Xi, \rho) \cap \mathcal{A} = \varnothing}{\mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho} \mid \Pi_{r\in\mathcal{L}} P_r \;\; {\mathcal{T}_1}^{F,\mathsf{Clk}} \;\; \mathsf{Nfs}^{\eta_1}_{F,\Xi,\mathsf{Clk},\rho} \mid \Pi_{r\in\mathcal{L}} Q_r}$$

(HONEST USERS)

$$\frac{\begin{array}{c}\mathtt{dom}(\sigma) = \mathtt{dom}(\sigma') = X \\ \forall x.\;\; x \in X \;\Rightarrow\; \exists F', \mathsf{Clk}', i \in \mathcal{I}, op.\;\; (F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk}) \;\wedge\; \sigma(x) = \mathsf{Clk}' \\ \wedge\; \Gamma(x) = \mathtt{Cert}(i, op) \;\wedge\; \sigma'(x) = \mathtt{cert}(F', i, op, \mathsf{Clk}')\end{array}}{\lceil C \rceil_\Gamma \sigma \;\; {\mathcal{T}_2}^{\Gamma,F,\mathsf{Clk}} \;\; C\sigma'}$$

$$\frac{i \in \mathcal{I} \qquad P \;\; {\mathcal{T}_2}^{\Gamma,F,\mathsf{Clk}} \;\; Q \qquad \Gamma(x) = \mathtt{Cert}(i, op)}{(\nu c)(c(x); P \mid \mathsf{TReq}(c)) \;\; \mathcal{T}_3' \;\; (\nu c)(c(x); Q \mid \mathsf{CReq}_i(op, c))}$$

(TRUSTED CODE)

$$\frac{P \;\; \mathcal{T}_1^{F,\mathsf{Clk}} \;\; Q \qquad P' \;\; \mathcal{T}_2^{\Gamma,F,\mathsf{Clk}} \;\; Q' \qquad \forall r \in \mathcal{L}.\;\; P_r \;\; \mathcal{T}_3' \;\; Q_r}{(\nu_{i\in\mathcal{I}} \alpha_i^\circ \beta_i^\circ \gamma_i^\circ)(P \mid P' \mid \Pi_{r\in\mathcal{L}} P_r) \;\; \mathcal{T}' \;\; (\nu_{i\in\mathcal{I}} \alpha_i \beta_i \gamma_i)(\nu K_{AS} \overline{K}_{AS})(Q \mid Q' \mid \Pi_{r\in\mathcal{L}} Q_r)}$$

(SYSTEM CODE)

$$\frac{P \;\; \mathcal{T}' \;\; Q}{(\nu \overrightarrow{n})(\sigma \mid P) \;\; \mathcal{T} \;\; (\nu \overrightarrow{n})(\sigma \mid (\nu_{j\in\mathbb{N}\backslash\mathcal{I}} \alpha_{j?} \beta_{j?} \gamma_{j?})(Q \mid \uparrow^{\mathrm{NS}}_{\mathrm{IS}}))}$$

Simulation relation for Lemma 7.4.1.3 $\phi[\psi[\_]] \preccurlyeq \_$

$$\frac{\mathtt{fn}(\kappa, M) \cap \mathcal{A} = \varnothing}{\mathsf{DReq}(\kappa, M)^{\eta_2} \;\; {\mathcal{U}_1'}^{F,\mathsf{Clk}} \;\; \mathsf{Req}(\kappa, M)} \qquad \frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(op, M) \cap \mathcal{A} = \varnothing \qquad F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk}}{\overline{\beta^\circ_{j?}}\langle op, \mathsf{Clk}', M\rangle \;\; {\mathcal{U}_1'}^{F,\mathsf{Clk}} \;\; \mathsf{Req}(\mathtt{cert}(F', j, op, \mathsf{Clk}'), M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(op, M) \cap \mathcal{A} = \varnothing \qquad F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk}}{\mathsf{DReq}_j(op, \mathsf{Clk}', M)^{\eta_1 \oplus \eta_2} \;\; {\mathcal{U}_1'}^{F,\mathsf{Clk}} \;\; \mathsf{Req}(\mathtt{cert}(F', j, op, \mathsf{Clk}'), M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(op, M) \cap \mathcal{A} = \varnothing \qquad F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk}}{(\nu c)(c(x); [\mathbf{msg}(x).3 \le \mathsf{Clk}']\, \overline{\beta_{j?}}\langle x, M\rangle \mid \mathsf{CReq}_j(op, c)) \;\; {\mathcal{U}_1'}^{F,\mathsf{Clk}} \;\; \mathsf{Req}(\mathtt{cert}(F', j, op, \mathsf{Clk}'), M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(op, M) \cap \mathcal{A} = \varnothing \qquad F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk} \qquad N = \mathbf{mac}(\langle j, op, \mathsf{Clk}'\rangle, K_?)}{(\nu c)(c(x); [\mathbf{msg}(x).3 \le \mathsf{Clk}']\, \overline{\beta_{j?}}\langle x, M\rangle \mid \overline{c}\langle N\rangle) \;\; {\mathcal{U}_1'}^{F,\mathsf{Clk}} \;\; \mathsf{Req}(\mathtt{cert}(F', j, op, \mathsf{Clk}'), M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(op, M) \cap \mathcal{A} = \varnothing \qquad F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk}}{\overline{\beta_{j?}}\langle \mathbf{mac}(\langle j, op, \mathsf{Clk}'\rangle, K_?), M\rangle \;\; \mathcal{U}_1'^{F,\mathsf{Clk}} \;\; \mathsf{Req}(\mathtt{cert}(F', j, op, \mathsf{Clk}'), M)}$$

$$\frac{k \in \mathbb{N} \qquad \mathtt{fn}(op, M) \cap \mathcal{A} = \varnothing \qquad F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk}}{\mathsf{Req}(\mathbf{mac}(\langle k, op, \mathsf{Clk}'\rangle, K_?), M) \;\; \mathcal{U}_1'^{F,\mathsf{Clk}} \;\; \mathsf{Req}(\mathtt{cert}(F', k, op, \mathsf{Clk}'), M)}$$

$$\frac{\mathtt{fn}(L, op, M) \cap \mathcal{A} = \varnothing}{\mathsf{EOk}(L, op, M) \;\; \mathcal{U}_1'^{F,\mathsf{Clk}} \;\; \mathsf{EOk}(L, op, M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(op, M) \cap \mathcal{A} = \varnothing}{(\nu m)\, \overline{\alpha^\circ_{j\,?}}\langle m\rangle; m(x); \overline{M}\langle \mathbf{mac}(\langle j, op, x\rangle, K_?)\rangle \;\; \mathcal{U}_1'^{F,\mathsf{Clk}} \;\; \mathsf{CReq}_j(op, M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(op, M) \cap \mathcal{A} = \varnothing \qquad F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk}}{(\nu m)\, (m(x); \overline{M}\langle \mathbf{mac}(\langle j, op, x\rangle, K_?)\rangle \mid \overline{m}\langle \mathsf{Clk}'\rangle) \;\; \mathcal{U}_1'^{F,\mathsf{Clk}} \;\; \overline{M}\langle \mathtt{cert}(F', k, op, \mathsf{Clk}')\rangle}$$

$$\frac{\mathtt{fn}(adm, M) \cap \mathcal{A} = \varnothing}{\mathsf{AReq}_k(adm, M) \;\; \mathcal{U}_1'^{F,\mathsf{Clk}} \;\; \mathsf{AReq}_k(adm, M)}$$

(FILE SYSTEMS)

$$\frac{\forall r \in \mathcal{L}.\;\; P_r \;\; \mathcal{U}_1'^{F,\mathsf{Clk}} \;\; Q_r \qquad \mathtt{fn}(\Xi, \rho) \cap \mathcal{A} = \varnothing}{\uparrow_{\mathrm{NS}}^{\mathrm{IS}\ \eta_2} \mid \uparrow_{\mathrm{IS}}^{\mathrm{NS}\eta_1 \oplus \eta_2} \mid \mathsf{Nfs}^{\eta_1}_{F,\Xi,\mathsf{Clk},\rho} \mid \Pi_{r\in\mathcal{L}} P_r \;\; \mathcal{U}_1^{F,\mathsf{Clk}} \;\; \mathsf{Nfs}_{F,\Xi,\mathsf{Clk},\rho} \mid \Pi_{r\in\mathcal{L}} Q_r}$$

(HONEST USERS)

$$\frac{\begin{array}{c}\lceil C\rceil_\Gamma = C^\circ \\ \forall x.\;\; x \in \mathtt{dom}(\sigma) \;\Rightarrow\; \exists F', \mathsf{Clk}', i \in \mathcal{I}, op.\;\; (F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk}) \\ \qquad\qquad \wedge\; \Gamma(x) = \mathtt{Cert}(i, op) \;\wedge\; \sigma(x) = \mathtt{cert}(F', i, op, \mathsf{Clk}')\end{array}}{C\sigma \;\; \mathcal{U}_2^{\Gamma,F,\mathsf{Clk}} \;\; C\sigma}$$

$$\frac{i \in \mathcal{I} \qquad \Gamma(x) = \mathtt{Cert}(i, op) \qquad P \;\; \mathcal{U}_2^{\Gamma,F,\mathsf{Clk}} \;\; Q}{(\nu c)(c(x); P \mid \mathsf{CReq}_i(op, c)) \;\; \mathcal{U}_3^{F,\mathsf{Clk}} \;\; (\nu c)(c(x); Q \mid \mathsf{CReq}_i(op, c))}$$

(TRUSTED CODE)

$$\frac{P \;\; \mathcal{U}_1^{F,\mathsf{Clk}} \;\; Q \qquad P' \;\; \mathcal{U}_2^{F,\mathsf{Clk}} \;\; Q' \qquad \forall r \in \mathcal{L}.\;\; P_\ell \;\; \mathcal{U}_3^{F,\mathsf{Clk}} \;\; Q_\ell}{(\nu_{i\in\mathcal{I}} \alpha_i\beta_i\gamma_i)(\nu K_{AS}\overline{K}_{AS})(P \mid P' \mid \Pi_{r\in\mathcal{L}} P_r) \;\; \mathcal{U}'^{F,\mathsf{Clk}} \;\; (\nu_{i\in\mathcal{I}} \alpha_i\beta_i\gamma_i)(Q \mid Q' \mid \Pi_{r\in\mathcal{L}} Q_r)}$$

(SYSTEM CODE)

$$\frac{P \;\; \mathcal{U}'^{F,\mathsf{Clk}} \;\; Q \qquad \forall x, N.\;\; (\exists \sigma'.\;\; \sigma \equiv \{^N/_x\} \mid \sigma') \;\Rightarrow\; N :_{\mathcal{F},F,\mathsf{Clk}} \mathtt{Export}}{(\nu\, \overrightarrow{n})(\nu K_?)(\eta_3(\sigma) \mid (\nu_{j\in\mathbb{N}\backslash\mathcal{I}} \alpha^\circ_{j\,?}\beta^\circ_{j\,?}\gamma^\circ_{j\,?}\alpha_{j_?}\beta_{j_?}\gamma_{j_?})\; P) \;\; \mathcal{U} \;\; (\nu\, \overrightarrow{n})(\nu K_{AS}\overline{K}_{AS})(\sigma \mid Q)}$$

We prove that the relations $\mathcal{S}$, $\mathcal{T}$, and $\mathcal{U}$ are included in the simulation preorder. Lemma 7.4.1 follows. So by Proposition 7.3.5, $\mathcal{R}$ is secure.

Some interesting points in those proofs are listed below.

- When an operation request is sent in IS^d we *wait* after sending an appropriate authorization request in NS^d (see $\mathcal{T}$); we continue only when that operation request in IS^d is processed, when we obtain a capability in NS^d, send an execution request with that capability, and process the execution request.

  Why wait? Suppose that the operation request in IS^d carries a time bound ∞; now if we obtain a capability in NS^d before the operation request in IS^d is processed, we commit to a finite time bound, which breaks the simulation.

- $\phi[\psi]$ forces a fresh capability to be acquired for every execution request by filtering execution requests in NS^d through IS^d and back. When an execution request is sent in NS^d under $\phi[\psi]$ we send an execution request with the same capability in NS^d (see $\mathcal{U}$). But under $\phi[\psi]$ a fresh capability is obtained and the execution request is sent again with the fresh capability. If the capability in the original request expires before the fresh capability, the simulation breaks. Fortunately operation requests in IS^d carry time bounds, so we can communicate this expiry bound through IS^d. In fact there seems to be no way around this problem *unless* time bounds can be specified in operation requests in IS^d!

Simulation relation for Lemma 7.4.3 $\psi[\phi[\_]] \preccurlyeq \_$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(op,\tau,M) \cap \mathcal{A} = \varnothing \qquad \mathbf{msg}(\kappa).3 = \tau'}{(\nu c)\,\overline{\alpha_{j?}}\langle op, c\rangle; c(\kappa); [\tau' \le \tau]\,\overline{\beta_{j?}}\langle \kappa, M\rangle \;\; \mathcal{V}_1'^{F,\mathsf{Clk}} \;\; \mathsf{Req}_j(op,\tau,M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(op,\tau,M) \cap \mathcal{A} = \varnothing \qquad N = \mathbf{mac}(\langle j, op, x\rangle, K_?) \qquad \mathbf{msg}(\kappa).3 = \tau'}{(\nu c)(c(\kappa); [\tau' \le \tau]\,\overline{\beta_{j?}}\langle \kappa, M\rangle \mid (\nu m)\,\overline{\alpha^{\circ}_{j?}}\langle m\rangle; m(x); \overline{c}\langle N\rangle) \;\; \mathcal{V}_1'^{F,\mathsf{Clk}} \;\; \mathsf{Req}_j(op,\tau,M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(op,\tau,M) \cap \mathcal{A} = \varnothing \qquad N = \mathbf{mac}(\langle j, op, x\rangle, K_?) \qquad \mathbf{msg}(\kappa).3 = \tau'}{(\nu c)(c(\kappa); [\tau' \le \tau]\,\overline{\beta_{j?}}\langle \kappa, M\rangle \mid (\nu m)(m(x); \overline{c}\langle N\rangle \mid \mathsf{TReq}(m))) \;\; \mathcal{V}_1'^{F,\mathsf{Clk}} \;\; \mathsf{Req}_j(op,\tau,M)}$$

$$\frac{\begin{array}{c} j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(op,\tau,M) \cap \mathcal{A} = \varnothing \\ F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk} \qquad N = \mathbf{mac}(\langle j, op, \mathsf{Clk}'\rangle, K_?) \qquad L = \mathtt{perm}(F', j, op) \qquad \mathbf{msg}(\kappa).3 = \tau' \end{array}}{(\nu c)(c(\kappa); [\tau' \le \tau]\,\overline{\beta_{j?}}\langle \kappa, M\rangle \mid (\nu m)(m(x); \overline{c}\langle N\rangle \mid \overline{m}\langle \mathsf{Clk}'\rangle)) \;\; \mathcal{V}_1'^{F,\mathsf{Clk}} \;\; [\mathsf{Clk} \le \tau]\;\mathsf{EOk}(L, op, M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(op,\tau,M) \cap \mathcal{A} = \varnothing \qquad F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk} \qquad L = \mathtt{perm}(F', j, op) \qquad \mathbf{msg}(\kappa).3 = \tau'}{(\nu c)(c(\kappa); [\tau' \leq \tau]\, \overline{\beta_{j?}}\langle \kappa, M\rangle \mid \overline{c}\langle \mathbf{mac}(\langle j, op, \mathsf{Clk}'\rangle, K_?)\rangle)\ \ \mathcal{V}_1'^{F,\mathsf{Clk}}\ \ [\mathsf{Clk} \leq \tau]\, \mathsf{EOk}(L, op, M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(op, M) \cap \mathcal{A} = \varnothing \qquad F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk} \qquad N = \mathbf{mac}(\langle j, op, \mathsf{Clk}'\rangle, K_?) \qquad L = \mathtt{perm}(F', j, op)}{\overline{\beta_{j?}}\langle N, M\rangle\ \ \mathcal{V}_1'^{F,\mathsf{Clk}}\ \ \mathsf{EOk}(L, op, M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(op, M) \cap \mathcal{A} = \varnothing \qquad F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk} \qquad N = \mathbf{mac}(\langle j, op, \mathsf{Clk}'\rangle, K_?) \qquad L = \mathtt{perm}(F', j, op)}{\mathsf{DReq}(N, M)^{\eta_1 \oplus \eta_2}\ \ \mathcal{V}_1'^{F,\mathsf{Clk}}\ \ \mathsf{EOk}(L, op, M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(op, M) \cap \mathcal{A} = \varnothing \qquad F', \mathsf{Clk}' \rightsquigarrow F, \mathsf{Clk} \qquad L = \mathtt{perm}(F', j, op)}{\overline{\beta^{\circ}_{j?}}\langle op, \mathsf{Clk}', M\rangle\ \ \mathcal{V}_1'^{F,\mathsf{Clk}}\ \ \mathsf{EOk}(L, op, M)}$$

$$\frac{\mathtt{fn}(op, M) \cap \mathcal{A} = \varnothing}{\mathsf{EOk}(L, op, M)\ \ \mathcal{V}_1'^{F,\mathsf{Clk}}\ \ \mathsf{EOk}(L, op, M)}$$

$$\frac{\mathtt{fn}(adm, M) \cap \mathcal{A} = \varnothing}{\mathsf{AReq}_k(adm, M)\ \ \mathcal{V}_1'^{F,\mathsf{Clk}}\ \ \mathsf{AReq}_k(adm, M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(M) \cap \mathcal{A} = \varnothing}{(\nu c)\, \overline{\alpha_{j?}}\langle M, c\rangle; c(y); \overline{M}\langle \mathbf{msg}(y).3\rangle\ \ \mathcal{V}_1'^{F,\mathsf{Clk}}\ \ \mathsf{TReq}(M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(M) \cap \mathcal{A} = \varnothing}{(\nu c)(c(y); \overline{M}\langle \mathbf{msg}(y).3\rangle \mid (\nu m)\, \overline{\alpha^{\circ}_{j?}}\langle m\rangle; m(x); \overline{c}\langle \mathbf{mac}(\langle j, M, x\rangle, K_?)\rangle)\ \ \mathcal{V}_1'^{F,\mathsf{Clk}}\ \ \mathsf{TReq}(M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(M) \cap \mathcal{A} = \varnothing}{(\nu c)(c(y); \overline{M}\langle \mathbf{msg}(y).3\rangle \mid (\nu m)(m(x); \overline{c}\langle \mathbf{mac}(\langle j, M, x\rangle, K_?)\rangle \mid \mathsf{TReq}(M)))\ \ \mathcal{V}_1'^{F,\mathsf{Clk}}\ \ \mathsf{TReq}(M)}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(M) \cap \mathcal{A} = \varnothing \qquad \mathsf{Clk}' \leq \mathsf{Clk}}{(\nu c)(c(y); \overline{M}\langle \mathbf{msg}(y).3\rangle \mid (\nu m)(m(x); \overline{c}\langle \mathbf{mac}(\langle j, M, x\rangle, K_?)\rangle \mid \overline{m}\langle \mathsf{Clk}'\rangle))\ \ \mathcal{V}_1'^{F,\mathsf{Clk}}\ \ \overline{m}\langle \mathsf{Clk}'\rangle}$$

$$\frac{j \in \mathbb{N}\backslash\mathcal{I} \qquad \mathtt{fn}(M) \cap \mathcal{A} = \varnothing}{(\nu c)(c(y); \overline{M}\langle \mathbf{msg}(y).3\rangle \mid \overline{c}\langle \mathbf{mac}(\langle j, M, \mathsf{Clk}'\rangle, K_?)\rangle)\ \ \mathcal{V}_1'^{F,\mathsf{Clk}}\ \ \overline{M}\langle \mathsf{Clk}'\rangle}$$

(FILE SYSTEMS)

$$\frac{\forall r \in \mathcal{L}.\ P_r\ \ \mathcal{V}_1'^{F,\mathsf{Clk}}\ \ Q_r \qquad \mathtt{fn}(\Xi, \rho) \cap \mathcal{A} = \varnothing}{\uparrow^{\mathrm{NS}}_{\mathrm{IS}}{}^{\eta_1} \mid \uparrow^{\mathrm{IS}}_{\mathrm{NS}}{}^{\eta_1 \oplus \eta_2} \mid \mathsf{Ifs}^{\eta_2}_{F,\Xi,\mathsf{Clk},\rho} \mid \Pi_{r \in \mathcal{L}} P_r\ \ \mathcal{V}_1^{F,\mathsf{Clk}}\ \ \mathsf{Ifs}_{F,\Xi,\mathsf{Clk},\rho} \mid \Pi_{r \in \mathcal{L}} Q_r}$$

(HONEST USERS)

$$\frac{\forall x.\ x \in \mathtt{dom}(\sigma)\ \Rightarrow\ \exists \mathsf{Clk}', i \in \mathcal{I}, op.\ \mathsf{Clk}' \leq \mathsf{Clk}\ \wedge\ \Gamma(x) = \mathtt{Cert}(i, op)\ \wedge\ \sigma(x) = \mathsf{Clk}'}{\lceil C \rceil_\Gamma \sigma\ \ \mathcal{V}_2^{\Gamma,F,\mathsf{Clk}}\ \ \lceil C \rceil_\Gamma \sigma}$$

$$\frac{i \in \mathcal{I} \qquad \Gamma(x) = \mathtt{Cert}(i, op) \qquad P\ \ \mathcal{V}_2^{\Gamma,F,\mathsf{Clk}}\ \ Q}{(\nu c)(c(x); P \mid \mathsf{TReq}(c))\ \ \mathcal{V}_3^{F,\mathsf{Clk}}\ \ (\nu c)(c(x); Q \mid \mathsf{TReq}(c))}$$

(SYSTEM CODE)

$$\frac{P\ \ \mathcal{V}_1^{F,\mathsf{Clk}}\ \ Q \qquad P'\ \ \mathcal{V}_2^{F,\mathsf{Clk}}\ \ Q' \qquad \forall r \in \mathcal{L}.\ P_\ell\ \ \mathcal{V}_3^{F,\mathsf{Clk}}\ \ Q_\ell}{P'' = (\nu_{i \in \mathcal{I}} \alpha_i^\circ \beta_i^\circ \gamma_i^\circ)(\nu K_?)(P \mid P' \mid \Pi_{r \in \mathcal{L}} P_r) \qquad Q'' = (\nu_{i \in \mathcal{I}} \alpha_i^\circ \beta_i^\circ \gamma_i^\circ)(Q \mid Q' \mid \Pi_{r \in \mathcal{L}} Q_r)}{(\nu\, \overrightarrow{n})(\sigma \mid (\nu_{j \in \mathbb{N} \setminus \mathcal{I}} \alpha_{j?}^\circ \beta_{j?}^\circ \gamma_{j?}^\circ \alpha_{j?} \beta_{j?} \gamma_{j?})\ P'')\ \ \mathcal{V}\ \ (\nu\, \overrightarrow{n})(\sigma \mid Q'')}$$

Further, we prove that the relation $\mathcal{V}$ is included in the simulation preorder. Lemma 7.4.3 follows. So by Lemmas 7.4.1.1–2 and Corollary 7.3.6, $\mathcal{R}$ is safe and fully abstract.

Bibliography

M. Abadi and Z. Manna. Temporal logic programming. *Journal of Symbolic Computing*, 8(3):277–295, 1989.

Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

Martín Abadi. Protection in programming-language translations. In *ICALP'98: International Colloquium on Automata, Languages and Programming*, pages 868–883. Springer LNCS, 1998.

Martín Abadi and Bruno Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 298(3):387–415, 2003.

Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005.

Martín Abadi and Luca Cardelli. An imperative object calculus. In *TAPSOFT'95: Theory and Practice of Software Development*, pages 471–485. Springer LNCS, 1995.

Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *POPL'01: Principles of Programming Languages*, pages 104–115. ACM, 2001.

Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: the spi calculus. *Information and Computation*, 148(1):1–70, 1999.

Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *POPL'90: Principles of Programming Languages*, pages 31–46. ACM, 1990.

Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *ICFP'96: Functional Programming*, pages 83–91. ACM, 1996.

Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In *LICS'98: Logic in Computer Science*, pages 105–116. IEEE, 1998.

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *POPL'99: Principles of Programming Languages*, pages 147–160. ACM, 1999.

Martín Abadi, Cédric Fournet, and Georges Gonthier. Authentication primitives and their compilation. In *POPL'00: Principles of Programming Languages*, pages 302–315. ACM, 2000.

Xavier Allamigeon and Bruno Blanchet. Reconstruction of attacks against cryptographic protocols. In *CSFW'05: Computer Security Foundations Workshop*, pages 140–154. IEEE, 2005.

Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(5):181–185, 1985.

Michael Backes, Christian Cachin, and Alina Oprea. Lazy revocation in cryptographic file systems. In *SISW '05: Security in Storage Workshop*, pages 1–11. IEEE, 2005.

Michael Backes, Christian Cachin, and Alina Oprea. Secure key-updating for lazy revocation. In *ESORICS'06: European Symposium on Research in Computer Security*, pages 327–346. Springer LNCS, 2006.

Michael Backes, Agostino Cortesi, and Matteo Maffei. Causality-based abstraction of multiplicity in security protocols. In *CSF'07: Computer Security Foundations Symposium*, pages 355–369. IEEE, 2007.

A. Banerjee and D. Naumann. Using access control for secure information flow in a Java-like language. In *CSFW'03: Computer Security Foundations Workshop*, pages 155–169. IEEE, 2003.

Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. In *ESORICS'07: European Symposium On Research In Computer Security*, pages 2–18. Springer LNCS, 2007.

Moritz Becker, Cedric Fournet, and Andrew Gordon. Design and semantics of a decentralized authorization language. In *CSF'07: Computer Security Foundations Symposium*, pages 3–15. IEEE, 2007.

D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE, 1975.

Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT'00: Theory and Application of Cryptology & Information Security*, pages 531–545, 2000.

K. J. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, MITRE, 1977.

Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *ACM SIGOPS Operating Systems Review*, 27(5):217–230, 1993.

Bruno Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *CSF'07: Computer Security Foundations Symposium*, pages 97–111. IEEE, 2007a.

Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 2007b. To appear. Technical report available at `http://eprint.iacr.org/2005/401`.

Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *CSFW'01: Computer Security Foundations Workshop*, pages 82–96. IEEE, 2001a.

Bruno Blanchet. From secrecy to authenticity in security protocols. In *SAS'02: Static Analysis Symposium*, pages 342–359, 2002.

Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW'01: Computer Security Foundations Workshop*, pages 82–96. IEEE, 2001b.

Bruno Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 2008. To appear. Technical report available at `http://arxiv.org/abs/0802.3444v1`.

Bruno Blanchet and Avik Chaudhuri. Automated formal analysis of a protocol for secure file sharing on untrusted storage. In *S&P'08: Symposium on Security and Privacy*, pages 417–431. IEEE, 2008.

Paolo Di Blasio and Kathleen Fisher. A calculus for concurrent objects. In *CONCUR'96: Concurrency Theory*, pages 655–670. Springer LNCS, 1996.

Matt Blaze. A cryptographic file system for Unix. In *CCS'93: Computer and Communications Security*, pages 9–16. ACM, 1993.

Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

Dan Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society*, 46(2):203–213, 1999.

Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1-2):109–130, 2002.

C. Braghin, D. Gorla, and V. Sassone. A distributed calculus for role-based access control. In *CSFW'04: Computer Security Foundations Workshop*, pages 48–60. IEEE, 2004.

Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Access control for mobile agents: the calculus of boxed ambients. *ACM Transactions on Programming Languages and Systems*, 26(1):57–124, 2004a.

Michele Bugliesi, Dario Colazzo, and Silvia Crafa. Type based discretionary access control. In *CONCUR'04: Concurrency Theory*, pages 225–239. Springer LNCS, 2004b.

Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989. A preliminary version appeared as Digital Equipment Corporation Systems Research Center report No. 39, February 1989.

Ran Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *FOCS'01: Foundations of Computer Science*, pages 136–145, 2001.

Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. *Information and Computation*, 196(2):127–155, 2005. ISSN 0890-5401. doi: http://dx.doi.org/10.1016/j.ic.2004.08.003.

Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *OSDI'06: Operating Systems Design and Implementation*, pages 147–160. USENIX, 2006.

Avik Chaudhuri. On secure distributed implementations of dynamic access control. Technical Report UCSC-CRL-08-01, University of California at Santa Cruz, 2008a. Available at `http://arxiv.org/abs/0805.4665`.

Avik Chaudhuri. Dynamic access control in a concurrent object calculus. In *CONCUR'06: Concurrency Theory*, pages 263–278. Springer LNCS, 2006.

Avik Chaudhuri. On secure distributed implementations of dynamic access control. In *FCS-ARSPA-WITS'08: Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis, and Issues in the Theory of Security*, pages 93–107, 2008b.

Avik Chaudhuri and Martín Abadi. Formal security analysis of basic network-attached storage. In *FMSE'05: Formal Methods in Security Engineering*, pages 43–52. ACM, 2005.

Avik Chaudhuri and Martín Abadi. Formal analysis of dynamic, distributed file-system access controls. In *FORTE'06: Formal Techniques for Networked and Distributed Systems*, pages 99–114. Springer, 2006a.

Avik Chaudhuri and Martín Abadi. Secrecy by typing and file-access control. In *CSFW'06: Computer Security Foundations Workshop*, pages 112–123. IEEE, 2006b.

Avik Chaudhuri, Prasad Naldurg, and Sriram Rajamani. A type system for data-flow integrity on Windows Vista. Technical Report TR-2007-86, Microsoft Research, 2007. Available at `http://arxiv.org/abs/0803.3230`.

Avik Chaudhuri, Prasad Naldurg, and Sriram Rajamani. A type system for data-flow integrity on Windows Vista. In *PLAS'08: Programming Languages and Analysis for Security*, pages 89–100. ACM, 2008a.

Avik Chaudhuri, Prasad Naldurg, Sriram Rajamani, G. Ramalingam, and L. Velaga. EON: Modeling and analyzing dynamic access control systems with logic programs. Technical Report MSR-TR-08-21, Microsoft Research, 2008b.

Avik Chaudhuri, Prasad Naldurg, Sriram Rajamani, G. Ramalingam, and L. Velaga. EON: Modeling and analyzing dynamic access control systems with logic programs. In *CCS'08: Computer and Communications Security*. ACM, to appear, 2008c.

David D. Clark and David R. Wilson. A comparison of commercial and military computer security poli cies. In *S&P'87: Symposium on Security and Privacy*, pages 184–194. IEEE, 1987.

Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *CSF'08: Computer Security Foundations Symposium*, pages 51–65. IEEE, 2008.

James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA'07: International Symposium on Software Testing and Analysis*, pages 196–206. ACM, 2007.

Matthew Conover. Analysis of the windows vista security model. Technical report, Symantec, 2007. Available at `www.symantec.com/avcenter/reference/Windows_Vista_Security_Model_Analysis.pdf`.

Rocco de Nicola, Gian Luigi Ferrari, and R. Pugliese. KLAIM: A kernel language for

agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.

Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(12):198–208, 1983.

Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *IJCAR'06: International Joint Conference on Automated Reasoning*, pages 632–646. Springer LNCS, 2006.

Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *SOSP'05: Symposium on Operating Systems Principles*, pages 17–30. ACM, 2005.

M. Felleisen. The theory and practice of first-class prompts. In *POPL'88: Principles of Programming Languages*, pages 180–190. ACM, 1988.

William Ferreira, Matthew Hennessy, and Alan Jeffrey. A theory of weak bisimulation for core cml. *Journal of Functional Programming*, 8(5):447–491, 1998.

Cormac Flanagan. Hybrid type checking. In *POPL'06: Principles of programming languages*, pages 245–256. ACM, 2006.

Cormac Flanagan and Martín Abadi. Object types against races. In *CONCUR'99: Concurrency Theory*, pages 288–303. Springer LNCS, 1999.

Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. In *ESOP'05: European Symposium on Programming*, pages 141–156. Springer LNCS, 2005.

Kevin Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, 1999.

Kevin Fu, Seny Kamara, and Yoshi Kohno. Key regression: enabling efficient key distribution for secure distributed storage. In *NDSS'06: Network and Distributed System Security*. ISOC, 2006.

H. Gobioff. *Security for a High Performance Commodity Storage Subsystem*. PhD thesis, Carnegie Mellon University, 1999.

H. Gobioff, G. Gibson, and J. Tygar. Security for network attached storage devices. Technical Report CMU-CS-97-185, Carnegie Mellon University, 1997.

J. A. Goguen and J. Meseguer. Security policies and security models. In *S&P'82: Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.

Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing remote untrusted storage. In *NDSS'03: Network and Distributed System Security Symposium*. ISOC, 2003.

Shafi Goldwasser and Mihir Bellare. Lecture notes in cryptography, 2001. Available at `http://www.cs.ucsd.edu/users/mihir/papers/gb.html`.

Andrew Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–521, 2003a.

Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: reduction and typing. In *HLCL'98: High-Level Concurrent Languages*, volume 16(3), pages 248–264. Elsevier, 1998.

Andrew D. Gordon and Alan Jeffrey. Secrecy despite compromise: types, cryptography, and the pi-calculus. In *CONCUR'05: Concurrency Theory*, pages 186–201. Springer LNCS, 2005.

Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300(1-3):379–409, 2003b.

D. Gorla and R. Pugliese. Resource access and mobility control with dynamic privileges acquisition. In *ICALP'03: International Colloquium on Automata, Languages, and Programming*, pages 119–132. Springer LNCS, 2003.

Yuri Gurevich and Itay Neeman. Dkal: Distributed-knowledge authorization language. *CSF'08: Computer Security Foundations Symposium*, pages 149–162, 2008.

Christian Haack and Alan Jeffrey. Timed spi-calculus with types for secrecy and authenticity. In *CONCUR'05: Concurrency Theory*, pages 202–216. Springer LNCS, 2005.

Shai Halevi, Paul A. Karger, and Dalit Naor. Enforcing confinement in distributed storage and a cryptographic model for access control. Technical Report 2005/169, Cryptology ePrint Archive, 2005. Available at `http://eprint.iacr.org/2005/169`.

Alon Y. Halevy, Inderpal Singh Mumick, Yehoshua Sagiv, and Oded Shmueli. Static analysis in datalog extensions. *Journal of the ACM*, 48(5):971–1012, 2001.

Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Certified in-lined reference monitoring on .net. In *PLAS'06: Programming languages and analysis for security*, pages 7–16. ACM, 2006.

Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. On protection in operating systems. In *SOSP'75: Symposium on Operating systems Principles*, pages 14–24. ACM, 1975.

Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. In *HLCL'98: High-Level Concurrent Languages*, volume 16(3), pages 174–188. Elsevier, 1998.

Matthew Hennessy and James Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Transactions on Programming Languages and Systems*, 24 (5):566–591, 2002.

Matthew Hennessy, Massimo Merro, and Julian Rathke. Towards a behavioural theory of access and mobility control in distributed systems. In *FOSSACS'03: Foundations of Software Science and Computational Structures*, pages 282–298. Springer LNCS, 2003.

Matthew Hennessy, Julian Rathke, and Nobuko Yoshida. SafeDpi: A language for controlling mobile code. *Acta Informatica*, 42(4-5):227–290, 2005.

Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *POPL'02: Principles of Programming Languages*, pages 81–92. ACM, 2002.

Daisuke Hoshina, Eijiro Sumii, and Akinori Yonezawa. A typed process calculus for fine-grained resource access control in distributed computation. In *TACS'01: Theoretical Aspects of Computer Software*, pages 64–81. Springer, 2001.

Michael Howard and David LeBlanc. *Writing Secure Code for Windows Vista*. Microsoft Press, 2007.

Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *FAST'03: File and Storage Technologies*, pages 29–42. USENIX, 2003.

Mahesh Kallahalla, Erik Riedel, and Ram Swaminathan. System for enabling lazy-revocation through recursive key generation. United States Patent 7203317. Available at `http://www.freepatentsonline.com/7203317.html`, 2007.

Z. D. Kirli. Confined mobile functions. In *CSFW'01: Computer Security Foundations Workshop*, pages 283–294. IEEE, 2001.

Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.

Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

B. W. Lampson. Protection. *ACM Operating Systems Review*, 8(1):18–24, Jan 1974.

Jean-Jacques Lévy. *Réductions correctes et optimales dans le lambda-calcul*. PhD thesis, Université Paris 7, 1978.

Ninghui Li, William H. Winsborough, and John C. Mitchell. Beyond proof-of-compliance: Safety and availability analysis in trust management. In *S&P'03: Symposium on Security and Privacy*, page 123. IEEE, 2003.

Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *POPL'05: Principles of Programming Languages*, pages 158–170. ACM, 2005.

Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS'96: Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166, 1996.

Sergio Maffeis. *Dynamic Web Data: A Process Algebraic Approach*. PhD thesis, Imperial College London, August 2006.

David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. In *PODC'02: Principles of Distributed Computing*, pages 108–117. ACM, 2002.

Ethan L. Miller, William E. Freeman, Darrell D. E. Long, and Benjamin C. Reed. Strong security for network-attached storage. In *FAST'02: File and Storage Technologies*, pages 1–14. USENIX, 2002.

R. Milner. The polyadic pi-calculus: a tutorial. In *Logic and Algebra of Specification*, pages 203–246. Springer LNCS, 1993.

Robin Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4(1):1–22, 1977.

A. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *CSFW'04: Computer Security Foundations Workshop*, pages 172–186. IEEE, 2004.

Prasad Naldurg, Stefan Schwoon, Sriram Rajamani, and John Lambert. NETRA: seeing through access control. In *FMSE'06: Formal Methods in Security Engineering*, pages 55–66. ACM, 2006.

Dalit Naor, Amir Shenhav, and Avishai Wool. Toward securing untrusted storage without public-key operations. In *StorageSS'05: Storage Security and Survivability*, pages 51–56. ACM, 2005.

George C. Necula. Proof-carrying code. In *POPL'97: Principles of Programming Langauges*, pages 106–119. ACM, 1997.

R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1–2):83–133, 1984.

Rocco De Nicola, GianLuigi Ferrari, Rosario Pugliese, and Betti Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, 2000.

Mehmet A. Orgun. On temporal deductive databases. *Computational Intelligence*, 12: 235–259, 1996.

Larry C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

Marco Pistoia, Anindya Banerjee, and David A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *S&P'07: Symposium on Security and Privacy*, pages 149–163. IEEE, 2007a.

Marco Pistoia, Anindya Banerjee, and David A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *S&P'07: Symposium on Security and Privacy*, pages 149–163. IEEE, 2007b.

François Pottier and Sylvain Conchon. Information flow inference for free. In *ICFP'00: Functional Programming*, pages 46–57. ACM, 2000.

François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems*, 27(2):344–382, 2005.

D. D. Redell. Naming and protection in extendible operating systems. Technical Report MAC-TR-140, Massachusetts Institute of Technology, 1974.

Ronald Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

Mark Russinovich. *Inside Windows Vista User Access Control*. Microsoft Technet Magazine, June 2007. Available at `http://www.microsoft.com/technet/technetmag/issues/2007/06/UAC/`.

Alejandro Russo and Andrei Sabelfeld. Securing interaction between threads and the scheduler. In *CSFW'06: Computer Security Foundations Workshop*, pages 177–189. IEEE, 2006.

A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

Beata Sarna-Starosta and Scott D. Stoller. Policy analysis for security-enhanced linux. In *WITS'04: Workshop on Issues in the Theory of Security*, pages 1–12. Informal record. Available at `http://www.cs.sunysb.edu/~stoller/WITS2004.html`, 2004.

Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

Umesh Shankar, Trent Jaeger, and Reiner Sailer. Toward automated information-flow integrity verification for security-critical applications. In *NDSS'06: Network and Distributed System Security Symposium*. ISOC, 2006.

Scott D. Stoller, Ping Yang, C.R. Ramakrishnan, and Mikhail I. Go fman. Efficient policy analysis for administrative role based access control. In *CCS'07: Conference on Computer and Communications Security*. ACM, 2007.

G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS'04: Architectural Support for Programming Languages and Ope rating Systems*, pages 85–96. ACM, 2004.

Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. In *S&P'04: Symposium on Security and Privacy*, pages 179–193. IEEE, 2004.

J. D. Ullman. *Principles of Database and Knowledge-base Systems, Volumes II: The New Technologies*. Computer Science, New York, 1989.

Vasco T. Vasconcelos. Typed concurrent objects. In *ECOOP'94: European Conference on Object-Oriented Programming*, pages 100–117. Springer LNCS, 1994.

Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS'07: Network and Distributed System Security Symposium*. ISOC, 2007.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Scheme'07: Workshop on Scheme and Functional Programming*, 2007.

David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In *Workshop on Electronic Commerce*, pages 29–40. USENIX, 1996.

L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. O'Reilly, 1996.

Windows Vista Tech Center. *Understanding and configuring User Account Control in Windows Vista*. Available at `http://technet.microsoft.com/en-us/windowsvista/aa905117.aspx`.

Thomas Y. C. Woo and Simon S. Lam. A semantic model for authentication protocols. In *S&P'93: Symposium on Security and Privacy*, pages 178–194. IEEE, 1993.

Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and E ngin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS'07: Computer and Communications Security*, pages 116–127. ACM, 2007.

Nobuko Yoshida. Channel dependent types for higher-order mobile processes. In *POPL'04: Principles of Programming Languages*, pages 147–160. ACM, 2004.

Steve Zdancewic and Andrew C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2/3):209–234, 2002.

Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *CSFW'03: Computer Security Foundations Workshop*, pages 29–43. IEEE, 2003.

Steve Zdancewic and Andrew C. Myers. Robust declassification. In *CSFW'01: Computer Security Foundations Workshop*, page 5. IEEE, 2001.

Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, 2002.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *OSDI'06: Operating Systems Design and Implementation*, pages 19–19. USENIX, 2006.

Lantian Zheng. Personal communication, July 2007.

Lantian Zheng and Andrew Myers. Dynamic security labels and noninterference. In *FAST'04: Formal Aspects in Security and Trust*, pages 27–40. Springer LNCS, 2004.