Foundations of Access Control for Secure Storage

Thesis Proposal

Avik Chaudhuri

Computer Science Department
University of California, Santa Cruz
avik@cs.ucsc.edu

Abstract

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. 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 networked 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. Certainly it would be nice if techniques developed for the study of secure communication can also be applied to study secure storage. We propose to make these connections explicit. On the other hand, 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, storage systems typically feature access control on store operations, for various reasons that are informally linked with security. We see an intriguing and challenging research opportunity in understanding the foundations of access control for security in such systems.

Therefore we propose a thorough investigation of formal techniques for the purposes of specifying, implementing, verifying, and exploiting access control in storage systems. We envisage two complementary lines of work: one that focuses on correctness proofs for various implementations of access control, and another that assumes correct "black-box" access control in proofs of end-to-end security properties. More specifically, we are interested in articulating and justifying precise security properties of several complex cryptographic access controls that appear in a variety of distributed storage designs. We are also interested in proof techniques that combine access control with static analysis for more concrete guarantees like secrecy and integrity. We report some preliminary work along these lines and outline related ongoing work. We also discuss work that remains to be done within the scope of this thesis—including work on consolidating and organizing the state of the art—and sketch a tentative plan of action that roughly spans the next two years. Finally, we summarize what we expect to be the main contributions of this thesis, and speculate on its likely impact on system security.

Contents

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 [10, 7, 4, 46, 39, 47, 25, 3]. 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 networked 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 contents M as a channel f that is ready to send message M; then f may be read by receiving a message on f and then sending the message back on f; and f may be written by receiving a message on f and sending back a new message M' on f. Certainly it would be nice if techniques developed for the study of secure communication can also be applied to study secure storage. In particular, previous work on asymmetric channels (channels with separate read and write capabilities) may be particularly useful [4, 18]. Moreover the use of cryptography for secure communication on untrusted channels is close to its use for secure storage on untrusted servers [55]. In general one might expect verification concepts and tools developed for the analysis of communication systems to be useful, perhaps with some suitable adaptations, for the analysis of storage systems as well. One must of course 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. We propose to make the connections between secure communication and secure storage explicit where possible.

On the other hand, 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, storage systems typically feature access control on store operations, for various reasons that are informally linked with security. 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. At the same time, 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 runtime mechanism for enforcing dynamic usage specifications. Indeed it is only realistic to expect a storage unit (such as a file or memory location) to have various uses over its lifetime. We see an intriguing and challenging research opportunity in understanding the foundations of access control for security in such systems.

Therefore we propose a thorough investigation of formal techniques for the purposes of specifying, implementing, verifying, and exploiting access control in storage systems. We envisage two complementary lines of work: one that focuses on correctness proofs for various implementations of access control, and another that assumes correct "black-box" access control in proofs of end-to-end security properties. More specifically, we are interested in articulating and justifying precise security properties of several complex cryptographic access controls that appear in a variety of distributed storage designs. We are also interested in proof techniques that combine access control with static analysis for more concrete guarantees like secrecy and integrity.

1.1 Thesis statement

Our thesis is the following.

A formal understanding of the foundations of access control in storage systems can significantly help in articulating, justifying, and enhancing the security of such systems.

Broadly, we propose to show that formal techniques can be applied to specify and verify security properties of a variety of storage systems. We expect some of those techniques to be adapted from existing ones developed for the study of secure communication. On the other hand, the study of secure storage poses some new problems, and we expect some of those problems to require the development of new techniques. Focusing on the foundations of access control for secure storage, we propose to explain the formal connections between various security properties and access control in storage systems via these techniques.

1.2 Outline of the proposal

As mentioned above, we envisage two complementary lines of work: one that focuses on the correctness of access controls in a variety of storage systems, and another that exploits correct access control in proofs of concrete end-to-end security properties. The motivation for the first line of work stems from the complexity of various access-control implementations in storage systems. There are various underlying assumptions and guarantees in those systems. The complexity of the implementations can be often attributed to the distributed nature of those systems and other complex design features for practical utility. Verifying the correctness of these implementations is typically not straightforward; we expect that the formal exercise can help understand the nuances of these implementations, uncover potential flaws, and articulate their precise properties. Going further, we expect that access control as a mechanism can be effective in achieving concrete security guarantees like secrecy and integrity. This consideration motivates the second line of work. We see the possibility of applying language-based techniques that relax conventional static analyses based on type systems by integrating dynamic access control in those analyses.

The outline of the remainder of the proposal is as follows. In Section 2 we report preliminary work along the proposed lines [28, 29, 30, 27] and outline related ongoing work. In Section 3 we discuss work that remains to be done within the scope of this thesis—including work on consolidating and organizing the state of the art—and sketch a tentative plan of action that roughly spans the next two years. We conclude in Section 4.

2 Preliminary results

In this section we review preliminary and ongoing work along the lines proposed in Section 1.2. Some of the material presented here appears in conference proceedings [28, 29, 30, 27].

2.1 On provable implementations of access control

We present some initial work on proving implementations of access control in networked storage systems. Specifically, we consider capability-based access controls in a file system with distributed storage, key-based access controls in a file system with untrusted storage, and label-based access controls in an operating system. The implementations rely on various delicate protocols based on cryptography and trust. We explore various techniques for their analysis.

2.1.1 Reduction analysis of distributed access controls in networked storage

Our first case study is a storage design that decouples file-system management from disk access. The design is popularly known as *network-attached storage* (NAS) [42], since disks are directly attached to the network (instead of being indirectly interfaced through the file system). Clients can request operations directly at the disks; such requests are guided by previous communication with file managers. The key advantage of such a scheme over conventional centralized storage is that each disk-access request need not pass through a file manager; information provided by the file managers can be reused for multiple disk-access requests. Not surprisingly, the scheme leads to remarkable improvements in performance.

However, decoupling file-system management from disk access makes analysis difficult. In particular, access control lists provide some obvious guarantees in conventional centralized storage, and it is not immediately clear whether those guarantees hold in the distributed architecture of NAS. We approach this problem by studying NAS as an implementation of "ideal" storage. More precisely, we prove that NAS systems preserve certain security properties of their abstractions as ideal storage systems. As a side effect, we simplify reasoning over NAS systems—since reasoning over the simpler specifications can suffice.

We analyze a basic NAS protocol using the reduction technique outlined above. Consider NAS systems with clients $C_i, C_j, \ldots$, a file manager M, and a disk D. Let op range over disk operations and K and K' be secret keys shared between M and D. The protocol is sketched in the following diagram.

$$\begin{array}{ccclll}
C_i & \rightarrow & M & : & op & (1) \\
M & \rightarrow & C_i & : & \mathbf{mac}(\langle i, op\rangle, K) & (2) \quad \text{if } \mathbf{access}(i, op) = \mathbf{allow} \\
M & \rightarrow & C_i & : & \mathbf{mac}(\langle i, op\rangle, K') & (2) \quad \text{if } \mathbf{access}(i, op) = \mathbf{deny} \\
 & \ldots & & & & \\
C_j & \rightarrow & D & : & \langle op', \kappa\rangle & (3) \\
D & \rightarrow & C_j & : & r & (4) \quad \text{if } \kappa = \mathbf{mac}(\langle \_, op'\rangle, K)
\end{array}$$

In the protocol,

1. C_i communicates to M its intent to request op.

2. M issues an appropriate *capability* if i has access to op. The capability is a message authentication code that contains the permission $\langle i, op\rangle$ signed with K; informally, it serves as an unforgeable proof of authorization for op to D. On the other hand, if i does not have access to op then M issues a fake "capability" that uses K' instead of K.

3. C_j requests D to service op' and sends κ as proof of authorization.

4. D replies if κ is a correct capability for op'.

We model this protocol in an applied pi calculus [7]. Clients $C_i, C_j, \ldots$ are modeled as processes in the calculus, that communicate with the special processes M and D on distinct authentication channels $\alpha_i, \alpha_j, \ldots$ and $\beta_i, \beta_j, \ldots$. A NAS system is specified as a process

$$(\nu \widetilde{n})\ (\nu_{i \in \overline{\mathcal{I}}} \alpha_i, \beta_i)\ (|_{i \in \overline{\mathcal{I}}} C_i \mid (\nu K)\ (M \mid D))$$

Here

- ν is a binder that, intuitively, generates fresh names or keys.
- $|$ is parallel composition.
- $\overline{\mathcal{I}}$ indexes clients that participate "honestly" in the protocol.

Other, possibly dishonest clients are left unspecified; we assume them to be part of an arbitrary attacker.

Now consider simpler, ideal storage systems with clients $C'_i, C'_j, \ldots$ and a file system S. The protocol is as follows.

$$\begin{array}{lllll} C'_j & \rightarrow & S & : & op' & (1) \\ S & \rightarrow & C'_j & : & r & (2) \quad \text{if } \mathbf{access}(j, op') = \mathbf{allow} \end{array}$$

In the protocol,

1. C'_j requests S to service op'.
2. S replies if j has access to op'.

As above, $C'_i, C'_j, \ldots$ are modeled as processes that communicate with the special process S on distinct authentication channels $\beta_i, \beta_j, \ldots$; an ideal storage system is specified as a process

$$(\nu \widetilde{n})\ (\nu_{i \in \overline{\mathcal{I}}} \beta_i)\ (|_{i \in \overline{\mathcal{I}}} C'_i \mid S)$$

where $\overline{\mathcal{I}}$ indexes clients that participate "honestly" in the protocol.

We reduce NAS systems to ideal storage systems and show that the former systems are fully abstract with respect to the latter ones. We define a simple syntactic translation $\lceil \cdot \rceil$ from honest clients in NAS to honest clients in ideal storage (*e.g.*, C_j is translated to C'_j). We lift this translation to systems:

$$\lceil (\nu \widetilde{n})\ (\nu_{i \in \overline{\mathcal{I}}} \alpha_i, \beta_i)\ (|_{i \in \overline{\mathcal{I}}} C_i \mid (\nu K)\ (M \mid D)) \rceil = (\nu \widetilde{n})\ (\nu_{i \in \overline{\mathcal{I}}} \beta_i)\ (|_{i \in \overline{\mathcal{I}}} \lceil C_i \rceil \mid S)$$

We observe behaviors via *tests*. A test is itself a process, and can use a special channel w. A process P passes a test E if $P \mid E \longrightarrow^{\star} \xrightarrow{\overline{w}}$, that is, if the parallel composition of P and E may eventually output on w. P is testing equivalent to another process Q (written $P \sim Q$) if P and Q pass the same tests. Intuitively, $P \sim Q$ means that P and Q cannot be distinguished by any environment. Several security properties are defined in terms of indistinguishability. We prove that NAS systems preserve all such properties of their specifications.

Theorem 2.1 (Full abstraction). *For any pair of NAS systems N_1 and N_2: $N_1 \sim N_2$ if and only if $\lceil N_1 \rceil \sim \lceil N_2 \rceil$.*

Full abstraction [64] has been studied as a powerful concept for implementing secure systems [2]; it has also been used for establishing security properties of various communication mechanisms [9, 8]. Intuitively, an implementation is fully abstract if it preserves observable information. For example, fake capabilities need to be issued to preserve observable information on static access control in NAS. Not surprisingly, full abstraction is fairly fragile, and can be broken by many reasonable implementations in practice. In particular, dynamic access control in NAS gives rise to counterexamples to full abstraction that appear impossible to avoid given any reasonable storage specification.

We briefly explain the previous comment. Suppose that a NAS client C communicates to the file manager M its intent to request op. M must then return an appropriate capability that depends non-trivially on dynamic permissions. Indeed, M must communicate some information on those permissions to the disk D via that capability. Now consider whether the capability leaks any information to the environment. If it does, then we need a reliable way of simulating this leak in the specification. For such a simulation to exist, the file system specification S must communicate similar information on dynamic permissions to the environment; moreover such communication must not modify the storage state, yet influence decisions for future service to correctly simulate future uses of the capability in NAS. Clearly such a specification does not remain "ideal". On the other hand, if the capability does not leak any information to the environment, then C cannot know whether the capability "works" without requesting D to service *op* and sending that capability as proof of authorization. Now we need a reliable way of simulating the capability's failures. Once again, the necessary specification does not remain "ideal".

In the latter case, however, it is possible to conservatively approximate such failures with a reasonably ideal storage specification. Fortunately, this approximation does not affect the preservation of a large class of safety properties that include security properties like secrecy and integrity. We now observe behaviors via *quizzes*. Quizzes distinguish finer than tests—a quiz is a test with a term and a set of names that occur "fresh" in the term. A process P passes a quiz $(E, u, \widetilde{n})$ if $P \mid E \longrightarrow^{\star} \xrightarrow{(\nu\widetilde{n})\,\overline{w}\langle u\rangle}$, that is, if the parallel composition of P and E may eventually output u on w with the names $\widetilde{n}$ fresh in u. Secrecy and integrity can be defined in terms of quiz failure. We prove that NAS systems preserve quiz failures of their specifications. Here $\lceil E \rceil$ composes E with a "wrapper" process, and $\lceil \cdot \rceil$ on terms preserves static equivalence.

Theorem 2.2 (Safe refinement)**.** *For any NAS system N: if N passes $(E, u, \widetilde{n})$ then $\lceil N \rceil$ passes $(\lceil E \rceil, \lceil u \rceil, \widetilde{n})$.*

2.1.2 Automated proofs for key-based access controls in untrusted storage[1]

Our next case study is a storage design that does not rely on storage servers to provide confidentiality. Contents of files are cryptographically secured, and keys for writing and reading those contents are managed by the owners of those files. Moreover, the protocol is designed for economic key distribution and cryptography in the presence of dynamic access control by file owners. The special schemes introduced for these purposes complicate the protocol and its security properties. Implementations of (variants of) the protocol can be found in several file systems with untrusted storage [55, 61].

In the basic protocol, principals are qualified as owners, writers, and readers of files. The owner of a file generates and distributes keys for writing and reading contents of the file. The write-key is used to encrypt and sign contents; the read-key is used to verify and decrypt those contents. The keys can be revoked by the owner to dynamically control access to the file; a new write-key and a new read-key are then generated and distributed appropriately. The new write-key is used for subsequently writing the file. Notably, the file is not immediately secured with the new write-key. As such, the previous read-key can be used to verify and decrypt the contents of the file until the file is re-written. This scheme, called *lazy revocation*, aims to prevent redundant cryptography and is justified by the following observations. Firstly, since the existing contents of the file come from the previous writers, signing those contents with the new write-key wrongly indicates that they come from the new writers. Secondly, encrypting the contents with the new write-key cannot guarantee secrecy from the previous readers, since the contents may be known to those readers before the revocation.

Further, the new readers can derive the previous read-key from the new read-key. This scheme, called *key rotation*, complements lazy revocation and aims to prevent redundant key distribution. Indeed, the new readers should be able to read the existing file contents, and key rotation relieves those readers from maintaining the previous read-key for that purpose. On the other hand, the previous write-key cannot be derived from the new write-key. (In any case, such a derivation does not appear to have any obvious use.)

We study the basic protocol and its properties in ProVerif [19], a tool designed by Bruno Blanchet for automatic verification of security protocols[2]. In the model, we abstract several cryptographic constructs and their properties with a few function symbols and term equations. Specifically, we use function symbols

[1]The material presented here is part of ongoing work that has not yet been published.

[2]We build a few optimizations into the tool for this particular enterprise.

`pkgen` and `skgen` to construct the correct write-key and read-key from a version number and a secret owner key; we model encryption and signing with a constructor `write`; we model verifying and decryption with a deconstructor `read` and the equation

```
reduc read(f,write(f,x,pkgen(v,sk)),skgen(v,sk)) = x.
```

We define a binary predicate `geq` that implements $\geq$ over version numbers. We write principals as applied pi-calculus processes with events [46]. Specifically, we define generic processes for owners, honest and dishonest readers, and honest and dishonest writers, and define an oracle for unwinding read-keys. (Honest principals are those that follow protocol; dishonest ones are those that go corrupt.) The code is sketched in Figure 1.

Properties of the protocol are written as correspondence queries over events [78, 46]. We automatically prove the following queries.

```
query let mx = m;
      attacker:mx ==>
                  ev:isreader(skx, rx, vx)
                & ev:corrupt(rx,vx)
                & ev:iswriter(skx, wx, vy)
                & ev:puts(wx, mx, vy)
                & geq:vx,vy.

query ev:gets(rx, mx, vx) ==>
                  ev:isreader(skx, rx, vx)
                & ev:iswriter(skx, wx, vx)
                & ev:puts(wx, mx, vx)
              |
                  ev:isreader(skx, rx, vx)
                & ev:iswriter(skx, wx, vx)
                & ev:corrupt(wx,vx).
```

The queries formalize the main security properties of the protocol. The first is a secrecy property. Informally,

> *If the attacker gets a term intended to be secret, then that term must have been written by an honest writer, and a reader must have gone corrupt after that term was written.*

The second is an integrity property. Informally,

> *If an honest reader gets a term, then that term must have been written by an honest writer, or a writer must have just gone corrupt.*

The protocol is implemented in the file system Plutus with RSA cryptography. Unfortunately, the implementation is not faithful to the model above. In particular, terms that correspond to read-keys can be derived from other terms that are available to both readers and writers. Consequently, writers can become readers. We expect a weaker secrecy property in this model. Informally,

> *If the attacker gets a term intended to be secret, then that term must have been written by an honest writer, and a reader or a writer must have gone corrupt after that term was written.*

However proving this theorem for a detailed model of the Plutus implementation in ProVerif appears to be difficult. Specifically, overloading several primitive cryptographic equations in the model appears to cause termination problems. However we can prove this theorem for a model of an abstract implementation in ProVerif. We expect to manually prove the preservation of security properties of interest for this reduction, as in Section 2.1.1.

```
let processOwner =
                  (* owner creates a new group and serves requests for that group *)
    new sk;
    ...
    ( ... |       (* initializes version to zero *)
     ( ! ... )    (* revokes keys, increments version *)
    |
     ( !
         ...      (* if r is not a revoked reader and if the current version is not less than v *)
         event isreader(sk, r, v);
         ...      (* returns read-key (skgen(v, sk), v) to reader r on a secure channel *)
     )
    |
     ( !
         ...      (* if w is not a revoked writer and if the current version is not less than v *)
         event iswriter(sk, w, v);
         ...      (* returns write-key (pkgen(v, sk), v) to writer r on a secure channel *)
     )
    ).

let processHonestReader =
                  (* reader r obtains a read-key (rkey,v) and uses key to read file *)
    ...
    in(f,crypt);
    let n = read(f,crypt,rkey) in
    event gets(r, n, v).

let processDishonestReader =
                  (* reader r obtains a read-key (rkey,v) and leaks key to the attacker *)
    ...
    event corrupt(r,v);
    out(c, rkey).

let processHonestWriter =
                  (* writer w obtains a write-key (wkey,v) and uses key to write file *)
    ...
    new m;
    event puts(w, m, v);
    ! out(f,write(f,m,wkey)).

let processDishonestWriter =
                  (* writer w obtains a write-key (wkey,v) and leaks key to the attacker *)
    ...
    event corrupt(w,v);
    out(c, wkey).

let processUnwind =
                  (* unwinds a read-key to a read-key for a previous version *)
    ...

process
     ( (! processOwner)
     | (! processHonestReader) | (! processDishonestReader)
     | (! processHonestWriter) | (! processDishonestWriter)
     | (! processUnwind)
     )
```

Figure 1: Principals as applied pi calculus processes in ProVerif

2.1.3 Automatic logical analysis of label-based access controls in an operating system[3]

Next we outline a study of a commercial operating system with dynamic label-based access control. There are access control models that guarantee protection against information flow vulnerabilities by design. These include the *multi-level security* (MLS) models of Bell-La Padula [16] for confidentiality and Biba [17] for integrity. In these models, processes and resources are associated with immutable markers called *labels*. Labels denote trust levels as well as protection boundaries, and form a lattice—the simplest such lattice has two elements `LO` and `HI`. Processes and resources created by trusted users are given the label `HI`, and those created by untrusted users are given the label `LO`. Access control policies based on labels are enforced throughout the system. For instance, a strict MLS policy mandates that a `HI` process can only read, write and execute `HI` resources, and a `LO` process can only read, write and execute `LO` resources, so that bad information flows are prevented by design. However conventional MLS models are impractical. It can often be advantageous to change labels of either processes or resources depending on the security context. For example, a file downloaded from the Internet can receive the label `LO` (untrusted), but can later be *classified* by elevating the label to `HI` if the integrity of the file can be established by other means (e.g., by validating a digital certificate). Conversely, a resource can be created by a higher-level process and *declassified* for use by a lower-level process. A process created by a high-level principal can run with a lower label to implement the principle of *least privilege* (which states that a process should be given only the required level of privilege to execute its task at hand) and subsequently access high-level resources via explicit label elevations.

The Microsoft Windows Vista operating system (henceforth referred to as "Vista") implements MLS with integrity labels. The MLS model features customized rules that constrain how labels can evolve over time. The model aims to prevent privilege-escalation attacks and code tampering by downloaded viruses, among other flows. At the same time, labels are flexible enough to allow safe classification and declassification. For example, if f is an untrusted executable downloaded from the Internet and consequently labeled `LO`, an administrator can still execute f by creating a process p, declassifying p to `LO`, and subsequently execute f in p with `LO` privileges. As such, executing f cannot do much damage to the system (since sensitive operations such as deleting files or making privileged system calls are denied to a `LO` thread).

While dynamic label changes enable more functionality, they also open the door for information-flow vulnerabilities that can be exploited over time. For example, the `LO` thread p running the downloaded file f as described above cannot directly modify a `HI` resource r—yet, it can trick another `HI` process q (perhaps via a `LO` graphical interface) to classify f to `HI`, and then execute f. In particular, this thread can now modify the `HI` resource r.

Designers of MLS systems experiment with various rules for classification and declassification. For each such rule, there might be hidden vulnerabilities that can be exploited by changing labels on existing processes and resources, or by creating new processes and resources, or both. We provide a uniform analysis technique to study such rules, and automatically detect all possible runtime exploits on information flow properties. In particular, we use this technique to analyze the Vista MLS model and enumerate various possible attacks on this model.

At the core of our technique is a new language called Dynamic Datalog, that extends standard constrained Datalog with temporal operators for creating and modifying simple objects. We code information flow violations as queries in this language, and use query evaluation to find potential attacks. Dynamic Datalog has some carefully designed restrictions—names can be created only through unary predicates, only unary predicates can be transformed, and transition guards need to satisfy some monotonicity conditions. (Fortunately all these restrictions are satisfied by Vista's label dynamics.) We show that query evaluation for Dynamic Datalog is decidable. Our crucial insight is that with these restrictions, it is possible to reduce the temporal query evaluation problem to the query satisfiability problem in a subset of standard (static) Datalog. Then, we adapt a decision procedure due to Halevy *et al* [50] to decide this satisfiability problem.

[3]The material presented here is part of ongoing work that has not yet been published.

2.1.4 Related work

Various cryptographic implementations of distributed access control have been proposed as part of the security designs of NAS [43, 74, 63, 80, 71, 59]. However, the security analyses of these implementations have been at best semi-formal. An exception is Gobioff's security analysis of a NAS protocol using belief logics [43].

Abadi and Lamport study the concept of refinement mappings for correct implementations [11]. There, a specification implements another if the behaviors of the former are contained in those of the latter. We borrow some of their ideas in our study of NAS, although their notion of correctness does not strictly apply in our case. Abadi studies the concept of full abstraction [64] for secure implementations [2]. We use variants of the pi calculus [65] to study security properties of systems; related techniques have been developed in [7]. We employ a variation of may-tests to observe the behavior of systems. Proofs based on may-testing for safety and security properties have also been studied elsewhere (*e.g.*, [70, 10]).

Various cryptographic implementations of decentralized access control have been proposed as part of the security designs of untrusted storage [61, 55, 40, 41]. Again, the security analyses of most of these implementations have been at best semi-formal. An exception is the work of Mazières and Shasha on data integrity for untrusted storage [61]. There, they formalize a notion of integrity called *fork consistency*, and propose a protocol that realizes this notion. Informally, fork consistency implies that the storage server cannot act dishonestly in an undetectable manner; any such act prevents certain users from seeing any future version of certain files. Several systems with untrusted storage (including Plutus [55]) rely on this protocol for data integrity. Another exception is the work on computational proofs of lazy revocation schemes for untrusted storage by Backes *et al.* [14, 13]. The schemes in these studies build on that proposed in [55] and described in Section 2.1.2, and are proved to be computationally secure. In contrast, we focus on symbolic security; symbolic security proofs often have computational counterparts, as first studied in [12].

Blanchet's Proverif [19] is a powerful tool that can analyze security protocols written in the applied pi-calculus, and has been successfully applied in several contexts [5, 58]. Logical analyses of access control models also appear elsewhere (*e.g.*, [34, 75, 68]).

2.2 On access control and types for security

We now present some initial work on type systems that exploit access control for secrecy and integrity. Specifically, we consider a typed pi calculus with file-system constructs where secrecy relies on an interplay between permissions and knowledge of names. We then consider a system of polymorphic types for a concurrent object calculus that allows methods to implement dynamic specifications via safe dynamic access control. Finally we consider a integrity-typed calculus for protection in an operating system with label-based access control.

2.2.1 Permissions and secrecy types in a file system environment

Secrecy properties can be guaranteed through a combination of static and dynamic checks. We present a study of the interplay of such checks in a pi calculus with file-system constructs. The calculus supports both access-control 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 of secrecy types for the calculus. While the typing is static, it applies to a program subject to dynamic access-control checks.

We consider "ideal" storage systems similar to those of Section 2.1.1, where clients interact among themselves and with a common file system. The file system organizes files in directories. We assume that clients are indexed by a set ∞, so that the set of clients is $\{C_k \mid k \in \infty\}$. There is a set of channels $\{\beta_k \mid k \in \infty\}$ on which clients may send requests to the file system. Upon receiving a request on β_k, the file system decides whether the request is allowed by the access-control policy for the index k. Access-control rules come in three flavors: rules of the first kind give access rights to specific files; rules of the second kind give default access rights to arbitrary files under specific directories; rules of the third kind give rights to assert rules of the other two kinds. We focus on two operations on file contents, read and write, and an operation grant on file permissions.

We refer to certain subsets of client indices as groups. Some of those sets are induced by an access-control policy—for instance, the set 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 let a proper subset $\mathcal{I}$ of ∞ index honest clients. The remaining clients are assumed to be dishonest; in general they may make up an unknown, arbitrary attacker. *Secrecy groups* span either only subsets of honest clients (thus excluding all dishonest clients) or all clients (the group "public").

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 secrecy 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.

Let $\mathcal{G}$ range over subsets of $\mathcal{I}$, and $\mathcal{H}$ range over $\mathcal{G}$ and ∞. The grammar of types is as follows.

$T^\nu ::=$	declared types
$\quad \mathcal{G}[\widetilde{T}]$	polyadic channel
$\quad \mathsf{Un}$	untrusted
$\quad \mathcal{H}\{T\}$	file name
$T ::=$	types
$\quad T^\nu$	declared type
$\quad \mathcal{H}'/\mathcal{H}$	directory name
$\quad \mathsf{Wr}(T)$	write contents
$\quad \mathsf{Rd}(T)$	read contents
$\quad \mathsf{Gr}_k$	grant permissions
$\quad \mathsf{Req}_i \ (i \in \mathcal{I})$	request channel
$\quad \#\mathcal{H}'/\mathcal{H}\{T\}$	file path

Declared types are types with which new names can be declared in the calculus. Types have the following informal meanings.

- The type $\mathcal{H}\{T\}$ is given to a file name; this typing means that the file name may be shared within the group $\mathcal{H}$, while the contents may be shared within the reach of the type T. This type construct is somewhat similar in form to the traditional type construct $\mathcal{G}[\widetilde{T}]$ for symmetric channels [26]. However, the name of a symmetric channel can never be less secret than its contents, since by knowing its name one can know what it carries. On the other hand, a file with a publicly known name may contain secrets, and these contents may be protected by dynamic access control.

- The type $\mathcal{H}'/\mathcal{H}$ is given to a directory name; this typing means that the directory name may be shared within the group $\mathcal{H}'$, and may contain files whose names may be shared within the group $\mathcal{H}$.

- The type $\#\mathcal{H}'/\mathcal{H}\{T\}$ is given to a file path $\mathsf{file}(d/f)$; this typing means that the directory name d has type $\mathcal{H}'/\mathcal{H}$, and the file name f has type $\mathcal{H}\{T\}$. The file path may be shared within the group $\mathcal{H}' \cap \mathcal{H}$, while the contents may be shared within the reach of the type T.

- The types $\mathsf{Wr}(T)$ and $\mathsf{Rd}(T)$ are respectively given to commands for writing and reading file contents of type T.
- The type Gr_k is given to commands that grant permissions to index k.
- The type Req_i is given to an honest client's request channel β_i.
- The type Un is given to untrusted terms, typically those that an attacker may know.

For a type T, the group within which the inhabitants of that type may be shared is given by its reach $\|T\|$, defined next:

$$\begin{array}{rclrclrcl}
\|\mathcal{G}[\widetilde{T}]\| & = & \bigcap \|\widetilde{T}\| \cap \mathcal{G} & \|\mathcal{H}'/\mathcal{H}\| & = & \mathcal{H}' & \|\mathcal{H}\{T\}\| & = & \mathcal{H} \\
\|\mathsf{Rd}(T)\| & = & \|T\| & \|\mathsf{Wr}(T)\| & = & \|T\| & \|\mathsf{Gr}_k\| & = & \infty \\
\|\#\mathcal{H}'/\mathcal{H}\{T\}\| & = & \mathcal{H}' \cap \mathcal{H} & \|\mathsf{Req}_i\| & = & \{i\} & \|\mathsf{Un}\| & = & \infty
\end{array}$$

Any type whose reach is ∞ is called a *public* type; a term that belongs to ∞ is a public term.

Typechecking a system involves typechecking client code and the file-system state under the same assumptions Γ. Specifically, typechecking imposes discipline on the code of honest clients and restricts the permissions of dishonest clients. Not surprisingly, the partition between honest and dishonest clients plays a central role in typechecking the system. Client code as well as the file-system state generate typing constraints that finally determine whether the partition is valid, *i.e.*, whether all honest clients are well-typed, and whether permissions are 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 permissions. 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.

Informally, typing ensures that any term used by an honest client C_i must have a type whose reach includes i. A subtyping rule allows any public type to be treated as untrusted. As a result, public directories and public file names may be used by and shared between all clients. Subtyping is not invertible; in particular, an honest client may not receive a public file name on an untrusted channel and then request a file operation on it. Typing also ensures that dishonest clients can access only those public files whose contents are public. Accordingly, dishonest clients may not have default permissions on public directories with public files. Further, dishonest clients may not grant themselves any of these potentially dangerous permissions.

A well-typed system remains well-typed during execution. This property is crucial to our main result. 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 [4, 26] for similar analyses). Let S range over such sets of names.

Definition 2.3 (S-adversary)**.** *A closed process E is an S-adversary if all declared types in E have reach ∞ and* $\mathsf{fn}(E) \subseteq S$.

Next, we provide a definition of secrecy, using the usual notion of escape (similar to that in, *e.g.*, [4, 26]). A term is revealed if it may eventually be published on a channel known to the adversary. The notion of adversary is slightly generalized.

Definition 2.4 (Secrecy)**.** *Let P and Q be closed processes, S be a set of names, ζ be a file-system state, and u be a closed term. Let $\widetilde{n} = \mathsf{fn}(u) - \mathsf{fn}(P, Q, S, \zeta)$.*

1. *Under the assumptions $\widetilde{n} : \widetilde{T}$, P reveals u to (Q, S) via ζ if $P \mid Q \mid E \mid \zeta \longrightarrow^\star \xrightarrow{(\nu\widetilde{n})\,\overline{c}\langle u\rangle}$ for some S-adversary E and $c \in \mathsf{fn}(Q) \cup S$.*
2. *If u has type T under assumptions Γ and $\|T\| \subseteq \mathcal{G}$, then u is a $(\Gamma \triangleright \mathcal{G})$-secret.*

Theorem 2.5 (Secrecy by typing and access control)**.** *Suppose that the file-system state ζ and each honest client C_i ($i \in \mathcal{I}$) are well-typed under assumptions Γ. Further suppose that $\|\Gamma(s)\| = \infty$ for each $s \in S$. Then for any trusted group $\mathcal{G}$ and fresh assumptions Γ', $\Pi_{i \in \mathcal{G}} C_i$ never reveals a $(\Gamma, \Gamma' \triangleright \mathcal{G})$-secret to $(\Pi_{i' \in \mathcal{I} - \mathcal{G}} C_{i'}, S)$ via ζ.*

In other words, in a well-typed system a secret meant to be shared only within a subset of honest clients is never revealed to the other (honest and dishonest) clients.

2.2.2 Dynamic access control and polymorphic types in a concurrent object calculus

Our next topic is a study of dynamic access control for safe administration and usage of shared resources. We use a convenient characterization of access control in terms of capabilities: a resource may be accessed if and only if a corresponding capability is shown for its access. This view provides an immediate low-level abstraction of access control "by definition"; moreover the view is independent of higher level specifications on resource usage (say, in terms of types, or identities of principals). The separation facilitates higher level proofs, since it suffices to guarantee that the flow of a capability that protects a resource respects the corresponding high-level intention on resource usage. The proofs in turn rely on a sound low-level implementation of access control in terms of capabilities. Fortunately, to that end, a capability for a resource can be identified with a pointer to that resource. Exporting a direct link to a resource, however, poses problems for dynamic access control, as discussed by Redell in his dissertation (1974). Redell suggests a simple alternative that uses indirection: export indirect pointers to a local, direct reference to the resource, and overwrite this local pointer to modify access to that resource [73]. We revisit that idea here.

We study safe dynamic access control in a concurrent object language. Resources are often built over other resources; dependencies between resources may entail dependencies on their access assumptions for end-to-end safety. A natural way to capture such dependencies is to group the related resources into objects. We develop a variant of Gordon and Hankin's concurrent object calculus **conc**ς [45] for our study. In **conc**ς, as in most previous object calculi (*e.g.*, [6, 20, 77]), a method is accessed by providing the name of the parent object and a label that identifies the method. For example, for a timer object t in the calculus, with two methods, tick and set, knowing the name t is sufficient to call (or even redefine) both methods (t.tick, t.set). We may, however, want to restrict access to set to the owner of t, while allowing other users to access tick; such requirements are not directly supported by **conc**ς.

Our calculus provides *veils* for implementing flexible access control of methods. Veils are similar to Redell's indirect access pointers. More specifically, a veil is an alias (or "handle") for the label that identifies a particular method inside an object definition. A method is invoked by sending a message on its veil; method access is modified by re-exporting a different veil for its label. A method call crucially does not require the name of the parent object. An object name, on the other hand, is required for access modification and redefinition of methods—thus object names are similar to Redell's local references (or "capabilities"). In the sequel, informally, a *capability* is a reference to an object, and veils are indirect references to its methods. A capability is meant to be shared between the owner and other administrators of an object, and veils are meant to be made available to the users of its methods. Dependencies between object methods often require their redefinitions and access modifications to be simultaneous—thus the calculus features a general "administration" primitive.

We show a type system for the resulting language that guarantees safe manipulation of objects with respect to dynamically changing specifications. Informally, we allow object methods to change their exported "type views" at runtime via dynamic binding. Resource administrators can not only control resource usage at runtime, but also dynamically specify why they do so. This flexibility is desirable since persistent resources (*e.g.*, file systems, memory) are typically used in several different contexts over time. For example, files are often required to pass through intervals of restricted access; memory locations are dynamically allocated/deallocated to map different data structures over several program executions. By a combination of access control (as provided by the language) and static discipline (provided by the type system) we can show that the intentions of the users and administrators of those resources are respected through and between such phases of flux. In particular, by labeling types with secrecy groups, we show that well-typedness guarantees secrecy under dynamic access control, even in the presence of possibly untyped, active environments.

We begin by presenting the calculus. The syntax is as follows.

$u, v ::=$	results
x	variable
n	name (capability or veil)
$d ::=$	denotations
$\widetilde{u}[\widetilde{\ell} \Mapsto \varsigma(x)\widetilde{(y)b}]$	object
$a, b ::=$	expressions
u	result
$n \longmapsto d$	denomination
$(\nu n)\, a$	restriction
$a \upharpoonright b$	fork
let $x = a$ in b	evaluation
$\ell(u)$	internal method call
$\ell \Mapsfrom (y)b$	internal method update
$\tilde{v}\langle u \rangle$	external method call
$u \mapsfrom d$	external update ("administration")

In the object $\widetilde{u}[\widetilde{\ell} \Mapsto \varsigma(x)\widetilde{(y)b}]$, the variable x abstracts "self"; for each label $\ell_i \in \widetilde{\ell}$, u_i is a veil for the method identified by that label; the method's body b_i takes the parameter y_i. Expressions have the following informal meanings.

- u is a result (a variable or name) that is returned by an expression.
- $p \longmapsto d$ attaches the capability p to an object d.
- $(\nu n)\, a$ creates a new name n that is bound in the expression a, and executes a.
- $a \upharpoonright b$ is the (non-commutative) parallel composition of the expressions a and b; it returns any result returned by b, while executing a for side-effect. This form, introduced in [45], is largely responsible for the compactness of the syntax, since it provides an uniform way to write expressions that return results, and "processes" that exhibit behaviors. (Of course, expressions that return results can also have side-effects.)
- let $x = a$ in b binds the result of the expression a to the variable x and then executes the expression b; here x is bound in b.
- $\ell(u)$ means a local method call inside an object; see external call.
- $\ell \Mapsfrom (y)b$ means a local method update inside an object; see external update.
- $\tilde{v}\langle u \rangle$ means an external call on the veil v, with argument u; in the presence of a denomination $p \longmapsto d$ where d exports v for a defined method, the corresponding method expression is exported by substituting veiled calls for internal calls, self updates for internal updates, p for the abstracted self variable, and u for the formal parameter.
- $u \mapsfrom d$ means an external update on the capability u; in the presence of a denomination $u \longmapsto d'$, the veils exported by d replace those exported by d', and the methods defined by d augment or override those defined by d'; the capability u is returned.

As mentioned above, we allow methods to change types at runtime: the type of a method corresponds dynamically to the type of the veil it exports. The following general principles govern our type discipline. One, an object update is consistent only if the types of the new veils match up with the types of the method definitions. Two, type consistency forces some methods to be overridden—we call these methods "flat".

Methods whose types are parametric with respect to the types of the flat methods need not be overridden. 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. We call these methods "natural". The primary goal of the type discipline is *flux robustness*, *i.e.*, type safety despite dynamic changes to type assumptions for methods. Dynamic access control is used in an integral way to enforce safety.

To specify and verify secrecy, we introduce a universe of indices ∞ and secrecy groups as in Section 2.2.1. Let $\mathcal{G}$ range over proper subsets of ∞ or group variables (ranged over by $\mathcal{X}$), and $\mathcal{H}$ range over $\mathcal{G}$ and ∞. Let ρ range over the qualifiers "flat" and "natural".

$S, T ::=$	types
X	type variable
$\mathsf{Obj}^{\mathcal{G}}[\widetilde{\ell} : \widetilde{(S)T^{\rho}}]$	capability type scheme
$\mathsf{Veil}^{\mathcal{G}}(u.\ell : (S)T)$	veil type
$(\exists x)T$	dependent union type
Null	null type
Un	untrusted type

Typed processes declare types for new names. Types have the following informal meanings.

- X ranges over type variables. Group and type variables appear in capability signatures (see below).

- A capability signature $\mathsf{Obj}^{\mathcal{G}}[\widetilde{\ell} : \widetilde{(S)T^{\rho}}]$ is a type scheme that assigns types $(S_i)T_i$ and qualifiers ρ_i to the methods $\ell_i \in \widetilde{\ell}$ of a denoted object. The group $\mathcal{G}$ corresponds to the set of administrators for that object. The scheme binds group and type variables that are shared by the types of the methods in the signature. We interpret a type scheme as an universally quantified type over its bound variables, while leaving the bound variables implicit (*à la* polymorphic types in ML [66]).

- A veil type $\mathsf{Veil}^{\mathcal{G}}(u.\ell : (S)T)$ is dependent on a capability u, and instantiates the type scheme for a method ℓ in the signature of that capability. The veil expects an argument of type S and returns a result of type T. The group $\mathcal{G}$ corresponds to the set of users—the "access-control list"—for the method referenced by the veil. We use dependence in the veil type to prevent the same veil from being exported by different objects. (A similar "no-confusion" property is required, for instance, of datatype constructors [32].)

- Dependent union types $(\exists x)T$ allow capability dependencies to be passed without explicit communication of the capabilities themselves. The type system thus supports the separation of roles of veils and capabilities (as intended) despite enforcing necessary dependencies between them.

- The type Null is given to an expression whose result, if any, is ignored.

- Finally, the type Un is given to any expression whose result, if any, is untrusted.

As before, the relationship between types and groups is made explicit by a *reach* function. Moreover we define group and type substitution; it is mostly standard, except for the substitution of ∞ for a group variable that annotates a type, which "rounds off" that type as untrusted.

We prove that well-typed code never leaks secrets beyond declared boundaries, even under arbitrary untrusted environments. As usual, the result relies on a standard but non-trivial preservation property: well-typed expressions preserve their types on execution.

2.2.3 Labels and integrity types in an operating system environment[4]

Next we outline a study of protection provided by a commercial operating system by combining runtime mechanisms with static discipline. The mechanisms are as discussed in Section 2.1.3; they complement con-

[4]The material presented here is part of ongoing work that has not yet been published.

ventional MLS controls with some non-standard features, *e.g.*, mutability of labels, for richer functionality. These features open the door for vulnerabilities that are eliminated by static discipline. For this purpose, we design a calculus that can express dynamic creation and relabeling of threads and locations, packing and execution of binaries, and other usual operating system constructs. In particular, the calculus may be seen as an intermediate language for writing and executing code on (an abstraction of) Vista. More broadly, the language is a concurrent imperative calculus with dynamic mandatory access control. We design a type system for this language that guarantees protection of trusted code from arbitrary untrusted environments. Protection is formalized as an integrity property on locations: loosely, locations whose contents are trusted can never contain data that flows from untrusted code. The type system itself formally articulates some of the so-called best practices in Vista [1], among other principles. The dynamics of labels pose several technical challenges (going well beyond standard declassification [67]). Indeed, labels protect both code and locations, and locations can in turn contain executable code—and in the untyped language, label casts can lead to integrity violations in various ways. We obtain a standard but non-trivial preservation theorem for the type system that provides the desired runtime protection guarantees.

2.2.4 Related work

Static analyses have been quite helpful in guaranteeing high-level safety properties of distributed systems: indeed, a significant body of work focuses specifically on safe resource usage [38, 21, 54, 57, 62, 22, 23, 24, 72]. Some analyses use access levels, as declared via static type annotations, to guarantee the absence of access violations at runtime [53, 69, 31, 23, 72]. In contrast, our type systems do not guarantee the success of access checks; indeed, type soundness depends on the failure of some of those checks. A similar approach is reflected in hybrid typechecking [37].

Our secrecy type systems borrow the concept of groups from an intermediate type system developed in the study of group creation [26]. Kirli's mobility regions [56] for distributed functional programs are also similar to groups as presented here. Bugliesi *et al.* develop another calculus that uses group creation to specify discretionary access-control policies [24]. Ideas similar to group creation also appear in the work of Braghin *et al.* on a calculus for role-based access control [22]. It is however not immediate to see how to apply these approaches to our setting. In [24], for example, clients declare access groups; in our setting clients can instead declare end-to-end secrecy intentions, and it is possible to verify that access control respects such intentions. Here secrecy is not defined as the absence of certain flows of information (that is, as some sort of non-interference property [44]). Rather, secrecy is presented as the impossibility of certain communication events (for instance, sending a message that contains a particular sensitive value). On the other hand, Banerjee and Naumann examine the use of access control for secure information flow [15].

Several other works emphasize distribution. Thus, in the language KLAIM [33], a type system checks that processes have been granted the necessary right to perform operations at specified localities [69]. Hennessy and Riely describe a typing system for a distributed pi-calculus that ensures that agents cannot access the resources of a system without first being granted the capability to do so [53]. Bugliesi *et al.* explore access-control types for the calculus of boxed ambients [23] with a typing relation similar in form to ours, but without dynamic access control—access control is specified in terms of static security levels.

Some other works allow specifications to be dynamic to reflect changing assumptions during execution. Dynamic specifications are often desirable when reasoning about persistent resources. When additional runtime guarantees can be exploited, dynamic specifications typically also allow finer analyses than static specifications. Along those lines, one body of work studies the enforcement of policies specified as security automata [76, 51]. Yet another studies systems with declassification, *i.e.*, conservative relaxation of secrecy assumptions at runtime [67]. Ling and Zdancewic [60] study some intriguing aspects of integrity in the presence of declassification. There is also some recent work on compromised secrets [47, 49] in the context of network protocols. In comparison, our analyses apply more generally to changing assumptions at runtime. Perhaps closest to our work are analyses developed for dynamic access control in languages with locality and migration [52, 48]. Similar ideas also appear in a type system for noninterference that allows the use of dynamic security labels [79]. Polymorphism in the context of dynamic typing has been studied by Duggan in [35, 36].

3 Towards defense of the thesis

We now review our initial work in the light of the thesis stated in Section 1.1, and discuss work that remains to be done to defend that thesis. We also provide a rough timeline.

3.1 Discussion on preliminary results and remaining work

In Section 2.1 we outline correctness proofs for three contrasting implementations of access control. The first of these implementations appears in a storage design with networked storage servers. There, the focus is on distributed enforcement of access control. Permissions are communicated from the site of maintenance to the site of enforcement via cryptographically secured capabilities. The second access-control implementation appears in a storage design with untrusted storage servers. There, the focus is on decentralized mechanisms for discretionary access control. Permissions are explicitly distributed as file-access keys, and files are cryptographically secured so that they can be accessed only by the correct keys. The third access-control implementation appears in a commercial operating system. There, the focus is on system-wide mandatory enforcement of access control. Permissions are based on dynamic protection labels.

These case studies highlight contrasting access-control designs. Each of those designs is appropriate for the underlying system with its particular assumptions and guarantees. Further, these studies highlight complementary proof techniques. The first of these is a reduction technique that simplifies security proofs for the systems under study by showing those systems preserve security properties of much simpler systems. The second technique uses an automatic tool ProVerif to prove concrete security properties for the systems under study. The third technique uses a decidable logic to automatically prove or disprove security properties of several experimental variants of the systems under study.

It might be instructive to compare these techniques and clarify their limitations. We plan to apply these techniques to other examples. For example, it may be possible to apply the reduction technique to show that systems with untrusted storage and cryptography (as described in Section 2.1.2) preserve security properties of simpler systems. Specifically, in Section 2.1.2 we mention the apparent limitations of ProVerif in automatically proving security theorems for a detailed concrete model of the protocol implemented in Plutus; on the other hand ProVerif can automatically prove those theorems for an abstract model with lesser details. We expect that this situation can be aptly resolved by a manual reduction analysis that shows that the concrete model is a faithful refinement of the abstract model with respect to the security properties of interest.

On the other hand, it may be possible to analyze various label-based operating-system access controls in ProVerif, rather than in a decidable logic. ProVerif uses an underlying logical resolution engine over a language of Horn clauses with function symbols; this language is undecidable. The resolution engine incorporates several sound optimizations for efficiency. We expect that ProVerif can find attacks in some of these experiments quite fast; however we do not expect ProVerif to terminate in all the experiments, or to find all possible attacks in any particular experiment.

Next, in Section 2.2 we outline three type systems that exploit runtime access-control support in the underlying language to prove secrecy and integrity properties. The first of these studies explores the interplay of file permissions and knowledge of file names for secrecy. The second study explores the role of dynamic access control on object methods for type safety in the presence of objects that implement dynamic specifications. The third study explores a combination of static types and dynamic labels for protection in an operating system.

Each of these typing analyses complement conventional static discipline with dynamic checks provided by the language runtime. In other words, violations of intended properties that are not caught statically are always caught at runtime. This verification method seems to be quite useful in practice. We plan to further investigate these techniques. In particular, we are interested in applying the type discipline for objects with dynamic access control to study systems with untrusted storage and key-based access control as described in Section 2.1.2. Indeed in the calculus described in Section 2.2.2, veils seem to be particularly close to access keys—both provide a mechanism for dynamic discretionary access control. We plan to make this connection explicit by modeling protocols for untrusted storage in the language. The type system can then be used to reason about systems that implement those protocols. Another possible application of this type discipline is in memory management, and we may have a closer look at this topic in the future. We also plan to show an abstract machine for the calculus that demonstrates the feasibility of implementing the calculus using threads, stacks, and heaps.

We are interested in exploring if actual Vista binaries can be certified using a possibly extended version of the typed process calculus for protection in (an abstraction of) Vista mentioned in Section 2.2.3. Specifically, we envision a two-tier system, wherein a model extractor extracts a process expression (in an extended version of the process calculus) from the binary, and the type system operates on the process expression to find information-flow violations.

Going further, we may explore other connections between information flow and access control. We have already shown some techniques that exploit access control for information-flow properties. Perhaps information flow can be suitably exploited for access-control properties. Indeed certain integrity properties are necessary to ensure robust declassification [67].

While following up on these lines of work, we also plan to group related results and provide unifying perspectives on those results. Further, we plan to publish some of these results in archival journals. Finally, we plan to survey the state of the art on formal techniques for secure storage, and present a coherent story about our contributions in the field.

3.2 Timeline

We propose the following tentative timeline that roughly spans the next two years.

Winter 2007 Expect to finish our study of the protocol implemented in Plutus, and finish ongoing work described in Sections 2.1.3 and 2.2.3.

Spring 2007 Plan to further explore connections between information flow and access control, and follow up on work described in Sections 2.2.2, 2.1.3, and 2.2.3.

Summer 2007 There is a possibility of doing an internship. Otherwise, plan to prepare previous work for publication in journals.

Fall 2007 Expect to continue preparing previous work for publication in journals, and plan to study peer-to-peer file systems for anonymity.

Winter 2008 Plan to seek several examples beyond the ones we study where our techniques can be applied, and others where our techniques seem inadequate.

Spring 2008 Expect to complete a survey of the state of the art on formal techniques for secure storage.

Summer 2008 Expect to finish writing the dissertation.

Fall 2008 Expect to defend our thesis.

4 Conclusion

We conclude by summarizing what we expect to be the main contributions of this thesis, and by speculating on its likely impact on system security.

Based on this thesis, we expect to be able to argue that formal techniques are effective for specifying and verifying security properties of a variety of storage systems. Our contributions will partly lie in adapting existing analysis techniques developed for the study of secure communication, where possible. Other contributions will lie in motivating, developing, and applying new techniques to security problems that arise in storage systems due to dynamic access control. Finally, and perhaps most importantly, we expect to present a coherent view of the state of the art; indeed we hope that one will be able to comment on the applicability of the studied techniques to storage systems beyond those considered in the thesis.

Most systems today rely on a combination of communication and storage. While secure communication is fairly well understood, existing technology for secure communication is not enough for system-wide security. We believe that what is needed beyond that technology is an understanding of the foundations of access control for secure storage. From this perspective, we expect techniques and insights developed in this thesis to impact the analysis of system security.

References

[1] *Windows Vista Developer Center*. http://msdn.microsoft.com/windowsvista/.

[2] M. Abadi. Protection in programming-language translations. In *ICALP'98: International Colloquium on Automata, Languages and Programming*, pages 868–883. Springer, 1998.

[3] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. In *POPL'02: Principles of programming languages*, pages 33–44. ACM, 2002.

[4] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 298(3):387–415, 2003.

[5] M. Abadi, B. Blanchet, and C. Fournet. Just fast keying in the pi calculus. In *ESOP'04: European Symposium on Programming*, pages 340–354. Springer, 2004.

[6] M. Abadi and L. Cardelli. An imperative object calculus. In *TAPSOFT'95: Theory and Practice of Software Development*, pages 471–485. Springer, 1995.

[7] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL'01: Principles of Programming Languages*, pages 104–115. ACM, 2001.

[8] M. Abadi, C. Fournet, and G. Gonthier. Authentication primitives and their compilation. In *POPL'00: Principles of Programming Languages*, pages 302–315. ACM, 2000.

[9] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1):37–83, 2002.

[10] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: the spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[11] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[12] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[13] M. Backes, C. Cachin, and A. Oprea. Secure key-updating for lazy revocation. In *ESORICS'06: European Symposium on Research in Computer Security*, pages 327–346. Springer, 2006.

[14] M. Backes and A. Oprea. Lazy revocation in cryptographic file systems. In *SISW '05: Security in Storage Workshop*, pages 1–11. IEEE, 2005.

[15] 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.

[16] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE Corp., 1975.

[17] K. J. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, MITRE Corp., 1977.

[18] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. *ACM SIGOPS Operating Systems Review*, 27(5):217–230, 1993.

[19] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW'01: Computer Security Foundations Workshop*, pages 82–96. IEEE, 2001.

[20] P. D. Blasio and K. Fisher. A calculus for concurrent objects. In *CONCUR'96: Concurrency Theory*, pages 655–670. Springer, 1996.

[21] P. D. Blasio, K. Fisher, and C. Talcott. A control-flow analysis for a calculus of concurrent objects. *IEEE Transactions on Software Engineering*, 26(7):617–634, 2000.

[22] 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.

[23] M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: the calculus of boxed ambients. *ACM Transactions on Programming Languages and Systems*, 26(1):57–124, 2004.

[24] M. Bugliesi, D. Colazzo, and S. Crafa. Type based discretionary access control. In *CONCUR'04: Concurrency Theory*, pages 225–239. Springer, 2004.

[25] M. Burrows, M. Abadi, and R. 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.

[26] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. *Information and Computation*, 196(2):127–155, 2005.

[27] A. Chaudhuri. Dynamic access control in a concurrent object calculus. In *CONCUR'06: Concurrency Theory*, pages 263–278. Springer, 2006.

[28] A. Chaudhuri and M. Abadi. Formal security analysis of basic network-attached storage. In *FMSE'05: Formal Methods in Security Engineering*, pages 43–52. ACM, 2005.

[29] A. Chaudhuri and M. Abadi. Formal analysis of dynamic, distributed file-system access controls. In *FORTE'06: Formal Techniques for Networked and Distributed Systems*, pages 99–114. Springer, 2006.

[30] A. Chaudhuri and M. Abadi. Secrecy by typing and file-access control. In *CSFW'06: Computer Security Foundations Workshop*, pages 112–123. IEEE, 2006.

[31] T. Chothia, D. Duggan, and J. Vitek. Type-based distributed access control. In *CSFW'03: Computer Security Foundations Workshop*, pages 170–184. IEEE, 2003.

[32] T. Coquand. Pattern matching with dependent types. In *TYPES'92: Types for Proofs and Programs*. Informal record. Online version of the proceedings at `http://citeseer.ist.psu.edu/71964.html`, 1992.

[33] R. de Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[34] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *IJCAR'06: International Joint Conference on Automated Reasoning*, pages 632–646. Springer, 2006.

[35] D. Duggan. Dynamic typing for distributed programming in polymorphic languages. *ACM Transactions on Programming Languages and Systems*, 21(1):11–45, 1999.

[36] D. Duggan. Type-based hot swapping of running modules (extended abstract). In *ICFP'01: International Conference on Functional Programming*, pages 62–73. ACM, 2001.

[37] C. Flanagan. Hybrid type checking. In *POPL'06: Principles of programming languages*, pages 245–256. ACM, 2006.

[38] C. Flanagan and M. Abadi. Object types against races. In *CONCUR'99: Concurrency Theory*, pages 288–303. Springer, 1999.

[39] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. In *ESOP'05: European Symposium on Programming*, pages 141–156. Springer, 2005.

[40] K. Fu. *Integrity and access control in untrusted content distribution*. PhD thesis, Massachusetts Institute of Technology, 2005.

[41] K. Fu, S. Kamara, and Y. Kohno. Key regression: enabling efficient key distribution for secure distributed storage. In *NDSS'06: Network and Distributed System Security*. The Internet Society, 2006.

[42] G. A. Gibson, D. P. Nagle, K. Amiri, F. W. Chang, E. Feinberg, H. G. C. Lee, B. Ozceri, E. Riedel, and D. Rochberg. A case for network-attached secure disks. Technical Report CMU–CS-96-142, Carnegie Mellon University, 1996.

[43] H. Gobioff. *Security for a High Performance Commodity Storage Subsystem*. PhD thesis, Carnegie Mellon University, 1999.

[44] J. A. Goguen and J. Meseguer. Security policies and security models. In *SSP'82: Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.

[45] A. D. Gordon and P. D. Hankin. A concurrent object calculus: reduction and typing. In *HLCL'98: High-Level Concurrent Languages*, pages 248–264. Elsevier, 1998.

[46] A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300(1-3):379–409, 2003.

[47] A. D. Gordon and A. Jeffrey. Secrecy despite compromise: types, cryptography, and the pi-calculus. In *CONCUR'05: Concurrency Theory*, pages 186–201. Springer, 2005.

[48] 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, 2003.

[49] C. Haack and A. Jeffrey. Timed spi-calculus with types for secrecy and authenticity. In *CONCUR'05: Concurrency Theory*, pages 202–216. Springer, 2005.

[50] A. Y. Halevy, I. S. Mumick, Y. Sagiv, and O. Shmueli. Static analysis in datalog extensions. *Journal of the ACM*, 48(5):971–1012, 2001.

[51] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .net. In *PLAS '06: Programming languages and analysis for security*, pages 7–16. ACM, 2006.

[52] M. Hennessy, M. Merro, and J. 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, 2003.

[53] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In *HLCL'98: High-Level Concurrent Languages*, pages 174–188. Elsevier, 1998.

[54] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Transactions on Programming Languages and Systems*, 24(5):566–591, 2002.

[55] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: scalable secure file sharing on untrusted storage. In *FAST'03: File and Storage Technologies*, pages 29–42. USENIX Association, 2003.

[56] Z. D. Kirli. Confined mobile functions. In *CSFW'01: Computer Security Foundations Workshop*, pages 283–294. IEEE, 2001.

[57] J. Kleist and D. Sangiorgi. Imperative objects as mobile processes. *Science of Computer Programming*, 44(3):293–342, 2002.

[58] S. Kremer and M. Ryan. Analysis of an electronic voting protocol in the applied pi calculus. In *ESOP'05: European Symposium on Programming*, pages 186–200. Springer, 2005.

[59] A. W. Leung and E. L. Miller. Scalable security for large, high performance storage systems. In *StorageSS'06: Storage Security and Survivability*, pages 29–40. ACM, 2006.

[60] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *POPL'05: Principles of Programming Languages*, pages 158–170. ACM, 2005.

[61] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *PODC'02: Principles of Distributed Computing*, pages 108–117. ACM, 2002.

[62] G. Miklau and D. Suciu. Controlling access to published data using cryptography. In *VLDB'03: Very Large Data Bases*, pages 898–909. Springer, 2003.

[63] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. Reed. Strong security for network-attached storage. In *FAST'02: File and Storage Technologies*, pages 1–13. USENIX Association, 2002.

[64] R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4(1):1–22, 1977.

[65] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[66] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT, 1997.

[67] A. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *CSFW'04: Computer Security Foundations Workshop*, pages 172–186. IEEE, 2004.

[68] P. Naldurg, S. Schwoon, S. Rajamani, and J. Lambert. Netra: seeing through access control. In *FMSE'06: Formal Methods in Security Engineering*, pages 55–66. ACM, 2006.

[69] R. D. Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, 2000.

[70] R. D. Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1–2):83–133, 1984.

[71] C. Olson and E. L. Miller. Secure capabilities for a petabyte-scale object-based distributed file system. In *StorageSS'05: Storage Security and Survivability*, pages 64–73. ACM, 2005.

[72] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems*, 27(2):344–382, 2005.

[73] D. D. Redell. Naming and protection in extendible operating systems. Technical Report MAC-TR-140, Massachusetts Institute of Technology, 1974.

[74] B. C. Reed, E. G. Chron, R. C. Burns, and D. D. E. Long. Authenticating network-attached storage. *IEEE Micro*, 20(1):49–57, 2000.

[75] B. Sarna-Starosta and S. D. Stoller. Policy analysis for security-enhanced linux. In *WITS'04: Workshop on Issues in the Theory of Security*, pages 1–12. Informal record. Online version of the paper at `http://www.cs.sunysb.edu/~stoller/WITS2004.html`, 2004.

[76] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[77] V. T. Vasconcelos. Typed concurrent objects. In *ECOOP'94: European Conference on Object-Oriented Programming*, pages 100–117. Springer, 1994.

[78] T. Y. C. Woo and S. S. Lam. A lesson on authentication protocol design. *ACM SIGOPS Operating Systems Review*, 28(3):24–37, 1994.

[79] L. Zheng and A. Myers. Dynamic security labels and noninterference. In *FAST'04: Formal Aspects in Security and Trust*, pages 27–40. Springer, 2004.

[80] Y. Zhu and Y. Hu. SNARE: A strong security scheme for network-attached storage. In *SRDS'03: Symposium on Reliable Distributed Systems*, pages 250–259. IEEE, 2003.