On Secure Distributed Implementations of Dynamic Access Control

Avik Chaudhuri
University of California at Santa Cruz
avik@cs.ucsc.edu

Abstract

Distributed implementations of access control abound in distributed 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 standard protocol based on capabilities, that can break security under a simple specification of access control. At the same time, we show a natural refinement of the specification for which a secure implementation of access control is possible. Our models and proofs are formalized in the applied pi calculus, following some new techniques that may be of independent interest.

Keywords dynamic access control, capabilities, testing equivalence, full abstraction

1 Introduction

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 paper, we reason about the extent to which such access control implementations preserve the character of local access checks. 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) [15, 17] 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 in this setting; dynamic access policies cause further problems that require considerable analysis to iron out. We study these cases separately, in detail, in Sections 2 and 3. 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 discover several problems in a standard implementation that cause safety and security to break under a simple specification of access control. On the other hand, we find a natural refinement of the specification under which a safe and secure implementation is possible.

We formalize our results in the applied pi calculus [2]. Basically, the correctness theorems imply that safety and security proofs for the specification carry over "for free" to the implementation. Our correctness proofs are built modularly, with carefully designed simulation proofs; we develop the necessary concepts and proof techniques in Section 4, and outline the proofs in Section 5.

From a high-level perspective, 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; further, we distill the key ideas behind those attacks and propose corrections in terms of useful design principles (Sections 2 and 3). We indicate how other state machines can be distributed just as well using those principles (Section 6).

Comparison with related work This paper culminates a line of work that we begin in [12] and continue in [13]. In [12], we show how to securely implement static access policies with capabilities; in [13], we present a safe (but not secure) implementation of dynamic access policies in that setting. In this paper, we carefully review those results, and 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 [13] that we can prove secure with appropriate assumptions. Further, guided by our analysis of access control, we outline how to automatically derive secure distributed implementations of other state machines. This approach is reminiscent of secure program partitioning [24], and deserves further investigation.

Access control for networked storage has been studied in lesser detail by Gobioff [15] using belief logics [10], and by Halevi, Karger, and Naor [17] using universal composability [11]. The techniques used in this paper are similar to those used by Abadi, Fournet, and Gonthier for secure implementation of channel abstractions [3] and authentication primitives [4], and by Maffeis to study the equivalence of communication patterns in distributed query systems [19]. These techniques rely on programming languages concepts, including testing equivalence [23] and full abstraction [1, 21]. 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 [18] in this paper. In file systems for untrusted storage, 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 [9, 20], as well as correctness proofs for the cryptographic techniques involved in such protocols [7, 8, 14].

2 Review: the case of static access policies

To warm up, let us focus on implementing access policies that are *static*. In this case, a secure implementation already appears in [12]. Below, we systematically reconstruct that implementation, focusing on a new, 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[1], NS^s, for networked storage. This protocol captures the essence of the NASD and OSD protocols [15, 17]; as we move along, we present more complicated variants of this protocol. Principals include users $U, V, W \ldots$, an access-control server A, and a storage server S. We assume that A privately maintains a static access policy Γ and S privately maintains a store ρ. Access decisions under Γ follow the relation $\Gamma \vdash_U op$ over users U and operations op. Execution of an operation op under ρ follows the 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 $\mathbf{mac}$ be a function over messages and keys that produces unforgeable message authentication codes (MACs) [16]. 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] In protocol names throughout this paper, 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.

$$
\begin{array}{llllll}
(1) & U & \rightarrow & A & : & op \\
(2) & A & \rightarrow & U & : & \mathbf{mac}(op, K_{AS}) \quad \text{if } \Gamma \vdash_U op \\
(2') & A & \rightarrow & U & : & \texttt{error} \quad \text{otherwise}
\end{array}
$$

$$
\begin{array}{llllll}
(3) & V & \rightarrow & S & : & \kappa \\
(4) & S & \rightarrow & V & : & r \quad \text{if } \kappa = \mathbf{mac}(op, K_{AS}) \text{ and } \rho[\![op]\!] \Downarrow \rho'[\![r]\!] \\
(4') & S & \rightarrow & V & : & \texttt{error} \quad \text{otherwise}
\end{array}
$$

Here a user U requests A for access to an operation op, and A returns a capability for op only if Γ 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 $U, V, W, \ldots$ and a server D. The access policy Γ and the store ρ are both maintained by D; the access and execution relations remain as above. There is no cryptography.

$$
\begin{array}{llllll}
\text{(i)} & V & \rightarrow & D & : & op \\
\text{(ii)} & D & \rightarrow & V & : & r \quad \text{if } \Gamma \vdash_V op \text{ and } \rho[\![op]\!] \Downarrow \rho'[\![r]\!] \\
\text{(ii}') & D & \rightarrow & V & : & \texttt{error} \quad \text{otherwise}
\end{array}
$$

Here a user V requests D to execute an operation op, and V executes op only if Γ specifies that V may access op. This protocol is correct "by definition"; so if NS^s implements this protocol, it is correct too.

What correctness criteria are appropriate here? A basic criterion is that of safety [5].

Definition 1 (Safety). *Under any context (adversary), the behaviors of a safe implementation are included in the behaviors of the specification.*

In practice, a suitable notion of inclusion 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 (*cf.* full abstraction [1, 21]).

Definition 2 (Security). *A secure implementation preserves behavioral equivalences of the specification.*

In this paper, we tie security to an appropriate may-testing equivalence [23]. 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 [5] (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 their IS^s instances.

A safety counterexample usually suffices to break security. For instance, we are in trouble if operations that cannot be executed in IS^s can somehow be executed in NS^s by manipulating capabilities. Suppose that $\Gamma \not\vdash_V op$ 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. We know that 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 $\Gamma \vdash_U op$. Therefore:

R1 Capabilities obtained by honest users must not be shared with dishonest users.

(However U can still share 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 must not be examined (by applying destructors) or compared.

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 Γ; in other words, A leaks information on Γ whenever it returns a capability![2] 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.

$$(2') \quad A \quad \rightarrow \quad U \quad : \quad \mathbf{mac}(op, \overline{K}_{AS}) \quad \text{if } \Gamma \not\vdash_U op$$

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 Γ. To prevent this leak:

R4 Capabilities for different users must be different.

For instance, a capability can mention the user whose access it authenticates. Making the meaning of a message explicit in its content is a good design principle for security [6], and we use it on several occasions in this paper. Accordingly, we modify the following messages in NS^s.

$$\begin{array}{lllllll} (2) & A & \rightarrow & U & : & \mathbf{mac}(\langle U, op\rangle, K_{AS}) & \text{if } \Gamma \vdash_U op \\ (2') & A & \rightarrow & U & : & \mathbf{mac}(\langle U, op\rangle, \overline{K}_{AS}) & \text{otherwise} \\ \\ (4) & S & \rightarrow & V & : & r & \text{if } \kappa = \mathbf{mac}(\langle _, op\rangle, K_{AS}) \text{ and } \rho[\![op]\!] \Downarrow \rho'[\![r]\!] \end{array}$$

(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 (*cf.* [12]).

Theorem 1. *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. Fortunately, 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.

[2] If we do not care about leaking information on Γ, then we must allow the same leak in the specification (see Section 3).

3 The case of dynamic access policies

We now consider the general problem of implementing dynamic access policies. Let Γ be dynamic; the following protocol, NS^d, is obtained by adding administration messages to NS^s. Execution of an administrative operation θ under Γ follows the relation $\Gamma[\![\theta]\!] \Downarrow \Gamma'[\![r]\!]$ over next policies Γ' and results r.

$$\begin{array}{lllcll} (5) & W & \rightarrow & A & : & \theta \\ (6) & A & \rightarrow & W & : & r \qquad \text{if } \Gamma \vdash_W \theta \text{ and } \Gamma[\![\theta]\!] \Downarrow \Gamma'[\![r]\!] \\ (6') & A & \rightarrow & W & : & \texttt{error} \qquad \text{otherwise} \end{array}$$

Here a user W requests A to execute an administrative operation θ, and A executes θ (perhaps modifying Γ) if Γ specifies that W controls θ. The following protocol, IS^d, is obtained by adding similar messages to IS^s.

$$\begin{array}{lllcll} (\text{iii}) & W & \rightarrow & D & : & \theta \\ (\text{iv}) & D & \rightarrow & W & : & r \qquad \text{if } \Gamma \vdash_W \theta \text{ and } \Gamma[\![\theta]\!] \Downarrow \Gamma'[\![r]\!] \\ (\text{iv}') & D & \rightarrow & W & : & \texttt{error} \qquad \text{otherwise} \end{array}$$

Unfortunately, NS^d does not remain a secure implementation of IS^d. Consider the NS^d pseudo-code below. Informally, `acquire` κ means "obtain a capability κ" and `use` κ means "request execution with κ"; `chmod` θ means "request access modification θ"; and `success` means "detect successful use of a capability". Here κ is a capability for an operation op and θ modifies access to op.

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". (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`.) However, 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`.

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 for storing short-term secrets, for example. Further, it allows us to prove the following new result, without complicating capabilities at all (see Section 5).

Theorem 2. *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.

[3] Some implementation details, such as (R3), are not required for safety.

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 very least, we expect that capabilities need to be more sophisticated. Below, we show how to recover safety by introducing *time*.

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 arrangement a "midnight-shift scheme", since the underlying idea is the same as that of periodically shifting guards at a museum or a bank. Implementing this scheme is straightforward. To implement (R5), capabilities carry *timestamps*. To implement (R6), administrative operations are executed on a "scratchpad" 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+}.

$$
\begin{array}{llll}
(2) & A \rightarrow U : \mathbf{mac}(\langle U, op, \mathsf{Clk}\rangle, K_{AS}) & \text{if } \Gamma \vdash_U op \\
(2') & A \rightarrow U : \mathbf{mac}(\langle U, op, \mathsf{Clk}\rangle, \overline{K}_{AS}) & \text{otherwise} \\
(4) & S \rightarrow V : r & \text{if } \kappa = \mathbf{mac}(\langle _, op, \mathsf{Clk}\rangle, K_{AS}) \text{ and } \rho[\![op]\!] \Downarrow \rho'[\![r]\!] \\
(6) & A \rightarrow W : r & \text{if } \Gamma \vdash_W \theta \text{ and } \Xi[\![\theta]\!] \Downarrow \Xi'[\![r]\!]
\end{array}
$$

Likewise, we modify the following message in IS^d to obtain the protocol IS^{d+}.

$$
(\text{iv}) \quad D \rightarrow W : r \quad \text{if } \Gamma \vdash_W \theta \text{ and } \Xi[\![\theta]\!] \Downarrow \Xi'[\![r]\!]
$$

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 good practice. However, recall that MACs can be decoded to retrieve their messages. In particular, one can tell the time in NS^{d+} by decoding capabilities. Clearly we require that:

R7 If it is possible to tell the time in NS^{d+}, it must also be possible to do so in IS^{d+}.

So we must make it possible to tell the time in IS^{d+}. (The alternative is to make it impossible 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+}.

$$
\begin{array}{llll}
(\text{v}) & U \rightarrow D & : & () \\
(\text{vi}) & D \rightarrow U & : & \mathsf{Clk}
\end{array}
$$

The following result can then be proved. A version of this result already appears in [13], but the definition of safety there is rather ad hoc; in Section 5, we prove this result again, for a stronger definition of safety.

Theorem 3. *NS^{d+} is a safe implementation of IS^{d+}.*

Unfortunately, beyond this result, [13] does not consider security. In the rest of this section, we analyze the difficulties that arise for security, and present new results.

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. Informally, `stale` means "detect any use of an expired capability". Here κ is a capability for operation op.

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 [23]. However, under some reasonable assumptions on fairness and communication reliability, blocking in fact becomes observable. Then, the only way to deal with this issue is to allow a similar observation in the specification, say by letting an execution request block non-deterministically. 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 instance, 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 `chmod` θ; `exec` op; `success`; `exec` op'; `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.

Security in a special case One way of plugging this leak is to consider that the elapse of time is altogether unobservable. (This prospect is not as shocking as it sounds, since time here is simply the value of a privately maintained counter.) What assumptions do we require to make this possible?

Note that we expect that executing an operation has some observable effect. If initially a user does not have access to an operation, but that access can be dynamically granted, then the elapse of time *can* be detected by observing the effect of executing that operation. So we must assume that:

A2 Accesses cannot be dynamically granted.

On the other hand, we should allow accesses to be dynamically revoked, since otherwise the access policy becomes static, and we do not need to consider time at all. Now, if initially a user has access to an operation, but that access can be dynamically revoked, then it is possible to detect the elapse of time if the *failure* to execute that operation is observable. So we must further assume that:

A3 Any unsuccessful use of a capability blocks (without any observable effect).

To validate our assumptions, let us try to adapt the counterexample above with (A2) and (A3). Suppose that initially all users have access to op and op', and θ specifies that no user may access op. Consider the NS^{d+} pseudo-code below. Informally, `failure` means "detect unsuccessful use of a capability".

t6 $\texttt{acquire}\ \kappa'; \texttt{chmod}\ \theta; \texttt{acquire}\ \kappa; \texttt{use}\ \kappa; \texttt{failure}; \texttt{use}\ \kappa'; \texttt{success}$

t7 $\texttt{chmod}\ \theta; \texttt{acquire}\ \kappa; \texttt{use}\ \kappa; \texttt{failure}; \texttt{acquire}\ \kappa'; \texttt{use}\ \kappa'; \texttt{success}$

Both (t6) and (t7) map to the same IS^{d+} pseudo-code $\texttt{chmod}\ \theta; \texttt{exec}\ op; \texttt{failure}; \texttt{exec}\ op'; \texttt{success}$. Fortunately, now (t6) and (t7) cannot be distinguished, since the intermediate `failure` event cannot be observed if true. (In contrast, recall that the intermediate `success` event in (t4) and (t5) forces a distinction between them.) Indeed, 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 5):

Theorem 4. *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.

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 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 instance, even if the suspicious-looking pseudo-code $\texttt{chmod}\ \theta; \texttt{acquire}\ \kappa; \texttt{use}\ \kappa; \texttt{success}$ in (t4) and (t5) is replaced by an innocuous pair of inputs on a public channel c, the adversary can still run that code in parallel and serialize it by a pair of outputs on c (which serve as "begin/end" signals).

- Even if we restrict the code run by honest users, such that every use of a capability can be serialized immediately after its acquisition, the adversary can still force time to elapse *after* a capability is sent to the file system and *before* it is examined. Unless we have a way to constrain this elapse of time, we are in trouble.

To see how the adversary can break security by interacting with honest users, consider the NS^{d+} pseudo-code below. Here κ is a capability for operation op, and θ modifies access to op; further, $c()$ and $\overline{w}\langle\rangle$ denote input and output on public channels c and w.

t8 $\texttt{acquire}\ \kappa; \texttt{use}\ \kappa; c(); \texttt{chmod}\ \theta; c(); \texttt{success}; \overline{w}\langle\rangle$

t9 $c(); c(); \overline{w}\langle\rangle$

Although $\texttt{use}\ \kappa$ immediately follows $\texttt{acquire}\ \kappa$ in (t8), the delay between $\texttt{use}\ \kappa$ and $\texttt{success}$ can be detected by the adversary to force time to elapse between those events. 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'_0 and κ'_1 are capabilities for op'.

$$\overline{c}\langle\rangle; \texttt{acquire}\ \kappa'_0; \texttt{use}\ \kappa'_0; \texttt{failure}; \texttt{chmod}\ \theta'; \texttt{acquire}\ \kappa'_1; \texttt{use}\ \kappa'_1; \texttt{success}; \overline{c}\langle\rangle$$

This context forces time to elapse between a pair of outputs on c. The context can distinguish (t8) and (t9) by testing output on w: in (t8), κ does not authorize access to op, so $\texttt{success}$ must be false and there is no output on w; on the other hand, in (t9), 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 (t8) and Q_2 run (t9). Then the specification instances of Q_1 and Q_2 run $\texttt{exec}\ op; c(); \texttt{chmod}\theta; c(); \texttt{success}; \overline{w}\langle\rangle$ and $c(); c(); \overline{w}\langle\rangle$. These instances cannot be distinguished, since the execution of op can always be delayed until θ is executed, so that $\texttt{success}$ 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 processed for progress; but operation requests in IS^{d+} make no such commitment. In the end, the only way to deal with this issue 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 processed within that time bound.

This assumption leads to a natural refinement of the current specification. Usual (unrestricted) requests simply 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 5):

Theorem 5 (Main theorem). *NS^{d+} is a secure implementation of IS^{d+} assuming* (A5).

While this result is quite pleasant, we should take care to understand 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, (A5) is not an unreasonable assumption; in fact, it is very common to implement (A5) in a file system.

- 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. Note that in contrast, we do not require (A5) to reduce safety proofs.

Discussion We now revisit the principles developed in Sections 2 and 3, and discuss some alternatives.

First recall (R3), where we introduce fake capabilities to prevent premature leaks of information about the access policy Γ. It is reasonable to consider that we do not care about such leaks, and wish to keep the original message $(2')$ in NS^s. But then we must allow those leaks in the specification. For instance, 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, so long as administrative operations that affect their correctness are not executed before those intervals elapse. Let $\mathbf{delay}$ be a function over users U, operations op, and clock values Clk that produces time intervals. We may have that:

R5 Any capability for U and op that is produced at time Clk expires at time $\mathsf{Clk} + \mathbf{delay}(U, op, \mathsf{Clk})$.

R6 If an administrative operation affects the access decision for U and op and is requested in the interval $\mathsf{Clk}, \ldots, \mathsf{Clk} + \mathbf{delay}(U, op, \mathsf{Clk}) - 1$, it is executed at the clock tick to time $\mathsf{Clk} + \mathbf{delay}(U, op, \mathsf{Clk})$.

This scheme is sound, since a capability for U and op at Clk, that expires at $\mathsf{Clk} + \mathbf{delay}(U, op, \mathsf{Clk})$, certifies a correct access decision for U and op between $\mathsf{Clk}, \ldots, \mathsf{Clk} + \mathbf{delay}(U, op, \mathsf{Clk}) - 1$.

Finally, the implementation details in Sections 2 and 3 are far from unique. Guided by the same underlying principles, we can design capabilities in various other ways. For instance, 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. Although this design involves more cryptography than the one in NS^{d+}, it reflects better practice: the access decision for U and op under Γ is explicit in the content of any capability that certifies that decision. What does this design buy us? Consider applications where the access decision is not a boolean, but a label, a decision tree, or some arbitrary 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 Section 6.

4 Definitions and proof techniques

Let us now develop formal definitions and proof techniques for security and safety; these serve as background for Section 5, where we outline formal proofs for security and safety of NS^{d+} under IS^{d+}.

Let $\preceq$ be a precongruence on processes and $\simeq$ be the associated congruence. A process P under a context φ is written as $\varphi[P]$. Contexts act as tests for behaviors, and $P \preceq Q$ means that any test that is passed by P is passed by Q, that is, "P has no more behaviors than Q".

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 [5].

Definition 3 (Full abstraction and security (*cf.* Definition 2))**.** *An implementation $\mathcal{R}$ is fully abstract if it satisfies:*

$$\begin{array}{rl} (\textsc{Preservation}) & \forall (P,Q) \in \mathcal{R}.\ \forall (P',Q') \in \mathcal{R}.\ P \preceq P' \Rightarrow Q \preceq Q' \\ (\textsc{Reflection}) & \forall (P,Q) \in \mathcal{R}.\ \forall (P',Q') \in \mathcal{R}.\ Q \preceq Q' \Rightarrow P \preceq P' \end{array}$$

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 interesting behaviors—if (P, Q) and (P', Q') are in a secure $\mathcal{R}$ and P has no more behaviors than P', then Q has no more behaviors than Q'. A fully abstract implementation moreover does not *eliminate* any interesting behaviors.

Any subset of a secure implementation is secure. Security implies preservation of $\simeq$. Finally, testing itself is trivially secure since $\preceq$ is closed under any context.

Proposition 6. *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 interesting behaviors by acting as a test for those behaviors. A fully abstract context does not do so; it merely *translates* behaviors.

Definition 4 (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 account for any benign differences between the implementation and the specification. In particular, we define an implementation to be safe if it does not introduce any behaviors modulo such a wrapper.

Definition 5 (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 instance, suppose that P' is a pi calculus process [22] that does not contain public channels. Further, suppose that $\{P'\}$ is taken to be the set of specification instances (so that any output on a public channel is unsafe). Let c be a public channel; suppose that $P = \overline{c}\langle\rangle; P'$ and $\phi = \bullet \mid !\, \overline{c}\langle\rangle$. 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 (Proof of security)**.** *Let ϕ and ψ be contexts such that for all $(P, Q) \in \mathcal{R}$, $Q \preceq \phi[P]$, $P \preceq \psi[Q]$, 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 $Q \preceq \phi[P] \preceq \phi[P'] \preceq \phi[\psi[Q']] \preceq Q'$. □

Intuitively, $\mathcal{R}$ is secure if $\mathcal{R}$ and $\mathcal{R}^{-1}$ both satisfy (INCLUSION), and the witnessing contexts "cancel" each other. A simple technique for proving full abstraction for contexts follows as a corollary.

Corollary 8 (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 6. □

5 Models and proofs

We now outline models of NS^{d+} and IS^{d+}, formalized in the applied pi calculus [2], and present proofs of our results. Other versions of these protocols that appear in Sections 2 and 3 are treated similarly.

We fix an equational theory that defines natural numbers and tuples, and contains exactly one equation that involves the symbol $\mathbf{mac}$, which is

$$\mathbf{msg}(\mathbf{mac}(x, y)) = x$$

Natural numbers are required to model time. Further, we identify the set of users by the set of natural numbers ($\mathbb{N}$), and consider any user not identified in a fixed subset $\mathbb{H}$ of $\mathbb{N}$ to be dishonest.

Models We conveniently model the file systems under study by parameterized process definitions [22]. Specifically, both file systems are parameterized by an access policy Γ, a scratchpad Ξ, a time Clk, and a store ρ. For the networked file system, $\mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho}$, the interface includes channels α_U, β_U, and γ_U for every user U; the user U may send authorization requests on α_U, execution requests on β_U, and administration requests on γ_U. For the ideal file system, $\mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho}$, the interface includes channels α°_U, β°_U, and γ°_U for every user U; the user U may send time requests on α_U, operation requests on β°_U, and administration requests on γ°_U. The adversary is an arbitrary evaluation context in the language [2].

We encode the relations $\Gamma \vdash_U op$, $\rho[\![op]\!] \Downarrow \rho'[\![r]\!]$, $\Gamma \vdash_U \theta$, and $\Xi[\![\theta]\!] \Downarrow \Xi'[\![r]\!]$ in the equational theory. A networked storage system is described as

$$NS^{d+}_{\Gamma,\rho}(C) \triangleq (\nu_{U\in\mathbb{H}}\alpha_U\beta_U\gamma_U)(C \mid (\nu K_{AS}\overline{K}_{AS})\,\mathsf{Nfs}_{\Gamma,\Gamma,0,\rho})$$

Here C is code run by honest users, Γ is an access policy, and ρ is a store; initially the scratchpad is Γ and the time is 0. Similarly, an ideal storage system is described as

$$IS^{d+}_{\Gamma,\rho}(C) \triangleq (\nu_{U\in\mathbb{H}}\alpha^\circ_U\beta^\circ_U\gamma^\circ_U)(C \mid \mathsf{Ifs}_{\Gamma,\Gamma,0,\rho})$$

Channels associated with honest users are hidden from the adversary. The adversary itself is left implicit; in particular, channels associated with dishonest users are available to the adversary.

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 w, so does $\varphi[Q]$. Let Γ and ρ range over terms that do not contain any interface channels, keys, or other private names used by the file systems under study. Let C range over well-formed code for honest users in NS^{d+} (see below), and let $\lceil_\rceil$ abstract such C in IS^{d+}. We define the implementation relation

$$\mathbb{IMP} \triangleq \bigcup_{\Gamma,\rho,C} \{(IS^{d+}_{\Gamma,\rho}(\lceil C\rceil), NS^{d+}_{\Gamma,\rho}(C))\}$$

The function $\lceil_\rceil$ can be described as a compiler from processes to processes. For every $U \in \mathbb{H}$, the compiler translates requests on α_U, β_U, and γ_U to appropriate requests on α°_U, β°_U, and γ°_U. Further, the compiler guarantees strong secrecy of capabilities obtained by user $U \in \mathbb{H}$.

We then show evaluation contexts ϕ and ψ such that:

Lemma 9. *For any Γ, ρ, and C, $NS^{d+}_{\Gamma,\rho}(C) \preceq \phi[IS^{d+}_{\Gamma,\rho}(\lceil C\rceil)]$, $IS^{d+}_{\Gamma,\rho}(\lceil C\rceil) \preceq \psi[NS^{d+}_{\Gamma,\rho}(C)]$, and $\phi[\psi[NS^{d+}_{\Gamma,\rho}(C)]] \preceq NS^{d+}_{\Gamma,\rho}(C)$.*

Specifically, we define processes $\uparrow^{\mathrm{IS}}_{\mathrm{NS}}$ and $\uparrow^{\mathrm{NS}}_{\mathrm{IS}}$ which, for every $V \in \mathbb{N} \setminus \mathbb{H}$, receive requests on α_V, β_V, and γ_V and send the requests by appropriate translation on α°_V, β°_V, and γ°_V, and vice versa. Then:

$$\begin{aligned} \phi &\triangleq (\nu_{V\in\mathbb{N}\setminus\mathbb{H}}\alpha^\circ_V\beta^\circ_V\gamma^\circ_V)\,(\bullet \mid (\nu K)\uparrow^{\mathrm{IS}}_{\mathrm{NS}}) \\ \psi &\triangleq (\nu_{V\in\mathbb{N}\setminus\mathbb{H}}\alpha_V\beta_V\gamma_V)\,(\bullet \mid \uparrow^{\mathrm{NS}}_{\mathrm{IS}}) \end{aligned}$$

where K is a dummy key that is secret to ϕ. Roughly, Lemma 9 is proved as follows: a networked storage system is simulated by an ideal storage system by forwarding public requests directed at Nfs to a private Ifs interface (via ϕ). Capabilities are simulated by terms that encode the same messages, but are signed with K. Conversely, an ideal storage system is simulated by a networked storage system by forwarding public requests directed at Ifs to a private Nfs interface (via ψ). Finally, a networked storage system simulates another networked storage system where requests directed at Nfs are filtered through a private Ifs interface before forwarding them to a private Nfs interface (via $\phi[\psi]$). This detour essentially forces capabilities to be acquired immediately before their use. Now, by Lemma 9 and Proposition 7:

Theorem 10 (*cf.* Theorem 5)**.** *$\mathbb{IMP}$ is secure.*

Further, we show that:

Lemma 11. *For any Γ, ρ, and C, $\psi[\phi[IS^{d+}_{\Gamma,\rho}(\lceil C \rceil)]] \preceq IS^{d+}_{\Gamma,\rho}(\lceil C \rceil)$.*

So, by Lemmas 9 and 11 and Proposition 7, $\mathbb{IMP}$ is fully abstract. Finally:

Theorem 12 (*cf.* Theorem 3)**.** *$\mathbb{IMP}$ is safe.*

Proof. Fix any Γ, ρ, and C. By Lemma 9, $NS^{d+}_{\Gamma,\rho}(C) \preceq \phi[IS^{d+}_{\Gamma,\rho}(\lceil C \rceil)]$. Further, by Lemma 9, $IS^{d+}_{\Gamma,\rho}(\lceil C \rceil) \preceq \psi[NS^{d+}_{\Gamma,\rho}(C)] \preceq \psi[\phi[IS^{d+}_{\Gamma,\rho}(\lceil C \rceil)]]$. So, by Lemma 11 and Corollary 8, ϕ is fully abstract (taking $\phi^{-1} = \psi$). The result follows by our definition of safety. □

6 Designing correct distributed protocols

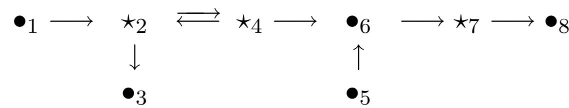
In the preceding sections, we present a thorough analysis of the problem of distributing access control. We now recycle that analysis on a more general problem.

Suppose that we are required to design a distributed protocol that correctly implements a given specification. For concreteness, we may assume that the specification is a state machine. We can solve this problem by partitioning the specification into smaller state machines, running those machines in parallel, and encoding the intermediate outputs of those machines so that they may be released and absorbed in any order. In particular, we can design NS^{d+} by partitioning IS^{d+} into access control and storage, running them in parallel, and encoding the intermediate outputs of access control as capabilities. The same principles should guide any such design. For instance, by (R3) and (R4), intermediate outputs should not leak information prematurely; by (R5) and (R6), such outputs must be timestamped and the states on which they depend must not change between clock ticks; and finally, by (A5), the specification must allow time-bounded requests.

We sketch a formalization of this idea below. (A detailed treatment is relegated to the appendix.) We describe a state machine as a directed graph G. In this graph, we refer to nodes of indegree 0 as *input nodes*, and nodes of outdegree 0 as *output nodes*; some of the other nodes are distinguished as *state nodes*. A *configuration* of G is a tuple (σ, τ), where σ assigns values to the nodes in G, and τ assigns times to the state nodes in G.

We label each node v in G by a function δ_v. The semantics of the state machine is then described as a binary relation over configurations of G. Intuitively, at every node v, the function δ_v produces a new value at v, consuming the values at its incoming nodes; further, if v is a state node, δ_v consumes the value at v, and increments the time at v.

For example, IS^{d+} may be described by the following graph:

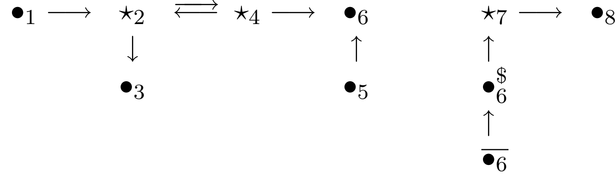


In this graph, $\bullet_1$ and $\bullet_5$ are input nodes, $\bullet_3$ and $\bullet_8$ are output nodes, and $\star_2$, $\star_4$, and $\star_7$ are state nodes. Intuitively, the values at $\star_2$, $\star_4$, and $\star_7$ are (respectively) scratchpads Ξ, access policies Γ, and stores ρ. The nodes $\bullet_1$ and $\bullet_3$ carry inputs and outputs for access modifications; the node $\bullet_6$ carries access decisions; and the nodes $\bullet_5$ and $\bullet_8$ carry inputs and outputs for store operations. We define the functions δ_v for each node v accordingly.

In the appendix, we describe an algorithm that can distribute a given state machine G along any cut of G, yielding a distributed state machine $G^{\$}$. Intuitively, this algorithm proceeds as follows. For every edge (v, u) in the cut, we replace (v, u) with the edges $(\overline{v}, v^{\$})$ and $(v^{\$}, u)$ in $G^{\$}$, where $\overline{v}$ and $v^{\$}$ are

fresh nodes, and v and $\overline{v}$ share the secret keys $K_{v,\overline{v}}$ and $E_{v,\overline{v}}$. Let r be any value at v that is produced in G by consuming some input values $\widetilde{i}$ and some state values at times $\widetilde{t}$. The encoded value at v that is produced in $G^{\$}$ is $\mathbf{mac}(\langle \widetilde{i}, \widetilde{t}, \{m, r\}_{E_{v,\overline{v}}}\rangle, K_{v,\overline{v}})$ for some fresh nonce m. As such, this value can be released at v and absorbed at $\overline{v}$, in parallel. The function $\delta_{v^{\$}}$ decodes this value back to r. In other words, the node $v^{\$}$ in $G^{\$}$ finally carries the same value r as the node v in G.

For instance, IS^{d+} can be distributed along the cut $\{(\bullet_6, \star_7)\}$ to obtain NS^{d+}, as follows:



In particular, $\delta_{\bullet_6}$ now produces capabilities of the form $\mathbf{mac}(\langle\langle U, op, \mathsf{Clk}\rangle, \{m, L\}_{E_{\bullet_6,\overline{\bullet_6}}}\rangle, K_{\bullet_6,\overline{\bullet_6}})$, where L is the access decision for U and op at Clk. These capabilities are decoded back by $\delta_{\bullet_6^{\$}}$.

We prove that any distributed state machine $G^{\$}$ derived by our algorithm is a safe and secure implementation of the state machine G. The proofs are similar to the ones presented in Section 5, and rely on a definition of $\preceq$ which is roughly as follows. A context is any graph that can produce values at the input nodes, consume values at the output nodes, and consume times at the state nodes of a graph. Two graphs are related by $\preceq$ if under all contexts, whenever an output is produced at the distinguished node w with the former graph, so is it with the latter graph.

7 Conclusion

In this paper, we present a comprehensive analysis of the problem of implementing distributed access control with capabilities. While in previous work, we already show how to implement static access policies securely [12] and dynamic access policies safely [13], here, 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 present new insights on the difficulty of implementing dynamic access policies securely (a problem that has hitherto remained unsolved). We show that such an implementation is in fact possible if the specification is slightly generalized. Finally, our analysis turns out to have other applications. Guided by the same basic principles, we show how to automatically derive secure distributed implementations of other state machines. This approach is reminiscent of secure program partitioning [24], and investigating its scope should be interesting future work.

Proof details for all the results in this paper appear in an extended technical report available online at `http://www.soe.ucsc.edu/~avik/papers/sdidac.pdf`.

Acknowledgments This work owes much to Martín Abadi, who formulated the original problem and co-authored our previous work in this area. Many thanks to him and Sergio Maffeis for helpful discussions on this work, and detailed comments on an earlier draft of this paper. Thanks also to him and Cédric Fournet for clarifying an issue about the applied pi calculus, which led to simpler proofs.

References

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

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

[3] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pages 105–116, 1998.
[4] M. Abadi, C. Fournet, and G. Gonthier. Authentication primitives and their compilation. In *POPL'00: Principles of Programming Languages*, pages 302–315. ACM, 2000.
[5] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
[6] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, Jan. 1996.
[7] 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.
[8] M. Backes and A. Oprea. Lazy revocation in cryptographic file systems. In *SISW '05: Security in Storage Workshop*, pages 1–11. IEEE, 2005.
[9] B. Blanchet and A. Chaudhuri. Automated formal analysis of a protocol for secure file sharing on untrusted storage. In *S&P'08: Proceedings of the 29th IEEE symposium on Security and Privacy*. To appear, 2008.
[10] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.
[11] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *FOCS'01: Foundations of Computer Science*, pages 136–145, 2001.
[12] 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.
[13] 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.
[14] 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.
[15] H. Gobioff, G. Gibson, and J. Tygar. Security for network attached storage devices. Technical Report CMU-CS-97-185, Carnegie Mellon University, 1997.
[16] S. Goldwasser and M. Bellare. Lecture notes in cryptography, 2001. See `http://www.cs.ucsd.edu/users/mihir/papers/gb.html`.
[17] S. Halevi, P. A. Karger, and D. Naor. Enforcing confinement in distributed storage and a cryptographic mo del for access control. Cryptology ePrint Archive, Report 2005/169, 2005. See `http://eprint.iacr.org/2005/169`.
[18] 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.
[19] S. Maffeis. *Dynamic Web Data: A Process Algebraic Approach*. PhD thesis, Imperial College London, August 2006.
[20] 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.
[21] R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4(1):1–22, 1977.
[22] R. Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.
[23] R. D. Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1–2):83–133, 1984.
[24] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3):283–328, 2002.

Appendix

In this appendix, we provide some additional details on the material of Sections 5 and 6, that may help the referees. Specifically, in Appendix A, we present the semantics of the applied pi calculus processes that appear in the models and proofs in Section 5. In Appendix B, we describe the algorithm for deriving secure distributed state machines, that is outlined in Section 6.

A Details on models and proofs

In Section 5, we consider the file systems $\mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho}$ and $\mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho}$ to be modeled by parameterized process definitions [22]. For convenience, we express these definitions as extensions of the usual semantic relations of structural equivalence $\equiv$ and reduction $\rightarrow$ in the applied pi calculus [2]. Processes that satisfy these extended relations modulo $\rightarrow^\star$ are guaranteed to exist. Further, some of the processes that we require are actually functions [22]; for convenience, we define their semantics directly as functions.

Models Figure 1 shows applied pi calculus models for the file systems under study. We implicitly assume that $U \in \mathbb{N}$. Recall that the file systems are parameterized by an access policy Γ, a scratchpad Ξ, a time Clk, and a store ρ. For the networked file system $\mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho}$, the interface includes channels α_U, β_U, and γ_U for every user U; the user U may send authorization requests on α_U, execution requests on β_U, and administration requests on γ_U. For the ideal file system $\mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho}$, the interface includes channels α°_U, β°_U, and γ°_U for every user U; the user U may send time requests on α_U, operation requests on β°_U, and administration requests on γ°_U. Other parameterized processes such as $\mathsf{CReq}_{U,op,M}$, $\mathsf{TReq}_{U,M}$, $\mathsf{EReq}_{\kappa,M}$, $\mathsf{OReq}_{U,op,T,M}$, $\mathsf{EOk}_{L,op,M}$, and $\mathsf{AReq}_{U,\theta,M}$ denote various internal states.

In the equational theory $\mathbf{auth}(\Gamma, U, op) = \mathbf{ok}$ means that U may access op under Γ; $\mathbf{auth}(\Gamma, U, \theta) = \mathbf{ok}$ means that U controls θ under Γ; $\mathbf{exec}(L, op, \rho) = \langle N, \rho' \rangle$ 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 scratchpad Ξ under decision L returns N and scratchpad Ξ'. We define the following functions:

$$\mathtt{perm}_{\Gamma,U,op} = \begin{cases} \mathbf{true} & \text{if } \mathbf{auth}(\Gamma, U, op) = \mathbf{ok} \\ \mathbf{false} & \text{otherwise} \end{cases} \qquad \mathtt{perm}_{\Gamma,U,\theta} = \begin{cases} \mathbf{true} & \text{if } \mathbf{auth}(\Gamma, U, \theta) = \mathbf{ok} \\ \mathbf{false} & \text{otherwise} \end{cases}$$

$$\mathtt{cert}_{\Gamma,U,op,\mathsf{Clk}} = \begin{cases} \mathbf{mac}(\langle U, op, \mathsf{Clk}\rangle, K_{AS}) & \text{if } \mathbf{auth}(\Gamma, U, op) = \mathbf{ok} \\ \mathbf{mac}(\langle U, op, \mathsf{Clk}\rangle, \overline{K}_{AS}) & \text{otherwise} \end{cases}$$

$$\mathtt{verif}_{\kappa} = \begin{cases} \mathbf{true} & \text{if } \kappa = \mathbf{mac}(\mathbf{msg}(\kappa), K_{AS}) \\ \mathbf{false} & \text{if } \kappa = \mathbf{mac}(\mathbf{msg}(\kappa), \overline{K}_{AS}) \end{cases}$$

A networked storage system may be described as

$$NS^{d+}_{\Gamma,\rho}(C) \triangleq (\nu_{U\in\mathbb{H}}\alpha_U\beta_U\gamma_U)(C \mid (\nu K_{AS}\overline{K}_{AS})\,\mathsf{Nfs}_{\Gamma,\Gamma,0,\rho})$$

Here C is code run by honest users, Γ is an access policy, and ρ is a store; initially the scratchpad is Γ and the time is 0. Similarly, an ideal storage system may be described as

$$IS^{d+}_{\Gamma,\rho}(C) \triangleq (\nu_{U\in\mathbb{H}}\alpha^\circ_U\beta^\circ_U\gamma^\circ_U)(C \mid \mathsf{Ifs}_{\Gamma,\Gamma,0,\rho})$$

Channels associated with honest users are hidden from the context. The context itself may be arbitrary and is left implicit; in particular, channels associated with dishonest users are available to the context.

(TIME REQUEST)

$$\mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \equiv \alpha^{\circ}_{U}(x); \mathsf{TReq}_{x} \mid \mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho}$$

(TIME)

$$\mathsf{TReq}_{M} \mid \mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \rightarrow \overline{M}\langle \mathsf{Clk} \rangle \mid \mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho}$$

(OPERATION REQUEST)

$$\mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \equiv \beta^{\circ}_{U}(op, T, x); \mathsf{OReq}_{U,op,T,x} \mid \mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho}$$

(EXECUTION OK)

$$\frac{\mathtt{perm}_{\Gamma,U,op} = L \qquad \mathsf{Clk} \leq T}{\mathsf{OReq}_{U,op,T,M} \mid \mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \rightarrow \mathsf{EOk}_{L,op,M} \mid \mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho}}$$

(EXECUTION)

$$\frac{\mathbf{exec}(L, op, \rho) = \langle N, \rho' \rangle}{\mathsf{EOk}_{L,op,M} \mid \mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \rightarrow \overline{M}\langle N \rangle \mid \mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho'}}$$

(ADMINISTRATION REQUEST)

$$\mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \equiv \gamma^{\circ}_{U}(\theta, x); \mathsf{AReq}_{U,\theta,x} \mid \mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho}$$

(ADMINISTRATION)

$$\frac{\mathtt{perm}_{\Gamma,U,\theta} = L \qquad \mathbf{exec}(L, \theta, \Xi) = \langle N, \Xi' \rangle}{\mathsf{AReq}_{U,\theta,M} \mid \mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \rightarrow \overline{M}\langle N \rangle \mid \mathsf{Ifs}_{\Gamma,\Xi',\mathsf{Clk},\rho}}$$

(TICK)

$$\mathsf{Ifs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \rightarrow \mathsf{Ifs}_{\Xi,\Xi,\mathsf{Clk}+1,\rho}$$

(AUTHORIZATION REQUEST)

$$\mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \equiv \alpha_{U}(op, x); \mathsf{CReq}_{U,op,x} \mid \mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho}$$

(AUTHORIZATION)

$$\frac{\mathtt{cert}(\Gamma, U, op, \mathsf{Clk}) = \kappa}{\mathsf{CReq}_{U,op,M} \mid \mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \rightarrow \overline{M}\langle \kappa \rangle \mid \mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho}}$$

(EXECUTION REQUEST)

$$\mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \equiv \beta_{U}(\kappa, x); \mathsf{EReq}_{\kappa,x} \mid \mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho}$$

(EXECUTION OK)

$$\frac{\mathtt{verif}_{\kappa} = L \in \{\mathbf{true}, \mathbf{false}\} \qquad \mathbf{msg}(\kappa) = \langle _, op, \mathsf{Clk} \rangle}{\mathsf{EReq}_{\kappa,M} \mid \mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \rightarrow \mathsf{EOk}_{L,op,M} \mid \mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho}}$$

(EXECUTION)

$$\frac{\mathbf{exec}(L, op, \rho) = \langle N, \rho' \rangle}{\mathsf{EOk}_{L,op,M} \mid \mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \rightarrow \overline{M}\langle N \rangle \mid \mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho'}}$$

(ADMINISTRATION REQUEST)

$$\mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \equiv \gamma_{U}(\theta, x); \mathsf{AReq}_{U,\theta,x} \mid \mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho}$$

(ADMINISTRATION)

$$\frac{\mathtt{perm}_{\Gamma,U,\theta} = L \qquad \mathbf{exec}(L, \theta, \Xi) = \langle N, \Xi' \rangle}{\mathsf{AReq}_{U,\theta,M} \mid \mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \rightarrow \overline{M}\langle N \rangle \mid \mathsf{Nfs}_{\Gamma,\Xi',\mathsf{Clk},\rho}}$$

(TICK)

$$\mathsf{Nfs}_{\Gamma,\Xi,\mathsf{Clk},\rho} \rightarrow \mathsf{Nfs}_{\Xi,\Xi,\mathsf{Clk}+1,\rho}$$

Figure 1: An ideal file system with local access control (above) and a networked file system with distributed access control (below)

Proofs Let Γ and ρ range over terms that do not contain any interface channels, keys, or other private names used by the file systems under study. Let C range over well-formed code for honest users in NS^{d+}, and let $\lceil _ \rceil$ abstract such C in IS^{d+} (see below). We define

$$\mathbb{IMP} = \bigcup_{\Gamma,\rho,C} \{(IS^{d+}_{\Gamma,\rho}(\lceil C \rceil), NS^{d+}_{\Gamma,\rho}(C))\}$$

We describe $\lceil _ \rceil$ as a typed compilation $\lceil _ \rceil_{\Gamma}$ under an appropriate type environment Γ. Let $\mathtt{Cert}(U, op)$ be the type of any capability obtained by user U for operation op. We show a fragment of the compiler below.

(AUTHORIZATION REQUEST TO TIME REQUEST)

$$\frac{c \notin \mathtt{fn}(Q) \qquad \lceil Q \rceil_{\Gamma, x:\mathtt{Cert}(U,op)} = P \qquad \ldots}{\lceil (\nu c)\, \overline{\alpha_U}\langle op, c \rangle; c(x); Q \rceil_{\Gamma} = (\nu c)\, \overline{\alpha^{\circ}_U}\langle c \rangle; c(x); P}$$

(EXECUTION REQUEST TO OPERATION REQUEST)

$$\frac{\Gamma(x) = \mathtt{Cert}(V, op) \qquad \lceil Q \rceil_{\Gamma} = P \qquad \ldots}{\lceil \overline{\beta_U}\langle x, M \rangle; Q \rceil_{\Gamma} = \overline{\beta^{\circ}_V}\langle op, x, M \rangle; P}$$

(DUMMY AUTHORIZATION REQUEST)
$\uparrow^{\text{IS}}_{\text{NS}} \equiv \alpha_V(op, x); (\nu m)\, \overline{\alpha^\circ_V}\langle m\rangle; m(\mathsf{Clk}); \overline{x}\langle \mathbf{mac}(\langle V, op, \mathsf{Clk}\rangle, K\rangle \mid \uparrow^{\text{IS}}_{\text{NS}}$

(DUMMY EXECUTION REQUEST)
$\uparrow^{\text{IS}}_{\text{NS}} \equiv \beta_V(\kappa, x); \mathsf{DReq}_{\kappa,x} \mid \uparrow^{\text{IS}}_{\text{NS}}$

(DUMMY OPERATION REQUEST)
$$\frac{\kappa = \mathbf{mac}(\mathbf{msg}(\kappa), K) \qquad \mathbf{msg}(\kappa) = \langle V, op, \mathsf{Clk}\rangle}{\mathsf{DReq}_{\kappa,M} \to \overline{\beta^\circ_V}\langle op, \mathsf{Clk}, M\rangle}$$

(DUMMY ADMINISTRATION REQUEST)
$\uparrow^{\text{IS}}_{\text{NS}} \equiv \gamma_V(op, x); \overline{\gamma^\circ_V}\langle op, x\rangle \mid \uparrow^{\text{IS}}_{\text{NS}}$

(DUMMY TIME REQ)
$$\frac{\mathbf{msg}(y) = \langle _, _, \mathsf{Clk}\rangle}{\uparrow^{\text{NS}}_{\text{IS}} \equiv \alpha^\circ_V(x); (\nu c)\, \overline{\alpha_V}\langle x, c\rangle; c(y); \overline{x}\langle \mathsf{Clk}\rangle \mid \uparrow^{\text{NS}}_{\text{IS}}}$$

(DUMMY OPERATION REQUEST)
$\uparrow^{\text{NS}}_{\text{IS}} \equiv \beta^\circ_V(op, T, x); (\nu c)\, \overline{\alpha_V}\langle op, c\rangle; c(\kappa); \mathsf{DReq}_{\kappa,T,x} \mid \uparrow^{\text{NS}}_{\text{IS}}$

(DUMMY EXECUTION REQUEST)
$$\frac{\mathbf{msg}(\kappa) = \langle _, _, \mathsf{Clk}\rangle \qquad \mathsf{Clk} \le T}{\mathsf{DReq}_{\kappa,T,M} \to \overline{\beta_V}\langle \kappa, M\rangle}$$

(DUMMY ADMINISTRATION REQUEST)
$\uparrow^{\text{NS}}_{\text{IS}} \equiv \gamma^\circ_V(op, x); \overline{\gamma_V}\langle op, x\rangle \mid \uparrow^{\text{NS}}_{\text{IS}}$

Figure 2: Wrappers

(ADMINISTRATION REQUEST TO ITSELF)
$$\frac{\lceil P\rceil_\Gamma = Q \qquad \ldots}{\lceil \overline{\gamma_U}\langle adm, M\rangle; Q\rceil_\Gamma = \overline{\gamma^\circ_U}\langle adm, M\rangle; Q}$$

The omitted fragment may be built from any type system that guarantees strong secrecy of terms of type $\texttt{Cert}(U, op)$ for any $U \in \mathbb{H}$ and op.

Next, we show contexts ϕ and ψ such that:

Restatement: Lemma 9. For any Γ, ρ, and C, $NS^{d+}_{\Gamma,\rho}(C) \preceq \phi[IS^{d+}_{\Gamma,\rho}(\lceil C\rceil)]$, $IS^{d+}_{\Gamma,\rho}(\lceil C\rceil) \preceq \psi[NS^{d+}_{\Gamma,\rho}(C)]$, *and* $\phi[\psi[NS^{d+}_{\Gamma,\rho}(C)]] \preceq NS^{d+}_{\Gamma,\rho}(C)$.

Figure 2 defines processes $\uparrow^{\text{IS}}_{\text{NS}}$ and $\uparrow^{\text{NS}}_{\text{IS}}$, which translate public requests from NS^{d+} to IS^{d+} and from IS^{d+} to NS^{d+}. There, we implicitly assume that $V \in \mathbb{N} \setminus \mathbb{H}$. We define

$$\begin{aligned}\phi &= (\nu_{V\in\mathbb{N}\setminus\mathbb{H}}\alpha^\circ_V\beta^\circ_V\gamma^\circ_V)\,(\bullet \mid (\nu K)\uparrow^{\text{IS}}_{\text{NS}})\\ \psi &= (\nu_{V\in\mathbb{N}\setminus\mathbb{H}}\alpha_V\beta_V\gamma_V)\,(\bullet \mid \ \uparrow^{\text{NS}}_{\text{IS}})\end{aligned}$$

B Details on designing correct distributed protocols

In Section 6, we briefly outline a method for distributing state machines. We provide a detailed treatment of this method below.

State machine as a graph We describe a state machine as a directed graph $G(\mathcal{V}, \mathcal{E})$. The *input nodes*, collected by $\mathcal{V}_i \subseteq \mathcal{V}$, are the nodes of indegree 0. The *output nodes*, collected by $\mathcal{V}_o \subseteq \mathcal{V}$, are the nodes of outdegree 0. Further, we consider a set of *state nodes* $\mathcal{V}_s \subseteq \mathcal{V}$ such that $\mathcal{V}_i \cap \mathcal{V}_s = \varnothing$. As a technicality, any node that is in a cycle or has outdegree > 1 must be in $\mathcal{V}_s$.

Nodes other than the input nodes run some code. Let $\mathcal{M}$ contain all terms and $\sqsubset$ be a strict total order on $\mathcal{V}$. We label each $v \in \mathcal{V} \setminus (\mathcal{V}_i \cup \mathcal{V}_s)$ with a function $\delta_v : \mathcal{M}^{\texttt{in}(v)} \to \mathcal{M}$, and each $v \in \mathcal{V}_s$ with a function $\delta_v : \mathcal{M}^{\texttt{in}(v)} \times \mathcal{M} \to \mathcal{M}$. Further, each state node carries a shared clock, following the midnight-shift scheme.

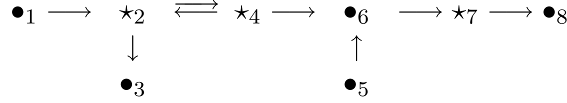
A *configuration* (σ, τ) consists of a partial function $\sigma : \mathcal{V} \to \mathcal{M}$ such that $\mathtt{dom}(\sigma) \supseteq \mathcal{V}_s$, and a total function $\tau : \mathcal{V}_s \to \mathbb{N}$. Intuitively, σ assigns values at the state nodes and some other nodes, and τ assigns times at the state nodes. For any $v \in \mathcal{V} \setminus \mathcal{V}_i$, the function δ_v outputs the value at v, taking as inputs the values at each incoming u, and the value at v if v is a state node; further, if such $u \notin \mathcal{V}_s$, the value at u is "consumed" on input. Formally, the operational semantics is given by a binary relation $\rightsquigarrow$ over configurations.

$$\frac{\begin{array}{c} v \in \mathcal{V} \setminus (\mathcal{V}_i \cup \mathcal{V}_s) \\ \forall k \in 1..\mathtt{in}(v).\ (u_U, v) \in \mathcal{E} \ \wedge\ \sigma(u_U) = t_U \qquad u_1 \sqsubset \ldots \sqsubset u_{\mathtt{in}(v)} \qquad \sigma^- = \sigma|_{\mathcal{V}_s \cup (\mathcal{V} \setminus \{u_1, \ldots, u_{\mathtt{in}(v)}\}} \end{array}}{(\sigma, \tau) \rightsquigarrow (\sigma^-[v \mapsto \delta_v(t_1, \ldots, t_{\mathtt{in}(v)})], \tau)}$$

$$\frac{\begin{array}{c} v \in \mathcal{V}_s \qquad \tau(v) = \mathsf{Clk} \qquad \sigma(v) = t \\ \forall k \in 1..\mathtt{in}(v).\ (u_U, v) \in \mathcal{E} \ \wedge\ \sigma(u_U) = t_U \qquad u_1 \sqsubset \ldots \sqsubset u_{\mathtt{in}(v)} \qquad \sigma^- = \sigma|_{\mathcal{V}_s \cup (\mathcal{V} \setminus \{u_1, \ldots, u_{\mathtt{in}(v)}\}} \end{array}}{(\sigma, \tau) \rightsquigarrow (\sigma^-[v \mapsto \delta_v(t_1, \ldots, t_{\mathtt{in}(v)}, t)], \tau[v \mapsto \mathsf{Clk} + 1])}$$

As usual, we leave the context implicit; the adversary is an arbitrary context that can write values at $\mathcal{V}_i$, read values at $\mathcal{V}_o$, and read times at $\mathcal{V}_s$.

For example, a graph that describes IS^{d+} is:



Here $\mathcal{V}_i = \{\bullet_1, \bullet_5\}$, $\mathcal{V}_o = \{\bullet_3, \bullet_8\}$, $\mathcal{V}_s = \{\star_2, \star_4, \star_7\}$, and $\mathcal{V} = \mathcal{V}_i \cup \mathcal{V}_o \cup \mathcal{V}_s \cup \{\bullet_6\}$. Intuitively, $\star_2$ carries scratchpads, and $\bullet_1$ and $\bullet_3$ carry inputs and outputs for access modifications; $\star_4$ carries access policies, and $\bullet_6$ carries access decisions; $\star_7$ carries stores, and $\bullet_5$ and $\bullet_8$ carry inputs and outputs for store operations. We define:

$$\begin{array}{rl} \delta_{\star_2}(\langle k, \theta \rangle, \Gamma, \langle _, \Xi \rangle) = \mathbf{exec}(\mathtt{perm}_{\Gamma,U,\theta}, \theta, \Xi) & \delta_{\bullet_3}(\langle N, \Xi \rangle) = N \\ \delta_{\star_4}(\langle _, \Xi \rangle, _) = \Xi & \delta_{\bullet_6}(\Gamma, \langle k, op \rangle) = \langle op, \mathtt{perm}_{\Gamma,U,op} \rangle \\ \delta_{\star_7}(\langle op, L \rangle, \langle _, \rho \rangle) = \mathbf{exec}(L, op, \rho) & \delta_{\bullet_8}(\langle N, \rho \rangle) = N \end{array}$$

Distribution as a graph cut Once described as a graph, a state machine can be distributed along any cut of that graph. For instance, IS^{d+} can be distributed along the cut $\{(\bullet_6, \star_7)\}$ to obtain NS^{d+}. We present this derivation formally in several steps.

Step 1 For each $v \in \mathcal{V}$, let $S(v) \subseteq \mathcal{V}_s$ be the set of state nodes that have paths to v, and $I(v) \subseteq \mathcal{V}_i$ be the set of input nodes that have paths to v without passing through nodes in $\mathcal{V}_s$. Then $G(\mathcal{V}, \mathcal{E})$ can be written in a form where, loosely, the values at $I(v)$ and the times at $S(v)$ are explicit in $\sigma(v)$ for each node v. Formally, the *explication* of G is the graph $\hat{G}(\hat{\mathcal{V}}, \hat{\mathcal{E}})$ where $\hat{\mathcal{V}} = \mathcal{V} \cup \{\hat{v} \mid v \in \mathcal{V}_i\} \cup \{\hat{u} \mid u \in \mathcal{V}_o\}$ and $\hat{\mathcal{E}} = \mathcal{E} \cup \{(\hat{v}, v) \mid v \in \mathcal{V}_i\} \cup \{(u, \hat{u}) \mid u \in \mathcal{V}_o\}$. We define:

$$\frac{v \in \mathcal{V}_i}{\hat{\delta}_v(t) = \langle t, t \rangle} \qquad \frac{v \in \mathcal{V}_o}{\hat{\delta}_{\hat{v}}(_, t) = t}$$

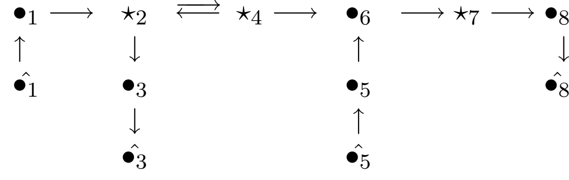
$$\frac{v \in \mathcal{V} \setminus (\mathcal{V}_i \cup \mathcal{V}_s) \qquad \delta_v(t_1, \ldots, t_{\mathtt{in}(v)}) = t}{\hat{\delta}_v(\langle I_1, t_1 \rangle, \ldots, \langle I_{\mathtt{in}(v)}, t_{\mathtt{in}(v)} \rangle) = \langle \langle I_1 \ldots I_{\mathtt{in}(v)} \rangle, t \rangle}$$

$$\frac{v \in \mathcal{V}_s \qquad \sigma(v) = \langle \mathsf{Clk}, t \rangle \qquad \delta_v(t_1, \ldots, t_{\mathtt{in}(v)}, t) = t'}{\hat{\delta}_v(\langle _, t_1 \rangle, \ldots, \langle _, t_{\mathtt{in}(v)} \rangle, \langle \mathsf{Clk}, t \rangle) = \langle \mathsf{Clk} + 1, t' \rangle}$$

This translation is sound and complete.

Theorem 13. *$\hat{G}$ is fully abstract with respect to G.*

For example, the explication of the graph for IS^{d+} is:



Here $\sigma(\bullet_6)$ is of the form $\langle\langle k, op, \mathsf{Clk}\rangle, \langle op, \mathtt{perm}_{\Gamma,U,op}\rangle\rangle$ rather than $\langle op, \mathtt{perm}_{\Gamma,U,op}\rangle$; the "input" $\sigma(\hat{\bullet}_5) = \langle k, op\rangle$, the "time" $\tau(\star_4) = \mathsf{Clk}$, and the "output" $\langle op, \mathtt{perm}_{\Gamma,U,op}\rangle$ of an access check are all explicit in $\sigma(\bullet_6)$. A capability can be conveniently constructed from this form (see below).

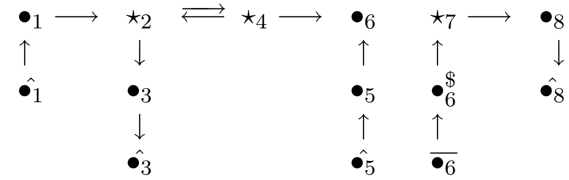
Step 2 Next, let $\mathcal{E}_0$ be any cut. As a technicality, we assume that $\mathcal{E}_0 \cap ((\mathcal{V}_i \cup \mathcal{V}_s) \times \mathcal{V}) = \varnothing$. The *distribution* of G along $\mathcal{E}_0$ is the graph $G^\$(\mathcal{V}^\$, \mathcal{E}^\$)$, where $\mathcal{V}^\$ = \hat{\mathcal{V}} \cup \{\overline{v} \mid (v, _) \in \mathcal{E}_0\} \cup \{v^\$ \mid (v, _) \in \mathcal{E}_0\}$ and $\mathcal{E}^\$ = (\hat{\mathcal{E}} \setminus \mathcal{E}_0) \cup \{(\overline{v}, v^\$) \mid (v, _) \in \mathcal{E}_0\} \cup \{(v^\$, u) \mid (v, u) \in \mathcal{E}_0\}$. Let K_v and E_v be secret keys shared by v and $v^\$$ for every $(v, _) \in \mathcal{E}_0$. We define:

$$\frac{(v, _) \in \mathcal{E}_0 \qquad \hat{\delta}_v(t_1, \ldots, t_{\mathtt{in}(v)}) = \langle t, t'\rangle \qquad m \text{ is fresh}}{\lambda^\$_v(t_1, \ldots, t_{\mathtt{in}(v)}) = \mathbf{mac}(\langle t, \{m, t'\}_{E_v}\rangle, K_v)}$$

$$\frac{(v, _) \in \mathcal{E}_0 \qquad \tau(S(v)) \text{ is included in } t}{\lambda^\$_{v^\$}(\langle t, \mathbf{mac}(\langle t, \{_, t'\}_{E_v}\rangle, K_v)\rangle) = \langle t, t'\rangle}$$

$$\frac{v \in \mathcal{V} \setminus \mathcal{V}_i \qquad (v, _) \notin \mathcal{E}_0}{\lambda^\$_v = \hat{\delta}_v}$$

Intuitively, for every $(v, _) \in \mathcal{E}_0$, $v^\$$ carries the same values in $G^\$$ as v does in G; those values are encoded and released at v, absorbed at $\overline{v}$, and decoded back at $v^\$$. For example, the distribution of the graph for IS^{d+} along the cut $\{(\bullet_6, \star_7)\}$ is:



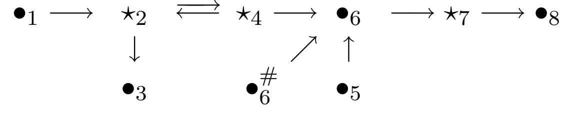
This graph describes a variant of NS^{d+}. In particular, the node $\bullet_6$ now carries a capability of the form $\mathbf{mac}(\langle\langle k, op, \mathsf{Clk}\rangle, \{m, \langle op, \mathtt{perm}_{\Gamma,U,op}\rangle\}_{E_{\bullet_6}}\rangle, K_{\bullet_6})$, that secures the input, time, and output of an access check.

Step 3 Finally, G is revised following (A5). The *revision* of G along $\mathcal{E}_0$ is the graph $G^\#(\mathcal{V}^\#, \mathcal{E}^\#)$, where $\mathcal{V}^\# = \mathcal{V} \cup \{v^\# \mid (v, _) \in \mathcal{E}_0\}$ and $\mathcal{E}^\# = \mathcal{E} \cup \{(v^\#, v) \mid (v, _) \in \mathcal{E}_0\}$. We define:

$$\frac{(v, u) \in \mathcal{E}_0 \qquad \tau(S(v)) \leq T}{\lambda^\#_v(t_1, \ldots, t_{\mathtt{in}(v)}, T) = \delta_v(t_1, \ldots, t_{\mathtt{in}(v)})}$$

$$\frac{v \in \mathcal{V} \setminus \mathcal{V}_i \qquad (v, _) \notin \mathcal{E}_0}{\lambda^\#_v = \delta_v}$$

Intuitively, for every $(v, _) \in \mathcal{E}_0$, progress at v requires that the times at $S(v)$ do not exceed the time bounds at $v^{\#}$. For example, the revised form of the graph for IS^{d+} is:



Here $\bullet_6^{\#}$ carries a time bound T, and $\lambda^{\#}_{\bullet_6}(\Gamma, \langle k, op\rangle, T) = \delta_{\bullet_6}(\Gamma, \langle k, op\rangle)$ if $\tau(\star_4) \leq T$.

We prove the following correctness result.

Theorem 14. *$G^{\$}$ is fully abstract with respect to $G^{\#}$.*

By Theorem 14, the graph for NS^{d+} is fully abstract with respect to the revised graph for IS^{d+}.

Similarly, we can design NS^s from IS^s. The induced subgraph of IS^{d+} without $\{\bullet_1, \star_2, \bullet_3, \star_4\}$ describes IS^s. We define $\delta_{\bullet_6}(\langle k, op\rangle) = \langle op, \mathtt{perm}_{\Gamma,U,op}\rangle$ for some static Γ. Distributing along the cut $\{(\bullet_6, \star_7)\}$, we obtain the induced subgraph of NS^{d+} without $\{\hat{\bullet}_1, \bullet_1, \star_2, \bullet_3, \hat{\bullet}_3, \star_4\}$. This graph describes a variant of NS^s, with $\sigma(\bullet_6)$ of the form $\mathbf{mac}(\langle\langle k, op\rangle, \{m, \langle op, \mathtt{perm}_{\Gamma,U,op}\rangle\}_{E_{\bullet_6}}\rangle, K_{\bullet_6})$. (Here capabilities do not carry timestamps.) By Theorem 14, the graph for NS^s is fully abstract with respect to a trivially revised graph for IS^s, where $\lambda^{\#}_{\bullet_6}(\langle k, op\rangle, \langle\rangle) = \delta_{\bullet_6}(\langle k, op\rangle)$.