2008 IEEE Symposium on Security and Privacy

Automated Formal Analysis of a Protocol for Secure File Sharing on Untrusted Storage

Bruno Blanchet
CNRS, École Normale Supérieure, INRIA*
blanchet@di.ens.fr

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

Abstract

We study formal security properties of a state-of-the-art protocol for secure file sharing on untrusted storage, in the automatic protocol verifier ProVerif. As far as we know, this is the first automated formal analysis of a secure storage protocol. The protocol, designed as the basis for the file system Plutus, features a number of interesting schemes like lazy revocation and key rotation. These schemes improve the protocol's performance, but complicate its security properties. Our analysis clarifies several ambiguities in the design and reveals some unknown attacks on the protocol. We propose corrections, and prove precise security guarantees for the corrected protocol.

1. Introduction

Much research in recent years has focused on the security analysis of communication protocols. In some cases, attacks have been found on old, seemingly robust protocols, and these protocols have been corrected [23, 33, 40]; in other cases, the security guarantees of those protocols have been found to be misunderstood, and they have been clarified and sometimes even formalized and proved [4, 33, 37]. More generally, this line of work has underlined the difficulty of designing secure communication protocols, and the importance of verifying their precise security properties.

While protocols for secure communication have been studied in depth, protocols for secure storage have received far less attention. Some of these protocols rely on secure communication, and we expect the usual techniques for secure communication to apply to such protocols as well. But some distinctive features of storage pose problems for security that seem to go beyond those studied in the context of communication protocols. Perhaps the most striking of these features is dynamic access control. Indeed, while most storage systems feature dynamic access control in some form, its consequences on more abstract security properties like secrecy and integrity are seldom evaluated in detail.

In this paper, we show that protocols for secure storage are worth analyzing, and study an interesting example. Specifically, we analyze a state-of-the-art file-sharing protocol that exploits cryptographic techniques for secure storage on an untrusted server. The protocol is the basis for the file system Plutus [32]. This setting is interesting for several reasons. First, compromise of storage servers is a reasonably common threat today, and it is prudent not to trust such servers for security [34]. Next, the protocol we study has a very typical design for secure file sharing on untrusted storage, where data is stored encrypted and signed, and keys for encrypting, signing, verifying, and decrypting such data are managed by users. Several file systems follow this basic design, including SNAD [35], SiRiUS [28], and other cryptographic file systems since the 1990s [15]. Finally, beyond the basic design, the protocol features some promising new schemes like lazy revocation and key rotation that improve the protocol's performance in the presence of dynamic access control, but in turn complicate its security properties. These features are worthy of study. For instance, our analysis reveals that lazy revocation allows more precise integrity guarantees than the naïve scheme in [28]. With lazy revocation, if an untrusted writer is revoked, readers can distinguish contents that are written after the revocation from previous contents that may have been written by that writer; consequently, they can trust the former contents even if they do not trust the latter contents. On a different note, the computational security of key rotation schemes has generated a lot of interest recently [6, 7, 26]. Our analysis reveals some new integrity vulnerabilities in the protocol that can be exploited even if the key rotation scheme is secure.

Formal techniques play a significant role in our analysis. We model the protocol and verify its security properties in the automatic protocol verifier ProVerif. ProVerif is based on solid formal foundations that include theory for the applied pi calculus and proof theory for first-order logic. The

*Bruno Blanchet's work has been done within the INRIA ABSTRACTION project-team (common with the CNRS and the ÉNS).

978-0-7695-3168-7 /08 $25.00 © 2008 IEEE
DOI 10.1109/SP.2008.12

formal language forces us to specify the protocol precisely, and prove or disprove precise security properties of the protocol. This level of rigor pays off in several ways:

- We find a new integrity attack on the protocol, and show that it can have serious practical consequences. That this attack has eluded discovery for more than four years is testimony to the difficulty of finding such attacks "by hand".
- We propose a fix and prove that it corrects the protocol. Both the attack and the correction are relative to a formal specification of integrity that is not immediately apparent from the informal specification in [32]. We also prove a weaker secrecy guarantee than the one claimed in [32] (and show that their claim cannot be true).
- The formal exercise allows us to notice and clarify some ambiguities in [32]; it also allows us to find some new, simpler attacks where more complex ones were known. These discoveries vastly improve our understanding of the protocol's subtleties.
- Finally, the use of an automatic verifier yields a much higher level of confidence in our proofs than manual techniques, which have been known to be error-prone.

More generally, our results reconfirm that informal justifications (such as showing resistance to specific attacks) are not sufficient for protocols. As far as we know, our study is the first automated formal analysis of a secure storage protocol; we expect our approach to be fruitful for other protocols in this area.

Related work There is a huge body of work on formal methods for the verification of security protocols, *e.g.*, [1, 4, 8, 16, 29, 33, 37]. We refer the reader to [14] for more information on this work, and we focus here on more closely related work on the design and verification of secure file systems.

In file systems based on the network-attached (object) storage protocols (NASD, OSD) [27, 30], distributed access control is implemented on trusted storage via cryptographic capabilities. A semi-formal security analysis of this protocol appears in [30], while [19–21] present formal models and manual security proofs for this protocol in the applied pi calculus.

Among other protocols for secure file sharing on untrusted storage, the closest to the one we study here are those behind the file systems Cepheus [25], SiRiUS [28], and SNAD [35]. Lazy revocation first appears in Cepheus; see [31] for a summary of the origins of lazy revocation, and its limitations. Keys for reading and writing files in SiRiUS are the same as those in Plutus. However, those keys are stored and distributed securely by the server ("in-band"), instead of being directly distributed by users ("out-of-band"). Moreover, revocation in SiRiUS is immediate, instead of lazy. In SNAD, keys for reading files are distributed in-band as in SiRiUS. However, unlike Plutus and SiRiUS, there are no keys for writing files—any user can write contents by signing those contents with its private key, and the storage server is trusted to control access to writes.

While the protocol we study partially trusts the storage server to prevent so-called rollback attacks (where contents received from the file system are not the most recent contents sent to the file system), the protocol behind the file system SUNDR [34] specifically provides a guarantee called *fork consistency*, that allows users to detect rollback attacks without trusting the storage server. The correctness of that protocol is formally proved in [34]. SUNDR does not focus on other secrecy and integrity guarantees.

Recently several schemes for key rotation have been proposed and manually proved in the computational model of security [6, 7, 26], and various alternative schemes for key distribution and signatures have been designed to eliminate public-key cryptography in this context [36]. Mechanically verifying these schemes should be interesting future work.

Finally, to guarantee stronger information-flow properties than the ones studied in this paper, access control must be complemented by precise code analysis. Recently, several type systems have been designed for such purposes [18, 22, 38, 42]. The type system in [18] is particularly suitable for proving such properties in the presence of dynamic access control and untrusted storage.

Organization The rest of the paper is organized as follows. In Section 2, we outline the protocol behind Plutus. In Section 3, we give an overview of ProVerif, and present our model of Plutus in ProVerif. In Section 4, we specify and analyze secrecy and integrity properties of Plutus in ProVerif, and present our results and observations. Finally, in Section 5, we discuss our contributions and conclude.

2. Plutus

The file system Plutus [32] is based on a storage design that does not rely on storage servers to provide strong secrecy and integrity guarantees. Instead, contents of files are cryptographically secured, and keys for writing and reading such contents are managed by the owners of those files. Special schemes are introduced to economize key distribution and cryptography in the presence of dynamic access control; those schemes complicate the protocol and its security properties.

In Plutus, principals are qualified as owners, writers, and readers. Every file belongs to a group[1], and all files in a

[1]There is a difference between the informal interpretation of a group

group have the same writers and readers. The owner of a group generates and distributes keys for writing and reading contents for that group; those keys are shared by all files in that group. Specifically, a *write key* is used to encrypt and sign contents, while a *read key* is used to verify and decrypt such contents. These keys can be revoked by the owner to dynamically control access to those files; a new write key and a new read key are then generated and distributed appropriately. However, the new write key is used only for subsequent writes: unlike SiRiUS [28], the files are not immediately secured with the new write key, so that the previous read key can be used to verify and decrypt the contents of those files until they are re-written. This scheme, called *lazy revocation*, avoids redundant cryptography and is justified by the following observations:

- Encrypting the existing contents with the new write key does not guarantee secrecy of those contents from the previous readers, since those contents may have been cached by the previous readers.
- More subtly, since the existing contents come from the previous writers, signing those contents with the new write key would wrongly indicate that they come from the new writers.

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

Concretely, a write key is of the form (sk, lk), where sk is part of an asymmetric key pair (sk, vk), and lk is a symmetric encryption key; the complementary read key is (vk, lk). Here sk, vk, and lk are a *sign key*, a *verify key*, and a *lockbox key*. Contents are encrypted with lk[2] and signed with sk; those contents are verified with vk and decrypted with lk. Plutus uses the RSA cryptosystem [39], so we have $sk = (d, n)$ and $vk = (e, n)$, where the modulus n is the product of two large primes p and q, and the exponents d and e are inverses modulo $(p-1)(q-1)$, that is, $ed \equiv 1 \mod (p-1)(q-1)$. Thus, the functions $x \mapsto x^d \mod n$ and $y \mapsto y^e \mod n$ are inverses. Given a hash function hash, a message M is signed with sk by computing $S = \mathsf{hash}(M)^d \mod n$, and S is verified with vk by checking that $S^e \mod n = \mathsf{hash}(M)$. We call (p, q) the RSA seed. In general, e may be chosen randomly, relatively prime to $(p-1)(q-1)$, and d may be computed from e, p, and q. However in Plutus, e is uniquely determined by n and lk as follows: given a pseudo-random sequence $\langle r_i \rangle$ generated with seed lk, e is the first prime number in the sequence $\langle r_i + \sqrt{n} \rangle$. We denote this algorithm by $\mathsf{genExp}(n, lk)$. To sum up, a sign/verify key pair (sk, vk) is generated from a random RSA seed (p, q) and a lockbox key lk, by computing $n = pq$, $e = \mathsf{genExp}(n, lk)$, $vk = (e, n)$, and $sk = (d, n)$, where d is the inverse of e modulo $(p-1)(q-1)$.

The owner of a group distributes (sk, lk) to writers and lk to readers; users can further derive vk from n and lk using genExp. Note that n is already available to writers from sk. Further, the owner distributes a signed n to writers, which they attach whenever they write contents to the file system—so *any* user can obtain n from the file system and verify its authenticity. Thus writers can act for readers in Plutus, although in [32] it is wrongly claimed that writers cannot derive vk (implying that read access is disjoint from writer access). It is already known that writers can act for readers in SiRiUS in a similar way [28, 36].

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

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

Version v **to version** $v+1$ When keys for version v are revoked, a new lockbox key lk_{v+1} is generated by "winding" the previous lockbox key lk_v with the owner's private key: $lk_{v+1} = lk_v^D \mod N$. The previous lockbox key can be retrieved by "unwinding" the new lockbox key with the owner's public key: $lk_v = lk_{v+1}^E \mod N$. In particular, a reader with a lockbox key $lk_{v'}$ for any $v' \geq v$ can generate the verify key vk_v by obtaining the modulus n_v from the file system, recursively unwinding $lk_{v'}$ to lk_v, and deriving vk_v from n_v and lk_v using genExp. The new sign/verify key pair (sk_{v+1}, vk_{v+1}) is generated from a random RSA seed (with modulus n_{v+1}) and lk_{v+1}.

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

in [32], and the formal interpretation of a group in this paper. In fact, the interpretation in [32] is inconsistent; see Section 4.4 for a more detailed discussion of this issue.

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

revert the store to a previous state, or garbage the current state.

3. ProVerif and a formal model of Plutus

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

3.1. ProVerif

The automatic verifier ProVerif [2, 11, 12, 14] is designed to verify security protocols. The protocol is specified in an extension of the pi calculus with cryptography, a dialect of the applied pi calculus [3]. The desired security properties can be specified, in particular, as correspondence assertions [41], which are properties of the form "if some event has been executed, then other events have been executed". (We illustrate this input language below.) Internally, the protocol is translated into a set of Horn clauses, and the security properties are translated into derivability queries on these clauses: the properties are proved when certain facts are not derivable from the clauses. ProVerif uses a resolution-based algorithm to show this non-derivability.

ProVerif relies on the formal, so-called Dolev-Yao model of protocols [24], in which messages are modeled as terms in an algebra. This rather abstract model of cryptography makes it easier to automate proofs than the more concrete, computational model, in which messages are modeled as bitstrings. Consequently, ProVerif can handle a wide variety of cryptographic primitives specified by rewrite rules or equations over terms. Moreover:

- When ProVerif proves a property, the proof is valid for an unbounded number of sessions of the protocol and an unbounded message size.

- When the proof fails, ProVerif provides a derivation of a fact from the clauses. It also tries to reconstruct, from this derivation, a trace of the protocol that shows that the property is false [5]. When trace reconstruction fails, ProVerif gives no definite answer. Such a situation is unavoidable due to the undecidability of the problem. In our study, whenever this situation happened, manual inspection of the derivation provided by ProVerif allowed us to reconstruct an attack against the said property: the failure of the ProVerif proof always corresponded to an attack.

We refer the reader to [14] for detailed information on ProVerif and the theory behind it.

3.2. A model of Plutus in ProVerif

We now present a model of Plutus in ProVerif; its security properties are specified and studied in Section 4.

3.2.1. Cryptographic primitives, lists, and integers

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

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

models the fact that any term of the form $\mathsf{enc}(M, K)$ can be decrypted with K to obtain M. (Here x and y are variables that can match any M and K.) The term $\mathsf{hash}(M)$ denotes the hash of message M. The term $\mathsf{exp}(M, (R, N))$ denotes the result of computing $M^R \mod N$. We abstract random RSA seeds as fresh names. The term $\mathsf{N}(s)$ denotes the modulus of seed s. The term $\mathsf{e}(s, K)$ denotes the unique exponent determined by the modulus $\mathsf{N}(s)$ and base K by the algorithm described in Section 2; this fact is modeled by the rewrite rule:

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

The term $\mathsf{d}(s, K)$ is the inverse exponent, as explained in Section 2. This fact is modeled by the equations:

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

Finally, the rewrite rule

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

models the fact that a modulus $\mathsf{N}(s)$ can be efficiently "factored" to obtain the RSA seed s if both exponents $\mathsf{e}(s, K)$ and $\mathsf{d}(s, K)$ are known [17].

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

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

$$\mathsf{geq}(x, x);$$
$$\mathsf{geq}(x, y) \Rightarrow \mathsf{geq}(\mathsf{succ}(x), y).$$

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

3.2.2. The protocol

We model principals as applied pi-calculus processes with events [14]. Informally:

- $\mathbf{out}(u, M); P$ sends the message M on a channel named u and continues as the process P; a special case is the process $\mathbf{out}(u, M)$, where there is no continuation.
- $\mathbf{in}(u, X); P$ receives a message M on a channel named u, matches M with the pattern X, and continues as the process P with variables in X bound to matching terms in M. Here X may be a variable x, which matches any message and stores it in x; a pattern $=N$, which matches only the message N; or even a more complex pattern like $(=N, x)$, which matches any pair whose first component is N and stores its second component in x.
- $\mathbf{new}\ m; P$ creates a fresh name m and continues as the process P.
- $\mathbf{event}\ e(M_1, \ldots, M_n); P$ executes the event $e(M_1, \ldots, M_n)$ and continues as the process P. A special case is the process $\mathbf{event}\ e(M_1, \ldots, M_n)$, where there is no continuation. The execution of $e(M_1, \ldots, M_n)$ merely records that a certain program point has been reached for certain values of $M_1, \ldots, M_n$. Such events are used for specifying security properties, as explained in Section 4.1.
- $\mathbf{if}\ M = M'\ \mathbf{then}\ P\ \mathbf{else}\ Q$ executes P if M evaluates to the same term as M'; otherwise it executes Q. A special case is the process $\mathbf{if}\ M = M'\ \mathbf{then}\ P$, where there is no $\mathbf{else}$ continuation.
- $\mathbf{let}\ X = M\ \mathbf{in}\ P$ evaluates M, matches it with the pattern X and, when the matching succeeds, continues as P with the variables in X bound to matching terms in the value of M.
- $P \mid Q$ runs the processes P and Q in parallel.
- $!P$ runs an unbounded number of copies of the process P in parallel.

In Figures 1, 2, and 3, we define processes that model the roles of owners, writers, and readers; the protocol is specified as the parallel composition of these processes. (The storage server is assumed to be untrusted at this point, and therefore not modeled. We study server-verified writes and their properties later.) The network is modeled by a public channel net; as usual, we assume that the adversary controls the network. Likewise, the file system is modeled by a public channel fs. On the other hand, private (secure) channels are not available to the adversary. For instance, $\mathsf{rprivchannel}(r)$ and $\mathsf{wprivchannel}(w)$ are private channels on which an owner sends keys to reader r and writer w, respectively. We limit the number of revocations that are possible in any group to $\max_{\mathrm{rev}}$. (Thus the number of versions is bounded. At this level of detail, ProVerif does not terminate with an unbounded number of versions. We managed to obtain termination with an unbounded number of versions for a more abstract treatment of cryptography, thanks to an extension of ProVerif that takes advantage of the transitivity of geq in order to simplify the Horn clauses. However, we do not present that abstract model here because it misses some of the attacks that are found with the more detailed model below.)

First, Figure 1 shows the code for owners. An owner creates its private/public key pair (lines 2–5), and then creates groups on request (lines 7–9). For each group, the owner maintains some state on a private channel $currentstate$. (The current state is carried as a message on this channel, and the owner reads and writes the state by receiving and sending messages on this channel.) The state includes the current version number, the lists of allowed readers and writers, the lockbox key, and the sign key for that group. The owner creates the initial version of keys for the group (lines 12–14), generates at most $\max_{\mathrm{rev}}$ subsequent versions on request (lines 17–21), and distributes those keys to the allowed readers and writers on request (lines 25–30 and 34–40). The generation and distribution of keys follow the outline in Section 2. Moreover, the owner signs the modulus of each version with its private key (line 38), sends the signed modulus to writers of that version (line 40), and sends its public key to readers so that they may verify that signature (line 30). Events model runtime assertions in the code: for instance, $\mathsf{isreader}(r, g, v)$ and $\mathsf{iswriter}(w, g, v)$ assert that r is a reader and w is a writer for group g at version v.

Next, Figure 2 shows the code for writers. A writer for group g at version v obtains the lockbox key, the sign key, and the owner-signed modulus for v from the owner of g (lines 46–47). To write data, an honest writer encrypts that data with the lockbox key (line 50), signs the encryption with the sign key (line 51), and sends the signed encryption to the file system with a header that includes the owner-signed modulus (lines 52–54). The event $\mathsf{puts}(w, M, g, v)$ asserts that an honest writer w for group g sends data M to the file system using keys for version v. In contrast, a dishonest writer leaks the lockbox key, the sign key, and the owner-signed modulus (line 59); the adversary can use this information to act for that writer. The event $\mathsf{corrupt}(w, g, v)$ asserts that a writer w for group g is corrupt at version v.

```
1  let processOwr =
2    new seed1; new seed2;                                   (* create owner's RSA key pair *)
3    let ownerpubkey = (e(seed1, seed2), N(seed1)) in
4    let ownerprivkey = (d(seed1, seed2), N(seed1)) in
5    out(net, ownerpubkey);                                  (* publish owner's RSA public key *)
6    (
7    ! in(net, (= newgroup, initreaders, initwriters));      (* receive a new group creation request;
                                initreaders and initwriters are the initial lists of allowed readers and writers, respectively *)
8     new g;                                                 (* create the new group g *)
9     out(net, g);                                           (* publish the group name g *)
10    new currentstate;                                      (* create a private channel for the current state for group g *)
11    (
12     ( new initlk;                                         (* create initial lk *)
13       new seed3; let initsk = (d(seed3, initlk), N(seed3)) in   (* generate initial sk *)
14       out(currentstate, (zero, initreaders, initwriters, initlk, initsk))
                                                             (* store state for version 0 on channel currentstate *)
15    )
16    |                                                      (* Next, we move from version 0 to version 1 *)
17    ( in(net, (= revoke, = g, newreaders, newwriters));    (* receive a revoke request for group g;
                                newreaders and newwriters are the new lists of allowed readers and writers *)
18      in(currentstate, (= zero, oldreaders, oldwriters, oldlk, oldsk));   (* read state for version 0 *)
19      let newlk = exp(oldlk, ownerprivkey) in              (* wind old lk to new lk *)
20      new seed3; let newsk = (d(seed3, newlk), N(seed3)) in   (* generate new sk *)
21      out(currentstate, (succ(zero), newreaders, newwriters, newlk, newsk))
                                                             (* store state for version 1 on channel currentstate *)
22    )
23    | ... |                                                (* Similarly, we move from version 1 to version 2, and so on *)
24    (
25    ! in(net, (= rkeyreq, r, = g));                        (* receive read key request for reader r and group g *)
26      in(currentstate, (v, readers, writers, lk, sk));     (* get the current state *)
27      out(currentstate, (v, readers, writers, lk, sk));
28      if member(r, readers) then                           (* check that the reader r is allowed *)
29       ( event isreader(r, g, v);                          (* assert that r is a reader for group g and version v *)
30         out(rprivchannel(r), (g, v, lk, ownerpubkey)) )   (* send lk and owner's public key to r *)
31    )
32    |
33    (
34    ! in(net, (= wkeyreq, w, = g));                        (* receive write key request for writer w and group g *)
35      in(currentstate, (v, readers, writers, lk, sk));     (* get the current state *)
36      out(currentstate, (v, readers, writers, lk, sk));
37      if member(w, writers) then                           (* check that the writer w is allowed *)
38       ( let (_, n) = sk in let sn = exp(hash(n), ownerprivkey) in   (* sign the modulus *)
39         event iswriter(w, g, v);                          (* assert that w is a writer for group g and version v *)
40         out(wprivchannel(w), (g, v, lk, sk, sn)))         (* send lk, sk, and signed modulus to w *)
41    )
42   )
43  ).
```

Figure 1. Code for owners

```
44 let processWtr =
45  ! in(net, (w, g));                                   (* initiate a writer w for group g *)
46    out(net, (wkeyreq, w, g));                         (* send write key request *)
47    in(wprivchannel(w), (= g, v, lk, sk, sn));         (* obtain lk, sk, and signed modulus *)
48    (
49     ( new m;                                          (* create data to write *)
50       let encx = enc(m, lk) in                        (* encrypt *)
51       let sencx = exp(hash(encx), sk) in              (* sign *)
52       event puts(w, m, g, v);                         (* assert that data m has been written by w for group g at version v *)
53       let (dx, n) = sk in
54       out(fs, (g, v, n, sn, encx, sencx))             (* send content to file system *)
55     )
56     |
57     ( in(net, = (corrupt, w));                        (* receive corrupt request for w *)
58       event corrupt(w, g, v);                         (* assert that w has been corrupted for group g at version v *)
59       out(net, (lk, sk, sn))                          (* leak lk, sk, and signed modulus *)
60     )
61    ).
```

Figure 2. Code for writers

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

Figure 3. Code for readers

Finally, Figure 3 shows the code for readers. A reader for group g at version v obtains the lockbox key for v from the owner of g (lines 64–65). To read data, an honest reader obtains content from the file system (line 67), and parses that content to obtain a signed encryption and a header that contains g, a version number vx, and a signed modulus. It verifies the signature of the modulus with the owner's public key (line 68); it then generates the verify key for vx from the modulus and the lockbox key (lines 69–71), verifies the signature of the encryption with the verify key (line 72), and decrypts the encryption with the lockbox key (line 73). The generation of the verify key for vx from the modulus for vx and the lockbox key for v follows the outline in Section 2: the lockbox key lk for vx is obtained from the lockbox key for v by unwinding it $v - vx$ times (line 70), after which genExp generates the required exponent (line 71). In Figure 3, we detail only the case where $v = 1$ and $vx = 0$ (lines 69–75), in which case we unwind the lockbox key once (line 70); the ProVerif script includes a similar block of code for each $vx \leq v \leq \max_{\text{rev}}$, located at line 76 and omitted in Figure 3. The event $\mathsf{gets}(r, x, g, vx)$ asserts that an honest reader r for group g receives data x from the file system using keys for version vx. In contrast, a dishonest reader leaks the lockbox key (line 81); the adversary can use this information to act for that reader. The event $\mathsf{corrupt}(r, g, v)$ asserts that a reader r in group g is corrupt at version v.

4. Security results on Plutus

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

4.1. Correspondences

Properties of the protocol are specified as correspondences [41]. The verifier ProVerif can prove such correspondences [14]. A simple example is the correspondence $e(M_1, \ldots, M_n) \rightsquigarrow e'(M'_1, \ldots, M'_{n'})$, which means that in any trace of the protocol in the presence of an adversary, the event $e(M_1, \ldots, M_n)$ must not be executed unless the event $e'(M'_1, \ldots, M'_{n'})$ is executed. More generally, correspondences may include equality tests of the form $M = M'$, atoms of the form $pred(M_1, \ldots, M_n)$ that rely on user-defined predicates $pred$ (such as geq and member), and atoms of the form $\mathsf{attacker}(M)$, which mean that the attacker knows the term M.

Definition 4.1 (Correspondences)**.** *Let $\mathcal{T}$ range over traces, σ over substitutions, and ϕ over formulas of the form* $\mathsf{attacker}(M)$*,* $e(M_1, \ldots, M_n)$*,* $pred(M_1, \ldots, M_n)$*,* $M = M'$*,* $\phi_1 \wedge \phi_2$*, or* $\phi_1 \vee \phi_2$*.*

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

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

In a correspondence $\phi \rightsquigarrow \phi'$, the variables of ϕ are universally quantified (because σ is universally quantified), and the variables of ϕ' that do not occur in ϕ are existentially quantified (because σ' is existentially quantified). ProVerif can prove correspondences $\phi \rightsquigarrow \phi'$ of a more restricted form, in which ϕ is of the form $\mathsf{attacker}(M)$ or $e(M_1, \ldots, M_n)$. This corresponds to the formal definition of correspondences proved by ProVerif given in [14, Definition 3], except for two extensions: we allow atoms of the form $\mathsf{attacker}(M)$, $M = M'$, and $pred(M_1, \ldots, M_n)$ to occur in ϕ' and we do not require that ϕ' be in disjunctive normal form.

In order to prove correspondences, ProVerif translates the process and the actions of the adversary into a set of Horn clauses $\mathcal{R}$. In these clauses, messages are represented by *patterns*[3] p, which are terms in which names a have been replaced with functions $a[\ldots]$. Free names are replaced with constants $a[\,]$, while bound names created by restrictions are replaced with functions of the messages previously received and of session identifiers that take a different value at each execution of the restriction—so that different names are represented by different patterns. The clauses use the following kinds of facts:

- $\mathsf{attacker}(p)$, which means that the adversary may have the message p;
- $\mathsf{message}(p, p')$, which means that the message p' may be sent on channel p;

[3] Note that the meaning of "pattern" in this context is different from the usual meaning, *e.g.* in Section 3.2.2.

- $\mathsf{event}(e(p_1, \ldots, p_n))$, which means that the event $e(p_1, \ldots, p_n)$ may have been executed;
- $\mathsf{m\text{-}event}(e(p_1, \ldots, p_n))$, which means that the event $e(p_1, \ldots, p_n)$ must have been executed;
- the facts $\mathsf{geq}(p, p')$ and $\mathsf{member}(p, p')$, which are defined in Section 3.2.1.

The clauses that define geq and member are shown in Section 3.2.1. The other clauses in $\mathcal{R}$ are generated automatically by ProVerif from the process and from the definitions of the function symbols; see [14, Section 5.2] for details. ProVerif establishes security properties by proving that certain facts are derivable from these clauses only if certain hypotheses are satisfied. The derivability properties are determined by a resolution-based algorithm, described in [14, Section 6]. Specifically, ProVerif computes a function $\mathrm{solve}_{P,Init}(F)$ that takes as argument a process P, the initial knowledge of the adversary $Init$, and a fact F, and returns a set of Horn clauses that determines which instances of F are derivable. More precisely, let $\mathcal{F}_{\mathrm{me}}$ be any set of m-event facts, which are supposed to hold. An instance F_0 of F is derivable from $\mathcal{R} \cup \mathcal{F}_{\mathrm{me}}$ if and only if there exist a clause $H \Rightarrow C$ in $\mathrm{solve}_{P,Init}(F)$ and a substitution σ_0 such that $F_0 = \sigma_0 C$ and the facts in $\sigma_0 H$ are derivable from $\mathcal{R} \cup \mathcal{F}_{\mathrm{me}}$. In particular, if $\mathrm{solve}_{P,Init}(F) = \emptyset$, then no instance of F is derivable from $\mathcal{R} \cup \mathcal{F}_{\mathrm{me}}$ for any $\mathcal{F}_{\mathrm{me}}$. Other values of $\mathrm{solve}_{P,Init}(F)$ give information on which instances of F are derivable and under which conditions. In particular, the m-event facts in the hypotheses of clauses in $\mathrm{solve}_{P,Init}(F)$ must be in $\mathcal{F}_{\mathrm{me}}$ in order to derive an instance of F (since $\mathcal{R}$ contains no clause that concludes m-event facts), so the corresponding events must have been executed.

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

Theorem 4.2 (Correspondences)**.** *Let P be a closed process. Let $\phi \rightsquigarrow \phi'$ be a correspondence, where ϕ is $\mathsf{attacker}(M)$ or $e(M_1, \ldots, M_n)$. Let $F = \mathsf{attacker}(p)$ if $\phi = \mathsf{attacker}(M)$ and $F = \mathsf{event}(e(p_1, \ldots, p_n))$ if $\phi = e(M_1, \ldots, M_n)$, where $p, p_1, \ldots, p_n$ are the patterns obtained from the terms $M, M_1, \ldots, M_n$ respectively, by replacing names a with patterns $a[\,]$. Let ψ' be the formula obtained from ϕ' by replacing names a with patterns $a[\,]$.*

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

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

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

This theorem is an extension of [14, Theorem 4] to the case in which ϕ' may contain atoms $\mathsf{attacker}(M)$, $M = M'$, and $pred(M_1, \ldots, M_n)$, and ϕ' may not be in disjunctive normal form. Intuitively, if $\mathcal{T}$ satisfies $\sigma_M\phi$, then $\sigma_p F$ is derivable, where σ_p is the substitution on patterns that corresponds to the substitution on terms σ_M. So there exist a clause $H \Rightarrow C$ in $\mathrm{solve}_{P,Init}(F)$ and a substitution σ_0 such that $\sigma_p F = \sigma_0 C$ and the facts $\sigma_0 H$ are derivable. Since $H \vdash \sigma\psi'$, we also have $\sigma_0\sigma\psi'$. Moreover, $C = \sigma F$, so $\sigma_p F = \sigma_0\sigma F$. So, letting $\sigma'_p = \sigma_0\sigma$, we have $\sigma_p F = \sigma'_p F$ and $\sigma'_p\psi'$, so $\sigma_M\phi = \sigma'_M\phi$ and $\mathcal{T}$ satisfies $\sigma'_M\phi'$, where σ'_M is the substitution on terms that corresponds to the substitution σ'_p on patterns. Hence the correspondence $\phi \rightsquigarrow \phi'$ is satisfied.

In this paper, we use the more general language of correspondences of Definition 4.1, and show how to exploit the more limited queries that ProVerif can prove in order to prove the correspondences that we need.

4.2. Main security properties of Plutus

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

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

satisfies the correspondence

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

Unfortunately, here writers can act for readers (see Section 2), so a corrupt writer at v leaks (at least) as much information as a corrupt reader at v. Note that on the contrary, it is intended in [32] that read access be disjoint from write access. Moreover, since the read key for v can be obtained from the read key for any $v' \geq v$ by unwinding, even a corrupt reader (or writer) at such v' leaks as much information as a corrupt reader at v. Of course, if the set of readers does not increase, a reader at v' is already a reader at v, so this situation is not surprising. (Indeed, this is the case that motivates key rotation in [32].) On the other hand, increasing the set of readers may result in unintended declassification of secrets. In light of these observations, we must weaken our specification of secrecy.

Definition 4.3 (Secrecy)**.** *Secrecy is preserved in Plutus if, for all g and v, any secret m written by an honest writer for g using keys for v is leaked only if a reader or writer for g is corrupt at some $v' \geq v$,* i.e.*, the process modeling Plutus satisfies the correspondence*

$$\mathsf{puts}(w, m, g, v) \ \wedge \ \mathsf{attacker}(m) \ \rightsquigarrow \\ v' \geq v \ \wedge \ \mathsf{corrupt}(a, g, v') \\ \wedge \ (\mathsf{isreader}(a, g, v') \ \vee \ \mathsf{iswriter}(a, g, v')) \tag{1}$$

This weaker property is proved as follows.

Theorem 4.4. *Secrecy is preserved in Plutus.*

Proof. Let $m[g = G, v = V]$ denote the name m created in line 49 when the variables g and v in lines 45 and 47 are bound to the terms G and V, respectively. (This notation can be used directly in ProVerif, exploiting ProVerif's internal representation of bound names by patterns. It is detailed and justified in [14].) ProVerif automatically proves the following correspondence:

$$\mathsf{attacker}(m[g = x_g, v = x_v]) \ \rightsquigarrow \\ v' \geq x_v \ \wedge \ \mathsf{corrupt}(a, x_g, v') \\ \wedge \ (\mathsf{isreader}(a, x_g, v') \ \vee \ \mathsf{iswriter}(a, x_g, v')) \tag{2}$$

By the semantics of the input language, for any terms W, M, G, and V, if $\mathsf{puts}(W, M, G, V)$ is executed, then $M = m[g = G, v = V]$. Thus, for all substitutions σ, if a trace $\mathcal{T}$ satisfies $\sigma\mathsf{puts}(w, x_m, x_g, x_v)$ and $\sigma\mathsf{attacker}(x_m)$, then $\sigma x_m = \sigma m[g = x_g, v = x_v]$; so $\mathcal{T}$ satisfies $\sigma\mathsf{attacker}(m[g = x_g, v = x_v])$; so by correspondence (2), $\mathcal{T}$ satisfies $\sigma'(v' \geq x_v \wedge \mathsf{corrupt}(a, x_g, v') \wedge (\mathsf{isreader}(a, x_g, v') \vee \mathsf{iswriter}(a, x_g, v')))$ for some substitution σ' such that $\sigma' x_g = \sigma x_g$ and $\sigma' x_v = \sigma x_v$. Hence, correspondence (1) is satisfied. □

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

Definition 4.5 (Integrity)**.** *Integrity is preserved in Plutus if for all g and v, any data x read by an honest reader for g using keys for v is written by an honest writer for g using keys for v unless a writer for g is corrupt at v,* i.e.*, the process modeling Plutus satisfies the correspondence*

$$\mathsf{gets}(r, x, g, v) \ \rightsquigarrow \\ \mathsf{iswriter}(w, g, v) \\ \wedge \ (\mathsf{puts}(w, x, g, v) \ \vee \ \mathsf{corrupt}(w, g, v)) \tag{3}$$

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

Theorem 4.6. *Integrity is not preserved in Plutus,* i.e.*, the correspondence* (3) *is not satisfied.*

Proof. When ProVerif is given the query (3), it cannot prove this query, and outputs a derivation of $\mathsf{gets}(r, m, g, 0)$ from facts that do not include $\mathsf{puts}(w, m, g, 0)$ or $\mathsf{corrupt}(w, g, 0)$ for any w; we manually check that this derivation corresponds to an attack. Briefly, a reader for g is corrupted at version 0 and a writer for g is corrupted at version 1; the adversary then constructs a bogus write key for version 0 and writes content that can be read by r using the read key for version 0. In more detail:

1. A reader for group g is corrupted at version 0 to get the lockbox key lk_0 for version 0.
2. Next, a writer for g is corrupted at version 1 to get the lockbox key lk_1, the sign key $(\mathsf{d}(s_1, lk_1), \mathsf{N}(s_1))$, and the owner-signed modulus $sn_1 = \mathsf{exp}(\mathsf{hash}(\mathsf{N}(s_1)), ownerprivkey)$ for version 1 (where s_1 is the RSA seed for version 1 and $ownerprivkey$ is the private key of the owner).
3. The exponent $\mathsf{e}(s_1, lk_1)$ is computed as $\mathsf{genExp}(\mathsf{N}(s_1), lk_1)$.
4. Next, the RSA seed s_1 is computed as $\mathsf{crack}(\mathsf{e}(s_1, lk_1), \mathsf{d}(s_1, lk_1), \mathsf{N}(s_1))$.
5. Now a bogus sign key sk' is constructed as $(\mathsf{d}(s_1, lk_0), \mathsf{N}(s_1))$.

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

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

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

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

The above attack can have serious consequences, since it implies that *a writer for an arbitrary group can act as a legitimate writer for a target group simply by colluding with a reader for that group*. Here, we consider a model without server-verified writes, that is, we assume that the server is compromised and colludes with the adversary. As argued in [28, 34], server compromise is a realistic possibility, so the above attack can be quite damaging. Worse, integrity is not preserved even in a model extended with server-verified writes. However with server-verified writes, the consequences are less serious—in order to write data for a group, the adversary needs to obtain the current write token for that group, for which it needs to corrupt a current writer for that group. Still, the attack has the same undesirable effect as allowing rotation of write keys. Specifically, it allows a corrupt writer at a later version to modify data in such a way that readers date the modified data back to an earlier version; in other words, the modified data appears to be older than it actually is to readers. This situation can be dangerous. Suppose that a reader trusts all writers at version 0, but not some writer at version 1 (say because the corruption of that writer at version 1 has been detected and communicated to the reader). The reader may still trust data written at version 0. However, the above attack shows that such data cannot be trusted: that data may in fact come from a corrupt writer at version 1.

We propose a simple fix $\mathbb{F}$ to correct the protocol: owners must sign each modulus with its correct group and version. More concretely, the term bound to sn at line 38 of the code for owners must be $\mathsf{exp}(\mathsf{hash}(n, g, v), ownerprivkey)$, and conversely, line 68 of the code for readers must check that $\mathsf{hash}(n, g, v) = \mathsf{exp}(sn, ownerpubkey)$. The corrected model preserves integrity as shown by Theorem 4.7 below. (Moreover, Theorem 4.4 continues to hold for the corrected model, with an unchanged proof.)

Theorem 4.7. *Integrity is preserved in Plutus with fix* $\mathbb{F}$.

Proof. Under the given conditions, ProVerif automatically proves the correspondence (3). □

While Definition 4.5 restricts the source of data read by honest readers, it still allows the adversary to replay stale data from a cache; in particular, content written by a writer at version v may be cached and replayed by the adversary at a later version v', when that writer is revoked. Unfortunately, in the model above we cannot associate contents that are read from the file system with the versions at which they are written to the file system. Such associations are possible only if the file system is (at least partially) trusted, as with server-verified writes.

Below we specify a stronger integrity property that we expect to hold in a model extended with server-verified writes; the property not only restricts the source of data read by honest readers, but also requires that such data be fresh. The code for the extended model is available online at `http://www.soe.ucsc.edu/~avik/projects/plutus/`. Briefly, we define a process to model the storage server, and extend the code for owners so that for any group g, a new write token is created for each version v, communicated to the server, and distributed to writers for g at v. Corrupt writers leak their write tokens. A writer must send contents to the server with a token; the contents are written to the file system only if that token is verified by the server to be the write token for the current version. Honest readers securely obtain server-verified contents from the server. (Of course, those contents are also publicly available from the server.) To verify the stronger integrity property, we replace the event $\mathsf{gets}(r, x, g, vx)$ in the code for readers (line 74) with a more precise event $\mathsf{gets}(r, x, g, vx, v')$. The latter event subsumes the former, and further asserts that the relevant contents are written to the file system after server-verification at v'. We expect that $v' = vx$, where vx is the version of keys used to read those contents, unless a writer for g is corrupt at v'; in the latter case, the adversary is able to replay at v' data that is originally written using keys for vx, so we may have $v' \geq vx$.

Definition 4.8 (Strong integrity)**.** *Strong integrity is preserved in Plutus if for all g and v, any data x read by an honest reader for g using keys for v is written by an honest writer for g using keys for v, unless a writer for g is corrupt at v; and further, such data is written either at v or at some version $v' \geq v$ at which a writer is corrupt,* i.e.*, the process modeling Plutus satisfies the correspondence*

$$\begin{aligned}
&\mathsf{gets}(r, x, g, v, v') \rightsquigarrow \\
&\quad \mathsf{iswriter}(w, g, v) \\
&\quad \wedge\ (\mathsf{puts}(w, x, g, v)\ \vee\ \mathsf{corrupt}(w, g, v)) \\
&\quad \wedge\ (v' = v\ \vee\ (v' \geq v\ \wedge \\
&\qquad \mathsf{iswriter}(w', g, v')\ \wedge\ \mathsf{corrupt}(w', g, v')))
\end{aligned} \tag{4}$$

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

Figure 4. Running times of ProVerif

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

Theorem 4.9. *Strong integrity is preserved in Plutus with server-verified writes and fix* $\mathbb{F}$.

Proof. Under the given conditions, ProVerif automatically proves the correspondence (4). □

Further, we show (using a correspondence omitted here) the correctness of server-verified writes: for any group g, only writers for g at the current version v can write data for g at v. (Such writes must be authorized by the current write token for g, which is distributed only to the current writers for g.) Consequently, server-verified writes prevent at least two kinds of attacks:

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

Running times of ProVerif Figure 4 presents the running times of ProVerif 1.14pl4 for the scripts above, in "minutes:seconds" format, on a 2.6 GHz AMD machine with 8 GB memory. We test models with or without fix $\mathbb{F}$, and with or without server-verified writes. We already find attacks assuming $\max_{\text{rev}} = 1$ for models without fix $\mathbb{F}$. On the other hand, models with fix $\mathbb{F}$ are tested assuming $\max_{\text{rev}} \leq 5$, so our security proofs apply only to those models (although we expect them to hold with larger values of $\max_{\text{rev}}$ as well). Memory usage increases significantly with server-verified writes; for example, the script with $\max_{\text{rev}} = 5$, fix $\mathbb{F}$, and server-verified writes takes around 2.2 GB of memory. For $\max_{\text{rev}} = 6$, ProVerif runs out of memory on this 8 GB machine.

4.3. Analysis of some design details

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

4.3.1. Why should a new modulus be created for each version?

The following explanation is offered by [32]:

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

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

1. A writer for g is corrupted at version 0, and a reader for g is corrupted at version 1. Thus the adversary obtains the lockbox key lk_0 and sign key (d_0, n) for version 0, and the lockbox key lk_1 for version 1. We may assume that the writer corrupted at 0 is revoked at 1. Let there be another writer for g at version 1 that publishes some content, so that the adversary also knows the owner-signed header sn_1 for version 1.
2. The adversary computes the exponent $e_0 = \mathsf{genExp}(n, lk_0)$, the RSA seed $s = \mathsf{crack}(e_0, d_0, n)$, and the sign key $sk_1 = (\mathsf{d}(s, lk_1), \mathsf{N}(s))$ for version 1. (Since the modulus n is unchanged, the RSA seed s is the same for versions 0 and 1.) Finally, choosing some fresh data m the adversary sends the following content to the file system, where $M = \mathsf{enc}(m, lk_1)$:
$$(g, 1, sn_1, n, M, \mathsf{exp}(\mathsf{hash}(M), sk_1))$$
3. An honest reader for g reads m using keys for version 1.

However, we have two comments on this attack:

- With server-verified writes, the sentence of [32] quoted above is not quite true: in order to become a valid writer, one additionally needs to obtain a write token at some version $v \geq 1$, which can be done only by corrupting a writer at some version $v \geq 1$.
- But by corrupting a writer at version $v \geq 1$, the adversary can mount a much simpler attack. Indeed, the adversary can compute the RSA seed s and all keys for version 1 from the keys for such v—without corrupting a writer at version 0 or a reader at version 1. We

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

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

ProVerif does not exhibit the former attack mentioned in [32] because it stops with this simpler attack.

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

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

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

4.4. Additional remarks

Below we list some more observations on the paper that describes Plutus [32]:

- The following sentence appears in [32, Section 3.1]:

  > *With filegroups, all files with identical sharing attributes are grouped in the same filegroup ...*

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

- The following sentence appears in [32, Section 3.4]:

  > *A revoked reader ... will never be able to read data updated since ... [its] revocation.*

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

- The following sentence appears in [32, Section 3.5.2]:

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

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

- The following sentence appears in [32, Section 3.2]:

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

  We clarify that contents should be signed after being encrypted (as in our model), for security in the computational model of cryptography. Indeed, signing encrypted contents allows one to use a weaker encryption scheme: the encryption scheme needs to be only IND-CPA (indistinguishable under chosen plaintext attacks), with the signature providing integrity of the ciphertext. Signing contents in the clear instead requires a stronger security assumption for the encryption scheme, that allows the adversary to call the decryption oracle. This point is similar to the fact that when the encryption is IND-CPA and the MAC is UF-CMA (unforgeable under chosen message attacks), encrypt-then-MAC (in which the MAC is applied to the ciphertext) guarantees the secrecy of the

plaintext, while encrypt-and-MAC (in which the MAC is applied to the plaintext) does not [9]. Here, the signature plays the role of the MAC.

- As noted in [26, Section 3], the key rotation scheme in [32] is not provable in the computational model of cryptography under reasonable assumptions (one-wayness of RSA and IND-CPA symmetric encryption), because a key obtained by unwinding is not indistinguishable from a random key when one has access to other winded versions of this key. This problem is out of scope of our verification since we work in the Dolev-Yao model of cryptography. Recently several other rotation schemes have been proposed, and their cryptographic security properties have been formally studied [6, 7, 26]. One can note that the attacks discussed in this section do not depend on the specific scheme for generating, winding, and unwinding lockbox keys. Our results continue to hold if we change the rotation scheme to a hash-chaining scheme [26, Section 5.1], for instance. They also continue to hold if lockbox keys are hashed before they are used for encryption, as proposed in [26, Section 5.3] and [7, Section 4.2] to correct the key rotation scheme in [32].

The scripts used in this paper are available at `http://www.soe.ucsc.edu/~avik/projects/plutus/`.

5. Conclusion

We have formally studied an interesting, state-of-the-art protocol for secure file sharing on untrusted storage (in the file system Plutus), and analyzed its security properties in detail using the automatic verifier ProVerif. Our study demonstrates that protocols for secure storage are worth analyzing. Indeed, the analysis vastly improves our understanding of the above protocol; we formally specify and verify its security properties, find (and fix) some unknown attacks, and clarify some design details that may be relevant for other storage protocols. Working in the Dolev-Yao model allows a deep analysis of the security consequences of some promising new features of the protocol. At the same time, some consequences remain beyond the scope of a Dolev-Yao analysis. It should be interesting to study those consequences in the computational model, perhaps using an automated verifier like CryptoVerif [10, 13]. Unfortunately, our initial attempts at modeling the protocol in CryptoVerif indicate that the tool is presently not mature enough to prove the relevant properties. We therefore postpone that study to a point at which tools for proofs in the computational model are more developed.

Over the years, storage has assumed a pervasive role in modern computing, and understanding secure storage has become as important as understanding secure communication. The study of secure communication has taught us the importance of rigor in the design and analysis of protocols. This observation certainly applies to secure storage as well. As far as we know, we are the first to present an automated formal analysis of a secure storage protocol. Our approach should be fruitful for other secure storage protocols, and we expect to see further work in this new area.

Acknowledgments

We would like to thank Martín Abadi for helpful discussions on this work and comments on a draft of this paper. Bruno Blanchet's work was partly supported by the ANR project ARA SSIA FormaCrypt. Avik Chaudhuri's work was partly supported by the National Science Foundation under Grants CCR-0208800 and CCF-0524078.

References

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[2] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005.

[3] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 104–115. ACM, 2001.

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

[5] X. Allamigeon and B. Blanchet. Reconstruction of attacks against cryptographic protocols. In *Proc. IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 140–154. IEEE, 2005.

[6] M. Backes, C. Cachin, and A. Oprea. Lazy revocation in cryptographic file systems. In *Proc. IEEE Security in Storage Workshop (SISW'05)*, pages 1–11. IEEE, 2005.

[7] M. Backes, C. Cachin, and A. Oprea. Secure key-updating for lazy revocation. In *Proc. European Symposium on Research in Computer Security (ESORICS'06)*, volume 4189 of *LNCS*, pages 327–346. Springer, 2006.

[8] M. Backes, A. Cortesi, and M. Maffei. Causality-based abstraction of multiplicity in security protocols. In *Proc. IEEE Computer Security Foundations Symposium (CSF'07)*, pages 355–369. IEEE, 2007.

[9] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Proc. International Conference on the Theory and Application of Cryptology & Information Security (ASIACRYPT'00)*, volume 1976 of *LNCS*, pages 531–545. Springer, 2000.

[10] B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*. To appear. Technical report version

available as ePrint Report 2005/401, http://eprint.iacr.org/2005/401.

[11] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proc. IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 82–96. IEEE, 2001.

[12] B. Blanchet. From secrecy to authenticity in security protocols. In *Proc. International Static Analysis Symposium (SAS'02)*, volume 2477 of *LNCS*, pages 342–359. Springer, 2002.

[13] B. Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *Proc. IEEE Computer Security Foundations Symposium (CSF'07)*, pages 97–111. IEEE, 2007. Extended version available as ePrint Report 2007/128, http://eprint.iacr.org/2007/128.

[14] B. Blanchet. Automatic verification of correspondences for security protocols. Report arXiv:0802.3444v1, 2008. Available at http://arxiv.org/abs/0802.3444v1.

[15] M. Blaze. A cryptographic file system for unix. In *Proc. ACM Conference on Computer and Communications Security (CCS'93)*, pages 9–16. ACM, 1993.

[16] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[17] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society*, 46(2):203–213, 1999.

[18] A. Chaudhuri. Dynamic access control in a concurrent object calculus. In *Proc. International Conference on Concurrency Theory (CONCUR'06)*, pages 263–278. Springer, 2006.

[19] A. Chaudhuri. On secure distributed implementations of dynamic access control. Technical Report UCSC-CRL-08-01, University of California at Santa Cruz, 2008.

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

[21] A. Chaudhuri and M. Abadi. Formal analysis of dynamic, distributed file-system access controls. In *Proc. IFIP WG6.1 Conference on Formal Techniques for Networked and Distributed Systems (FORTE'06)*, volume 4229 of *LNCS*, pages 99–114. Springer, 2006.

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

[23] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.

[24] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(12):198–208, 1983.

[25] K. Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, 1999.

[26] K. Fu, S. Kamara, and Y. Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *Proc. Network and Distributed System Security Symposium (NDSS'06)*. Internet Society (ISOC), 2006.

[27] H. Gobioff, G. Gibson, and J. Tygar. Security for network attached storage devices. Technical Report CMU-CS-97-185, Carnegie Mellon University, 1997.

[28] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proc. Network and Distributed System Security sympositum (NDSS'03)*, pages 131–45. Internet Society (ISOC), 2003.

[29] A. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–521, 2003.

[30] S. Halevi, P. A. Karger, and D. Naor. Enforcing confinement in distributed storage and a cryptographic model for access control. Cryptology ePrint Archive, Report 2005/169, 2005. Available at http://eprint.iacr.org/2005/169.

[31] M. Kallahalla, E. Riedel, and R. Swaminathan. System for enabling lazy-revocation through recursive key generation. United States Patent 7203317. Details available online at http://www.freepatentsonline.com/7203317.html, 2007.

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

[33] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *LNCS*, pages 147–166. Springer, 1996.

[34] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC'02)*, pages 108–117. ACM, 2002.

[35] E. L. Miller, W. E. Freeman, D. D. E. Long, and B. C. Reed. Strong security for network-attached storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST'02)*, pages 1–14. USENIX, 2002.

[36] D. Naor, A. Shenhav, and A. Wool. Toward securing untrusted storage without public-key operations. In *Proc. ACM Workshop on Storage Security and Survivability (StorageSS'05)*, pages 51–56. ACM, 2005.

[37] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[38] M. Pistoia, A. Banerjee, and D. A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *Proc. IEEE Symposium on Security and Privacy (S&P'07)*, pages 149–163. IEEE, 2007.

[39] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[40] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *Proc. USENIX Workshop on Electronic Commerce*, pages 29–40. USENIX, 1996.

[41] T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *Proc. IEEE Symposium on Security and Privacy (S&P'93)*, pages 178–194. IEEE, 1993.

[42] L. Zheng and A. Myers. Dynamic security labels and noninterference. In *Proc. IFIP WG1.7 Workshop on Formal Aspects in Security and Trust (FAST'04)*, pages 27–40. Springer, 2004.