Composable Security Analysis of OS Services

Ran Canetti[2], Suresh Chari[1], Shai Halevi[1], Birgit Pfitzmann[1], Arnab Roy[1], Michael Steiner[1], and Wietse Venema[1]

[1] IBM T.J. Watson Research Center[⋆]
[2] Tel-Aviv University[⋆⋆]

Abstract. We provide an analytical framework for basic integrity properties of file systems, namely the binding of files to filenames and writing capabilities. A salient feature of our modeling and analysis is that it is *composable:* In spite of the fact that we analyze the filesystem in isolation, security is guaranteed even when the file system operates as a component within an arbitrary, and potentially adversarial system.
Our results are obtained by adapting the *Universally Composable* (UC) security framework to the analysis of software systems. Originally developed for cryptographic protocols, the UC framework allows the analysis of simple components in isolation, and provides assurance that these components maintain their behavior when combined in a large system, potentially under adversarial conditions.

1 Introduction

Contemporary software systems are complex, consisting of many millions of lines of code, spread across a myriad of components and sub-components. A natural approach for analyzing such large systems is by analyzing each component separately, and "hoping" to use the component-wise analysis to analyze the entire system. Unfortunately, applying this approach to security analysis is problematic. Even if a component is simple enough to analyze separately, its interaction with other components can yield unexpected results. Often, a component will be used in environments different from what its designers initially had in mind, alongside other components that perhaps did not even exist when the original component was analyzed, potentially violating some assumptions that were made in the analysis.

Ideally, we would like to analyze the behavior of a component in isolation, and have the assurance that this behavior remains intact *even when that component is embedded in a new environment.* Within the realm of cryptography, the frameworks of Reactive Simulatability [19, 1] and Universal Composability (UC) [4, 5] ensure just that. These frameworks are aimed at capturing the security of cryptographic primitives and protocols, ranging from authentication and key exchange, to public-key encryption and signatures, zero-knowledge, and more (see [5] for many examples.) However, many of the features of these frameworks appear at first to be specific to the realm of cryptographic protocols. A natural question is whether the "composable security" approach sketched above can be carried out in a meaningful way even outside the limited domain of cryptography. In particular:

Can we obtain meaningful composable security in the context of general software systems?

A positive answer could significantly reduce the overhead in analyzing the security of large systems, while at the same time provide better overall security guarantees.

In this work we demonstrate that this can indeed be done, in the context of guaranteeing some basic integrity properties of filesystems. For this purpose we adapt the UC framework to software systems by establishing new conventions for modeling process management and scheduling. The current work is one of just a few attempts to apply the UC formalism to a large and complex software system, and we believe that it will enable further application of the UC formalism to other software systems.

[⋆] This work is supported in part by the Department of Homeland Security under the grant FA8750-08-2-0091.
[⋆⋆] Research supported by the Check Point Institute for Information Security. Part of this work was done when visiting IBM T.J. Watson Research Center.

Analysis in the UC framework proceeds by defining an idealized specification model and an implementation, and then proving that the implementation realizes the idealized specification. Our main contribution is a very simple filesystem specification model, called SimpFS, that captures many integrity concerns in contemporary filesystems, together with an implementation over existing POSIX filesystems [20] and a proof that the implementation realizes the specification model.

> *The composability properties of our analysis imply that software systems that use our implementation over POSIX behave essentially the same as if they were using the simple, idealized specification system* SimpFS.

This is a very strong security guarantee. In particular, it allows analyzing software systems without worrying about how the filesystem is implemented, and without worrying about potential bad interactions between the analyzed system and the filesystem implementation.

Our filesystem model is geared toward ensuring integrity of files and their names, and in particular preventing filename manipulation attacks. In such attacks, a victim program expects a particular filename to have certain semantics. (E.g., a mail program may expect the file `/var/mail/root` to be the mail file for the super-user.) In the attack, the adversary creates a link by the same name in the filesystem, pointing to another file (e.g., `/var/mail/root -> /etc/passwd`), thereby "tricking" the victim program into accessing an unexpected file. (In the mail example, such a link may cause a naive mail program to write incoming email into the system's password file.) Such attacks were quite common in UNIX systems of old. Since creating links to files often takes lower permissions than accessing these files, this form of attack sometimes allows an attacker to leverage the permissions of a privileged victim program to read or write files that the attacker cannot access on its own.

Our implementation builds on the ideas presented in Chari et al. [7], who address the problem of privilege escalation attacks via filename manipulation. To counter these attacks, Chari et al. present a "safe" name resolution procedure, and deploy this system-wide on popular POSIX systems. The SimpFS interfaces are designed to tightly bind files with their names: files can be accessed *only* via the names they were created with, which means that filename manipulation attacks are impossible in our model. Our proof — showing that implementation based on [7] realizes the model — implies in particular that it indeed eliminates these filename manipulation attacks.

SimpFS offers a simple interface that captures enough filesystem primitives for application developers to build meaningful applications. The simplicity of SimpFS is due to its very narrow interface (only four commands) and the fact that *it does not have directories.* We argue that the murky relation between files and their names in plain POSIX systems stem to a large extent from the fact that pathnames consist of many directories, each with its own permissions, which are combined in a non-obvious manner to yield the effective permissions for the entire name. In contrast, a filename in SimpFS is just a single entity with explicitly specified permissions. Thus SimpFS provide applications with radically simplified semantics, making it easier to use the filesystem without falling into traps. At the same time, we argue that the vast majority of contemporary applications in POSIX systems *do not really need directories*, and can be implemented over the simple SimpFS interface without loss of functionality.

1.1 Related Work

Triggered by Joshi and Holzmann's mini-challenge [13], there is a lot of recent work on formalization and verifications of file systems. Most notably, Freitas et al [10] specify and prove a POSIX file store in Z/Eves. This body of work focuses mostly on the correctness aspects and does not address in depth the security and access control aspects of filesystems. In the broader perspective of (secure) operating systems, there is a long history of formalization and verification, from PSOS [16] to the recent seL4 [14]. While they make considerable progress toward high-assurance OS, these works are not based on frameworks that allow easy composition of components to form larger systems. Additionally, the focus in many of these works is on mandatory access control whereas we cover a discretionary control. (We stress that although our model addresses integrity concerns, these are very different from the Biba integrity model [3].)

An abstract model of another large standard systems, the browser, suitable for proofs of cryptographic protocols exists in [12]. It includes a model of information-flow properties under attack. However, the federated identity protocols built on top of it have only been proven secure with respect to specific security properties, not in a real-world / ideal-world setting [12].

Protocol Composition Logic (PCL) [8] is a comparable general approach on reasoning about (cryptographic) network protocols in a composable fashion. Recently, PCL was applied to analyze systems [9], more specifically integrity properties provided by TPM. The symbolic and axiomatic nature of PCL leads to a more axiomatic specification of security rather than the declarative form in UC. Furthermore, the composition theorems in PCL are weaker than in the UC framework.

A noteworthy contribution to secure composition of large systems is the CHATS project [17], that identifies architectural principles to guide the structuring and decomposition of trustworthy systems. That work is largely orthogonal to ours, as it does not focus on formal modeling or proofs.

There have been many more attempts to leverage well-established formalisms such as logic, typing or process calculi to model composability of certain system security properties, e.g., McLean [15] for non-interference properties or Bengtson et al [2] for cryptographic protocols and access control mechanisms. Many of them provide tool support; but they do not provide the same composition guarantees as in the UC framework.

2 The Universal Composability Framework

We briefly describe the relevant aspects of the framework of universally composable (UC) security. The reader is referred to [4] for more details. The framework describes two probabilistic games: The *real world* that captures the protocol flows and the capabilities of an attacker, and the *ideal world* that captures what we think of as a secure system. The notion of security asserts that these two worlds are essentially equivalent.

The real-world model. The players in the real-world model are all the entities of interest in the system (e.g., the nodes in a network, the processes in a software system, etc.), as well as *the adversary* A and *the environment* $\mathcal{Z}$. All these players are modeled as efficient, probabilistic, message-driven programs (formally, they are all interactive Turing machines).

The actions in this game should capture all the interfaces that the various participants can utilize in an actual deployment of this component in the real world. In particular, the capabilities of A should capture all the interfaces that a real-life attacker can utilize in an attack on the system. (For example, A can typically see and modify network traffic.) The environment $\mathcal{Z}$ is responsible for providing all the inputs to the players and getting all the outputs back from them. Also, $\mathcal{Z}$ is in general allowed to communicate with the adversary A. (This captures potential interactions where higher-level protocols may leak things to the adversary, etc.)

The ideal-world model. Security in the UC framework is specified via an "ideal functionality" (usually denoted $\mathcal{F}$), which is thought of as a piece of code to be run by a completely trusted entity in the ideal world. The specification of $\mathcal{F}$ codifies the security properties of the component at hand. Formally, the ideal-world model has the same environment as the real-world model, but we pretend that there is a completely trusted party (called "the functionality"), which is performing all the tasks that are required of the protocol. In the ideal world, participants just give their inputs to the functionality $\mathcal{F}$, which produces the correct outputs (based on the specification) and hands them back to the participants. $\mathcal{F}$ may interact with an adversary, but only to the extent that the intended security allows. (E.g., it can "leak" to the adversary things that should be publicly available, such as public keys.) Specifying the code of $\mathcal{F}$ is typically a non-trivial task. It is important that $\mathcal{F}$ satisfies all the desired security properties, but also that $\mathcal{F}$ does not impose unnecessary constraints: It is only too easy to write a functionality that describes "what we intuitively want", but is not realizable by any implementation.[3] Another crucial concern is the *simplicity* of the functionality $\mathcal{F}$, since we want $\mathcal{F}$ to capture the important security concerns, not the mundane implementation details.

[3] For example, to realize an abstract time-synchronization functionality that always returns the exact time, one needs to devise a protocol for perfect clock synchronization, which is impossible to achieve in our physical world due to the Heisenberg uncertainty principle.

UC-Security and the Composition Theorem. An implementation π **securely realizes** an ideal functionality $\mathcal{F}$ if no external environment can distinguish between running the protocol π in the real world and interacting with the trusted entity running the ideal functionality $\mathcal{F}$ in the ideal world. That is, for every adversary A in the real world, there should exist an adversary A' in the ideal world, such that no environment $\mathcal{Z}$ can distinguish between interacting with A and π in the real world and interacting with A' and $\mathcal{F}$ in the ideal world.

The striking feature of the UC framework is its ability to handle composition. Specifically, the composition theorem from [4] asserts the following: Let ρ be an arbitrary system that runs in the ideal world and uses (perhaps multiple copies of) the functionality $\mathcal{F}$. Next, consider the system ρ' in the real world, that is the same as ρ except that in ρ' each call to the ideal functionality $\mathcal{F}$ is replaced by executing the implementation π. Then, if π securely realizes $\mathcal{F}$ it is guaranteed that system ρ' behaves essentially the same as system ρ. In particular, all the security properties of ρ are inherited by protocol ρ'. This guarantee is the basis for the composable security guarantees provided by the UC framework.

2.1 Conventions for Software systems

We briefly describe some technicalities that must be resolved when attempting to apply the UC framework to software system, and the conventions that we use to address them. The "entities of interest" in our work are processes, which differ somewhat from the interactive Turing machines (ITMs) in common cryptographic models. One aspect relates to side-channels: whereas an ITM can only influence other ITMs by sending messages, a process shares some physical resources with other processes on the same machine, so it could influence them via side channels such as timing and concurrency. In this work we ignore that aspect, i.e. we do not have any side channels in our formal model. (This does not matter for our current SimpFS model, since we do not model any secrecy requirements.) We thus just let the adversary learn "whatever it needs," so it has no use for side channels.

A more important difference is preemptive multitasking: common crypto models postulate a sequential scheduling model, where an active ITM keeps the control until it sends a message, at which point the recipient becomes active. On the other hand, processes in contemporary OSes can be made to yield control involuntarily. Resolving this discrepancy is not as hard as it may seem, since (side-channels aside) an active entity has no effect on its surroundings until it sends a message, which means that influencing the surroundings only comes with losing the control. We use the standard sequential scheduling of the UC framework, but ensure that the adversary gets the control after every message is sent, and can decide when this message will be delivered. (This is somewhat similar to the "buffer scheduler" from [1].) Hence the adversary in our formal model is able to simulate the actions that would have happened in the actual deployed system, delay delivery messages until the simulation arrives at the point where they were delivered. We thus argue that the formal adversary in our model is able to induce any behavior that can happen in the actual deployed system.

Another difference is that some processing in real systems is done not by the processes themselves, but by the kernel on their behalf. Hence also in our model we postulate the existence of a "kernel component" that can do things on behalf of processes. In our filesystem example, this kernel component is only responsible for maintaining the process privileges: Whenever a process calls a filesystem function, the kernel adds the process-id and roles of the calling process to the list of arguments, and forwards everything to the filesystem. (The kernel component gets these roles from the environment.) We note that although we do not use it in our filesystem example, in general we could have several such "kernel components" in a system, representing several physical machines.

3 SimpFS: A Simple Idealized File-System

This section describes SimpFS, our simple filesystem model. SimpFS has a minimalistic interface with simple semantics, having only basic primitives to create, read, write and delete files. Still, we believe that the this file-system functionality is sufficient for most applications. (Other aspects — such as locking — can be implemented on top of our interface.) The SimpFS model includes file write permissions, hence capturing properties of *filesystem integrity*. We currently do not model read permissions, but we expect that this work can be extended to include read permissions without too much change.

An important feature of SimpFS is that *it does not have any directories, only files and their names.* As we mention in the introduction, we believe that directories have "inherently cumbersome semantics", hence decided to do away with them in order to keep the semantics as simple as possible. We stress that the model supports names that include '`/`' (so applications can still store their temporary data in files with names that begin with "`/tmp/`"). But a name such as "`/a/b/foo`" is viewed as just one entity, and its existence does not imply the existence of an object with name "`/a/b`." Of course, our *implementation* over POSIX still interprets '`/`' as a directory separator, and name creation induces the right associations between names and paths, in spite of symlinks, adversarial write permissions etc. While directories are a useful and convenient way to manage and organize systems, we argue that directory permissions are very rarely needed in applications (if ever), and most applications can therefore directly use the SimpFS interface.

A key security property of SimpFS is that it rules out filename manipulation attacks. Our focus on this property is motivated by the large number of privilege escalation attacks due to unsafe pathname resolution that were discovered in POSIX systems over the years. A classical example of this type of attacks is local mail delivery, where `/var/mail` may be world-writable, allowing an adversary to create a link from `/var/mail/root` to (say) `/etc/passwd`, thereby "tricking" a naive mail-delivery program (running as `root`) to write the content of incoming mail into `/etc/passwd`. Such attacks arise due to the opaque mapping of names to files in POSIX. SimpFS features a very tight binding between files and their names: a file can be manipulated *only* with the names it was created with.

We describe an implementation of SimpFS over contemporary POSIX filesystems and *rigorously prove* that this implementation realizes SimpFS, using the UC framework. The proof implies that processes that use our implementation will be protected against pathname manipulation attacks such as above even if adversarial processes use the same POSIX filesystem in arbitrary ways.

3.1 A formal model of SimpFS

SimpFS consists of files and their names. A newly created file is given some names, and thereafter the file can be accessed by any of these names. Existing names can be deleted, but one cannot add names to existing files. When deleting names, a file can end up with zero names, in which case it is not reachable anymore so we can consider it as deleted. We associate permissions with both the file names and the files themselves:

- Every file has a list of roles that can write in it, called the *Writers* list. A process can write to a file if it holds a role in the Writers list of the file.
- File names have a set of *Manipulators*, listing all the roles that have permission to delete that name.

In the current version we do not have read permissions, which means that SimpFS allows every process to read every file.

In more details, our ideal SimpFS maintains an array of files and an associative array of names: `files[]` is an array of files (indexed by integers). Each entry is a file, consisting of an array of bytes (i.e., a data blob) and a list of roles (specifying the Writers of this file). `names[]` is an associative array (indexed by strings). We refer to the index of an entry as a file-name, and each entry consists of a pointer to a file (i.e., an integer) and a list of roles (specifying the Manipulators of this name). The interface below constrains the Manipulator lists, making sure that all the names of the same file have the same set of Manipulators. (This choice is not very important, it is done mostly to simplify the presentation.)

In the initial state, the file-system is empty, with no files and no names (i.e., both arrays are empty). There are only four operations that are supported in SimpFS: `CreateFile` creates a new file with some names, `DeleteName` deletes an existing name, `Read` reads data from a file (specified by some name), and `Write` writes data to a file (specified by some name).

The semantics of these operations is described by the pseudo-code in Figure 1. As is the case with every formal UC functionality, the pseudo-code includes not only the intended functionality as seen by the

```
CreateFile(Writers, Manipulators, Names, pid, Roles)
{
  // Allow the adversary to fail the operation and decide the error code
  var retCode = AdversaryAction("CreateFile",Writers,Manipulators,Names,pid,Roles);
  if (retCode != OKAY) return retCode;

  var codes[] = empty;    // a local list of return codes, one per name
  var f = index of next available entry in the files[] array;
  files[f].data=empty, files[f].Writers=Writers;

  // Allow the adversary to decide whether to create each name
  for each fName in Names {
    var code = AdversaryAction("CreateOneName", fName);
    if (code!=OKAY) codes[i]=code;
    else {
      if (names[fName] already exists) codes[i] = FILE_EXISTS;
      else {
        names[fName].file=f, names[fName].Manipulators=Manipulators;
        codes[i]=OKAY;
  } } }
  call AdversaryAction("Done CreateFile") and then return codes;
}

DeleteName(fName, pid, Roles)
{
  // Allow the adversary to fail the operation and decide the error code
  var retCode = AdversaryAction("DeleteName",fName,pid,Roles);
  if (retCode != OKAY) return retCode;

  if (names[fName] does not exist) return FILE_DOESNT_EXIST;
  if (Roles intersect names[fName].Manipulators = emptyset) return NO_PERMISSION;

  delete names[fName]; // Note: no point deleting the file, even if not reachable
  call AdversaryAction("Done DeleteName") and then return OKAY;
}

Write(fName, atAddr, data, pid, Roles)
{
  // Allow the adversary to fail the operation
  var retCode = AdversaryAction("OpenWrite",fName,pid,Roles);
  if (retCode != OKAY) return retCode;

  if (names[fName] does not exist)  return FILE_DOESNT_EXIST;
  var f = names[fName].file;        // f serves as a "handle" to the file
  if (Roles intersect files[f].Writers = emptyset) return NO_PERMISSION;

  var numBytes = AdversaryAction("Write",fName,atAddr,data,pid,Roles);
  if (numBytes < length(data)) truncate data to numBytes bytes;   // only partial write

  var nBytes = length(data);
  if (atAddr < 0) atAddr = length(files[f].data);    // append
  else if (atAddr > length(files[f].data)) {
    prepend (atAddr-length(files[f].data)) zero bytes to data;
    atAddr = length(files[f].data);
  }
  write data to files[f].data starting at position atAddr;
  call AdversaryAction("Done Write") and then return [OKAY,nBytes];
}

Read(fName, fromAddr, nBytes, pid, Roles)
{
  // Allow the adversary to fail the operation or read less bytes
  var [retCode,numBytes] = AdversaryAction("Read",fName,fromAddr,nBytes,pid,Roles);
  if (retCode != OKAY)  return retCode;
  if (numBytes < nBytes) nBytes = numBytes;

  if (names[fName] does not exist) return FILE_DOESNT_EXIST;
  var f = names[fName].file;

  if (fromAddr < 0) fromAddr = 0;
  else if (fromAddr > length(files[f].data)) {
    fromAddr = length(files[f].data);
    nBytes = 0;
  }
  if (nBytes < 0)   // read to end-of-file
    nBytes = length(files[f].data) - fromAddr;

  data = content of files[f].data from fromAddr for nBytes;

  call AdversaryAction("Done Read") and then return [OKAY,nBytes,data];
}
```

Fig. 1. The SimpFS commands.

legitimate users of the system, but also all the interfaces that an adversary can utilize to attack it. This is codified by an `AdversaryAction` call, in which SimpFS "leaks" to the adversary the details of its operation, and also lets the adversary influence these operations.

A key feature of SimpFS is that a file can be accessed *only* using one of the names that were specified when the file was created, thus eliminating filename-manipulation attacks such as described above. Hence proving that an implementation realizes SimpFS implies in particular that such attacks cannot be successfully mounted against the implementation.

We make no liveness guarantees in SimpFS, so at the beginning of every operation the adversary is given the option to abort the operation and determine the error code. (This does not mean that an implementation of SimpFS cannot ensure some liveness properties, but it means that a proof that an implementation realizes SimpFS carries no such guarantees within itself.)

The pseudo-code includes with every call also the process-id and permissions (Roles) of the caller, which in our system model are filled by the kernel component, cf. Figure 2. (Formally there is also an implicit "invocation id" for each call of one of the four main operations, allowing SimpFS to handle messages received from the ideal-world adversary for different invocations.) Note also that the `AdversaryAction` at the beginning and end of every operation comply with our convention that the adversary gets the control before any message is delivered. Finally, we note that all the variables in the code in Figure 1 are local to that invocation, except for the global `files[]` and `names[]`.

Process corruption. Following the standard conventions of the UC framework, SimpFS has a special procedure to handle the case where the adversary corrupts a process. For our purposes it is more convenient to let the environment decide when a process is corrupted (as opposed to the adversary, which is the more common convention in UC-model works). When the environment corrupts a process, this process makes a call `IamCorrupted(pid,Roles)`, to inform SimpFS that "it belongs to the adversary" now. SimpFS informs the adversary of this call, and it remembers that this process and all its roles are now bad. Thereafter, the adversary is allowed to make all the usual calls to SimpFS (`CreateFile`,`DeleteName`,`Read`, `Write`) on behalf of that process. SimpFS will process these calls just as if it was the corrupted process that made the call, but will return the result to the adversary rather than to the environment.

Every call from the corrupted process (not via the adversary) will be routed directly to the adversary, and the adversary can always instruct SimpFS to send anything to the corrupted process (which will then be forwarded to the environment). Also, if the roles of the corrupted players change then the kernel component will notify SimpFS of this change. SimpFS will add any new role that a corrupted process acquires to its list of bad roles, but *it will not remove any roles from that list*, even if the corrupted process loses some of its roles. (This last aspect represents the fact that the corrupted process may already have used this role to introduce artifacts into the filesystem, that will remain even after the process no longer has this role.)

Atomicity of the SimpFS operations. The operations `DeleteName` and `Read` are atomic, whereas `CreateFile` and `Write` are not: In `DeleteName` and `Read`, once the adversary allows the operation to go through (by returning `OKAY`), SimpFS holds onto the control-flow throughout the name lookup and the operation itself, and only then it yields control back to the adversary.

In `Write`, on the other hand, the control is returned to the adversary after the file lookup (via the call `AdversaryAction("Write", ...)`), and only then is the operation carried out. Similarly in `CreateFile`, the adversary gets the control before the creation of any name. This choice was made so that we would be able to realize SimpFS over the POSIX interface that requires to open the file and then write in it. The real-world read can be made atomic by checking after the fact that the file did not change since it was opened, but for write such a check is meaningless since the file was already written. (See also the attack in Section 4.5 for another reason for the check after read.)

Mapping Unix permissions to roles. The interfaces of SimpFS above are defined with "generic roles" that encode permissions, with access control being a simple role inclusion. Our implementation over POSIX, of course, uses userids and groups, which are particular types of roles. The mapping is quite straightforward, roughly there is a different role for each userid and group in the system, and a process gets the role corresponding to its effective-uid and all the roles corresponding to its groups. There is also one role for "others", that every process has. Some care must be taken since POSIX permissions do not exactly follow role inclusion. (For example, if a file is not owner-readable then the owner cannot read it, even if the file is readable by "others".) Adjusting the mapping to this technicality is quite straightforward, and is omitted here.

4 Implementing SimpFS over POSIX

We describe simpfs, which is a concrete implementation of the SimpFS functionality over the POSIX filesystem interface [20]. The presentation below focuses on a user-space implementation, where each simpfs operation runs with the effective uid of its caller, but we point out that the same procedures can also be implemented in the kernel. (See Figures 2 and 3 for illustrations of the system model in both cases.)

Our implementation relies on the "safe pathname resolution" procedure of Chari et al. [7], that protects processes from opening adversarial links. While resolving paths this procedure ensures that an adversary can not manipulate the resolution to result in opening unintended components. In simpfs, very roughly speaking, each operation consists of first using that procedure to open the corresponding file and then performing the actual operation.

Before describing this implementation, we first introduce concepts that are used in the rest of the paper and describe some assumptions that we make on the POSIX filesystems underlying our implementation. Then in Section 4.2 we describe the safeDirOpen procedure, which is the heart of our implementation and builds on [7], and then in Section 4.3 we describe the rest of the implementation.

4.1 Concepts and Properties of POSIX

We assume that the reader is familiar with basic concepts of POSIX such as directories, pathnames, users and groups, hardlinks and symlinks, etc.

Definition 1 (Pathname Manipulators). *Let `/dir1/.../dirn/foo` be an absolute pathname. The* manipulators *of this pathname are all the roles (users and groups) that own, or have write permissions in, any directory visited during the resolution of this pathname.*

Note that the definition applies even when a pathname does not resolve, and that `root` is a manipulator of every pathname.

Definition 2 (Safe Names). *A pathname is* system safe *if its only manipulator is `root`. A pathname is* safe for U *(where U is a user-id) if its only manipulators are `root` and U. Otherwise, the pathname is unsafe for U.*

For example, in a typical UNIX system the pathname `/etc/passwd` is system safe, the pathname `/home/joe/mbox` is safe for user `joe`, and the pathname `/var/spool/mail/jane` is unsafe for everyone (as `/var/spool/mail` may be world- or group-writable).

Definition 3 (Simple Pathnames). *A pathname is* simple *if it is an absolute path that resolves to a regular file, its elements are only hard links (i.e., not symbolic links), no elements are named '`.`' or '`..`', and the pathname contains no repeated slashes '`//`'.*

Assumptions. We now list some properties that we assume on the underlying POSIX system, and use in our proof of security. Most of these assumptions are justified either by the fact that they are part of the POSIX specification itself, or by the fact that many contemporary POSIX filesystems seem to satisfy them.

Assumption 1 *The underlying filesystem does not contain multiple mount points to the same filesystem, and each directory has only one parent (i.e., one hard link with a name other than '`.`' or '`..`').*

Justification. Assumption 1 is justified by the fact that nearly all contemporary POSIX implementations either do not allow processes to create additional hard links to directories (e.g., FreeBSD, Linux) or restrict this operation to the super-user (e.g., Solaris, HP-UX). A notable exception is MacOS.

We observe that given Assumption 1, for every reachable hard link to a regular file there is a unique simple name that ends with that hard link. Moreover a resolution of any absolute name that ends with that hard link will visit all the directories in this unique simple pathname.

Assumption 2 (Permissions) *1. If an operation by a process affects the content of a file, then the process must have write permission for that file. 2. Let P be an absolute pathname. If an operation by a process affects the resolution of P or changes the permissions or ownership of any of the directories visited during its resolution, then that process must have a role which is a manipulator of P.*

Justification. The only operations that affect pathname resolution are creating, removing, or renaming pathname components, and they all require write permission in the containing directory. Also, note that only the owner of a directory (or `root`) can change the permissions of that directory, and in most systems only `root` can change ownership.

Corollary 1. *Let P be some pathname, denote by $\mathcal{M}(P)$ the set of manipulators for P (user-ids and groups), and let $\mathcal{B}$ be a set of roles such that $\mathcal{M}(P) \cap \mathcal{B} \neq \emptyset$. Then changing the manipulator set for P so that $\mathcal{M}(P) \cap \mathcal{B} = \emptyset$ requires an operation by a process with some role outside of $\mathcal{B}$.*

Proof. The only operations that change the manipulator-set of a pathname are changing the permissions or ownership of some visited directory, or moving, renaming, or removing some visited directory, symlink, or the last hardlink.

Denote by op the first system call after which the manipulator-set of of P is disjoint from $\mathcal{B}$. Denote by $\mathcal{M}'(P)$, $\mathcal{M}''(P)$ the manipulator set of P just before and just after the system call op, respectively, so $\mathcal{M}'(P) \cap \mathcal{B} \neq \mathcal{M}''(P) \cap \mathcal{B} = \emptyset$. Since op changes the manipulator set of P, it must have succeeded, hence the calling process must have had some role R^* with sufficient privileges for performing op.

Assume toward contradiction that the calling process has only roles in $\mathcal{B}$, and thus $R^* \in \mathcal{B}$. Since R^* has sufficient privileges for one of the manipulator-changing operations then by Assumption 2 $R^* \in \mathcal{M}'(P)$. We now have three cases: either op is `chown` (so R^* is `root` hence it remains a manipulator), or op is `chmod` (so R^* is the owner of the directory so it remains the owner), or op is any other manipulator-changing operation so R^* is a writer in the containing directory and it remains so after the operation. In each case R^* remains a manipulator, $R^* \in \mathcal{M}''(P) \cap \mathcal{B}$, hence $\mathcal{M}''(P) \cap \mathcal{B} \neq \emptyset$.

Assumption 3 *The hardlink to a directory in its parent directory can only be removed when the child directory is empty. Moreover, after the hardlink is removed from the parent directory, no further entries can be created in the child directory, even if some process still holds a handle to it.*

Justification. The last part of Assumption 3 is justified by the fact that `rmdir` implementations remove the entries '`.`' and '`..`' from the child directory before removing the hard link in the parent directory, and no new entries can be created in directories without '`.`' and '`..`' .

Corollary 2. *If a system call for creating an entry in a directory returns successfully, then the hard link for this directory in its parent directory could not have been removed before that system call, or removed after the call but before the newly-created entry is removed.*

4.2 The safeDirOpen procedure

Underlying our simpfs implementation is a procedure for *safe name resolution*, which is adapted from the work of Chari et al. [7]. Our safeDirOpen procedure takes an absolute pathname, resolves it "in a safe manner" and returns a handle to the directory containing the final hard link to the actual file, the name of that hard link, and additional information as discussed below. The top-level operations of simpfs first call safeDirOpen and then perform the requested operation on the final hard link.

safeDirOpen resolves a pathname one atom at a time, each time opening the next atom (or reading it, if it is a symlink), while keeping track of the owners and writers of the visited directories. (Below we identify the time that a directory was visited as the time when it was opened, and the time that a symlink was visited with the time that it was read.)

The procedure can be in one of three states: system-safe, safe-for-uid, or unsafe. When invoked (by a process with effective uid U), the procedure begins in a system-safe state, switching to safe-for-uid state upon visiting a directory where U is an owner or writer, and switching to unsafe state upon visiting a directory with any writer or owner other than `root` or U. Once in unsafe state it stays in that state for the duration of the current name resolution. Likewise, there is no transition from safe-for-uid to the system-safe state.

When safeDirOpen enters the unsafe state, it does not follow symlinks for the remainder of the current name resolution. Also, for technical reasons the procedure never accepts pathnames that contain multiple slashes '`//`' or have components named '`.`' or '`..`', and it refuses to visit any directory whose name begins with the special prefix `_SimpFS_ephemeral_`. In any of these cases, the procedure returns an error code.

Once safeDirOpen arrives at the final atom (and verifies that it is indeed the final atom and not a symlink), it ends successfully, returning a handle to the directory containing this last hard link, as well as the name of the hard link. In addition, safeDirOpen returns its current state (system-safe, safe-for-uid, or unsafe), the set of owners and writers of the directories that it visited, and an array of (handle,name) pairs, containing handles to all visited directories, and the names that were looked-up in those directories. (These names could belong to either a directory, a symlink, or the final hard link.)

Upon failure, safeDirOpen returns an error code, a handle to the last directory pathname component that was successfully resolved, the state (system-safe, etc.) and manipulators of that directory, and the unresolved remainder of the pathname. For example, when called to resolve `/a/b/c`, if it encountered an error after visiting `/a` but before visiting `/a/b`, then it will return a handle to directory `/a`, the state and manipulators of `/a`, and the remainder of the pathname argument "`b/c`". (Note that this will be the return value even if `/a/b` happens to be a symlink and the procedure visited more directories after `/a`, but could not completely resolve `/a/b`.)

4.3 Implementing the simpfs commands

createFile(`Writers,Manipulators,Names`). When called by a process with effective-uid U, the procedure begins by checking that U belongs to the set of manipulators specified by the `Manipulators` parameter. Then it creates a new file with an ephemeral name that begins with the special prefix `_SimpFS_ephemeral_`. This ephemeral name is created so that it is safe for U, thus ensuring that no other users can remove or rename it.[4]

Now createFile attempts to set the write permissions of the new file as specified in the `Writers` parameter. If this is successful, it proceeds to create the names, one at a time, by calling the subroutine createOneName for each name in `Names`. After all the calls to createOneName, the procedure createFile removes the ephemeral name that it created for the new file, and returns the vector of return codes that it received from all the calls to createOneName.

The subroutine createOneName(`fName`) begins by checking that the new name is an absolute name, and that it does not contain '`//`' or elements named '`.`' or '`..`', or elements that begin with `_SimpFS_ephemeral_`. Then it calls safeDirOpen(`fName`) thus obtaining a handle to the last successfully resolved directory on this pathname and the corresponding set of manipulators. If all the directories were resolved successfully, then createOneName checks that the set of manipulators equals the `Manipulators` parameter, and aborts if they differ.

If some directories were not resolved, createOneName verifies that the manipulator set of the prefix is not too large (i.e., it must be contained in the `Manipulators` parameter), aborting otherwise. Then createOneName attempts to create the remaining directories, one at the time, initially creating each one so that it is only writable by owner U with an ephemeral name that begins with `_SimpFS_ephemeral_`. Upon success, it tries to set the write permissions of the last directory so that the resulting set of manipulators will match the `Manipulators` parameter. Then it goes over all the newly created directories, top to bottom, renaming each one to the name that it is supposed to have according to `fName`.

Once all the directories exist and have the right set of manipulators and the right names, the procedure createOneName makes a `linkat` system call to create a hard link in the last directory, pointing to the new file. createOneName then returns whatever code was returned from the `linkat` system call.

[4] See Section 4.5 for a short discussion of this point.

If any operation fails, then createOneName attempts to clean-up after itself, trying to remove all the directories that still have names that begin with `_SimpFS_ephemeral_`. However, after a directory was renamed to its "permanent name", createOneName does not remove it.

In the proof of security in Section 5 we rely on the following properties of our implementation of createFile:

– The initial ephemeral name for the new file is safe for the effective-uid of the calling process.
– The procedure never creates symlinks, only directories and hard links.
– The procedure only changes permissions and/or removes pathname components if these components begin with the special prefix `_SimpFS_ephemeral_`.
– A name `fName` is created if and only if the `linkat` system call at the end of the subroutine createOneName(`fName`) is successful.

deleteName(`fName`). When called with effective-uid U, deleteName calls safeDirOpen(`fName`) and aborts if that function fails. Else deleteName has an array of pairs (handle,name), and the state with which safeDirOpen arrived at the final directory (system-safe, safe-for-uid, or unsafe). If the state is not system-safe, then deleteName checks that the final directory is either world-writable, or owner-writable and owned by U, and it aborts otherwise.[5] Also, if the state is unsafe then deleteName checks that the file that the hard link points to has only a single hard link, aborting otherwise.

Then deleteName attempts to delete the final hard link, followed by attempts to delete the directories higher-up on the path. deleteName returns when any system call to remove a name fails, or when any of these names resolves to a symlink, or when it is done deleting all the names in the array. The return code from deleteName is whatever was returned from the first `unlink` system call (i.e., the one that deleted the hard link at the end of `fName`).

We note that barring a race condition, this implementation of deleteName does not delete symlinks. In the proof in Section 5 we show that the only cases where these race conditions are possible are when the adversary already has permissions to delete these symlinks by itself.

read(`fName`,...). When called with effective-uid U, read calls safeDirOpen(`fName`) to get a handle for the final directory, the name of the hard link pointing to the actual file, and the state at which it arrived in this last directory: system-safe, safe-for-uid, or unsafe. Then read uses `openat`, `lstatat` and `fstat` to open the file and verify that it is still the same file (and not a symlink). In addition, if the state is not system-safe, then read checks that the file is either world-readable, or owner-readable and owned by U, and it aborts otherwise. Also, if the state is unsafe then read checks that the file has only a single hard link, aborting otherwise.

Then the procedure uses the `read` system call to read the file, and before closing the file it makes yet another `lstatat` system call to check that the hard link still points to the same inode as it did when it was opened. (See Section 4.5 for the reason for this last test.) If all these checks pass, then read returns the result from the `read` system call.

write(`fName`,...). The procedure write is almost identical to read except that it adds a write-permission check on the actual file, and it does not do the final check after writing to verify that the hard link still points to the same inode. (Indeed, such check is useless since the file was already written to.)

4.4 Consistency properties of the implementation

In the proof of security in Section 5, it is important to consider what changes may happen in the filesystem between the time that the safeDirOpen pathname resolver visits some directory and the time that the procedure that called safeDirOpen returns. An important technical observation is that if the procedure that called safeDirOpen was successful then none of those visited directories could have been removed during this time.

Lemma 4. *Consider an execution of one of the procedures createOneName, deleteName, read, or write on argument fName, and assume that the procedure succeeds (i.e., does not return an error code). Assume further that no symlink that was read during name resolution was later deleted or renamed during the execution of*

[5] This check is intended to protect against privilege-escalation attacks on `setgid` programs, cf. Section 4.5.

this procedure, and no directory was renamed after it was `open`*ed by this procedure. Then also none of these directories was deleted after it was* `open`*ed and before the time that the procedure issued the system call (respectively,* `linkat`*,* `unlinkat` *or* `openat`*) for the final hard link in* fName*.*

Moreover, for the procedures createOneName*,* read*, and* write*, as long as no symlinks are deleted or renamed, no directories are renamed, and the final hard link in* fName *exists in its original containing directory, then also none of these directories is deleted even after the operation returns.*

Proof. Assume not, and consider the first directory that was deleted after it was `open`ed. There are two cases to consider: this directory was deleted either before or after name resolution visited the next pathname component (i.e., symlink read, directory or file `open`ed).

By Assumption 3, the directory could not have been deleted before the next component was accessed, else the subsequent access would have failed. But it also could not have been deleted after the next pathname component was visited, since the directory must have been non-empty: If the next pathname component is a symlink then this follows from our assumption that symlinks were not removed or renamed, if it is a directory then it follows from our assumption that directories were not renamed and the fact that we consider the first directory to be removed, and if it is the final hard link then it follows from our assumption that it still exists in its containing directory.

Jumping ahead, we use Lemma 4 in the proof by noting that our SimpFS implementation never renames or removes symlinks, or renames directories, and hence no uncorrupted process will do any of these things. If in addition we know that no corrupted process has write permissions in any of the directories visited then also corrupted processes could not rename or remove symlinks or rename directories. Thus, we can apply Lemma 4 and conclude that all the directories stay put throughout the execution of createOneName, deleteName, read, or write.

4.5 Rationale and Discussion

Before proceeding to the formal proof of security, we discuss here some of the rationale for our implementation, including some specific attacks that the implementation was designed to foil.

Privilege-escalation attacks on `setgid` programs Our implementation of safeDirOpen only considers the effective-uid for the purpose of determining the safety of a directory, and thus we must consider the possibility of privilege-escalation attacks between processes with the same effective-uid. In contemporary UNIX systems, two processes with the same effective-uid can have different filesystem privileges only if one of them has a group-privilege that the other does not,[6] as would happen when one of these processes runs a `setgid` program.

To see the problem, consider two processes running with effective-uid of `joe`, one having the additional group privilege of `mail` while the other is compromised by an attacker (e.g., due to a buffer-overflow vulnerability). Ideally, we would like to argue that files which have read/write permissions for the `mail` group (but not user `joe`) are still protected against the compromised process.

Assume that the non-compromised process with `mail` group privileges needs to delete a file `/home/joe/dir/foo`. The compromised process can create a symlink `/home/joe/dir -> /var/mail`, "tricking" the other process into deleting `/var/mail/foo` (assuming that `/var/mail/` is writable by group `mail`). Embedding this attack in our formal model, we have a name `/var/mail/foo` for which `joe` is *not a manipulator*, and a good process that attempts to delete an unrelated name `/home/joe/dir/foo`, and yet by some action of a compromised process with `joe` privileges, this results in the deletion of `/var/mail/foo`.

We fix this problem by adding a check to the operations deleteName, read, and write, aborting if the name is not system-safe and group privileges are needed to perform the operation. Very roughly, this defense works because it prevents the use of group privileges after following symlinks that were created by non-`root` processes. (We note that we do not need this extra precaution in createOneName. This is because the SimpFS functionality restricts deletion of existing names, but puts no restrictions on the creation of names that do not exist.)

[6] We ignore the `fsuid` of Linux here.

An attack on open-then-read programs To understand the need for another check of the final hard link after a `read` system call in a `read` operation, we describe the following potential attack: Consider the three programs `sshd` that needs to read the file `/etc/passwd`, `passwd` that replaces the file `/etc/passwd` by a new file upon successful edit, and the MTA local delivery that needs to write into `/var/mail/root`. The `passwd` program runs with `root` privileges, because it is a `setuid`-root program, and the MTA local delivery runs with `root` privileges in order to append to the `/var/mail/root` mailbox file. Also, assume that the directory `/var/mail` is world writable and that initially `/var/mail/root` does not exist. The attack consists of the following sequence of steps:

1. The attacker creates a *hard link* `/var/mail/root`, pointing to the same file as `/etc/passwd`.
2. The attacker opens a new `ssh` connection, causing `sshd` to `open` the file `/etc/password` for read. (Note that since `/etc/passwd` is a system-safe name, the `open` will succeed even if there are multiple hard links.) At this point the attack relies on the `sshd` process to be switched out and remain inactive until Step 5 below.
3. The attacker then uses the `passwd` command to change its password, thereby causing the old `/etc/passwd` file to be replaced by a new file. (Note that the hard link `/var/mail/root` is now the only hard link still pointing to the old `/etc/passwd` file, and that the `sshd` process still holds a handle to that file.)
4. The attacker sends email to `root@localhost`, causing the MTA local delivery to append the content of that message to `/var/mail/root`.
5. The `sshd` process is now switched in again and reads from its handle to the old `/etc/passwd` file, thereby reading also the data that was written there by the MTA delivery agent.

To thwart this attack, we added the `lstatat` check between reading and closing the file, verifying that the hard link still points to the same file. We stress that it is possible to switch the link back and forth to foil this extra test, but it is sufficient for the purpose of our simpfs implementation. Non-adversarial processes will never attempt such a back-and-forth switch, and adversarial processes either do not have the privileges needed to foil the test, or else they have sufficient privileges to manipulate the file directly. (Our proof relies on this extra test in the analysis of the read operation on Page 19.)

Our treatment of symbolic links Our proof of security in the full version Section 5 relies in places on the assumption that good processes do not create symlinks. This is consistent with our simpfs implementation (that indeed does not create symlinks), but it begs the question why we allow safeDirOpen to follow symlinks at all.

The reason is that the implementation of simpfs is useful also in situations where the filesystem includes non-adversarial symlinks. A close inspection of our proof shows that the arguments remain valid also in the presence of non-adversarial symlinks, as long as the files that have non-adversarial symlinks in their names remain static (i.e., they are not deleted, removed, or moved). It is even possible to modify the semantics of SimpFS to accommodate non-adversarial symlinks in a dynamic filesystem, but the new semantics will not be as simple anymore.

Using the sticky bit Recall that the initial ephemeral name for a new file must be safe for the effective-uid of the calling process (denoted U). Such a name can perhaps be created in U's home directory, but not all uid's have one. A simple way of achieving the same result in contemporary UNIX systems is creating this ephemeral name in `/tmp`, relying on the fact that `/tmp` is owned by `root` and has the sticky bit on. This does not quite fit into our definition of "safe for U" (since `/tmp` is world-writable), but it suffices for the purpose of our proof of security. Specifically, what we need is to ensure that as long as the calling process holds a handle to the new file, only U or `root` can change the resolution of the ephemeral name.

5 Proof of Security

We next prove that our simpfs implementation realizes the SimpFS functionality over POSIX, given our assumptions from Section 4. The proof refers to a system model where simpfs is implemented in user-level code and relies on an incorruptible kernel component that handles process permissions; see Figure 2. Essentially the same proof shows that the simpfs procedures realize the SIMPFS functionality when implemented in the kernel (in which case permissions are handled by the environment, cf. Figure 3).

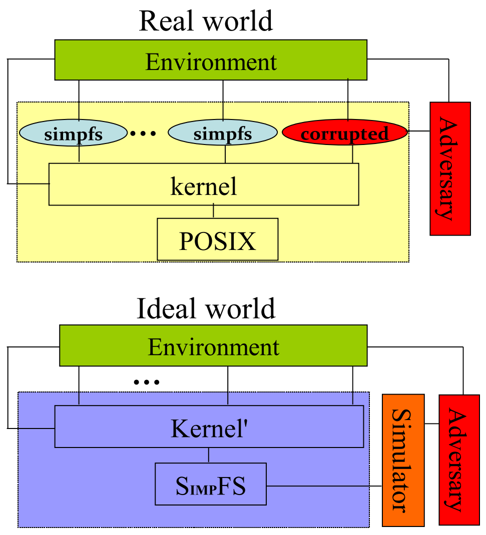

Fig. 2. The real and ideal worlds for a user-level implementation of simpfs. The kernel components that keep track of privileges are formally considered to be parts of the implementation and the ideal functionality.

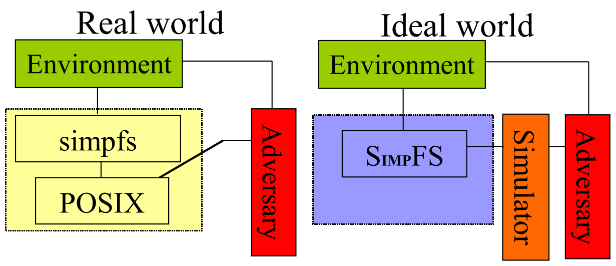

Fig. 3. The real and ideal worlds for a kernel implementation of simpfs. In this setting, process privileges are handled by the environment.

Theorem 1. *Our simpfs implementation realizes the* SIMPFS *functionality over the POSIX interface, provided that the underlying POSIX system satisfies Assumptions 1 through 3.*

To prove Theorem 1 we show that there exists an ideal-world simulator $\mathcal{S}$ such that for every real-world adversary $\mathcal{A}$, no environment $\mathcal{Z}$ can distinguish the behavior of the real world with $\mathcal{A}$ from that of the ideal world with $\mathcal{S}$ and $\mathcal{A}$. We first define a few concepts that will be important in the proof, then define the simulator $\mathcal{S}$, and finally prove the indistinguishability.

5.1 Useful Concepts

THE SIMULATED REAL WORLD. As usual, our simulator $\mathcal{S}$ interacts with the adversary $\mathcal{A}$, and it needs to simulate a complete picture of the real world as would be seen by this $\mathcal{A}$ and the environment $\mathcal{Z}$. Note that

$\mathcal{S}$ knows all the calls made by $\mathcal{A}$ to the underlying POSIX system. Also, the simulator knows the details of all the calls made by the legitimate players to the SimpFS functionality (by virtue of the `AdversaryAction` calls made by SimpFS). Hence $\mathcal{S}$ can simulate the corresponding real-world implementation for these calls, keeping a complete picture of the real-world POSIX system as it would exist in the real world at any point in time. Below we call this POSIX system that the simulator keeps the *simulated real world.*
Bad roles. Recall that the association between processes and roles (such as userid and groups) is *not one-to-one.* This raises the possibility that some roles are held by both corrupted and uncorrupted processes at the same time, and similarly a process can have both "good" and "bad" roles.[7] To handle these cases we introduce the following definition.

Definition 4 (Bad roles). *At any point in a run of the system, the set $\mathcal{B}$ of bad roles contains all the roles that were held by a corrupt process since the start of this run. The other roles are called good roles.*

Clearly, the set $\mathcal{B}$ is monotonically growing throughout the run of the system. The simulator can make calls to SimpFS using any role in $\mathcal{B}$, as per our process corruption interface.
Protected names and files. Throughout the simulation, some of the names in the simulated real world also exist in the SimpFS functionality, while the others exist for the most part only "in the simulator's head." Intuitively, the former are the *protected* names while the latter are *unprotected.* The formal notions of protected names (and also files) are defined next.

Definition 5 (Protected Names). *An absolute pathname `fName` that resolves to a regular file in the simulated real world at a given point in time is* protected *if no bad role in $\mathcal{B}$ is a manipulator for `fName`. Pathnames that resolve to regular files but are not protected are called* unprotected.

Definition 6 (Protected Files). *A file that exists in the simulated real world is* protected *if no bad role in $\mathcal{B}$ has permission to write in it. Otherwise it is* unprotected.

Unprotected names and files can exist only after some processes were corrupted. Also, a system-safe pathname is protected if and only if no `root` process was corrupted, and a pathname which is safe for U is protected if and only if no `root` or U processes were corrupted.

Note also that protected names must be created by uncorrupted processes, since no corrupted process has the permission to create them. This means that protected names can only be either the names that were specified as arguments to createFile, or the temporary names with special prefix `_SimpFS_ephemeral_` that are used inside the procedure createFile. Below we refer to the latter as *ephemeral*:

Definition 7 (Ephemeral Names). *A pathname in the simulated real world is called ephemeral if any of the pathname components begins with the prefix `_SimpFS_ephemeral_`.*

5.2 The Simulator

The simulator's strategy is to keep in the SimpFS functionality only protected names, while it simulates the unprotected names internally. When a player tries to access such an unprotected name, the simulator temporarily creates a file with that name in SimpFS by making a `CreateFile` call on behalf of a corrupted process. The simulator then allows the main operation to succeed and return an answer, and then deletes that temporary name using a `DeleteName` call on behalf of the same corrupted process.

In a few more details, when the simulator is informed by SimpFS that some process invoked an operation (`CreateFile`, `DeleteName`, `Read`, `Write`), it simulates the corresponding procedure of the simpfs implementation (including any interleaving events). For `DeleteName` and `Read` it returns OKAY if the procedure succeeds, for `CreateFile` it returns the first OKAY once the temporary file is created and set with right permissions and then returns OKAY for each name for which the `link` system call succeeded. For `Write` it

[7] For example, we could have a corrupted process with userid `jack` and group `users` and an uncorrupted process with userid `jane` and group `users`, so the role corresponding to group `users` is held by both an corrupted process and an uncorrupted one. Also the uncorrupted process holds both a "good" role (`jane`) and a "bad" role (`users`).

returns the first OKAY when the procedure `open`'s the file, and then again when the procedure successfully `write`'s to the file. In all cases, if the procedure in the simulated real world fails, the simulator returns the same error code.

For `DeleteName`, `Read`, and `Write`, before returning OKAY the simulator ensures that a file with the corresponding name exists in the SimpFS functionality. If this name is an unprotected, the simulator first creates a temporary file with this name in SimpFS and writes into it the content that it has in the simulated real world. The simulator also puts these temporary names in a list of names to-be-deleted, and deletes them from SimpFS as soon as it gets back the control. Similarly for `CreateFile`, if a successful createOneName creates an unprotected name then the simulator puts that name on its to-be-deleted list and deletes it from SimpFS once it gets back the control.

When receiving a `Done Write` call from SimpFS, the simulator goes over all the protected file names, looking for names for which the content of the corresponding file in the simulated real world differs from that in the SimpFS functionality. If the file is *unprotected* (i.e., the simulator has permissions to write in it) then the simulator makes a `Write` call to set the content of the file in the SimpFS functionality to match that of the simulated real world.

Process corruption. When the simulator learns from SimpFS that a process is corrupted, it goes over all the file names that exist in SimpFS, and deletes each name that the newly corrupted process can delete from SimpFS, using a call on behalf of that process. The simulator also remembers that this process is now corrupted.

Modifications of files with protected names. When a corrupted process modifies the content of a file that has a protected name in the simulated real world, the simulator makes a `Write` call to the SimpFS functionality on behalf of the same process, setting the content of the corresponding file inside SimpFS to match that of the simulated real world.

5.3 Proof of correctness

We show that with the simulator defined above, the view of the environment in the ideal and real worlds is identical. As we noted above, it is sufficient to argue about the simulated real world vs. the SimpFS functionality. We now prove a sequence of lemmas relating the names that exist in the simulated real world to those that exist in the SimpFS functionality.

Lemma 5. *Every protected name that exists in the simulated real world is either ephemeral or also exists in the* SimpFS *functionality.*

Proof. Recall that protected names must be created by uncorrupted processes, since corrupted processes do not have permission to write in the directories containing them. As per our implementation, the only names of regular files that are created by uncorrupted processes are either the names that are specified as parameter in createFile or ephemeral names. As to the former, they are created via a successful `link` system call in createOneName, at which point the simulator returns OKAY to the SimpFS functionality, which in turn then creates the name (if it does not already exist).

Below we say that the a particular link (hard or symbolic) that exists in the simulated real world *remains unchanged* during some time interval if it is not removed or renamed in its containing directory, and its permissions and ownership remained the same. A pathname remains unchanged if all the directories, links and filenames that are accessed during resolution of this pathname remain unchanged.

Lemma 6. *Every name `fName` that exists in the* SimpFS *functionality and no corrupted process has permission to delete it, also exists in the simulated real world and is protected. Moreover, `fName` remained unchanged since it was last created in the simulated real world.*

Proof. Fix any file name `fName` that satisfies the premise of the lemma. This cannot be temporary a name on the to-be-deleted list, since those can be deleted by corrupted processes. Thus the last time when it was created in SimpFS was after a successful `link` system call in createOneName, during a `CreateFile` call by an uncorrupted process. (Also `fName` is not ephemeral, since our implementation of createOneName does not create ephemeral names for regular files.)

Let $\mathcal{M}$ be the set of manipulators for `fName` in the SimpFS functionality, so by the premise of the lemma $\mathcal{M} \cap \mathcal{B} = \emptyset$. Also, $\mathcal{M}$ was the manipulator-set specified in the `CreateFile` call to SimpFS when `fName` was created. Recall now that the subroutine createOneName keeps track of all the owners/writers in all the directories that it visits, and only issues the final `link` system call if that set equals $\mathcal{M}$. Denote the directories visited during name resolution (in order) by $\texttt{dir}_1$,$\texttt{dir}_2$,...,$\texttt{dir}_n$ and the final filename by `foo`. Since $\mathcal{M} \cap \mathcal{B} = \emptyset$ then set of writers/owners in those directories at the time where createOneName visited them was disjoint of $\mathcal{B}$. We next show that all these directories (and also the final file) remained unchanged since createOneName visited them, thus completing the proof.

First, we claim that at the time of creation, `fName` was a simple pathname. That `fName` does not include '`.`' or '`..`' or '`//`' follows since createOneName does not create names that include any of them. Also, uncorrupted processes in our implementation never create symbolic links, so symbolic links can only be created in directories that are writable by some role in $\mathcal{B}$. This means that none of the directories $\texttt{dir}_i$ contained symbolic links when the name-resolution visited them during createOneName, so in particular all the pathname components visited (or created) by createOneName (except the final `foo`) were directories. Once these directories were visited, they were not moved (since only corrupted processes can move directories but none of them had permission to do so), hence by Lemma 4 they also not removed before the hard link `foo` was created. Hence also at the time that `foo` was created, the pathname `fName` was simple.

Next, assume toward contradiction that one of the directories $\texttt{dir}_i$ (or `foo`) was modified or erased since it was visited by createOneName, and consider the first of them that was modified or erased. By Assumption 2, the caller owned or had write permission in the parent directory at the time of the change. Since the set of owners/manipulators is disjoint of $\mathcal{B}$, it means that the process that first modified/erased that pathname component must have been uncorrupted.

In our implementation, system calls that modify permissions are used by uncorrupted processes only on ephemeral names, which `fName` is not. Therefore the first modification had to be a removal of a pathname component (by an uncorrupted process). Invoking Lemma 4 again, we know that none of the directories can be removed, thus the first pathname element to be deleted has to be the hard link `foo` itself. Note also that hard links to files are only deleted by uncorrupted processes during a successful `DeleteName` call to SimpFS.

Denote the pathname argument to the successful `DeleteName` call that deleted `foo` by `fName2`, and we argue that `fName2` must be the same as `fName`. Clearly, `fName2` cannot include '`.`' or '`..`' or '`//`' since safeDirOpen does not allow these. Also, recall that the deleteName procedure was run by a process that had permissions to delete the hard link `foo`, so it must have a different effective-uid from all the corrupted processes. Since only the adversary creates symlinks, then symlinks must reside in directories that are unsafe for the effective-uid of that process, hence safeDirOpen will not follow them. Therefore safeDirOpen encountered only hard links (to directories) as it resolved the name `fName2`.

We now argue that these directories must have been the same $\texttt{dir}_1$,$\texttt{dir}_2$,...,$\texttt{dir}_n$ as in `fName`, and moreover at the time of deletion the hard-link must have been called `foo` (as in `fName`). For `foo` itself, we already established above that the first modification to it since it was created was the time it was removed. Hence at the time of deletion it must have been called `foo` and must have resided at deletion in the same directory in which is was created.

As for the containing directories, at the time that `foo` was created none of them was writable or owned by corrupted players, which implies that none of them was writable or owned by corrupted player any any point since these directories themselves were created. (This follows from Corollary 2.) Thus these directories could not have been moved to their containing directory at `fName`, they must have been created there with ephemeral names and then renamed to their permanent name, which remained fixed at least as long as `foo` existed. By induction on the pathname components of `fName` (starting from $\texttt{dir}_n$ and going back), we therefore conclude that the deleteName procedure must have opened each $\texttt{dir}_i$ using a handle to $\texttt{dir}_{i-1}$ and the same name that $\texttt{dir}_i$ has in `fName`. Hence `fName2` and `fName` are the same.

Summing up, we had an uncorrupted player who made a successful call `DeleteName(fName)` to the SimpFS functionality. But this means that `fName` no longer exists in SimpFS, which is a contradiction.

Lemma 7. *At any point in time, two non-ephemeral protected names resolve to the same file in the simulated real world if and only if they belong to the same file in the* SimpFS *functionality.*

Proof. Fix any two non-ephemeral protected names that exist at some point in time in the simulated real world. By Lemma 5 they also exist in the SimpFS functionality. For each name, we look at the `CreateFile` call when it was *last* created in the SimpFS functionality, which was after the `link` system call returned successfully in the respective simulated createOneName subroutine.

If both createOneName subroutines were part of the same createFile procedure then they were created pointing to the same ephemeral filename, and since they are protected then also the ephemeral name was protected, which means that it was not deleted between the two `link` system calls. Hence they were created pointing to the same file. On the other hand, if the two subroutines were part of two different runs of createFile then they were created pointing to different files. By Lemma 6, the two pathnames remained unchanged since they were created. Hence, they still resolve to the same file if they were created in the same `CreateFile` call to the SimpFS functionality (and hence belong to the same file in SimpFS), and they still resolve to different files if they were created in two `CreateFile` calls (and hence belong to different files in SimpFS).

Lemma 8. *Consider a call `Write(fName,...)` from an uncorrupted process that returns `OKAY`, and consider the state of the simulated real world at the time when the `open` system call in the implementation returns a handle to the final hard link. If at that time `fName` is unprotected, but there exists a protected name that resolves to the same file, then the file itself is unprotected (i.e., there is some role in $\mathcal{B}$ with permission to write in it).*

Proof. If any `root` process is corrupted then all files and names are unprotected and we are done. Assume from now on that no `root` process is corrupted. It follows that when the last `open` system call returned, the name `fName` was not system-safe (else it would have been protected), so the `Write` procedure did not open `fName` in a system-safe mode. Let U denote the effective-uid of the calling process. The same argument as above shows that if no U process was corrupted (when the `open` system-call returned), the `Write` procedure could not have opened `fName` in a safe-for-U mode. Hence the only two cases that we need to consider are that some U process was corrupted, or that `safe-open` opened `fName` in unsafe mode.

In the former case, recall that `fName` was not opened in system-safe mode, so a `Write` could only succeed when the file is either world-writable or owned by U (and writable by owner). Either way the file is not protected (since it can be written by the corrupted U process). It is left to show that the latter case (where the file was opened in unsafe mode and no U process is corrupted) cannot happen.

Since the file had a protected name it also had a *simple* protected name, which we denote `fName2` = `/dir1/.../dirn/foo`. The hard link `foo` must also be the last hard link in `fName`, as opening a file in unsafe mode would fail if the file has multiple hard links. Finally, the resolution of `fName` could not have encountered directories unsafe for U *before* merging into the simple path `fName2`, else it would fail. But since no U or `root` process is corrupted, all these directories were still safe for U when the `open` system call returned, hence `fName` was protected, which is a contradiction.

Lemma 9. *The view of the environment is identical in the real and ideal worlds.*

Proof. We need to show that the answers that the environment sees when interacting with simpfs over POSIX and the adversary $\mathcal{A}$ are identical to what it sees from the SimpFS functionality with the simulator $\mathcal{S}$ and the same $\mathcal{A}$. Below we will argue about the simulated real world, since it is an exact replica of the real world.

From the description of the simulator, it is clear than whenever the implementation of some call returns an error code then the environment will see the same error code in the ideal world (since this is what the simulator returns to SimpFS). Also, it is clear that the results of all the calls that have *unprotected names* as arguments must be the same, since the simulator always creates the corresponding files in SimpFS on the fly to ensure this.

It is left to show that for operations that have *protected names* as argument, if they succeed in the real-world implementation then SimpFS will not return an error, and also that the content of successful `Read` operations is the same. We begin with error codes: The cases where the call to `AdversaryAction` returns `OKAY` but SimpFS returns an error are the following:

– In `CreateFile` when the filename already exists. By Lemma 6, if a protected filename exists in SimpFS then it also exists in the simulated real world, hence the `link` system call would fail and the simulator would not return `OKAY`.
– In `DeleteName/Read/Write` where the name does not exist, or the calling process does not have permission to delete the name or read/write the file.
  Recall that if a name does not exist but the operation in the real world succeeds, then the simulator creates the corresponding name with the right permissions in the SimpFS functionality before returning `OKAY`. So the only case that needs to be examined is when the name does exist (and does not have corrupted manipulators) but the calling process does not have the permissions to delete, read, or write. By Lemma 6, such names exist also in the simulated real world, and they remained unchanged since they were created. Moreover the createFile procedure ensures that the name and file have the same sets of manipulators/writers in the simulated real world as in the SimpFS functionality. Hence, if the calling process does not have permission to delete/read/write then the simulated procedure will also fail, and the simulator will not return `OKAY`.

Next we consider the content of files with protected names. By Lemma 6 this name also exists in the SimpFS functionality. We observe that the last time `fName` was created in the SimpFS functionality (prior to the successful `read` system call) could not have been between the `open` and `read` system calls, since otherwise the final `lstat` check would have failed and the `Read` would not have been successful. Hence the name (and the file) were created before the `open` system call.

We now examine the content of the file corresponding to `fName` since the last time it was created in the simulated real world. (This was when the temporary name for this file was created.) For each successful `write` system call for this file, we designate the beginning of the next successful `read` or `write` system call (for the same file) as "the point where the `write` operation ended." We prove by induction that at the time each `write` ended, the content of the file in SimpFS was identical to its content in the simulated real world.

We have two cases to consider: either the file is unprotected (i.e., one of the bad roles in $\mathcal{B}$ belongs to the `Writers` set), or it is protected. If the file is unprotected then the simulator would always make sure to adjust its content in the SimpFS functionality to whatever it would be in the simulated real world. We now claim that the last remaining case — where the file is protected but the name that was used to write in it is not — cannot happen.

If the `open` system call for the `Write` operation happened after the name `fName` was created in the simulated real world then we meet the conditions of Lemma 8, namely a successful `Write` to an unprotected name where the same file also has a protected name (the protected name is `fName`). If the `open` system call happened before the name `fName` was created then the temporary name for that file must have still existed at the time, which was itself protected, and again we meet the conditions of Lemma 8. In either case the file cannot be protected.

We have shown that the content of the file is identical at the end of every `write` operation. Since the `open` call for the `Read` happened after the file was created then the subsequent `read` system call returns the content of this file (specifically, the content after the last `write` system call), which is the same as the content that SimpFS has for that file. This completes the proof of Lemma 9 and also Theorem 1.

6 Conclusion

In this work we adapted the Universal Composability (UC) framework to the modeling of large software systems. Focusing on filesystem interfaces, we described SimpFS, which is a simple filesystem abstraction intended to capture *filesystem integrity* concerns. We describe an implementation of this abstraction over real POSIX filesystems and prove that the implementation realizes the SimpFS abstraction in the UC sense. SimpFS is a simple but useful interface and with a few small enhancements is sufficient to build real applications.

Our work demonstrates that formal security frameworks such as Universal Composability can be used also beyond the niche of cryptographic protocols. Our modeling of POSIX-based file systems is the first example of this scale.

References

1. M. Backes, B. Pfitzmann, M. Waidner. The reactive simulatability (RSIM) framework for asynchronous systems, *Inf. Comput.*, vol. 205, no. 12, 2007, pp 1685–1720.
2. J. Bengtson, K. Bhargavan, C. Fournet, A. Gordon, and S. Maffeis, Refinement types for secure implementations, 21st IEEE Computer Security Foundations Symposium (CSFS), pages 17-32, 2008.
3. K. Biba, Integrity considerations for secure computer systems, MITRE TR-3153, Bedford, MA, 1977.
4. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *FOCS*, 2001, pp 136–145.
5. R. Canetti. Security and Composition of Cryptographic Protocols, SIGACT News, vol. 37, no. 3&4, 2006.
6. R. Canetti, S. Chari, S. Halevi, B. Pfitzmann, A. Roy, M. Steiner and W. Venema. Composable Security Analysis of OS Services, Cryptology ePrint Archive, Report 2010/213, `http://eprint.iacr.org/`
7. S. Chari, S. Halevi and W. Venema. Where Do You Want to Go Today? Escalating Privileges by Pathname Manipulation. Proc. Symposium on Network and Distributed Systems Security (NDSS) 2010.
8. A. Datta, A. Derek, J. C. Mitchell, and A. Roy, Protocol composition logic (PCL), Electronic Notes in Theoretical Computer Science, 2007.
9. A. Datta, J. Franklin, D. Garg, and D. Kaynar, A logic of secure systems and its application to trusted computing, Proc. of the IEEE Symp. on Research in Security & Privacy, pp. 221–236, 2009.
10. L. Freitas, J. Woodcock, and Z. Fu, POSIX file store in Z/Eves, Science of Computer Programming, vol. 74, no. 4, pp. 238–257, 2009.
11. O. Goldreich. Foundations of Cryptography. Cambridge Press Vol 1(2001) and Vol. 2(2004)
12. T. Gross, B. Pfitzmann, and A.-R. Sadeghi, Browser model for security analysis of browser-based protocols, ESORICS 2005, pp. 489–508.
13. R. Joshi and G. J. Holzmann, A mini challenge: Build a verifiable filesystem, Formal Aspects of Computing, vol. 19, no. 2, pp. 269–272, 2007.
14. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, seL4: Formal verification of an OS kernel, SOSP 2009, pp. 207–220.
15. J. McLean, A general theory of composition for a class of "possibilistic" properties, IEEE Transactions on Software Engineering, vol. 22, no. 1, pp. 53–67, 1996.
16. P. Neumann and R. Feiertag, PSOS revisited, ACSAC 2003, pp 208–216.
17. P. Neumann, Principled assuredly trustworthy composable architectures, DARPA Project CHATS, Final Rep., 2004. `http://www.csl.sri.com/users/neumann/chats.html`
18. B. Pfitzmann and M. Waidner. A General Framework for Formal Notions of "Secure" Systems. Institut fur Informatik, Hildesheim University. Apr. 1994.
19. B. Pfitzmann and M. Waidner. Composition and Integrity Preservation of Secure Reactive Systems. In *Proc. CCS*, 2000. pp 245–254.
20. The Open Group Base Spec. Issue 7, IEEE Std 1003.1-2008. `http://www.opengroup.org/onlinepubs/9699919799/`.