Proving confidentiality in a file system using DISKSEC

Atalay İleri, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL

Abstract

SFSCQ is the first file system with a machine-checked proof of security. To develop, specify, and prove SFSCQ, this paper introduces DISKSEC, a novel approach for reasoning about confidentiality of storage systems, such as a file system. DISKSEC addresses the challenge of specifying confidentiality using the notion of *data noninterference* to find a middle ground between strong and precise information-flow-control guarantees and the weaker but more practical discretionary access control. DISKSEC factors out reasoning about confidentiality from other properties (such as functional correctness) using a notion of *sealed blocks*. Sealed blocks enforce that the file system treats confidential file blocks as opaque in the bulk of the code, greatly reducing the effort of proving data noninterference. An evaluation of SFSCQ shows that its theorems preclude security bugs that have been found in real file systems, that DISKSEC imposes little performance overhead, and that SFSCQ's incremental development effort, on top of DISKSEC and DFSCQ, on which it is based, is moderate.

1 Introduction

Many security problems today stem from bugs in software. Although there has been significant effort in reducing bugs through better testing, fuzzing, model checking, and so on, subtle bugs remain and continue to be exploited. Machine-checked verification is a powerful approach that can eliminate a large class of bugs by proving that an implementation meets a precise specification.

Prominent examples of machine-checked security proofs include verification of strict isolation (with confidentiality) for an OS kernel in CertiKOS [15], seL4 [26], and Komodo [17], as well as security proofs in Ironclad [20] about applications like a password hasher and a notary service. However, proving the security of systems with rich sharing semantics, such as file systems, is an open problem. For example, unlike prior examples that focus on strict isolation without controlled sharing, users in a file system can share files with one another, and the underlying implementation has shared data structures (such as a buffer cache or write-ahead log) that contain data from different users.

Proving security for a file system requires addressing two key challenges. The first challenge lies in *specifying* security. Integrity can be expressed as simply as a functional correctness property. Confidentiality is more challenging to specify. For example, consider a natural specification for `readdir`, which allows the file system to return the names in a directory in any order. This nondeterminism could be abused by a buggy or malicious file system to leak confidential file data through careful manipulation of the order of `readdir` results. Furthermore, nondeterminism is essential to a file system, because file systems must deal with crashes, which can occur nondeterministically at any time.

One approach to specifying confidentiality is to formulate it as a noninterference property, such as in most information-flow-control systems. This means that the execution of one process (a potential victim processing confidential data) cannot influence the execution of another process (an adversary trying to learn that data). Noninterference can be stated concisely, and is easy for applications to use. However, information-flow-control style guarantees are stronger than what file systems aim for. Instead, file systems aim for weaker notions of confidentiality, along the lines of discretionary access control on files that reveal some metadata, such as file lengths.

A second challenge lies in proving confidentiality. Confidentiality is a "two-safety" property [34], which requires reasoning about *pairs* of executions to show that an adversary cannot observe any differences correlated with confidential data. However, reasoning about pairs of executions is more complicated than reasoning about a single execution, which is sufficient for proving integrity and functional correctness.

This paper presents DISKSEC, an approach for proving the security, and specifically confidentiality, for storage systems, such as file systems. The paper demonstrates the benefits of DISKSEC by developing, specifying, and proving the security of a file system in a prototype called SFSCQ, based on the DFSCQ file system [13].

DISKSEC addresses the specification challenge by using a notion of *data noninterference* that both matches what file systems aim to provide and is concise and easy to use for applications. Data noninterference requires that an adversary's execution be independent of the contents of individual files, but it allows the adversary to observe other metadata, such as file length and directory entries, and allows for discretionary access control (i.e., a user can choose to disclose their data).

To address the challenge of proving security, DISKSEC factors out reasoning about confidentiality from all other properties, such as functional correctness. DISKSEC does so by introducing a notion of sealed blocks. This builds on the intuition that file systems do not look inside of the blocks that represent user file contents. As a result, DISKSEC is able to treat confidential file blocks as opaque in much of the file-system code, greatly reducing the need for manual proofs of two-safety that consider pairs of executions. The only manual proofs of two-safety are in the top-level `read` and `write` system calls.

We implemented DISKSEC and SFSCQ in the Coq proof assistant [35]. All proofs of security are machine-checked by Coq, eliminating the possibility of bugs that violate SFSCQ's specification. An evaluation of SFSCQ shows that its specifications are complete enough to prove confidentiality of a simple application. The evaluation also shows that DISKSEC's approach allowed us to develop SFSCQ with a modest amount of effort, and that SFSCQ achieves comparable performance to the DFSCQ file system that it is based on.

The contributions of this paper are:

- SFSCQ, the first file system with a machine-checked proof of confidentiality. SFSCQ has a concise specification that captures discretionary access control using data noninterference, and deals with nondeterminism due to crashes.
- DISKSEC, an approach for specifying and proving confidentiality for storage systems that reduces proof effort. DISKSEC uses the idea of sealed blocks to factor out reasoning about confidentiality from most of the file system code.
- An evaluation that demonstrates that DISKSEC's approach leads to negligible performance overheads in SFSCQ, that it precludes the possibility of confidentiality bugs that have been found in existing file systems, and that SFSCQ's specification allows applications to reason about their confidentiality.

Our SFSCQ prototype has several limitations. Since it relies on Coq's extraction to Haskell, inherited from DFSCQ, its trusted computing base (TCB) includes the Haskell runtime and compiler. The version of SFSCQ with fully machine-checked proofs does not support changing permissions. A newer version of SFSCQ supports dynamic permissions but has a few proofs that have not been repaired to reflect this change. Finally, SFSCQ's access-control mechanisms are relatively simple, supporting owned and public files but not groups or separate read and write permissions.

2 Related Work

DISKSEC builds on a large body of prior work in several dimensions, as we discuss in the rest of this section.

Data noninterference. DISKSEC's notion of data noninterference builds on prior work on formalizing noninterference properties [19, 25, 26, 29, 30, 32]. Specifically, data noninterference can be thought of as a specialization of abstract noninterference [18], relaxed noninterference [24], or observation functions [15]. One difference in our approach is that data noninterference stops at the file-system API boundary; applications are not subject to data-noninterference policies. This matches well the traditional discretionary access-control policies enforced by file systems.

Formalizing data noninterference requires reasoning about two executions, since confidentiality is a two-safety property [34]. In this context, our contribution lies in a specification and a proof style based on sealed blocks that helps us prove a data-noninterference two-safety property about the file system.

Machine-checked security in systems. Several prior projects have proven security (and, specifically, confidentiality) properties about their system implementations: seL4 [23, 26], CertiKOS [15], and Ironclad [20]. For seL4 and CertiKOS, the theorems prove complete isolation: CertiKOS requires disabling IPC to prove its security theorems, and seL4's security theorem requires disjoint sets of capabilities. In the context of a file system, complete isolation is not possible: one of the main goals of a file system is to enable sharing. Furthermore, CertiKOS is limited to proving security with deterministic specifications. Nondeterminism is important in a file system to handle crashes and to abstract away implementation details in specifications.

Ironclad proves that several applications, such as a notary service and a password-hashing application, do not disclose their own secrets (e.g., a private key), formulated as noninterference. Also using noninterference, Komodo [17] reasons about confidential data in an enclave and showing that an adversary cannot learn the confidential data. Ironclad and Komodo's approach cannot specify or prove a file system: both systems have no notion of a calling principal or support for multiple users and there is no possibility of returning confidential data to some principals (but not others). Finally, there is no support for nondeterministic crashes.

Information flow and type systems. Another approach to ensuring security is to rely on types or runtime enforcement mechanisms. Although this does not give a machine-checked theorem of security, we build on aspects of this approach, namely, the sealed disk has typed blocks.

Type systems and static-analysis algorithms, as with Jif's labels [27, 28] or the UrFlow analysis [14], have been developed to reason about information-flow properties of application code. However, these analyzers are static and would be hard to use for reasoning about data structures inside of a file system (such as a write-ahead log or a buffer cache) that contain data from different users.

Bug description	Filesystem(s)	year
anyone can change POSIX ACLs	btrfs [5], gfs2 [3]	2010
anyone can change POSIX ACLs	NFS [8]	2016
file permissions can be changed by writing to hidden file	reiserfs [2]	2010
truncated data can be accessed	btrfs [7]	2015
crash can expose deleted data	ext4 [9]	2017
crash can expose data	ext4 [22]	2014
can overwrite append-only file	ext4 [4], btrfs [6]	2010
can overwrite arbitrary files	ext4 [1]	2009

Figure 1: Bugs in various Linux file systems that can lead to data-disclosure or integrity violations.

Dynamic tools, such as Jeeves and Jacqueline [37, 38] and Resin [39], deal with dynamic data structures but require sophisticated and expensive runtime enforcement mechanisms. DISKSEC avoids the overhead of runtime enforcement and an additional trusted runtime checker.

Formalizing file-system security. Prior work has extensively studied the security guarantees provided by file systems, both formally and informally [10]. However, none of the prior work articulated a precise, machine-checkable model and specification for file-system security.

Symbolic models of cryptography. Our proof strategy is related to techniques introduced to reason about cryptographic protocols. Many cryptographic-protocol proofs are done in the Dolev-Yao model of perfect cryptography [16]. There programs are modeled as algebraic expressions, which developers reason about using equational axioms, like that decryption is the inverse of encryption, when called with identical symmetric keys. No equations allow breaking encryption without knowing the key. This model is attractive for its simplicity, and protocol-analysis tools like ProVerif [11] and Tamarin [33] build on it. HACL* [40] uses a similar proof strategy for proving its cryptographic library. DISKSEC's block-sealing abstraction extends this idea with the notion of a permission associated with each sealed block.

3 Motivation: bugs

File systems are an important building block for applications, which rely on the file system for security. For example, a mail server relies on the file system to ensure that data written to one user's mailbox file does not end up in some other user's mailbox file. Unfortunately, file systems have had bugs that allowed for data disclosure or modifying other users' files: we list several such bugs in Figure 1. In this section we describe several of these bugs in more detail.

File-system data leak. ext4 has an optimization called delayed allocation where new blocks for files are not actually allocated (but simply tracked) until they must be flushed to disk. It is important that even after a crash, blocks allocated in this manner have their new data written before ending up as part of the file; otherwise the old data in the block is leaked, potentially disclosing data from any user. For some time ext4 used its write-ahead log to ensure the new data was written atomically with the metadata changes to the file. An optimization introduced in 2012 removed this write-ahead logging [36], reasoning that the new data was always written to disk immediately with delayed allocation, before flushing the log. This optimization is incorrect: the disk may reorder writes so that the journal is actually written to disk first, exposing the old data on crash; the bug was fixed in 2016 by restoring the old behavior of writing the newly allocated blocks through the write-ahead log.

Access-control checks. File systems implement sophisticated policies for controlled sharing, such as file permissions, append-only or read-only files, and shared directories. It is easy for file-system developers to make mistakes in implementing these policies. For example, several file systems forgot to correctly implement append-only files when the file was being modified through a special interface for efficiently moving file data [4, 6]. In these examples, the file system did not read or write the file data itself but instead changed the data-block pointers inside of the file's inode. Another example is the privileged `nfsd` daemon, which forgot to check permissions when local users changed POSIX ACLs on a file [8]. A final example is a file system that stored metadata (including ACLs) in a separate file but failed to prevent users from directly modifying that separate file [2].

4 Goal

The goal of DISKSEC is to use machine-checked verification to ensure the absence of security bugs in file systems. Using a proof assistant (Coq) to check our proofs ensures that we consider all possible corner cases in our implementation when proving that it meets our specification. Thus, as long as our specification excludes the possibility of certain bugs, such as the ones described in the previous section, Coq will provide a high degree of assurance that no such bugs can exist in the implementation.

4.1 Threat model

From the perspective of verification, we would like to have confidence that the file system is secure purely based on the file system's security specification. This means that we have to treat the developer of the file system with an adversarial mindset. This subsumes all possible bugs that a well-meaning but error-prone developer might introduce into the implementation.

As a result, our threat model is that the adversary both develops the file system and runs an adversarial application on top of the file system in an attempt to obtain confidential file data. However, the adversary does provide a proof that their file-system implementation meets our security specification. The potential victim runs on top of the same file system but sets their permissions so that the confidential files are not accessible to the adversary's process. Our goal is to ensure that the security specification is so strong that it prevents leaks even when the file-system developer is colluding with adversarial processes running on top of the file system.

Our threat model is focused on proving that the file-system implementation has no confidentiality bugs, rather than proving the absence of bugs in the environment outside of the file system. Thus, we assume that our model of how the file-system implementation executes is correct. That is, we are not concerned with bugs in unverified software or hardware outside of the file system, or users mounting malicious disk images. We do prove that `mkfs` produces a correct image, but ensuring confidentiality on top of an intentionally corrupted file system image is difficult, even without formal verification. We also do not reason about timing channels, as we do not model time.

4.2 Challenges

The most difficult aspect of formally proving the security of a file system lies in guaranteeing confidentiality. This is difficult for several reasons.

Two-safety. First, proving confidentiality is more difficult than proving functional correctness: as mentioned earlier, confidentiality is a two-safety property. Functional correctness is a one-safety property because a violation of functional correctness can be demonstrated by a single execution. For instance, if an application wrote one byte to a file and then read back a different byte, this single execution shows that the file system is incorrect. Thus, functional correctness of a file system is a theorem that says that all executions meet the spec (i.e., there are no such violations). Integrity properties, such as ensuring that one user cannot corrupt another user's data, are an example of a one-safety functional-correctness property and can be handled using standard verification techniques.

In contrast, demonstrating a violation of confidentiality requires two executions, where the results observed by an adversary differ depending on the secret data. For instance, consider a file system with block-level deduplication that also exposes the number of free blocks. An adversary who wants to learn the contents of a victim's file could write their guess for the victim's block into the adversary's own file and then check whether the number of free blocks stayed the same or decreased. If the file system implemented deduplication across users, this attack allows an adversary to learn whether their guess block was already present in the file system, thus inferring whether the victim has that data.

In the above example, looking at a single execution does not allow one to directly conclude that data was leaked, because the system appears to be functioning correctly. Determining that data is leaking requires one to consider a pair of executions, in which the adversary performs the same operations, but the confidential user data is different. If these two executions produce different adversary-observable results, the adversary is able to infer information about confidential data.

By stating confidentiality as a two-safety property, the above deduplication example would violate confidentiality, and thus could not appear in an implementation that was proven to achieve confidentiality. Specifically, suppose the starting states of the two executions differed in the contents of a confidential file, where in one execution the file matched the adversary's guess and in the other execution it didn't match. In this case, the number of free blocks returned by the adversary would differ in the two executions, which would not be allowed by the confidentiality definition.

Nondeterminism and probabilities. Another complication in proving confidentiality lies in the fact that many specifications, including those in the file system, are nondeterministic. Some nondeterminism is unavoidable because file systems must deal with crashes (e.g., due to power failure), which can occur at any time. Thus, it is impossible to know what are the exact contents of the disk after a crash; the on-disk state could reflect any prefix of the writes issued by the file system. Modern disks complicate this situation even further by buffering writes in memory inside the disk controller; as a result, the writes can be made durable out-of-order, and the state of the disk after a crash might reflect some out-of-order writes. Even in the absence of crashes, the file system implementation may want to use randomness (e.g., to randomize directory hash tables), which makes the execution nondeterministic.

Other nondeterminism comes from specifications that hide irrelevant details. For instance, the inode allocator in the file system does not specify which precise inode number will be returned; instead, its specification simply states that it will return *some* inode number that is not already in use. As another example, the specification for `readdir` in a file system likely allows the files in a directory to be returned in any order. The use of nondeterminism is important for keeping specifications concise and for allowing implementations to change (e.g., to implement performance optimizations) without modifying the specification.

Any nondeterminism is a potential leak of confidential data. The nondeterministic specification of the block allocator from above does not preclude the allocator from leaking confidential data, because it could, in theory, choose the next inode number based on the confidential contents of files, without violating its specification (i.e., still returning some unused inode number). Similarly, the nondeterministic specification for `readdir` is also not a good confidentiality specification, because a bug might cause the order of entries returned by `readdir` to be affected by the contents of some confidential file.

Even the nondeterminism associated with the state of the disk after a crash can be taken advantage of by an adversarial file-system implementation to leak data. For instance, a high-performance file-system specification allows the file system to delay flushing data to disk. An adversarial implementation could choose whether to flush data immediately or defer the flush based on one bit of confidential data from a victim's file. To take advantage of this, an adversary could wait for the system to crash and, after the crash, check whether any writes appear to have been lost. If so, the adversary concludes the file system must have deferred the writes, which would have only happened if the confidential bit was zero. This, in turn, can allow the adversary to infer confidential bits.

More generally, the possibility of leaking confidential data arises because nondeterministic specifications capture what *might* be possible, but an adversary may have more precise information about the actual *probabilities* of different outcomes. For instance, consider a hypothetical system call that returned random data. An adversarial file-system implementation could leak confidential data through this system call by sometimes returning uniformly random data and sometimes returning confidential data from some file. A naïve view of this system call might be that, since any return value is possible, this system call is not leaking any data. However, an adversary can leverage the distribution of outcomes to learn confidential data over time, by invoking this system call many times and observing what value is being returned more frequently than one would expect from a uniform distribution.

Indirect disclosure. Yet another complication with confidentiality is that an adversarial file system might not immediately leak confidential data. For example, an adversarial file system may wait for a legitimate user to read confidential data, at which point the file system would be allowed to access this data, since it has to return it to the user. However, in addition to returning this data, an adversarial file system could also stash away a copy of it, so that the adversary can later retrieve it. For instance, the file system could change the order of entries in an on-disk directory structure, or change the allocated inode numbers or block numbers, based on the confidential data that it wants to leak. Preventing this attack is difficult because the adversarial file system appears to have legitimate access to the user's data when operating on behalf of that user.

File-system complexity. Finally, file systems are complex software. Linux ext4, for instance, consists of approximately 50,000 lines of code. Even the simple verified DFSCQ file system consists of thousands of lines of executable code [13]. The proofs of functional correctness for DFSCQ are already tens of thousands of lines of Coq code. The complexity of proving two-safety, which is a more challenging property, could easily spiral out of control.

5 Specification: data noninterference

To capture the notion of confidentiality in a file system, DISKSEC defines the notion of *data noninterference*. Loosely speaking, data noninterference states that two executions are indistinguishable with respect to specific confidential data (e.g., the contents of a file). Data noninterference allows an application to conclude that an adversary cannot learn the contents of a file from the file system but may be able to learn other information about the file (e.g., its length, its creation time, the fact that it was created at all, etc.). Furthermore, data noninterference does not place any restrictions on application code, which captures the discretionary aspect of typical file-system permissions. This notion intuitively corresponds to the security guarantees provided by Linux file systems.

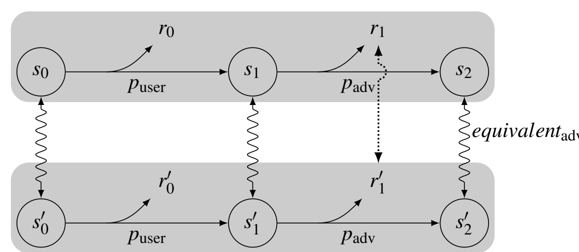

Figure 2: Overview of DISKSEC's approach to reasoning about confidentiality.

Two-safety formulation. DISKSEC formulates data noninterference in terms of two-safety, as shown in Figure 2. Specifically, data noninterference considers two executions that run the same code but start from different states. In Figure 2, the executions are shown as horizontal transitions between states, indicated by the gray outlines. The executions consist of a step by the user (running procedure p_{user}, corresponding to some system call) and then a step by the adversary (running p_{adv}, corresponding to some other system call). Although Figure 2 shows one particular pair of executions, DISKSEC's theorems consider all possible such pairs of executions.

The starting states in these two executions (s_0 and s'_0) agree on all data visible to the adversary but could have different contents of confidential files. We call these two states $equivalent_{\text{adv}}$, to indicate that they are equivalent with respect to the adversary. This equivalence is indicated by the squiggly line in Figure 2. The essence of data noninterference is allowing the states to differ in the contents of confidential data while requiring all other metadata (such as file length, directory order, etc.) to remain the same.

The definition of data noninterference consists of two requirements. The first is *state noninterference*, which requires that after every transition, the resulting states remain $equivalent_{\text{adv}}$. This is indicated in Figure 2 by the squiggly lines between s_1 and s'_1, as well as between s_2 and s'_2. This requirement ensures that confidential data from s_0 and s'_0 does not suddenly become accessible to the adversary in a subsequent state, and it addresses the indirect-data-disclosure challenge (e.g., an adversarial implementation of the `read` system call stashing away the results).

The second requirement is *return-value noninterference*, which requires that transitions by the adversary return exactly the same values in both executions. For example, Figure 2 shows that the adversary's p_{adv} returns r_1 in the top execution and r'_1 in the bottom execution. Return-value noninterference requires that $r_1 = r'_1$, as indicated by the dotted arrow. This prevents the adversary from learning any confidential data, such as through collusion with an adversarial file system that affects the order of `readdir` results, or through missing access control checks.

Capturing file-system security. Achieving the two requirements from data noninterference ensures that the adversary cannot obtain confidential data from the file system. This is because state noninterference maintains $equivalence_{\text{adv}}$ regardless of what the adversary does (i.e., the squiggly lines will continue to connect states in all possible pairs of executions), and any attempts by the adversary to observe information will produce identical results, based on return-value noninterference, because they run in $equivalent_{\text{adv}}$ states.

The discretionary nature of data noninterference shows up in the fact that legitimate users can obtain different results depending on the confidential data. For example, in Figure 2, the results of the user's execution of p_{user}, r_0 and r'_0, might be different, because p_{user} could correspond to the user reading a confidential file. At this point, a user has the discretion to disclose this information (e.g., by writing it to a public file). Data noninterference does not prevent this, by design, because it is attempting to model the standard discretionary access control in a POSIX file system.

Defining return-value noninterference. Figure 3 presents DiskSec's definition of return-value noninterference, in a simplified notation. This definition relies on the definition of `exec`, which describes how procedures execute. `exec` takes four arguments: the procedure that is executing (`p`), the principal on whose behalf `p` is running (`u`), the starting state (`st0`), and the randomness for this execution (`rand`). `exec` returns two things: the outcome and an *unseal trace*, which we describe later. The outcome can be either `Finished st' r`, indicating that the procedure ended in state `st'` and returned `r`, or `Crashed st'`, indicating that the system crashed in state `st'`. The unseal traces are irrelevant for now and are used only as part of the proof technique described in §6. This definition also relies on a notion of two states being equivalent for a particular principal, `equivalent_for_principal`, which captures the intuitive notion $equivalent_{\text{adv}}$ from above.

```
Definition equivalent_for_principal u st0 st1 :=
  (* all parts of st0 and st1 that are accessible to
     principal u are identical *).

Definition ret_noninterference '(p : proc T) :=
  forall u st0 st0' rand ret tr0 st1,
    exec p u st0 rand =
        Some (Finished st0' ret, tr0) ->
    equivalent_for_principal u st0 st1 ->
    forall st1' tr1,
      exec p u st1 rand =
          Some (Finished st1' ret', tr1) ->
      ret' = ret.
```

Figure 3: Definition of return-value noninterference, capturing that return values do not leak other users' confidential data.

The definition of return-value noninterference captures the intuition about the adversary not being able to learn information about confidential data: the return value obtained by the adversary by running some code does not depend on the confidential data. To make this precise, `ret_noninterference` of procedure `p` considers pairs of states, `st0` and `st1`, which are equivalent as far as some principal `u` is concerned. Here, `u` is representing the adversary, and confidential data is represented by the difference between `st0` and `st1` that the adversary should not be able to observe. If `u` runs procedure `p` in state `st0` and gets return value `ret`, then it must also have been possible for the adversary to get the same return value, `ret`, if he ran `p` in state `st1` instead.

Defining state noninterference. Figure 4 presents DiskSec's definition of state noninterference, which complements return-value noninterference. This definition helps DiskSec deal with the indirect-disclosure challenge from §4.2. This definition considers two principals: a `viewer` and a `caller`. The definition intuitively says that, by running procedure `p`, the caller will not create any state differences observable to `viewer`.

More formally, `state_noninterference` considers two executions by `caller`, running the same procedure `p`, with the same exact arguments (encoded inside of `p`). If the caller runs `p` in two states that appear equivalent to `viewer`, then the resulting states in `res0` and `res1` will still appear equivalent to `viewer`. This definition includes the possibility of a crash while running `p`.

```
Definition equiv_state_for_principal u res0 res1 :=
  exists st0 st1,
    equivalent_for_principal u st0 st1 /\
    (res0 = Crashed st0 /\ res1 = Crashed st1 \/
     exists v0 v1,
       res0 = Finished st0 v0 /\
       res1 = Finished st1 v1).

Definition state_noninterference `(p : proc T) :=
  forall viewer caller st0 rand res0 tr0 st1,
    exec p caller st0 rand = Some (res0, tr0) ->
    equivalent_for_principal viewer st0 st1 ->
    forall res1 tr1,
      exec p caller st1 rand = Some (res1, tr1) ->
      equiv_state_for_principal viewer res0 res1.
```

Figure 4: Definition of state noninterference, capturing that `caller` does not indirectly disclose state to `viewer`.

Handling non-determinism and probabilities. Both Figure 3 and Figure 4 quantify over an argument called `rand` that is passed to `exec`. `rand` is an oracle that supplies all non-determinism used during execution, including non-deterministic values used by the file system implementation (e.g., getting a random number), as well as non-determinism representing the effect of a crash (i.e., the point at which the crash occurred, and which recent writes made it to disk). The execution semantics, `exec`, queries the `rand` oracle whenever it needs to make a non-deterministic decision. The `exec` function is deterministic given a specific `rand`, but DISKSEC allows non-determinism by permitting different executions with different non-determinism oracles. One way to think of this `rand` oracle is that it represents a seed for a logical random-number generator.

Factoring out the randomness `rand` from the execution semantics `exec` helps DISKSEC handle probabilities without fully formalizing probabilistic reasoning in Coq. Since the `exec` function is deterministic (given a specific randomness oracle `rand`), the probability of a particular outcome is the sum of the probabilities of different `rand` oracles that lead to that outcome. Following the random-number generator seed analogy, the probability of an outcome is simply the fraction of seeds that lead to that outcome.

DISKSEC's theorem statements require that, for any choice of `rand`, both the return values and states are equivalent. This ensures that the probabilities of equivalent return values and states are also equal, since the probabilities of these outcomes are simply the sums of probabilities of individual `rand` values. Using the samples to relate the probabilities of outcomes is reminiscent of a coupling argument [21], although we do not explicitly reason about probabilities in DISKSEC. Since the probabilities of the outcomes are equal, this prevents an adversary from learning confidential data based on the observed probabilities of different outcomes.

6 Proof approach: sealed blocks

Proving that every system call in a file system satisfies `ret_noninterference` and `state_noninterference` would require a proof that reasons about two executions, which is complex. To reduce proof effort, DISKSEC introduces an implementation and proof approach called *sealed blocks*. This approach factors out reasoning about confidentiality of files from most of the file-system logic, by reasoning about the confidentiality of disk blocks. The intuition behind this approach is threefold. First, all confidential data lives in file blocks. Second, the file system itself rarely needs to look inside of the file blocks. Finally, permissions on files translate directly into permissions on the underlying blocks comprising the file.

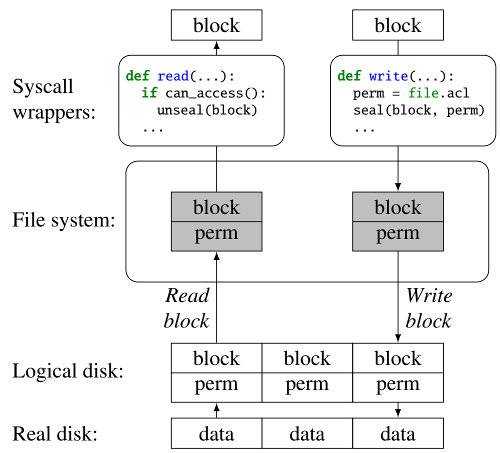

Figure 5: Overview of DISKSEC's proof approach using sealed blocks.

Figure 5 presents an overview of DISKSEC's block-sealing approach. There are three parts to the block-sealing approach. The first is to create a logical disk where every disk block is associated with a *permission*, which defines the set of principals that can access this block. Some permissions are public, indicating that the block is accessible to anyone. Other permissions might restrict access to some users, indicating that this block is storing confidential file data. DISKSEC is agnostic to the specific choice of principals or permissions; that is, all of DISKSEC is parameterized over arbitrary types for principals and permissions. The logical disk is purely a proof strategy and does not appear at runtime; the real disk, shown at the bottom of Figure 5, has no permissions.

The second part is a sealed-block abstraction, indicated by shaded blocks in Figure 5. A sealed block represents the raw block contents and the associated permission, but the file system cannot directly access a sealed block's contents. Instead, the file-system implementation must explicitly call `seal()` and `unseal()` to translate between sealed blocks and their raw contents. These `seal()` and `unseal()` functions are also purely part of the proof and do not appear at runtime.

The code of the file system can read and write arbitrary blocks on disk, but the result of a read is a sealed block that must be explicitly unsealed if needed. The file-system internals can unseal public blocks (e.g., containing allocator bitmaps or inodes) but cannot unseal private blocks. This avoids the need to reason about the file-system implementation when proving confidentiality, because the file-system implementation never has access to confidential data.

The third part is the wrappers for system calls that handle confidential data, namely, `read()` and `write()`. These wrappers are responsible for explicitly calling `seal()` and `unseal()` to translate between the raw data seen by the user (on top of the system call) and the sealed blocks that are handled in the rest of the file-system implementation.

DISKSEC's sealed-block approach is a good fit for the challenges outlined in §4.2. Specifically, there are very few places where a file system must access the actual contents of a file's disk block—namely, in the wrappers for the `read()` and `write()` syscalls. As a result, most specifications in a file system remain largely the same. The key difference is that the specifications promise that the procedure in question does not look inside of any confidential blocks. This means that any nondeterminism present in the specification cannot be used to leak confidential data.

This approach allows file-system developers to avoid proving explicit confidentiality theorems for most of the file system, but it still allows DISKSEC to conclude that confidentiality is not violated. DISKSEC provides a theorem that proves two-safety for any file-system implementation that correctly uses the sealed-block abstraction. As a result, the file-system developer need not reason about complex two-safety theorems and can limit their reasoning to single executions.

6.1 Formalizing sealed blocks

To formally define DISKSEC's sealed-block abstraction, DISKSEC uses the notion of a *handle* to represent a sealed block. DISKSEC requires the developer to perform two steps. The first is to modify their code to use the sealed-block abstraction: that is, to pass around handles for blocks and to call `seal()` and `unseal()` as necessary. The second is to prove that their code correctly follows the unsealing rules. This boils down to ensuring that sealed blocks are unsealed only when the principal has appropriate permission for that block.

DISKSEC models this by extending traditional Hoare logic to reason about unseal operations. Specifically, DISKSEC builds on CHL [12], where functional correctness specifications are written in terms of pre- and postconditions. DISKSEC, first, extends the execution semantics (as we describe next) to produce an *unseal trace* consisting of unseal operations and, second, extends the specifications to require that the unseal trace contain only allowed unseals.

We expect that systems built on top of DISKSEC would often group multiple blocks into a single object (e.g., multiple blocks comprising a single file in a file system). To help developers reason about all of these blocks sharing the same permissions, DISKSEC introduces the notion of a *domain*. This is a layer of indirection between blocks and permissions. Specifically, sealed blocks point to a domain ID (e.g., an inode number in the case of a file system), and the domain in turn specifies the permission for those blocks (e.g., the permission reflected in the inode's data structure).

Execution model. DISKSEC's execution model requires the implementation to be written in a domain-specific language, based on CHL and implemented inside of Coq, which provides several primitive operations. These operations include reading and writing the disk, manipulating sealed blocks by sealing and unsealing, as well as others for sequencing computation, returning values, flushing disk writes, etc.

Figure 6 shows a simplified version of DISKSEC's execution semantics. The semantics are defined as a function that takes the `code` being executed (of type `proc T`), the principal `u` running the operation (of type `Principal`), the starting state `st` (of type `State`), and a randomness oracle `rand`. The function produces a tuple consisting of a result (of type `result T`) and a trace of unsealed permissions (of type `trace`). The function is allowed to return `None` (as indicated by the `option` type) when there is no execution possible for the supplied randomness (e.g., the randomly chosen handle is already in use).

For example, consider the case that handles the `Read a` operation, which describes the execution of reading address `a` from disk. There are three sub-cases. If the address is out of bounds, the `Read` returns a handle for a zero block, with an empty unseal trace. If the handle `h` supplied by the randomness oracle is already in use, no execution is possible. Otherwise, the `Read` initializes the new handle to represent the block from address `a`, with the block's domain ID, and returns that handle, with an empty trace because no blocks were unsealed.

As another example, the `Unseal h` operation produces a nonempty trace, consisting of the permission of the sealed block whose handle `h` was unsealed, as long as the handle was valid (otherwise, `Unseal` returns zero). Since the sealed block points to a domain ID, `dom`, the semantics of `Unseal` look up the corresponding permissions of that domain. One omitted rule handles concatenation of unseal traces when a developer sequences one statement after another.

The `ChangePerm dom newperm` operation allows the developer to change permissions of a domain. This operation is used in implementing `chown`. The semantics of

```
Inductive nondet_decision :=
| RandomHandle (h:handle)
(* Other types of non-determinism omitted for space *)
| CrashHere.

Definition oracle := list nondet_decision.

Definition exec '(code:proc T) (u:Principal) (st:State)
           (rand:oracle) : option (result T * trace) :=
  match code, rand with
  | ChangePerm _ _, CrashHere => None
  | _, CrashHere => Some (Crashed st, [])
  | Read a, RandomHandle h =>
    if addr_out_of_bounds st a then
      Some (Finished st hzero, [])
    else if handle_used st h then
      None
    else
      let data := disk_block_data st a in
      let dom  := disk_block_dom  st a in
      let st' := install_handle st h (data, dom) in
      Some (Finished st' h, [])
  | Write a h, _ =>
    if handle_used st h then
      let data := handle_data st h in
      let dom  := handle_dom  st h in
      let st' := disk_block_write st a (data, dom) in
      Some (Finished st' tt, [])
    else
      Some (Finished st tt, [])
  (* Some transitions omitted for space reasons *)
  | Seal data dom, RandomHandle h =>
    if handle_used st h then
      None
    else
      let st' := install_handle st h (data, dom) in
      Some (Finished st' h, [])
  | Unseal h, _ =>
    if handle_used st h then
      let data := handle_data st h in
      let dom  := handle_dom  st h in
      let perm := domain_perm st dom in
      Some (Finished st data, [perm])
    else
      Some (Finished st zero, [])
  | ChangePerm dom newperm, _ =>
    let oldperm := domain_perm st dom in
    let st' := domain_set_perm st dom newperm in
    Some (Finished st tt, [oldperm])
  | _, _ => None
  end.
```

Figure 6: Execution semantics with logging of unseal operations.

ChangePerm modify the permission associated with the domain, and produce an unseal trace containing the domain's old permission, to reflect that data with that permission may have been disclosed. Since the domains are purely a proof construct, ChangePerm is a purely logical operation, which does not perform any actions at runtime.

Finally, exec describes the possible crash behaviors of the system. For example, the case for _, CrashHere states that it is possible to crash in the starting state, regardless of what code was being executed, if the randomness oracle tells us CrashHere. A combination of other rules, not shown, allow crashing in the middle of a sequence of operations. The very first case, for ChangePerm _ _, CrashHere, says that ChangePerm cannot crash. This reflects the fact that ChangePerm is a purely logical operation.

Specification and verification of unseal rules. DiskSec requires developers to write specifications for each procedure, using pre- and postconditions. The postcondition describes how the procedure modifies the state of the system, along with what must be true of the procedure's return value, assuming that the precondition (a predicate over the system state and the procedure's arguments) held at the start of the procedure.

To reason about what blocks a procedure might unseal, DiskSec augments specification postconditions with requirements about the permissions that appear in the unseal trace produced by the execution of the procedure.

```
Definition unseal_safe '(p : proc T) :=
  forall u st rand res tr,
    exec p u st rand = Some (res, tr) ->
      forall perm,
        In perm tr -> can_access u perm.
```

Figure 7: Definition of unseal safety.

Figure 7 shows DiskSec's definition of unseal safety. This definition says that procedure p is "unseal-safe" if, for every principal u that runs this procedure and any starting state st, all permissions produced by this procedure in its unseal trace tr will be accessible to the calling principal. Proving unseal safety leads to a proof obligation for the file-system developer—namely, proving that the implementation will unseal a block only if the current principal has access to it.

File-system implementation code falls into three categories with respect to proving unseal safety. The first category are procedures that do not invoke any Unseal operations. For these procedures, the resulting unseal trace is always empty, and DiskSec is able to prove unseal safety without any developer input. Most of the file-system code falls in this category.

The second category are procedures that unseal public blocks. Examples include accessing inodes, allocator bitmaps, directories, etc. These procedures do produce unseal traces containing permissions, but all of the permissions should be public. Thus, the developer's job is to show that these permissions are indeed public; once this is established, showing that the current principal has access is straightforward (since every principal has access to public permissions).

To prove that the permissions are indeed public, the developer relies on representation invariants of the file system. For example, the invariant for the block allocator states that all of the bitmap blocks are public. The developer can assume this invariant within any implementation of the block allocator API, which helps her prove that the block in question has public permissions. In turn the developer must prove that the invariant is preserved by every procedure (including across crashes and recovery), and show that it is established at initialization time by `mkfs`.

The final category are procedures that unseal private blocks. In a file system, this happens only in the implementation of the `read` system call, which returns file data to the caller. The implementation (wrapper) of the `read` system call contains explicit code to obtain the current principal, get the file's ACL (access control list) from the inode, and compare them. The developer's job is to prove that this code correctly performs the permission check. This proof typically relies on the file's representation invariant, which asserts that every file block is tagged with a permission matching the ACL stored in the inode.

```
Definition unseal_public `(p : proc T) :=
  forall u st rand res tr,
    exec p u st rand = Some (res, tr) ->
      forall perm,
        In perm tr -> perm = Public.
```

Figure 8: Definition of `unseal_public`.

DISKSEC also provides a stronger version of unseal-safety, as shown in Figure 8, called `unseal_public`. A procedure satisfies this definition if all of its code falls in the first two categories above: that is, the procedure either unseals no blocks at all or unseals only public blocks. This alternative definition is strictly stronger than unseal-safety; any procedure that satisfies `unseal_public` is also unseal-safe. The distinction between these two notions will help the developer prove noninterference theorems, as we will describe in §6.2.

Crashes. DISKSEC's approach naturally extends to reasoning about crashes. DISKSEC's disk-crash model builds on the CHL model of disk crashes [12, 13]. After a crash, disk blocks can be updated nondeterministically, as in CHL, based on outstanding writes that are in the disk's write buffer but have not been flushed yet to durable storage. However, domains always follow the data for pending writes; that is, logically, the content of the disk block is updated atomically together with its domain ID.

All handles are invalidated after a crash, to model the fact that the computer reboots and all in-memory state is lost. All recovery code, such as log replay or `fsck`, is proven correct in DISKSEC, which means that it must follow the same block-sealing rules as the rest of the file-system code. This ensures that no data can be disclosed by the recovery code.

6.2 Proving noninterference

To help the developer prove the two types of noninterference, DISKSEC provides helper theorems. Figure 9 shows the first one, which proves return-value noninterference based on unseal-safety. DISKSEC proves this theorem by considering all operations performed by procedure `p`. Each operation must produce the same result in the two executions being considered, since the states are equivalent for the principal in question, `u`. The only way in which the executions could differ is if they unsealed a block that was not accessible to `u`. However, `unseal_safe` says that this is impossible. This theorem also applies to procedures that are `unseal_public`, since that notion is strictly stronger than `unseal_safe`.

```
Theorem unseal_safe_to_ret_noninterference :
  forall `(p : proc T),
    unseal_safe p -> ret_noninterference p.
```

Figure 9: Theorem connecting unseal-safety to return-value noninterference.

Figure 10 shows the second theorem provided by DISKSEC, for reasoning about state noninterference. This theorem requires that the procedure satisfy the stronger definition, `unseal_public`, to ensure state noninterference. The intuition for why this theorem is true lies in the fact that a procedure that unseals only public blocks cannot obtain any confidential data in the first place. As a result, this procedure's execution will be identical regardless of the contents of confidential blocks, and thus the state after this procedure's execution will remain equivalent from the adversary's point of view. DISKSEC proves this theorem formally in Coq.

```
Theorem unseal_public_to_state_noninterference :
  forall `(p : proc T),
    unseal_public p -> state_noninterference p.
```

Figure 10: Theorem connecting `unseal_public` to state noninterference.

DISKSEC does not provide a general-purpose theorem for reasoning about state noninterference for procedures that satisfy only the weaker notion of unseal-safety (i.e., that unseal private blocks), such as the `read()` system call. Such procedures can indirectly disclose data as described in §4.2 to legitimately unseal confidential data on behalf of the currently executing principal but then stash a copy of it. It is up to the file-system developer to prove the state noninterference of those procedures. §7 will discuss in more detail how SFSCQ structures its implementation to simplify these proofs; in the case of SFSCQ, the only system call that requires this type of reasoning is `read`.

6.3 Code generation

To generate efficient executable code, DISKSEC must avoid explicitly sealing and unsealing blocks. To do so, DISKSEC eliminates any notion of handles, sealing, or unsealing at runtime. DISKSEC does so by representing each handle with the actual disk-block contents themselves, when generating executable code. DISKSEC's theorems ensure that the code does not look at the disk contents at runtime unless it has the appropriate permissions. As a result, it is safe to perform this elimination.

Similarly, this allows the sealing and unsealing operations also to be eliminated from runtime code.

7 Case study: File system

To evaluate whether DISKSEC allows specifying and proving confidentiality for a file system, we applied DISKSEC to the DFSCQ verified file system, producing the SFSCQ verified secure file system, as described below.

7.1 Specifying security

The core specification of confidentiality for SFSCQ lies in the `write` system call, as shown in Figure 11. This specification says that the *data* argument to the `write` system call remains confidential. This is stated formally by considering two different executions, starting from the same state `st`, where different data (`data0` and `data1`) are written to the same offset `off` of the same file `f`. The results, `res0` and `res1`, must be equivalent for any adversary `adv` that does not have permission to access file `f`. Since `equivalent_state_for_principal` considers both crashing and noncrashing executions, this definition ensures that the data passed to `write` remains confidential regardless of whether the system crashes or not.

```
Theorem write_confidentiality :
  forall f off data0 data1 caller st rand res0 tr0,
    exec (write f off data0) caller st rand =
                              Some (res0, tr0) ->
      exists res1 tr1,
        exec (write f off data1) caller st rand =
                              Some (res1, tr1) /\
        forall adv,
          ~ can_access adv (file_perm st f) ->
          equiv_state_for_principal adv res0 res1.
```

Figure 11: Confidentiality specification for the `write` system call.

The other part of the security specification lies in the `chown` system call, which changes the permissions on existing files, and thus affects what data is or is not confidential. Because `chown` can disclose the contents of a previously confidential file, the standard definition of state non-interference from Figure 4 does not hold for `chown`. Specifically, even if an adversary `viewer` could not distinguish states `st0` and `st1` before some `caller` executed `chown`, the adversary may nonetheless be able to distinguish `st0` and `st1` after the `chown` runs because the adversary may now have permission to read the previously confidential file.

The security of `chown` is defined by a specialized version of state non-interference, which considers three cases. The first case is that the adversary `viewer` does not have access to the file after the `chown` (i.e., is not the new owner). In this case, state non-interference holds. The second case is that the adversary `viewer` does gain access to the file after `chown` (i.e., is the new owner), but the file had the same contents in the two executions (i.e., in states `st0` and `st1`). In this case, state non-interference holds as well. Finally, the adversary `viewer` may gain access to the file *and* the files had different contents in the two executions. In this case, state non-interference does not apply. Figure 12 summarizes this formally.

```
Definition chown_state_noninterference f new_owner :=
  forall viewer caller st0 rand res0 tr0 st1,
    exec (chown f new_owner) caller st0 rand =
                              Some (res0, tr0) ->
    ( file_data st0 f = file_data st1 f \/
      viewer <> new_owner ) ->
    equivalent_for_principal viewer st0 st1 ->
      exists res1 tr1,
        exec (chown f new_owner) caller st1 rand =
                              Some (res1, tr1) /\
        equiv_state_for_principal viewer res0 res1.
```

Figure 12: Confidentiality specification for the `chown` system call.

The `write` and `chown` specifications, shown above, are the only parts of the security specification that are specific to the file system, because they define where confidential data enters the system in the first place, and how permissions on that confidential data can change. Somewhat counter-intuitively, no special treatment is required in the specifications of other system calls, such as `read`. Instead, it suffices to prove the two general noninterference theorems for all system calls (i.e., `ret_noninterference` and `state_noninterference`). This is because we do not want to consider specific attacks, such as whether `read` has a missing access-control check. Instead, DISKSEC's noninterference definitions ensure that confidential data cannot be disclosed regardless of what system calls the adversary tries to use.

Integrity of the file system is a functional-correctness property and thus is covered by SFSCQ's specifications, alongside other correctness properties. Integrity did not require SFSCQ to use any machinery from DISKSEC for reasoning about confidential data.

7.2 Modifying the implementation

Changing representation invariants. DFSCQ consists of many modules, such as the write-ahead log, the bitmap allocator, the inode module, etc. Each module has its own invariant that describes how that module's state is represented in terms of blocks. For example, the bitmap allocator describes how the free bits are packed into disk blocks, where they are stored on disk, and the semantics of each bit.

For SFSCQ, we modified all invariants that describe disk blocks to state the domain IDs that go along with those blocks. For instance, we modified the invariant of the allocator to state that the bitmap blocks are public. We modified the write-ahead log layer to expose the underlying domain IDs on disk blocks to modules implemented on top of the write-ahead log (in addition to modifying the log invariant to state that the log metadata is public).

The only nonpublic data is the file contents. We modified the file invariant to state that the domain ID of every file block matches the file's inode number, and the permissions for a particular domain ID match the ACL stored in the inode with the inode number matching the domain ID.

One surprising issue that we encountered came up in the DFSCQ write-ahead log. For performance, DFSCQ's write-ahead log used checksums to verify block contents after a crash. As a result, the recovery procedure unsealed blocks from the write-ahead log after a crash, including blocks that contain confidential data.

To address this issue, we switched to a barrier-based write-ahead log instead, which is the default design of Linux ext4. Instead of using checksums, the barrier-based write-ahead log issues a disk flush between writing the contents of new log entries and updating the log header. (DFSCQ already included an implementation of this barrier-based write-ahead log but did not use it by default.)

Modifying code. Loosely speaking, DFSCQ modules handle two kinds of blocks: blocks that they manipulate (e.g., the bitmap allocator manipulating the bitmap blocks) and blocks that they pass through (e.g., the write-ahead log handling reads and writes as part of a transaction, or the file layer handling file reads and writes). The first category required a module to access the block contents, so we added `Seal` and `Unseal` operations accordingly. Virtually all operations that fell in this category involved sealing and unsealing public data. For the second category, we did not seal or unseal the data and instead transparently passed through the handle representing the block; as a result, the module was oblivious to the domain IDs associated with the disk block.

Private data is sealed and unsealed at the top of the SFSCQ implementation; that is, in the implementation of the `read` and `write` system calls. We modified the `write` system-call implementation to `Seal` the blocks with the file's inode number as the domain ID, before processing them further. We modified the `read` system call to implement the permission-checking logic—i.e., reading the ACL from the file's inode, checking whether the currently running principal has access to the file, and unsealing the block only if the check passes.

Changing intermediate specifications. We augmented the Hoare-logic specifications of all internal SFSCQ procedures to require that the procedure be `unseal_public`. This change required little manual effort, because we simply changed the underlying definition of the Hoare-logic specification to require `unseal_public`. For the write-ahead log, we added additional constraints in the specification of the `log_write` procedure, requiring that the blocks written as part of a transaction must be public, as described above.

7.3 Proving security

Reproving functional correctness. Many existing proofs in DFSCQ broke after we made the above changes. The proofs broke for three reasons: there were now additional `Seal` and `Unseal` operations in the code (e.g., the bitmap allocator now sealed and unsealed its bitmap blocks), the logical representation of a block changed to include a domain ID, and the specification changed (e.g., augmenting the invariant to state the domain ID of a block). This required manually tweaking most of the proofs to fix them. The proof changes were simple since the code's logic and the proof argument remained unchanged.

Proving unsealing. In addition to fixing existing proofs, SFSCQ's specifications required us to prove that the `Unseal` operation was used correctly. For most procedures, the specification required that the procedure satisfy `unseal_public`. Proving that only public blocks were unsealed required us to demonstrate that the block was indeed public by referring to the invariant.

For the implementation of the `read` system call, which unseals private data, we had to prove that `read` correctly implements the permission check in its code. This means proving that `read` calls `Unseal` only after checking permissions, and that the code for the permission check returns "allowed" only if the current principal really does have permission to access the file contents. This proof mostly boiled down to showing that the code implementing the access-control check in `read` matches the logical permission required by the specification.

Proving noninterference. Proving that SFSCQ provides confidentiality required us to prove three theorems. The first is that `write` implements the specification from Figure 11. This shows that SFSCQ will treat data passed by an application to `write` as confidential. The second is that system calls satisfy `ret_noninterference`. This shows that an adversary cannot use any of SFSCQ's system calls to learn confidential data. The final is that all system calls satisfy `state_noninterference`. This shows that SFSCQ will not indirectly leak a user's data when the user invokes an otherwise-benign system call. Taken together, these theorems allow an application to formally conclude that its data remains private, as we show in §9.

Proving `ret_noninterference` was the easiest, using DISKSEC's theorem from Figure 9. All SFSCQ procedures are proven to be unseal safe, so no further proof effort is required.

Proving `state_noninterference` was simple for all system calls except `read`, because those system calls satisfy `unseal_public`, allowing us to apply DISKSEC's theorem from Figure 10. For `read`, we structured the system-call implementation in two parts: a `read_helper`, which returns the handle to the data read from the file, and a wrapper around `read_helper` that unseals the data and returns it to the user. `read_helper` is `unseal_public`, allowing us to apply DISKSEC's theorem from Figure 10. The wrapper required a manual proof, but the proof was short since the wrapper is two lines of code.

Finally, to prove that `write` meets its confidentiality specification, we similarly split `write` into a wrapper and a `write_helper`. The wrapper's job is to seal all input data and pass the handles to `write_helper`. Much as with `read`, this reduced the proof effort to just the wrapper.

8 Implementation

We implemented DISKSEC by extending the CHL framework from FSCQ [12]. The changes involved modifying the model of the disk to keep track of logical permissions, adding primitive operations to seal and unseal blocks, and changing the execution semantics to keep track of unseal permissions, as shown in Figure 6. We also changed the meaning of Hoare-logic specifications to require either unseal-safety or the stronger `unseal_public` notion. The source code of DISKSEC and SFSCQ is publicly available at https://github.com/mit-pdos/fscq.

We developed SFSCQ by modifying the DFSCQ file system [13], making the changes described in §7. In particular, as mentioned in §7.2, we switched from DFSCQ's checksum-based write-ahead log to a two-barrier-based log in SFSCQ (which is also the default for Linux ext4). SFSCQ retains all other optimizations from DFSCQ, including log-bypass writes, deferred commit, etc (with proofs). As with DFSCQ, we produce executable code by extracting the Coq implementation to Haskell and running it on top of FUSE. To erase the block sealing and unsealing operations at runtime, DISKSEC uses the `Extract Constant` command in Coq to represent DISKSEC's handles using the raw blocks themselves, and it implements `Seal` and `Unseal` as no-ops.

We built two versions of DISKSEC and SFSCQ. The first version is fully proven, but lacks support for changing permissions on an existing file (i.e., changing the permissions on a file would require copying the file's data into a new file with the new permissions), and lacks support for randomness oracles. The second version extends the first version with support for randomness oracles and dynamic permissions. These changes caused existing proofs to break, and a few of them have not been repaired. See the source code for details.

The DISKSEC approach worked reasonably well for SFSCQ because the underlying FSCQ file system does not unseal user data unless the user explicitly reads it. The one exception was in the checksum-based write-ahead log, as mentioned above. Other file system features that look at file contents might also be a challenge for DISKSEC, such as proactive checksum verification of file contents, de-duplication, storing small file contents in the inode itself, etc.

9 Evaluation

This section experimentally answers the following questions:

- Are SFSCQ's specifications trustworthy? That is, are SFSCQ's theorems sufficient for applications to prove confidentiality of their own data? What assumptions do these proofs rely on?
- How much effort was required to develop DISKSEC, and to use DISKSEC to prove the security of SFSCQ?
- How much runtime overhead does DISKSEC's approach impose in SFSCQ?

9.1 Specification trustworthiness

To evaluate the trustworthiness of SFSCQ's specifications, we performed several analyses.

End-to-end application confidentiality. To demonstrate that SFSCQ's specifications capture confidentiality in a useful way, we developed a simple application on top of SFSCQ that copies a file, wrote a confidentiality specification for this application (namely, that the application does not leak the data of the copied file), and proved it. This application tests two aspects of SFSCQ's specs. The first is, does SFSCQ's specification actually guarantee confidentiality? The second has to do with SFSCQ's discretionary access control model: can application developers demonstrate that they are not inadvertently leaking data, despite having the discretion to do so?

We were able to prove the correctness and security of our implementation of `cp`. This suggests that SFSCQ's specifications capture sufficient information for `cp` to conclude that its data remains confidential, and that it is possible for application developers to show that they do not abuse their discretionary privileges by leaking data.

Bug case study. To evaluate whether SFSCQ's specifications would eliminate real security bugs, we qualitatively analyzed the bugs presented in §3 to determine whether SFSCQ's theorems preclude the possibility of that bug. Figure 13 shows the results. Functional correctness theorems preclude the possibility of integrity bugs. DISKSEC state noninterference precludes the possibility of all confidentiality bugs in our study. No bugs were prevented by return-value noninterference, because return-value noninterference captures a particularly simple kind of bug, such as the file system forgetting to check the ACL on `open()`. No file-system developers made this mistake in our study. Nonetheless, return noninterference is important for completeness of SFSCQ's theorems. Overall, the results demonstrate that SFSCQ's theorems preclude the possibility of all studied bugs.

Description	Theorem violated
anyone can change POSIX ACLs [3, 5, 8]	state NI
reiserfs permissions can be changed by writing to hidden file [2]	state NI
truncated data can be accessed [7]	state NI
crash can expose deleted data in ext4 [9]	state NI
crash can expose data in ext4 [22]	state NI
can overwrite append-only file in ext4, btrfs [4, 6]	integrity
can overwrite arbitrary files in ext4 [1]	integrity

Figure 13: Security bugs in Linux file systems and which SFSCQ theorem precludes them.

Trusted computing base. SFSCQ assumes the correctness of several components. SFSCQ assumes that Coq's proof checking kernel is correct, because it verifies SFSCQ's proofs. SFSCQ assumes that the Haskell runtime and support libraries (and the underlying Linux kernel) do not have bugs, since SFSCQ generates executable code through extraction to Haskell. SFSCQ assumes that DISKSEC's model of the disk is accurate. In particular, all non-determinism in DISKSEC's execution semantics must be "realizable," in the sense that it is actually possible for an execution to observe all specified non-determinism (e.g., crashing at any point), and this non-determinism must be independent of confidential data. All proofs in DISKSEC and SFSCQ are checked by Coq.

9.2 Effort

To understand how much effort was required to verify DISKSEC and SFSCQ, we compared SFSCQ to the implementation of DFSCQ on which SFSCQ is based. Figure 14 shows the results (counting the sum of lines removed and lines added), breaking down the differences into several categories. The core infrastructure, including improvements to DFSCQ's CHL, amounted to around 9,300 lines. We made significant changes to DFSCQ to develop SFSCQ, but many of these changes were mechanical fixes to proofs to address small changes. In addition, using DISKSEC in SFSCQ required around 1,900 lines of new code and proofs. Porting DFSCQ to the first version of DISKSEC (without support for changing permissions) took one author about 3 months, and another 2 months to mostly finish support for permission changes.

Component	Changes to DFSCQ
DISKSEC	9,283
DFSCQ proof fixes	$-10,471$, $+26,433$ (36,094 total)
SFSCQ impl. and proofs	1,837
Verified `cp` application	407

Figure 14: Lines of code change required to implement DISKSEC and apply it to build SFSCQ. Counts measure the diff between DFSCQ and SFSCQ.

9.3 Performance

We expect that the performance overhead of DISKSEC is nearly zero, because most of its code changes (such as handles, sealing, and unsealing) are eliminated in the process of generating executable code. (All of the Seal and Unseal operations turn into `return` statements.) The only exception is checking permissions when reading data from a file; the original DFSCQ implementation had no permission checks, which we added in SFSCQ.

To check that DISKSEC introduces almost no overhead, we used two microbenchmarks (LFS smallfile and largefile benchmarks [31] as modified by DFSCQ [13]). As a baseline, we compare with two versions of DFSCQ, on which SFSCQ is based. The first is unmodified DFSCQ. The second is a version of DFSCQ with a two-disk-barrier write-ahead log (instead of its default checksum-based log). This matches the modification we made to SFSCQ, as mentioned in §7.2. For comparison with other file systems, such as Linux ext4, we refer the reader to the detailed evaluation in the DFSCQ paper [13: §7.4].

Figure 15 shows the results, which confirm that SFSCQ performs nearly identically to DFSCQ in the same logging configuration. The use of a two-disk-barrier write-ahead log incurs some performance overhead for smallfile; largefile performance is not impacted because its file data writes bypass the log.

Filesystem	smallfile	largefile
DFSCQ	446 files/s	108 MB/s
DFSCQ (no checksums)	295 files/s	109 MB/s
SFSCQ	299 files/s	100 MB/s

Figure 15: Benchmarks showing performance of SFSCQ compared to DFSCQ and a version of DFSCQ with a comparable logging implementation. Numbers shown are the median of 30 runs.

10 Conclusion

SFSCQ is the first file system with a machine-checked proof of security. DISKSEC enabled us to specify and prove SFSCQ's confidentiality with modest effort. DISKSEC's key techniques are the use of a sealed block abstraction, as well as the notion of data noninterference as the top-level theorem statement, which is a good fit for discretionary file access control. Experimental evaluation shows that SFSCQ's theorems would preclude security bugs that have been found in real file systems, that SFSCQ's development effort was moderate, and that there is little performance impact of using DISKSEC.

Acknowledgments

Thanks to the PDOS group, the anonymous reviewers, and to our shepherd Jay Lorch, for improving this paper. This research was supported by NSF awards CNS-1563763 and CNS-1812522, and by Google.

References

[1] CVE-2009-4131, 2009. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-4131.

[2] CVE-2010-1146, 2010. http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2010-2066.

[3] CVE-2010-2017, 2010. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1641.

[4] CVE-2010-2066, 2010. http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2010-2066.

[5] CVE-2010-2017, 2010. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2071.

[6] CVE-2010-2537, 2010. http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2010-2066.

[7] CVE-2015-8374, 2015. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8374.

[8] CVE-2016-1237, 2016. http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2010-2066.

[9] CVE-2017-7495, 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7495.

[10] J. Barkley. Introduction to POSIX security, Oct. 1994. http://ftp.gnome.org/mirror/archive/ftp.sunet.se/pub/security/docs/nistpubs/800-7/node18.html.

[11] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001.

[12] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–37, Monterey, CA, Oct. 2015.

[13] H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 270–286, Shanghai, China, Oct. 2017.

[14] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 105–118, Vancouver, Canada, Oct. 2010.

[15] D. Costanzo, Z. Shao, and R. Gu. End-to-end verification of information-flow security for C and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 648–664, Santa Barbara, CA, June 2016.

[16] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proceedings of the 22nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 350–357, Nashville, TN, Oct. 1981.

[17] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 287–305, Shanghai, China, Oct. 2017.

[18] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL)*, pages 186–197, Venice, Italy, Jan. 2004.

[19] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 3rd IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, Apr. 1982.

[20] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, Broomfield, CO, Oct. 2014.

[21] J. Hsu. *Probabilistic Couplings for Probabilistic Reasoning*. PhD thesis, University of Pennsylvania, Nov. 2017.

[22] J. Kara. [PATCH] ext4: Forbid journal_async_commit in data=ordered mode. http://permalink.gmane.org/gmane.comp.file-systems.ext4/46977, Nov. 2014.

[23] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM*

Transactions on Computer Systems, 32(1):2:1–70, Feb. 2014.

[24] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 158–170, Long Beach, CA, Jan. 2005.

[25] J. McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–57, Jan. 1992.

[26] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: from general purpose to a proof of information flow enforcement. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013.

[27] A. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 129–147, Saint-Malo, France, Oct. 1997.

[28] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Computer Systems*, 9(4):410–442, Oct. 2000.

[29] A. Popescu, J. Hölzl, and T. Nipkow. Proving concurrent noninterference. In *Proceedings of the 2nd International Conference on Certified Programs and Proofs (CPP)*, pages 109–125, Kyoto, Japan, Dec. 2012.

[30] A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings of the 16th IEEE Symposium on Security and Privacy*, pages 114–127, Oakland, CA, May 1995.

[31] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, Pacific Grove, CA, Oct. 1991.

[32] J. M. Rushby. Proof of separability: A verification technique for a class of security kernels. In *Proceedings of the 5th International Symposium on Programming*, pages 352–367, Turin, Italy, Apr. 1982.

[33] B. Schmidt, S. Meier, C. Cremers, and D. Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, pages 78–94, Cambridge, MA, June 2012.

[34] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Proceedings of the 12th International Static Analysis Symposium (SAS)*, pages 352–367, London, UK, Sept. 2005.

[35] The Coq Development Team. *The Coq Proof Assistant, version 8.8.0*, Apr. 2018. URL https://doi.org/10.5281/zenodo.1219885.

[36] T. Ts'o. [PATCH] ext4: remove calls to ext4_jbd2_file_inode() from delalloc write path. http://lists.openwall.net/linux-ext4/2012/11/16/9, Nov. 2012.

[37] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL)*, Philadelphia, PA, Jan. 2012.

[38] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong. Precise, dynamic information flow for database-backed applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 631–647, Santa Barbara, CA, June 2016.

[39] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 291–304, Big Sky, MT, Oct. 2009.

[40] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.