Verifying Confidentiality Under Nondeterminism for Storage Systems

by

Atalay Mert Ileri

B.S., Bilkent University (2014)
S.M., Massachusetts Institute of Technology (2016)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2023

Author .....
Department of Electrical Engineering and Computer Science
January 25, 2023

Certified by .....
M. Frans Kaashoek
Charles Piper Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Nickolai Zeldovich
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Verifying Confidentiality Under Nondeterminism for Storage Systems

by

Atalay Mert Ileri

Submitted to the Department of Electrical Engineering and Computer Science
on January 25, 2023, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Storage systems must often store confidential data for their users. It is important to ensure that confidentiality of the stored data is maintained in the presence of bugs and malicious adversaries. This thesis tackles this problem using formal verification, a technique that involves proving a software system always satisfy certain requirements.

There are numerous challenges in specifying what it means a system being confidential and proving that a system satisfies that specification: nondeterministic behavior, indirect leakage of the data, system complexity, and others. Nondeterminism in particular creates unique challenges by making probabilistic leakage possible. This dissertation introduces the following to address these challenges:

- Two novel confidentiality specifications for storage systems with nondeterministic behavior: *data nonleakage* and *relatively deterministic noninfluence*. Both definitions accommodate discretionary access control and intentional disclosure of the system metadata.

- Two techniques accompanying these specifications: *sealed blocks* and *nondeterminism oracles*. These techniques addressed the challenges encountered in proving the confidentiality of the systems, and also reduced the proof effort required for said proofs. These techniques are formalized and implemented in two frameworks: DiskSec and ConFrm. Both frameworks contain metatheory to help the developer to prove that their implementation satisfies the specification.

- The first confidential, crash-safe, and formally verified file systems with machine-checkable proofs: SFSCQ and ConFs. SFSCQ uses data nonleakage and ConFs

uses relatively deterministic noninfluence as their confidentiality specifications. Both are implemented and verified in Coq.

An evaluation shows relatively deterministic noninfluence has 9.2x proof overhead per line of implementation code. Experiments with multiple benchmarks show that our systems perform better compared to FSCQ verified file system but worse compared to ext4 file system.

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Nickolai Zeldovich
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

First, I want to thank my advisors Frans Kaashoek and Nickolai Zeldovich for introducing the world of formal verification to me, and to Adam Chlipala for his valuable guidance and contributions to the foundations of this thesis. Second, I am incredibly appreciative of my parents Nalan and Ali Ileri for supporting me throughout my entire life, sacrificing their time and sleep so I could get to this place in my life. They always trusted me in my choices and let me forge my own path. I am eternally grateful for this. Finally, my girlfriend Jade Eryn Rose Lovett, who always kept me sane, even when things got crazy. She helped me stay focused and motivated, and was my light in the darkest of times. I couldn't ask for a better partner.

I want to thank all the past and present members of PDOS for accepting me without question as I kept disappearing and reappearing in the office. I am especially thankful for my board game group: Nathan Beckmann, Tej Chajed, Jon Chang, Jon Gjengset, and Max Wolf, for countless hours of entertainment and companionship. Also, thanks to my roommates Akshay Agarwal and Aviv Adler for all the fun conversations we had. I am extraordinarily lucky to have Hoon Cho and Albert Kim as my friends, who stayed by my side even when I was not that pleasant to be around.

Last but not least, I want to thank my best friend Ovunc Metin, whom I lost to cancer during my PhD, for unconditionally supporting me through the worst years of my life and helping me find my way. I miss you dearly.

Contents

List of Figures

Chapter 1

Introduction

Storage systems are an integral part of many software systems we use day-to-day. Users expect their data stored in such systems to stay secret. In this thesis, we investigate the challenges surrounding the verification of storage system safety under nondeterminism and present solutions to those challenges.

1.1 Motivation

There are many factors that can undermine the confidentiality of a storage system. Due to their prevalence among all the software, we will focus on the software bugs.

1.1.1 Software Bugs

Storage systems have had numerous bugs that allowed for data disclosure: we list several such bugs in Figure 1-1 and describe some of these bugs in more detail below. All the presented bugs are obtained from CVE database [1].

Bug description	**Filesystem(s)**	**year**
Access to deleted files' data	fuse-exFAT [8]	2022
Data leak via unaligned file lengths	xfs [7]	2021
Can set incorrect permissions on new filesystem objects	nfs [4]	2020
Data leak through uninitialized memory	ext4 [2]	2019
A local user may create files that belongs to another user	xfs [6]	2021
A local user may be able to read arbitrary files	APFS [5]	2021
Information leak due to permission bypass	Android fs [3]	2019

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

exFAT allows users to grow a file in stream extension beyond its valid length. This operation immediately allocates the blocks but they do not get zeroed-out until the file is closed. exFAT specification states that if the contents beyond the valid length is read, null bytes should be returned. An implementation of exFAT in Linux contains a bug that returns the newly allocated blocks' residual contents instead of null bytes [8]. In another file system, such a result can be achieved by setting file length to a value that is unaligned with the block boundary then extending it with `ioctl` system call [7].

Some of the bugs are due to improper permission setting and handling. Such bugs allow attackers to access confidential data of other users by bypassing access control mechanisms in the storage systems. Both Android fs and and APFS had bugs that can leak confidential data due to faulty permission checking [5, 3]. xfs file system allows users to create files that belong to other users and groups that are writable by the creator [6].

1.2 Verification and Threat Model

This dissertation looks at how we could avoid these bugs using software verification by providing precise descriptions of how such a system should behave to ensure confidentiality of the stored data. These descriptions are called confidentiality specifications. A confidentiality specification of a system both forces a system to maintain certain properties, and informs users about the safety guarantees that system provides. Figure 1-2 illustrates how formal verification works on a high level.

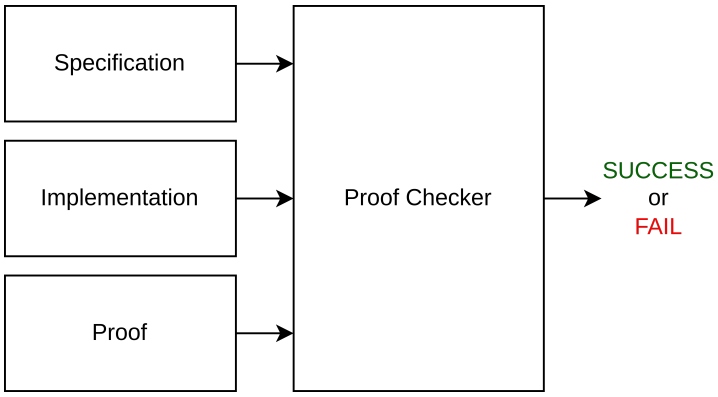

Figure 1-2: Formal verification workflow.

From the perspective of verification, we would like to have confidence that the storage system is secure purely based on the storage system's confidentiality specification. This can be achieved by adopting a strong threat model. To reach our goal, we take an unconventional approach: treating the developer of the storage system with an adversarial mindset. More specifically, we assume the developer may be malicious and intentionally deliver an implementation that can be exploited in the future, which we call an *adversarial implementation*. This subsumes all possible bugs that a well-meaning but error-prone developer might introduce into the implementation as well as any implementation that is designed to be exploited.

As a result, our threat model is that the adversary both develops the storage

system and runs an adversarial application on top of the storage system in an attempt to obtain confidential data. However, the adversary must provide a proof that their implementation meets our confidentiality specification. The potential victim runs on top of the same storage 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 confidentiality specification is so strong that it prevents leaks even when the storage system developer is colluding with adversarial processes running on top of the storage system.

Our threat model is focused on proving that the storage system implementation has no confidentiality bugs, rather than proving the absence of bugs in the environment outside of the storage system. Thus, we assume that our model of how the storage system implementation executes is correct. That is, we are not concerned with bugs in unverified software or hardware outside of the storage system, or users mounting malicious disk images. We do prove that an initialization process 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.

1.3 Challenges

In addition to the standard challenges like confidentiality being a two-execution property and the complexity of storage systems, nondeterminism poses unique challenges in both specifying and proving confidentiality. In this section we will focus on two challenges that arises from two types of nondeterminism: nondeterminism in the specification and nondeterminism in the implementation.

1. Nondeterministic specifications. A challenge in proving confidentiality lies in the fact that many specifications, including those in a storage system, are nondeterministic. This nondeterminism is necessary to abstract implementation details so it can be read, understood, and reviewed by humans. A deterministic specification runs the risk of being overly verbose to a degree that the specification is the implementation, obscuring the important parts of the specification, making it difficult to read and comprehend, and defeating one of the key purposes of writing a specification.

To demonstrate the possible pitfalls of writing a nondeterministic specification that can prevent malicious implementations, consider the `create` operation from our confidential file system ConFs. `create` takes an owner and creates an empty file owned by the provided owner. Upon successful completion, it returns the inode number that identifies the file. A natural correctness specification for its return value could be "`create` returns the index of a previously unused inode that now corresponds to the newly created file." If we were only interested in functional correctness, this would be an acceptable specification. However, there is a substantial confidentiality problem associated with it. `create` is allowed to return any inode number as long as it was unused at the time of the call. A malicious implementation of the file system can take advantage of this nondeterminism in the specification to pick the returned inode number and subsequently leak confidential data, e.g., last byte of the return value being equal to the first byte of a block that belongs to another user.

Even the nondeterminism associated with the state of the disk after a crash can be taken advantage of by an adversarial storage system implementation to leak data. For instance, a high-performance file system specification may allow 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 only have happened if the confidential bit was zero. This, in turn, can allow the adversary to infer confidential bits.

2. Nondeterminism in implementation: secret-dependent outcome probabilities. Another challenge arises because an adversary may infer the secret information stored in the system if the distribution of the outcome of a function is dependent on a secret. Such a vulnerability may exist even when the possibility of observing a particular return value is independent from the secret. A simple example of this challenge can be seen in Figure 1-3.

```
if (get_random_bit() == 1)
  return secret_bit
else
  return get_random_bit()
```

Figure 1-3: Example of a program that leaks information via return-value probabilities.

If we assume that `get_random_bit()` outputs 0 or 1 with equal probability, then this code leaks the secret bit 50% of the time and outputs a random bit 50% of the time. It is also important to note that it can output 0 or 1 independently of the secret value. This way, any (state, return) pair represents a successful execution. Therefore, by observing just a single return value, an adversary cannot infer the value of the secret bit with 100% certainty. However, the probabilities of the output values in Table 1-4 show that they correlate with the value of the secret bit.

Secret bit	Output % 0	Output % 1
0	75%	25%
1	25%	75%

Figure 1-4: Distributions of return values for each state.

Any adversary who is aware of this behavior can infer the value of the secret bit with certain confidence.

These types of vulnerabilities are not limited to usage of randomization. They also manifest themselves when there are other random events that can affect the behavior of the system. For example, in storage systems, source of randomness comes from the possibility of a system crash at any point of the execution, not from an operation with a randomised behavior. In real world, at any point in time, there is a certain probability of system crashing. Moreover, this probability depends on many complex factors that make it hard to estimate precisely. When modeling the system, this hardship can be circumvented by modeling crashes as nondeterministic events, instead of random ones. However, such a modeling choice does not change the fact that there is an unknown probability associated with the materialization of each nondeterministic event in the model. Since the probabilities are unknown, a technique that addresses these type vulnerabilities should work no matter what the actual distribution may be.

Other challenges. There are other challenges that are standard in the literature. The first challenge is that confidentiality is a two-execution property, also known as a hyperproperty, while functional correctness is a one-execution property [16]. A two-execution property is a property that is defined over two executions of a program. This makes confidentiality proofs harder and more complex compared to functional correctness proofs, which include only one execution.

The second challenge is the system complexity. Storage systems are complex software. For instance, consists of approximately 50,000 lines of code. Proving properties of such software require techniques to handle the complexity.

1.4 State of the Art: Noninfluence

The de facto approach to specifying confidentiality is through information-flow control. An information-flow-control policy restricts and regulates transfer of information between different parts of the system as well as to its users. There are wide variety of information-flow control specifications that formalizes different notions of security [31]. One that is of particular interest is *noninfluence* [38].

Noninfluence is introduced as a comprehensive specification to ensure the confidentiality of a system that both processes and stores confidential data. It is a specification that regulates both the data stored in the system, and the data newly introduced to the system. Noninfluence achieves this goal via two separate properties, one for each type of confidential data: *nonleakage* and *noninterference*.

Nonleakage ensures the confidentiality of the data stored in the system. Informally, nonleakage states that "if two states of the system differ only in the aspects that are not visible to a user, then such a user shouldn't be able to distinguish between those states if he runs the **exact same operation** from both states". This is a necessary property for ensuring confidentiality of a storage system and is a part of our specifications as well. However, it is insufficient because it doesn't restrict system's behavior regarding the different actions the same user takes. For example, it doesn't dictate the system behavior when a user writes 0 or 1 to a location.

Noninterference regulates the behavior regarding the addition of the new data. Informally, it states that "the users of a system should observe the same behavior from the system regardless of other users' actions in the system". This restriction is

not suitable for many storage systems. The reason is that it prohibits any transfer of information, both direct and indirect. However, storage systems sometimes expose effects of the users indirectly via metadata or some public information of the system like available free space. For example, in a file system, if a user creates a file and writes to it, other users of the file system may see that he created a file, even though they may not see the file's contents. For many applications, this behavior is completely acceptable, even though noninterference prohibits it. Because of this, storage systems require a more flexible confidentiality specification for introducing new data to the system.

1.5 Solution Approaches

This thesis explores two different approaches to addressing the challenges mentioned above: (1) sealed blocks and data nonleakage, and (2) nondeterminism oracles, abstractions and refinements. These approaches complement each other, and together offer a solution to each of the challenges. We briefly explain all of the approaches below. We present more detailed explanations in the following chapters.

1.5.1 Data Nonleakage and Sealed Blocks

Data nonleakage and sealed blocks are our first attempt at addressing the challenges. Our implementation of these approaches addresses the first challenge but also reveals the intricacies of specifying and proving confidentiality of storage systems, most notably the second challenge. Sealed blocks and data nonleakage are briefly explained below.

Data Nonleakage We define data nonleakage as a solution to traditional noninfluence definition being too strong for storage-systems confidentiality. Data nonleakage

relaxes the requirement of any sequence of actions to be noninterfering by allowing the specialization of specification to two particular sequences. For example, it is possible to state that `write` should be confidential but `create` operation is public. As a result, a system has a set of specifications, one for each pair of actions that are not allowed to interfere. Data nonleakage will be explained more in-depth in Chapter 3.

Sealed Blocks Sealed blocks is a proof technique based on the idea that if a program's behavior does not depend on confidential data, then it is not possible for it to leak confidential data, since it will behave the same no matter what the data is. Sealed blocks achieve this via two mechanisms: (1) providing a way to handle data without accessing its contents, and (2) enforcing access control when the contents are accessed.

The first mechanism is an operation that we call *sealing*. Sealing takes a piece of data and its owner, and turns it into an abstract object that does not provide any functionality other than *unsealing* it. This way, showing that a program's behavior does not depend on confidential data reduces to showing that it only handles the sealed data, which can be done by a static analysis of the program.

However, a file system has code that must inspect the contents of the data, especially for systems that uses on-disk data structures. An example of this is the bitmap of an allocator. To be able to allocate or free any resources, it has to inspect and modify the contents of a bitmap. The second mechanism ensures the program that uses the file system does not access any data that it is not permitted to access. Since sealed data require unsealing to access the contents, each unseal operation can be detected and recorded during the execution, creating an *unseal trace*. With unseal traces, showing that a program does not leak confidential data reduces to showing that its unseal trace only contains unseals that it is permitted to perform.

Overview and Limitations The first approach – *data nonleakage and sealed blocks* – addresses the two-execution, nondeterministic specifications, and partially the system complexity challenges. We used the sealed-blocks approach to address the two-execution challenge, that confidentiality proofs are harder and more complex than integrity proofs, by turning two-execution proofs into trace properties, which require one-execution proofs. Sealed blocks reduce the two-execution property to a trace property, which is a one-execution property. Data nonleakage allows nondeterministic specifications while preventing exploitation of such nondeterminism by implementations. Both of them reduces required proof effort significantly but don't allow certain implementation optimizations.

1.5.2 Nondeterminism Oracles, Abstractions and Refinements

The second approach addresses challenges that are left unresolved by the first approach. Nondeterminism oracles, abstractions and refinements incorporate the insights we obtained from our first attempt to obtain stronger specifications.

Nondeterminism Oracles The second approach addresses challenge 2 by using a model that allows reasoning about each possible sequence of nondeterministic events individually. Such a model enables confidentiality specifications that can require existence of a matching execution with same return-value for each such sequence. If the matching exists for each sequence of nondeterministic events, then the probability of materializing of an execution that leads to a particular return value will be equal for equivalent states.

We achieved this via a technique we call *nondeterminism oracles*. A nondeterminism oracle is an abstract object where whenever the execution needs to make a nondeterministic choice, it "asks" oracle and chooses based on the oracle's answer. Therefore, different oracles lead to different executions of the same program. Simi-

larly, for a fixed oracle, execution becomes deterministic. This allows us to enforce return-value probabilities to be the same for equivalent states by requiring return-value equality for each oracle.

Limitations There are two important limitations regarding the nondeterminism oracle approach. First, the oracle approach prohibits some implementations that are secure for a specific distribution but may be insecure for other distributions. Figure 1-5 shows an implementation that is secure only if the generated bit is uniformly random. Oracle approach prohibits the implementation in the figure because implementation is not secure for any other distribution. It is important to note that this limitation does not weaken the guarantees of the oracle approach, but renders certain secure implementations unviable.

```
if (get_random_bit() == 1)
  return secret_bit
else
  return negate(secret_bit)
```

Figure 1-5: Example of a program that is secure if the generated bit is uniformly random.

Second, oracle approach assumes that the probability distribution of nondeterministic events in real life is independent of the secret data stored in the system. For example, in a system, if writing a block of zeroes to the disk makes it more likely to crash than writing any other value, then oracle approach is not a fitting model for that system.

Abstractions and Refinement To tackle the complexity of file systems, we use abstractions. Being able to define abstractions at desired points helps contain and compartmentalize the complexity in the system and reduce the proof complexity as

implementations stack on top of each other. In our work, we use refinement for connecting abstractions to implementations where correctness of the refinement is established by simulation proofs. Refinements and simulations are well-established techniques in the literature.

One complication is that it is known that noninterference is not necessarily preserved under simulations [23]. However, nondeterminism oracles lead to a confidentiality specification that is preserved under simulations given that refinement satisfies a certain property. This condition will be explained in more detail in the following chapters. This result is one of the contributions of this thesis.

The second approach solves the challenges the first approach falls short on. Nondeterminism oracles enable reasoning about individual nondeterministic events. This ability leads to a stricter confidentiality specification that prevents challenge 2, while still being flexible enough to prevent overly verbose specifications and avoid implementation restrictions. Abstractions and refinements encapsulate the complexity in small components and allow complex systems to be implemented in a modular fashion while keeping proof complexity in check.

1.6 Implementation

We implement our solution approaches in two frameworks: sealed blocks and data nonleakage are implemented in DiskSec, and nondeterminism oracles, abstractions and refinements are implemented in ConFrm. Both frameworks include storage system models and operational semantics of the operations on them as well as formalization of their respective specifications.

We implement two file systems using the frameworks: SFSCQ and ConFs. SFSCQ is implemented by porting DFSCQ verified file system to DiskSec and modifying the implementation to add multi-user support and access control mechanisms. ConFs

is implemented from scratch using ConFrm. Its implementation consists of multiple abstraction levels and takes advantage of ConFrm's abstraction mechanisms. Both systems use their respective frameworks' confidentiality specifications. These are the first formally verified confidential software systems with nondeterminism.

All the frameworks and systems are implemented in the Coq proof assistant. They are extracted to Haskell to produce runnable code. Details of these frameworks and file systems will be explained in their respective chapters. Source code of Disksec & SFSCQ can be found in `https://github.com/mit-pdos/fscq/tree/security`, and source code of ConFrm & ConFs can be found in `https://github.com/Atalay-Ileri/ConFrm`.

Limitations. The frameworks come with some limitations that impact their applicability. DiskSec's helper theorems apply only to implementations that don't compare confidential data. Therefore to take advantage of the full power of DiskSec, the implementation should avoid such comparisons as much as possible.

ConFrm has two important limitations. The first one is that it doesn't provide any additional support for proving confidentiality of the top abstraction level. This may lead to lengthy proofs if the top abstraction layer has a complex structure.

The second limitation is due to our design choices when implementing the nondeterminism oracles. ConFs instantiates ConFrm with a mixed embedding where disk and memory operations are deeply embedded and the rest is shallowly embedded [15]. Because of this embedding, ConFrm can only apply nondeterminism oracles to memory operations. As a consequence, the implementation in Figure 1-6 would be considered secure under such embedding.

```
if (secret_bit == 1) {
  // loop 1000 times
}
read_disk (a)
```

Figure 1-6: Example of a program that has secret-dependent crash probability.

Above implementation has a different crash probability depending on the secret bit. If the bit is 1, function it will take much longer for function to complete, therefore it will be more likely to crash during the function. Such dependence may leak the value of the bit. However, since the existence of the loop is transparent to ConFrm, it will fail to catch the leakage. This limitation can be overcame by using a fully deeply embedded language for the implementations.

Finally, both our file systems have simple access-control mechanisms and termination-insensitive specifications.

1.7 Contributions

There are two groups of contributions of this thesis where each one contains three components.

Data Nonleakage, DiskSec, and SFSCQ. The first group is our first attempt at tackling nondeterminism and consists of:

- **Data Nonleakage**, a confidentiality specification based on noninfluence that captures discretionary access control and deals with nondeterminism due to crashes.
- **Sealed blocks**, a proof approach that factors out reasoning about confidentiality from most of the storage-system code.

- **DiskSec**, a framework for specifying and proving confidentiality for storage systems that reduces proof effort by using the sealed-block approach.
- **SFSCQ**, the first file system with a machine-checked proof of confidentiality. SFSCQ uses data nonleakage as its specification, and DiskSec for proving its confidentiality.

RDNI, ConFrm, and ConFs. The second group improves upon the first group and includes:

- **RDNI**, a confidentiality specification based on noninfluence that incorporates oracles, reboot functions, and crash-reboot-recovery processes. RDNI provides stronger guarantees than data nonleakage under nondeterminism.
- **Nondeterminism oracles**, a modeling technique for reasoning about specific series of nondeterministic events.
- **A metatheory** for preservation of RDNI through abstractions, including a modified simulation definition that incorporates oracles, reboot functions, and crash-reboot-recovery processes.
- **ConFrm**, a framework for specifying and proving confidentiality of storage systems with RDNI specifications. ConFrm implements RDNI and its metatheory as well as the support for implementing systems as layers of abstractions, defining refinements, and proving simulations.
- **ConFs**, the first file system that uses checksum-based, encrypted logging with a machine-checked confidentiality proof. ConFs is implemented in ConFrm and uses RDNI as its confidentiality specification.

Finally, thesis has an evaluation that compares SFSCQ and ConFs to demonstrate the proof effort required and performance overheads of each approach.

Two groups of contributions complement each other. DiskSec is suitable for the systems that handle the confidential data without examining its contents and doesn't need intermediate abstraction layers . ConFrm is more suitable for the systems that process confidential data and multiple layers of abstractions.

1.8 Outline

Chapter 2 presents the related work in the literature. Chapter 3 explains proof ideas behind Disksec and its implementation details. Chapter 4 presents SFSCQ, the first formally verified file system with a confidentiality specification. Chapter 5 describes ConFrm, a confidentiality framework for implementing systems modularly and with strong guarantees regarding nondeterminism. Chapter 6 presents the structure and implementation of ConFs. Chapter 7 evaluates the performance of both file systems. Chapter 8 discusses some possible future work. Chapter 9 concludes the dissertation.

Chapter 2

Related Work

Our work builds on a diverse body of prior work. We will explain these works throughout this section.

Confidentiality properties. There is a significant body of work formalizing noninterference properties [22, 27, 28, 32, 33, 34]. DiskSec and ConFrm's efinitions build upon this existing work. Specifically, data nonleakage and relatively deterministic noninfluence can be thought of as a specialization of Oheimb's nonleakage and noninfluence, respectively [38]. One difference in our approach is that our specifications stop at the file-system API boundary; applications are not subject to our policies. This matches well the traditional discretionary access-control policies enforced by file systems.

Formalizing data nonleakage requires reasoning about two executions, since confidentiality is a two-safety property [37]. In this context, our contribution lies in a specification and proof style based on sealed blocks that helps us prove a data nonleakage two-safety property about the file system.

ConFrm's definition is different from its predecessors in how it treats nondeterminism in its formalism. ConFrm takes a more fine-grained approach in relating

nondeterministic executions by requiring a strong coupling between executions for each nondeterministic execution branch.

Machine-checked security in systems. Several prior projects have proven security (and specifically confidentiality) properties about their system implementations: seL4 [26, 28], CertiKOS [17], and Ironclad [24]. 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 via 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 [19] reasons about confidential data in an enclave and shows 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.

DiskSec supports nondeterministic crashes, discretionary access control, and shared data structures. However, it lacks support for branching on confidential data (e.g. hash-based logging), abstraction layers, and stronger crash guarantees .

ConFrm's contributions complement this line of work. ConFrm provides tools that allow developers to preserve confidentiality while creating abstraction layers. However, it uses a specific confidentiality definition. Even though the definition can

be customized via defining different state-equivalence relations, it may not express an arbitrary confidentiality specification.

Information-flow and type systems. Another approach to ensuring confidentiality involves relying on type systems. An advantage of this approach is that type checking can be automated to reduce proof load for the developer.

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 [30, 29] or the UrFlow analysis [14], have been developed to reason about information-flow properties of application code. UrFlow is specialized for the database backed web applications and uses a querying language to define the policies. Jif's analyzer would be hard to use for reasoning about dynamic data structures inside of a file system (such as a write-ahead log or a buffer cache) that contain data from different users.

Dynamic tools, such as Jeeves and Jacqueline [40, 39] and Resin [41], deal with dynamic data structures but require sophisticated and expensive runtime enforcement mechanisms. DiskSec and ConFrm avoid the overhead of runtime enforcement and an additional trusted runtime checker.

SeLoc [20] uses double weakest preconditions to prove noninterference for fine-grained concurrent programs. It is built on top of IRIS [25], a separation logic-based framework that proves correctness of fine-grained concurrent programs. It provides a confidentiality-ensuring type system and a wide array of tools to the developers. However, employing SeLoc requires using IRIS, which adds a substantial entry barrier. Conversely, both Disksec and ConFrm are standalone and lightweight but do not offer the full array of tools SeLoc offers.

Formalizing file-system security. Prior work has extensively studied the security guarantees provided by file systems, both formally and informally [9]. 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 in DiskSec is related to the techniques introduced to reason about cryptographic protocols. Many cryptographic-protocol proofs are done in the Dolev-Yao model of perfect cryptography [18]. 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 [35] build on it. HACL* [42] 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.

Sequential composability and confidentiality-preserving refinements. Since it is known that traditional noninterference is not preserved in simulation-based refinements, there is a body of work that tries to identify the conditions that make refinements noninterference-preserving.

Sun et. al. [36] proposes two confidentiality properties for interface automata: SIR-GNNI and RRNI. Both properties are based on refinements and defined relative to arbitrary security lattices. They also provide sufficient conditions that make SIR-GNNI and RRNI sequentially compositional.

Baumann et. al. [10] formulates noninterference as an epistemic logic over trace sets. They define ignorance-preserving refinements and prove that it is a sufficient condition to preserve the noninterference of abstraction. They also show that ignorance-

preserving refinements are not compositional w.r.t. sequential composition. They propose another class of refinements called "relational refinements", which are sequentially compositional.

ConFrm's definition is also sequentially compositional. Our suggested property differs from existing work in two points. First, it provides stronger guarantees for nondeterministic executions. Second, it supports reasoning about crashes and recovery of the storage system.

Chapter 3

DiskSec

3.1 Specification: Data Nonleakage

To capture the notion of confidentiality in a file system, DiskSec defines the notion of *data nonleakage*. Loosely speaking, data nonleakage states that two executions are indistinguishable with respect to specific confidential data (e.g., the contents of a file). Data nonleakage 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 nonleakage 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.

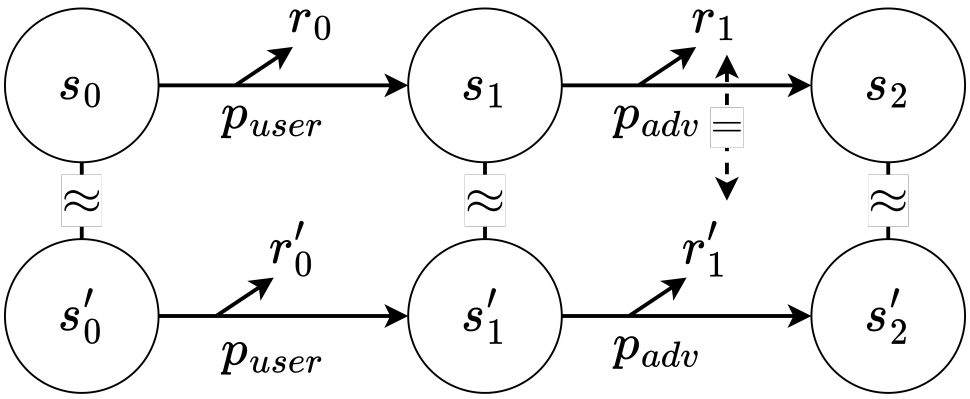

Figure 3-1: Overview of DiskSec's approach to reasoning about confidentiality.

Two-safety formulation. DiskSec formulates data nonleakage in terms of two-safety, as shown in Figure 3-1. Specifically, data nonleakage considers two executions that run the same code but start from different states. In Figure 3-1, the executions are shown as horizontal transitions between states, indicated by the gray outlines. An execution consists 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 3-1 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 $\mathit{equivalent}_{\text{adv}}$, to indicate that they are equivalent with respect to the adversary. This equivalence is indicated by the squiggly line in Figure 3-1. The essence of data nonleakage 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 nonleakage consists of two requirements that is based on the

two parts of noninfluence. The first is *state nonleakage*, which requires that after every transition, the resulting states remain $equivalent_{\text{adv}}$. This is indicated in Figure 3-1 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.

The second requirement is *return-value nonleakage*, which requires that transitions by the adversary return exactly the same values in both executions. For example, Figure 3-1 shows that the adversary's p_{adv} returns r_1 in the top execution and r'_1 in the bottom execution. Return-value nonleakage 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.

Capturing file-system security. Achieving the two requirements from data nonleakage ensures that the adversary cannot obtain confidential data from the file system. This is because state nonleakage maintains $equivalence_{\text{adv}}$ regardless of what the adversary does, and any attempts by the adversary to observe information will produce identical results, based on return-value nonleakage, because they run in $equivalent_{\text{adv}}$ states.

The discretionary nature of data nonleakage shows up in the fact that legitimate users can obtain different results depending on the confidential data. For example, in Figure 3-1, 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 nonleakage does not prevent this, by design, to model the standard discretionary access control in a POSIX file system.

Defining return-value nonleakage. Figure 3-2 presents DiskSec's definition of return-value nonleakage, in a simplified notation. This definition relies on the definition of `exec`, which describes how procedures execute. `exec` relates the procedure that is executing (`p`), the principal on whose behalf `p` is running (`u`), and the starting state (`st0`) to an 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 Section 3.2. 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_nonleakage (p: proc T) :=
  ∀ u st0 st0' st1 ret tr0,
    equivalent_for_principal u st0 st1 →
    exec p u st0 (Finished st0' ret, tr0) →

    ∃ st1' ret' tr1,
    exec p u st1 (Finished st1' ret', tr1) ∧
    ret' = ret
```

Figure 3-2: Definition of return-value nonleakage, capturing that return values do not leak other users' confidential data.

The definition of return-value nonleakage 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_nonleakage` 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 nonleakage. Figure 3-3 presents DiskSec's definition of state nonleakage, which complements return-value nonleakage. This definition helps DiskSec deal with the indirect-disclosure challenge from Section 1.3. 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`.

```
Definition equiv_state_for_principal u res0 res1 :=
    ∃ st0 st1,
      equivalent_for_principal u st0 st1 ∧
      (res0 = Crashed st0 ∧ res1 = Crashed st1 ∨
      ∃ v0 v1,
        res0 = Finished st0 v0 ∧
        res1 = Finished st1 v1).

Definition state_nonleakage (p: proc T) :=
  ∀ viewer caller st0 res0 tr0 st1,
    equivalent_for_principal viewer st0 st1 →
    exec p caller st0 (res0, tr0) →

    ∃ res1 tr1,
      exec p caller st1 (res1, tr1) ∧
      equiv_state_for_principal viewer res0 res1.
```

Figure 3-3: Definition of state nonleakage, capturing that `caller` does not indirectly disclose state to `viewer`.

More formally, `state_nonleakage` 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`.

3.2 Proof Approach: Sealed Blocks

Proving that every system call in a file system satisfies `ret_nonleakage` and `state_nonleakage` 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.

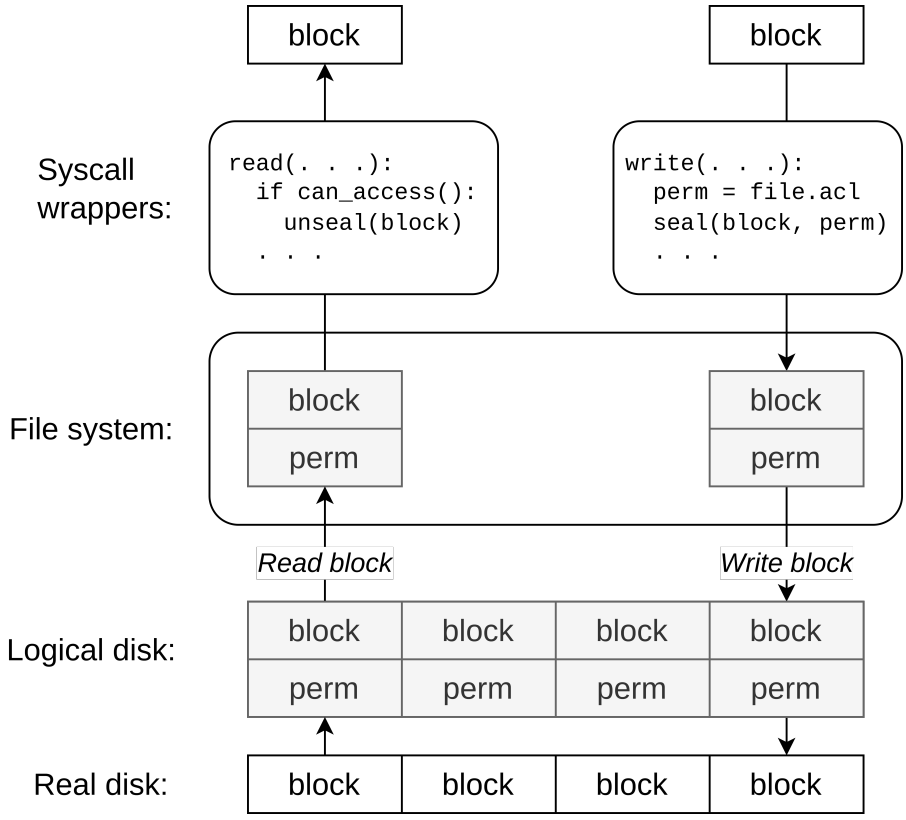

Figure 3-4: Overview of DiskSec's proof approach using sealed blocks.

Figure 3-4 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 3-4, has no permissions.

The second part is a sealed-block abstraction, indicated by shaded blocks in Figure

3-4. 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 Section 1.3. 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.

3.2.1 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 [13], 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, a sealed block points 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.

```
Inductive exec :
    forall T, proc T → Principal →
    State → (result T * trace) → Prop :=
| ExecReadSuccess :
    ∀ u st a h,
    a < disk_size →
    ~ handle_used st h →
    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
    exec (Read a) u st (Finished st' h, [])
| ExecWriteSuccess :
    ∀ u st a h,
    a < disk_size →
    handle_used st h →
    let data := handle_data st h in
    let dom := handle_dom st h in
    let st' := disk_block_write st a (data, dom) in
    exec (Write a h) u st (Finished st' tt, [])
| ExecSeal :
    ∀ u st h data dom,
    ~ handle_used st h →
    let st' := install_handle st h (data, dom) in
    exec (Seal data dom) u st (Finished st' h, [])
| ExecUnseal :
    ∀ u st h,
    handle_used st h →
    let data := handle_data st h in
    let dom := handle_dom st h in
    let perm := domain_perm st dom in
    exec (Unseal h) u st (Finished st data, [perm])
| ExecChangePerm :
    ∀ u st dom newperm,
    let oldperm := domain_perm st dom in
    let st' := domain_set_perm st dom newperm in
    exec (ChangePerm dom newperm) u st (Finished st' tt, [oldperm])
 (* Some rules omitted for space reasons *)
| ExecCrash :
    ∀ T p u st,
    (∀ dom perm, p <> ChangePerm dom perm) →
    exec p u st (Crashed st, [])
```

Figure 3-5: Execution semantics with logging of unseal operations.

Figure 3-5 shows a simplified version of DiskSec's execution semantics. The semantics are defined as a relation that relates the `code` being executed (of type `proc T`), the principal `u` running the operation (of type `Principal`), and the starting state `st` (of type `State`) to a tuple consisting of a result (of type `result T`) and a trace of unsealed permissions (of type `trace`).

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 genereated handle `h` 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 `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. `ExecCrash` rule states that, for any operation other than `ChangePerm`, it is possible to crash in

the starting state. Impossibility of crash for `ChangePerm` operation reflects the fact that `ChangePerm` is a purely logical operation. A combination of other rules, not shown, allow crashing in the middle of a sequence of operations.

Specification and verification of unseal rules. DiskSec requires developers to write a specification 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) :=
    ∀ u st res tr,
    exec p u st (res, tr) →
    (∀ perm, In perm tr → can_access u perm).
```

Figure 3-6: Definition of unseal safety.

Figure 3-6 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) :=
    ∀ u st res tr,
    exec p u st (res, tr) →
    (∀ perm, In perm tr → perm = Public).
```

Figure 3-7: Definition of `unseal_public`.

DiskSec also provides a stronger version of unseal-safety, as shown in Figure 3-7, 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 nonleakage theorems, as described in Section 3.2.2.

Crashes. DiskSec's approach naturally extends to reasoning about crashes. DiskSec's disk-crash model builds on the CHL model of disk crashes [13, 12]. 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.

3.2.2 Proving Nonleakage

To help the developer prove the two types of nonleakage, DiskSec provides helper theorems. Figure 3-8 shows the first one, which proves return-value nonleakage based on unseal-safety. We proved 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_nonleakage :
    ∀ (p: proc T),
    unseal_safe p → ret_nonleakage p.
```

Figure 3-8: Theorem connecting `unseal_safe` to return-value nonleakage.

Figure 3-9 shows the second theorem provided by DiskSec, for reasoning about state nonleakage. This theorem requires that the procedure satisfy the stronger definition, `unseal_public`, to ensure state nonleakage. 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_nonleakage :
    ∀ (p: proc T),
    unseal_public p → state_nonleakage p.
```

Figure 3-9: Theorem connecting `unseal_public` to state nonleakage.

DiskSec does not provide a general-purpose theorem for reasoning about state nonleakage 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 Section 1.3 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 nonleakage of those procedures. Chapter 4 discusses 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`.

Chapter 4

SFSCQ

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.

4.1 Specifying Security

The core specification of confidentiality for SFSCQ lies in the write system call, as shown in figure 4-1. 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 :
  ∀ f off data0 data1 caller st res0 tr0,
    exec (write f off data0) caller st (res0, tr0) →

    ∃ res1 tr1,
      exec (write f off data1) caller st (res1, tr1) ∧
      ∀ adv,
        ~ can_access adv (file_perm st f) →
        equiv_state_for_principal adv res0 res1.
```

Figure 4-1: 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 nonleakage from figure 3-3 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 nonleakage 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 nonleakage 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 nonleakage does not apply. Figure 4-2 summarizes this formally.

```
Definition chown_state_noninterference f new_owner :=
 ∀ viewer caller st0 res0 tr0 st1,
   exec (chown f new_owner) caller st0 (res0, tr0) →
   (file_data st0 f = file_data st1 f ∨ viewer <> new_owner) →
   equivalent_for_principal viewer st0 st1 →

   ∃ res1 tr1,
     exec (chown f new_owner) caller st1 (res1, tr1) ∧
     equiv_state_for_principal viewer res0 res1.
```

Figure 4-2: 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 counterintuitively, no special treatment is required in the specifications of other system calls, such as `read`. Instead, it suffices to prove the two general nonleakage theorems for all system calls (i.e., `ret_nonleakage` and `state_nonleakage`). This is because we do not want to consider specific attacks, such as whether `read` has a missing access-control check. Instead, Disksec's nonleakage 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.

4.2 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.

4.3 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 nonleakage. Proving that SFSCQ provides confidentiality required us to prove three theorems. The first is that `write` implements the specification from figure 4-1. This shows that SFSCQ will treat data passed by an application to `write` as confidential. The second is that system calls satisfy `ret_nonleakage`. 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_nonleakage`. 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 chapter 7.

Proving `ret_nonleakage` was the easiest, using Disksec's theorem from figure 3-8. All SFSCQ procedures are proven to be unseal safe, so no further proof effort is required.

Proving `state_nonleakage` was simple for all system calls except `read`, because those system calls satisfy `unseal_public`, allowing us to apply Disksec's theorem from figure 3-9. 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 3-9. 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.

4.4 Limitations

As we stated before, `data_nonleakage` does not prevent leakage from non-uniform outcome probabilities. We can demonstrate it by revisiting the example in section 1.3. `data_nonleakage` considers all states in the example equivalent to each other, because the state is a single secret bit. Figure 4-3 illustrates all possible executions

and how each execution matches with another from an equivalent state. Each arrow represents a possible execution of the program. On the labels, numbers in the brackets represent generated random bits, and the number following the brackets denotes the return value. Executions are colored based on their return values.

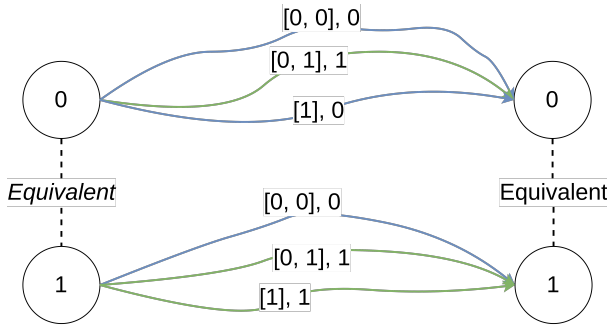

Figure 4-3: Visualisation of Figure 1-3 satisfying `data_nonleakage`.

We can see that, for each execution, there is an execution with the same color from the equivalent state. Therefore, this implementation satisfies `data_nonleakage`, but the secret bit can be determined as described in Section 1.3.

Another limitation was in proving confidentiality of `change_owner` and `delete` operations. Since each block has an owner in the DiskSec model, a `change_owner` operation require changing owners of multiple blocks atomically. A crash in the middle of such an operation can leave the system in an inconsistent state. We overcame this by adding an indirection in ownership tracking.

The limitation with `delete` operations comes from leftover data in freed blocks. Each free block is publicly owned. This requires freed blocks to be overwritten with zero blocks to prevent leakage, even though they are not accessible through the file-system API. This reduced `delete`'s performance significantly. It is worth noting that this was a shortcoming of the sealed-block technique, rather than a bug in the implementation.

Chapter 5

ConFrm

ConFrm is a framework for proving confidentiality of storage systems. It contains a new confidentiality definition as well as structures such as layer templates, and execution semantics, helping to implement file systems with confidentiality proofs.

5.1 Specification: Relatively Deterministic Noninfluence

Figure 4-3 shows that `data_nonleakage` leads to leakage of confidential data when probability of observing a return value depends on a secret. To address this challenge, this thesis introduces a new confidentiality definition that we call *Relatively Deterministic Noninfluence* (RDNI) that takes return-value frequencies into account. In the following section we will progressively build this new definition.

5.1.1 Basic Definition

Noninfluence can be interpreted as "matching" of executions from equivalent states to equivalent states for some chosen relation between two states[1]. More specifically, for each execution from a state, a *matching execution* from an equivalent state is an execution with the same return value where resulting states are equivalent as well. The traditional nondeterministic noninfluence definition allows multiple executions to be matched with a single execution. For example, in Figure 4-3, the same execution is matched with two other executions. This flexibility allows matching two sets of executions from equivalent states to equivalent states as long as their sets of possible return values are the same, but it ignores the probabilities of the return values.

Since the nondeterminism is what leads to the multiple possible executions, each execution should be the result of some specific sequence of nondeterministic events. In other words, each sequence of nondeterministic events uniquely identifies an execution, that is, executions are deterministic relative to a sequence of nondeterministic events. Therefore, for a particular sequence of nondeterministic events, there can be at most one execution from each equivalent state.

One way to ensure that the return-value probabilities are the same is, for each possible return value, requiring that the number of executions that returns it being the same from the equivalent states. This can be achieved by enforcing a 1-to-1 matching between executions from the equivalent states. If an execution can be matched with exactly one execution from an equivalent state, then we can conclude that the return value probabilities.

This requirement is the core idea behind the RDNI definition, and indeed is sufficient to address our challenge. Figure 5-1 visualizes how the example 1-3 does not

[1]In reality, noninfluence definition does not the require relation to be an equivalence. We use "equivalent states" instead of "related states" to make it clear that it is this relation we are referring to when we say "equivalent states" in other chapters.

satisfy the new definition, although it satisfies conventional noninfluence. There is no matching execution from the equivalent state if the first generated random bit is 1. There is exactly one execution from each state when the generated bit is 1 and they have different return values.

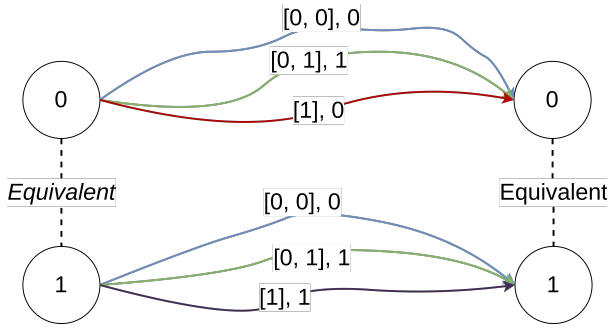

Figure 5-1: There is no corresponding execution for the red and black executions.

RDNI formalizes the above notion by using an execution relation that takes a sequence of nondeterministic events, which we call an *oracle*, and refers to it whenever it needs to make a nondeterministic choice (e.g., crashing or successfully executing). One important requirement is that the oracle must capture all the nondeterminism in the system. If all possible nondeterminism in the system is captured by the oracle, it is possible to reason about specific sequence of nondeterministic events by reasoning about the oracle itself. This requirement is enforced by ConFrm through the construction of a layer. Figure 5-2 shows the formalization of this approach.

```
Definition simple_RDNI {T} (u: user) (p: prog T)
        (eqv: state → state → Prop) :=
  ∀ (o: oracle) (s1 s2: state) (res1: Result T),
    exec u o s1 p res1 →
    eqv s1 s2 →

    ∃ res2,
        exec u o s2 p res2 ∧
        eqv (extract_state res1) (extract_state res2) ∧
        extract_ret res1 = extract_ret res2.
```

Figure 5-2: Simple relatively deterministic noninfluence.

Figure 5-2 states that a program p satisfies `simple_RDNI` for any two states s_1 and s_2 related by eqv, if there is an execution of p by user u from s_1 with oracle o that results in res_1, then there is an execution of p by user u from s_2 with oracle o with a result res_2 such that states of res_1 and res_2 are equivalent by eqv, and return values res_1 and res_2 are equal.

5.1.2 Crash, Reboot, and Recovery

Since we will be reasoning about crash-safe systems, RDNI should be extended to take crashes, reboots and recovery into account. We achieve this by replacing the execution semantics in the definition with one that captures the entire process of crash-reboot-recovery. There are important differences in the new execution semantics that needs to be explained:

1. Execution relation taking two program arguments, program to run and a recovery program,

2. How the state after a crash followed by a reboot is handled. Effects of a reboot of a system may be nondeterministic. One example of this is an asynchronous disk. When a system crashes and reboots, the disk can be in one of the multiple

possible states nondeterministically due to buffered and reordered writes. To capture and quantify this source of nondeterminism, we introduce *reboot state functions* — or *reboot functions* for short. A reboot function takes a state after a crash and returns the state that the system will be in after a reboot. Reboot functions make effects of the reboot on a state deterministic, due to the fact that the outcome of a function application is deterministic. Similar to the oracles, different outcomes of a nondeterministic reboot are represented by different reboot functions.

Although they both represent a form of nondeterminism in the system, we decided to separate reboot functions from oracles in our implementation. This separation simplifies the refinement definitions in ConFs and makes incorporating some assumptions on reboot function outputs easier.

3. Execution semantics of a crash-reboot-recovery process must capture multiple crash and recovery attempts. One way to achieve this is providing semantics for execution of the original program followed by the multiple consecutive executions of a recovery program. Since each execution requires an oracle and each crash requires a reboot function to determine after-reboot state, the new execution semantics will take a list of oracles and a list of reboot functions.

We will explain further details of how executions with recovery are implemented in the following section. Figure 5-3 shows the formalization with recovery executions.

```
Definition RDNI_with_recovery {T} (u: user) (p: prog T)
               (rec: prog unit)
               (eqv: state → state → Prop) :=
  ∀ (l_o: list oracle)  (l_rf: list (state → state))
  (s1 s2: state) (res1: Result T),
    exec_with_recovery u l_o s1 l_rf p rec res1 →
    eqv s1 s2 →
    ∃ res2,
        exec_with_recovery u l_o s2 l_rf p rec res2 ∧
        eqv (extract_state res1) (extract_state res2) ∧
        extract_ret res1 = extract_ret res2.
```

Figure 5-3: RDNI with recovery executions.

This definition differs from `simple_RDNI` in the way that it allows equivalence between the states to be broken temporarily in case of a crash, as long as the recovery program also reestablishes the equivalence. In other words, `RDNI_with_recovery` permits after-reboot states to be distinguishable. This is in line with our threat model, which assumes that adversaries cannot interact with the system until recovery is successfully completed.

5.1.3 Incorporating Noninterference

All the RDNI definitions up to this point were a variant of nonleakage. Therefore they didn't specify system's behavior regarding the new confidential data. We made two changes to incorporate noninterference into the definition.

First is conditioning return-value equality on a predicate for the user. This way, we can require return value equivalence to hold only for certain users, which is needed to state confidentiality of multi user systems. For example, return-value equality is required when the theorem is about adversaries' executions but not needed when it is about normal users' executions.

Second is changing the definition to be about the execution of two programs

instead of the same one. This change enables us to reason about functions with different input arguments, because a ConFrm program is a function and its arguments together. With these changes, we reach our final definition for RDNI, which is shown in Figure 5-4.

```
Definition RDNI T (u: user) (p1 p2: prog T)
        (rec: prog unit)
        (eqv: state → state → Prop)
        (cond: user → Prop):=
  ∀ (l_o: list oracle) (l_rf: list (state → state))
  (s1 s2: state) (res1: Result T),
    exec_with_recovery u l_o s1 l_rf p1 rec res1 →
    eqv s1 s2 →
    ∃ res2,
        exec_with_recovery u l_o s2 l_rf p2 rec res2 ∧
        eqv (extract_state res1) (extract_state res2) ∧
        (cond u → extract_ret res1 = extract_ret res2).
```

Figure 5-4: Final definition of RDNI.

5.1.4 Termination Sensitivity

The RDNI definition requires for each execution an execution to exist from any equivalent state. This was the case for all the definitions presented so far. This requirement is called *termination sensitivity*. In its essence, termination sensitivity implies that an adversary cannot learn any confidential information by observing if the program terminates or not. An example of a non-termination-sensitive implementation could be a function looping infinitely based on some secret bit. If an implementation contains such a loop, then an adversary who calls the function can learn the value of the secret bit by waiting a reasonable amount of time to see whether the function terminates.

More generally, termination sensitivity is not restricted to infinite loops. Any

behavior that causes one execution to "get stuck" while another execution finishes will violate termination sensitivity. Which executions can "get stuck" depends on how the semantics are defined. For example, if the semantics of a read operation on a disk are defined only for addresses that are in-bounds, then any execution that attempts an out-of-bounds access will get stuck.

It is also worth pointing out that getting stuck is not the same as returning an error for an invalid operation. In the latter case, semantics are still defined. So, in the above example, if the semantics are defined to return an error in case of an out-of-bounds access, then one in-bounds and one out-of-bounds execution would not violate termination sensitivity.

Although it captures an important aspect of a possible breach of confidentiality, termination sensitivity is not a necessity in confidentiality specifications. Many different systems in the literature use termination-insensitive definitions for their specifications [31]. A termination-insensitive definition requires any pair of existing executions from equivalent states to have the same return value and result in equivalent states. However, a program now has freedom to get stuck in some equivalent states.

However, these definitions are generally variants of deterministic specifications, where the termination-insensitive variant is a weaker specification. In the nondeterministic case, termination insensitivity is overly restrictive. The requirement of **any** pair of executions from any two equivalent states to have the same return value and also result in equivalent states diminishes the power of nondeterminism greatly. For example, an abstraction of an allocation function where an unused resource is nondeterministically allocated would not satisfy a termination-insensitive specification. This restriction makes termination-insensitive nondeterministic specifications an unfitting confidentiality specification in many cases.

The relatively deterministic nature of RDNI allows us to define a termination-insensitive variant that is not overly restrictive. Formal definition of termination-

insensitive RDNI can be found in Figure 5-5. In this variant, pairs of executions with the same oracles are allowed to be termination-insensitive, but there is no requirement on pairs of executions with different oracles. In other words, a pair of executions with different oracles from equivalent states can have different return values and also can result in nonequivalent states. This freedom enables abstractions that are similar to the above example, since return value will be determined by the oracle.

```
Definition Termination_Insensitive_RDNI T (u: user) (p1 p2: prog T)
         (rec: prog unit)
         (eqv: state → state → Prop)
         (cond: user → Prop):=
  ∀ (l_o: list oracle) (l_rf: list (state → state))
  (s1 s2: state) (res1 res2: Result T),
    exec_with_recovery u l_o s1 l_rf p1 rec res1 →
    exec_with_recovery u l_o s2 l_rf p2 rec res2 →
    eqv s1 s2 →
    eqv (extract_state res1) (extract_state res2) ∧
    (cond u → extract_ret res1 = extract_ret res2).
```

Figure 5-5: Termination-insensitive variant of RDNI.

5.2 Definitions and Metatheory

On top of RDNI, ConFrm also includes structures and metatheory that developers can use to implement confidential and crash-safe storage systems. This portion consists of two parts: (1) support for abstraction, and (2) the metatheory that provides relevant theorems to prove confidentiality of an implementation from the confidentiality of an abstraction. We will first present the infrastructure for defining abstractions and then explain the metatheory.

5.2.1 Abstraction Structures

Cores. ConFrm introduces cores as the main way to model the abstract state of the system and the operations that can be performed on it. A core has four components,

1. the state the system
2. the list of possible operations that can be performed,
3. the list of possible nondeterminism tokens,
4. the execution semantics of each operation.

Also, to ensure that tokens capture all the nondeterminism in the semantics, a proof that shows, given a token, execution semantics are deterministic is required. Figure 5-6 shows the formal definition of a core.

```
Record Core :=
  {
    token : Type;
    state : Type;
    operation : Type → Type;
    exec : ∀ T, user → token → state →
        operation T → Result state T → Prop;

    exec_deterministic_wrt_token :
      ∀ u o s T (p: operation T) ret1 ret2,
        exec u o s p ret1 →
        exec u o s p ret2 →
        ret1 = ret2;
  }.
```

Figure 5-6: Definition of a core.

An example core for a cache can be seen in figure 5-7.

```
Definition Cache_Core :=
  {
    token := Continue | Crash ;
    state := address → option block ;
    operation := Read a | Write a b | Flush ;
    exec := . . . ;
    exec_deterministic_wrt_token := . . .
  }.
```

Figure 5-7: An example core for a cache.

This cache has three operations: `Read`, `Write`, and `Flush`. Its state is a partial function from addresses to blocks, to model possible cache misses. Its tokens are simple, `Continue` for a successful execution and `Crash` for crashing on that operation. Execution semantics and the proof of determinism are omitted for brevity.

Crashes. ConFrm provides support for crash semantics by defining two different execution results: `Finished` and `Crashed`. A `Finished` result means that program has successfully completed and contains a state and a return value. A `Crashed` result means that the program crashed during its execution and contains only a state, which represents the state of the system after the crash happened but before rebooting.

Developers define the crash semantics of the system by defining execution rules that lead to a `Crashed` result. It is the developer's responsibility to ensure that the defined execution semantics correctly models the system's both normal and crash behavior.

Layers. ConFrm also includes the machinery that turns a core to a full layer by equipping it with `Bind` and `Return` operations. This eliminates the repetitive work that must to be done to define layers. It also allows the framework to provide core-agnostic theorems and tactics to be used in proofs.

Semantics of the layer are derived from the semantics of its core. A new semantics takes a list of tokens (i.e., an oracle) and consumes exactly one at each step. Figure 5-8 shows the derived execution semantics for a layer.

```
Inductive exec :
    ∀ T, user → oracle → state' →
    prog T → @Result state T → Prop :=

  | ExecOp :
      ∀ T (p : core.operation T) u o d d' r,
        core.exec u o d p (Finished d' r) →
        exec' u [OpToken o] d (Op T p) (Finished d' r)

  | ExecRet :
      ∀ d T (v: T) u,
        exec u [Cont] d (Ret v) (Finished d v)

  | ExecBind :
      ∀ T T' (p1: prog T) (p2: T → prog T')
        u o1 d1 d1' o2 r ret,
        exec u o1 d1 p1 (Finished d1' r) →
        exec u o2 d1' (p2 r) ret →
        exec u (o1++o2) d1 (Bind p1 p2) ret

  (* Crash semantics are omitted. *)
```

Figure 5-8: Execution semantics of a language.

It instruments core operations and tokens to convert them into layer programs and tokens, respectively. `Ret` and `Bind` have standard definitions, augmented with oracles.

ConFrm also provides some theorems regarding determinism of an execution as well as the relationship between oracles and executions like how two executions relate to each other if one's oracle is a prefix of the other's.

Recovery semantics. ConFrm provides predefined recovery semantics for the systems and adds these semantics when it generates a layer from a core. To distinguish recovery semantics from the semantics of the execution of a single program, we will refer to recovery semantics as executing-with-recovery. In ConFrm's recovery model, only two outcomes are possible when executing-with-recovery: (1) execution can finish without any crashes, or (2) execution crashes then recovers after a certain number of attempts. To represent these two outcomes, ConFrm uses two types of recovery result: `RFinished` and `Recovered`. `RFinished` corresponds to case (1), and `Recovered` corresponds to case (2). Since there is no rule for crashing infinitely many times, the provided semantics implicitly assume that recovery eventually will succeed.

```
Inductive exec_with_recovery :
    ∀ T, user → list oracle → state →
    list (state → state) → prog T → prog unit →
    @Recovery_Result state T → Prop :=

  | ExecFinished :
      ∀ T (p: prog' T) p_rec u o d d' t,
        exec u o d p (Finished d' t) →
        exec_with_recovery u [o] d [] p p_rec (RFinished d' t)

  | ExecRecovered :
      ∀ T (p: prog' T) p_rec u o lo d d' get_reboot_state l_grs ret,
        exec u o d p (Crashed d') →
        exec_with_recovery u lo (get_reboot_state d')
            l_grs p_rec p_rec ret →
        exec_with_recovery u (o::lo) d (get_reboot_state::l_grs)
            p p_rec (Recovered (extract_state ret)).
```

Figure 5-9: Recovery semantics in ConFrm.

Figure 5-9 displays the formal definition. Semantics for (1) are stated in the `ExecFinished` rule. It is quite straightforward. If the program successfully executes, then it successfully executes-with-recovery. Semantics for (2) are stated in the

`ExecRecovered` rule and are more involved. It is inductively defined to capture repeated attempts of recovery until it succeeds. The rule states that, if the original program crashes, and the recovery program executes-with-recovery to some result, then the original program executes-with-recovery to the state of that result. Execution uses a new oracle and a new reboot function every time a crash-reboot-recovery cycle happens. Therefore, lengths of those lists implicitly determine how many times the recovery will crash until it succeeds.

Refinements. ConFrm's main mechanism for relating abstractions and implementations is refinements. ConFrm defines a refinement as an object between an implementation layer and a core abstracting it. We extend the standard refinement definition to accommodate both crashes and oracles.

As shown in Figure 5-10, a refinement has four components that correspond to the four components of a core, and a theorem states that a successful execution preserves the state-refinement relation. The four components are

- a `compile` function that turns an abstract operation into its implementation program,
- a `refines` relation that relates an abstract state to an implementation state,
- a `refines_reboot` relation that relates an abstract reboot state to an implementation reboot state,
- and `token_refines` relation that relates an abstract token to an implementation oracle.

```
Record CoreRefinement C_imp (L_imp: Layer C_imp) (C_abs: Core) :=
  {
    compile_core : ∀ T, C_abs.operation T → L_imp.prog T;

    refines_core: L_imp.state → C_abs.state → Prop;

    refines_reboot_core: L_imp.state → C_abs.state → Prop;

    token_refines: ∀ T, user → L_imp.state →
        C_abs.operation T →
        (L_imp.state → L_imp.state) →
        L_imp.oracle →
        C_abs.token → Prop;

    exec_compiled_preserves_refinement_finished_core :
      ∀ T (p2: C_abs.operation T) o1 s1 s1' r u,
        (∃ s2, refines_core s1 s2) →
        L_imp.exec u o1 s1 (compile_core T p2) (Finished s1' r) →
        (∃ s2', refines_core s1' s2');
  }.
```

Figure 5-10: Definition of a core refinement.

Both `compile` and `refines` are part of the standard definition. However, `refines_reboot` and `token_refines` relations require more explanation.

We separate `refines_reboot` from `refines` because, in general, a `refines` relation is too strong to hold for after-reboot states, but we also needed a relation between them to ensure that recovery restores the original `refines` relation. For example, if a write-ahead log implementation uses a log cache to speed up the reads, its `refines` relation may contain a proposition that the values stored in the cache are exactly the values stored in the log. In this case, the `refines` relation doesn't hold after a reboot because the cache will not contain any data.

The `token_refines` relation correspond to representing a set of oracles that lead to the same behavior with a single token. Informally, an abstract token is a concise

representation of multiple sequences of nondeterministic events that lead to the same outcome, which can be thought as having the cumulative probability of the sequences it represents in the mental model.

The `token_refines` is more complicated than `refines` and `refines_reboot`. On top of the oracle and the token it relates, it takes the following parameters

- a user,
- an implementation state,
- an abstract operation,
- and an implementation reboot function.

All these parameters are necessary to capture the intricate relationship between abstract tokens and implementations' crash and recovery behavior. We can demonstrate the roles they play by examining the following example.

Assume that we are abstracting an implementation of a checksum-based log on an asynchronous disk with a `write` function. A crash during a write to a checksum-based log may leave the log in such a state that whether the `write` succeeded or not would depend on which blocks made it to the disk before the crash (which is determined by the after-reboot state of the implementation). In other words, success of a write after crash depends on (1) state of the disk just after the crash, and (2) state of the disk after reboot. To determine (1), we need to know the user, the starting state, and the data being written, which are in the operation. To determine (2), we need to know the reboot function. Therefore, capturing the behavior of the write in this particular case requires all the parameters listed above. Other operations may require some or all of those parameters as well.

Similar to generating a layer from a core, ConFrm can automatically generate a refinement between two layers given a core refinement between an implementation

layer and an abstraction core. A refinement for a layer differs from a refinement for a core in three places. First, the `compile` function transforms programs from the abstraction layer to implementation layer. Second, `token_refines` turns into `oracle_refines`, which relates an abstract oracle and an implementation oracle. Third, a finished execution of any compiled program should preserve the refinement. ConFrm also provides a `recovery_oracles_refine` relation, which relates lists of implementation oracles to lists of abstraction oracles by `oracle_refines` inductively. Figures 5-11 and 5-12 visualize how refinement is defined for normal and recovery executions respectively.

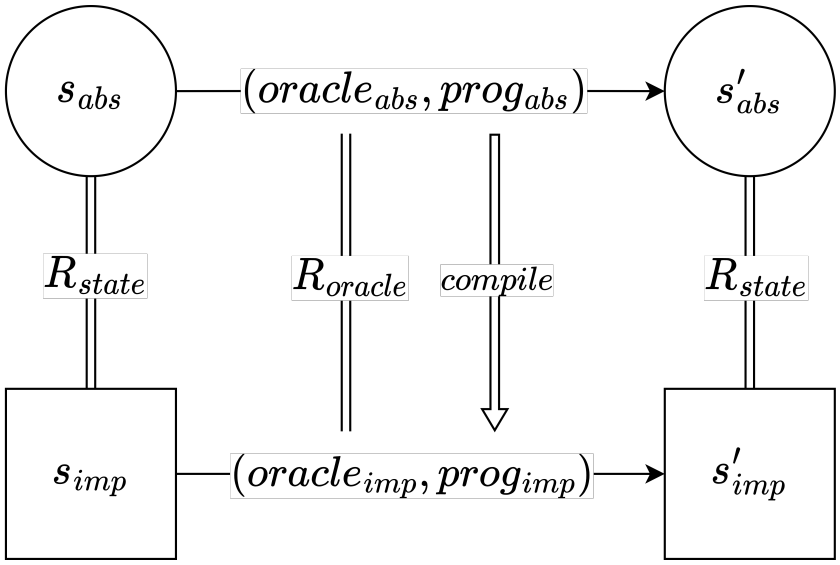

Figure 5-11: Refinement for normal executions.

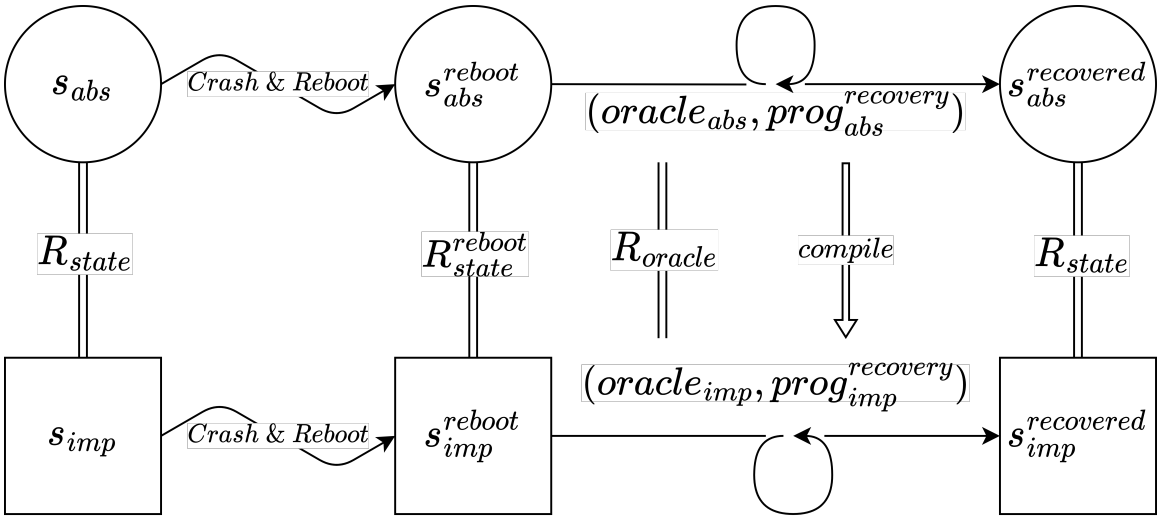

Figure 5-12: Refinement for reboot and recovery executions.

Horizontal Compositions. To enable modular implementations, ConFrm provides automatic derivation of a new, composite core from two given cores via horizontal composition. A state of the composite core is a pair that contains the state of each of the component cores. This capability allows developers to develop the system in small, self-contained parts that can be combined at will when desired without much overhead. A layer derived from a composite ConFrm core contains support for "lifting" the programs written in a layer of one of the component cores to the layer of the composite core. Similarly, it allows automatic derivation of a refinement between the two composite layers if a component of the first layer is a refinement of a component of the second layer.

5.2.2 Metatheory

At the heart of ConFrm lies the Theorem 5-15, which derives the confidentiality of a compiled program from the confidentiality of its abstraction. The theorem reveals

sufficient conditions for preserving RDNI through refinement. The two conditions are:

1. there should be a simulation between implementation and abstraction with respect to refinement relations, and

2. if a list of implementation oracles refine a list of abstract oracles from a state with the first program, then it should refine the same oracle from any state that is equivalent to the first state with the second program.

The first condition ensures that there is no execution of a compiled program that is not captured by an execution of an abstract program. This is necessary for a property of any abstract execution to imply a property of any implementation execution. If there were an implementation execution that does not correspond to an abstract execution, then it would not be possible to reason about such an execution through an abstract execution.

The second condition can be interpreted as the necessity that abstraction does not inject dependency on the confidential data into abstract oracles. Abstractions modelling some deterministic behaviors of an implementation as nondeterminism is a common pattern. For example, an abstraction of a resource allocator may model the allocation function to return an unused resource nondeterministacally, even though the implementation's behavior is actually deterministic (e.g., returning the first available one).

This property makes sure that the developer does not abstract a behavior that depends on the confidential data in such a way. If such action would be permitted, then two implementation executions from equivalent states with the same implementation oracles could correspond to two abstraction executions with different oracles. In such a case, the noninfluence of the abstraction with the same oracles wouldn't be strong enough to establish the same fact in the implementation, due to the fact

that noninfluence of the abstraction does not state anything about executions with different oracles. Formalization of this condition can be seen in figure 5-13.

```
Definition oracle_refines_same_from_equivalent
    (u: user) T (p1_abs p2_abs: L_abs.prog T)
    rec_abs l_get_reboot_state_imp equivalent_states_abs :=

    ∀ l_o_imp l_o_abs l_o_abs' s1_imp s2_imp,

      refines_equivalent equivalent_states_abs s1_imp s2_imp →

      recovery_oracles_refine
        u s1_imp p1_abs rec_abs
        l_get_reboot_state_imp
        l_o_imp l_o_abs →

      recovery_oracles_refine
        u s2_imp p2_abs rec_abs
        l_get_reboot_state_imp
        l_o_imp l_o_abs' →

      recovery_oracles_refine
        u s2_imp p2_abs rec_abs
        l_get_reboot_state_imp
        l_o_imp l_o_abs.
```

Figure 5-13: Formalization of oracle refinement being independent of confidential data.

Simulations. The first condition above states that a simulation must exist between the abstraction and the implementation. Since we introduced oracles and crash-and-recovery into execution relations, we modify the standard simulation definition to accommodate those changes.

```
Definition Simulation
   u T (p_abs: L_abs.prog T) (rec_abs : L_abs.prog unit)
   l_get_reboot_state_imp l_get_reboot_state_abs
   eqv_begin eqv_end :=

   ∀ l_o_imp s_imp  s_imp' s_abs,

    eqv_begin s_imp s_abs →

    L_imp.(exec_with_recovery) u l_o_imp s_imp
        l_get_reboot_state_imp (compile p_abs)
        (compile rec_abs) s_imp' →

    ∃ l_o_abs s_abs',
        recovery_oracles_refine u s_imp p_abs rec_abs
            l_get_reboot_state_imp l_o_imp l_o_abs ∧

        L_abs.(exec_with_recovery) u l_o_abs s_abs
            l_get_reboot_state_abs p_abs rec_abs s_abs' ∧

        eqv_end (extract_state_r s_imp') (extract_state_r s_abs') ∧
        extract_ret_r s_imp' = extract_ret_r s_abs').
```

Figure 5-14: ConFrm's simulation definition with oracles and execution-with-recovery.

As shown in figure 5-14, the first change is that the modified simulation definition has three simulation relations, one for the starting states, one for the end states, and one for the oracles. We separate the relation that relates the starting and end state to be able to reason about recovery where the relations that hold at the beginning and at the end are different.

In the context of RDNI transfer of a recovery program, start and end simulation relations coincide with the state-refinement relations `refines_reboot` and `refines`, respectively. We also define a two-relation variant to use in definition 5-15, where start and end relations are both a `refines` relation.

The second change is that a simulation is defined over an entire execution-with-

recovery. This allows the simulation relation to be broken temporarily after a crash, as long as it is restored by the recovery process. This change is necessary because crashes may expose states that will never appear during a normal execution. This way, a refinement relation can only consider the states that appear during normal execution. How to represent crash states is entirely left to the developer.

```
Lemma RDNI_transfer:
  ∀ C_imp C_abs (L_imp: Layer C_imp) (L_abs: Layer C_abs)
    (R: Refinement L_imp L_abs)
    u T (p1_abs p2_abs: L_abs.prog T) rec_abs
    l_get_reboot_state_imp
    l_get_reboot_state_abs
    equivalent_states_abs cond,

    RDNI
      u p1_abs p2_abs rec_abs
      equivalent_states_abs
      cond l_get_reboot_state_abs →

    Simulation R
      u p1_abs rec_abs
      l_get_reboot_state_imp
      l_get_reboot_state_abs →

    Simulation R
      u p2_abs rec_abs
      l_get_reboot_state_imp
      l_get_reboot_state_abs →

    oracle_refines_same_from_equivalent R
      u p1_abs p2_abs rec_abs
      l_get_reboot_state_imp
      equivalent_states_abs →

    RDNI
      u (R.compile p1_abs)
      (R.compile p2_abs)
      (R.compile rec_abs)
      (refines_equivalent equivalent_states_abs)
      cond l_get_reboot_state_imp.
```

Figure 5-15: RDNI transfer theorem.

5.3 Using ConFrm

In this section, we explain how ConFrm can be used to ensure an implementation is secure. Our explanation will contain two parties: a checker and a developer. The checker is the party who is responsible for ensuring the security of the system and the developer is the party who will implement the system using ConFrm. For clarity, we divide the implementation components into two groups: the group that will be defined by the checker and the group that will be provided by the developer. In case the entire system is implemented by the developer, checker's group corresponds to the components that needs to be inspected manually to ensure their correctness.

To use ConFrm, the checker has to implement the model of the disk and the memory, the top level model of the system, and RDNI specifications for confidentiality. The model of the disk and the memory ensures that the required assumptions about the nondeterminism holds. The top level model is where semantics of each operation in the system is defined and serves as a functional specification for the system. It also contains an initialization and recovery operation. Both models are implemented as ConFrm layers. Writing RDNI specifications include selecting the equivalence relation that encodes what is confidential and what is not. In summary, the checker is responsible for specifying the system and the computer it will run on.

In return, the developer provides an implementation for each system operation along with a simulation proof, a proper initialization proof and proofs of RDNI specifications for the implementation. Proper initialization proof establishes that the implementation creates an initial state that refines an abstract state. Figure 5-16 shows the formal statement.

```
Theorem proper_initialization :
  ∀ u o_imp s s_imp_init r,

    L_imp.(exec) u o_imp s (compile Init) (Finished s_imp_init r) →

    ∃ s_abs,
      refines s_imp_init s_abs.
```

Figure 5-16: Proper initialization theorem.

Proper initialization theorem is a simulation-like requirement with the differences that it holds for any start state and it only applies to non-crashing executions. Such a proof is required to guarantee system starts from a well-formed state.

Simulation proofs show that the implementation provides the intended functionality of each operation. Finally, developer provides proofs for RDNI specifications, one implementation specification for each top-level specification. Implementation specifications are in the form of the conclusion of `RDNI_transfer` theorem. Developer can derive the proofs from the simulation and oracle refinement independence proofs. All these components are machine-checkable and doesn't require manual inspection.

Specifications and proper initialization proof combined guarantee that the system initializes correctly, its state will refine abstract states throughout the execution, states that refine equivalent abstract states will continue to refine equivalent abstract states, and it will return the same values from the states that refine equivalent abstract states. The checker doesn't need to know the details of how an implementation state and oracle refines an abstract states and oracle, respectively. It is sufficient to know that such a refinement exists to conclude that the implementation is secure.

In this chapter, we covered formalization of RDNI and other components of ConFrm, as well as how to use it to implement confidential storage systems. Next, we will present ConFs, our confidential file system implemented and proved confidential in ConFrm.

Chapter 6

ConFs File System

ConFs is the first confidential file-system with a checksum-based log with machine-checkable proofs. The first half of the chapter will explain its design and implementation. The second half will explain its confidentiality specifications and the effort that went into proving them.

6.1 Design

ConFs consists of three components: (1) a checksum-based write-ahead log with a log cache, (2) a transaction system, and (3) file-system structures like block allocators and inodes. These components embody different challenges from section 1.3.

The full design of ConFs can be seen in Figure 6-1. Solid boxes depict ConFrm cores. Shaded boxes represent implementation components. Colors distinguish different ConFrm layers. Each shaded box uses functions and operations from the boxes directly below it. For example, the log cache uses both implemented log functions and cache operations from the cache core. A solid box on top of a shaded box represents an abstraction (e.g. transactional disk abstracts the functions of transactions into operations).

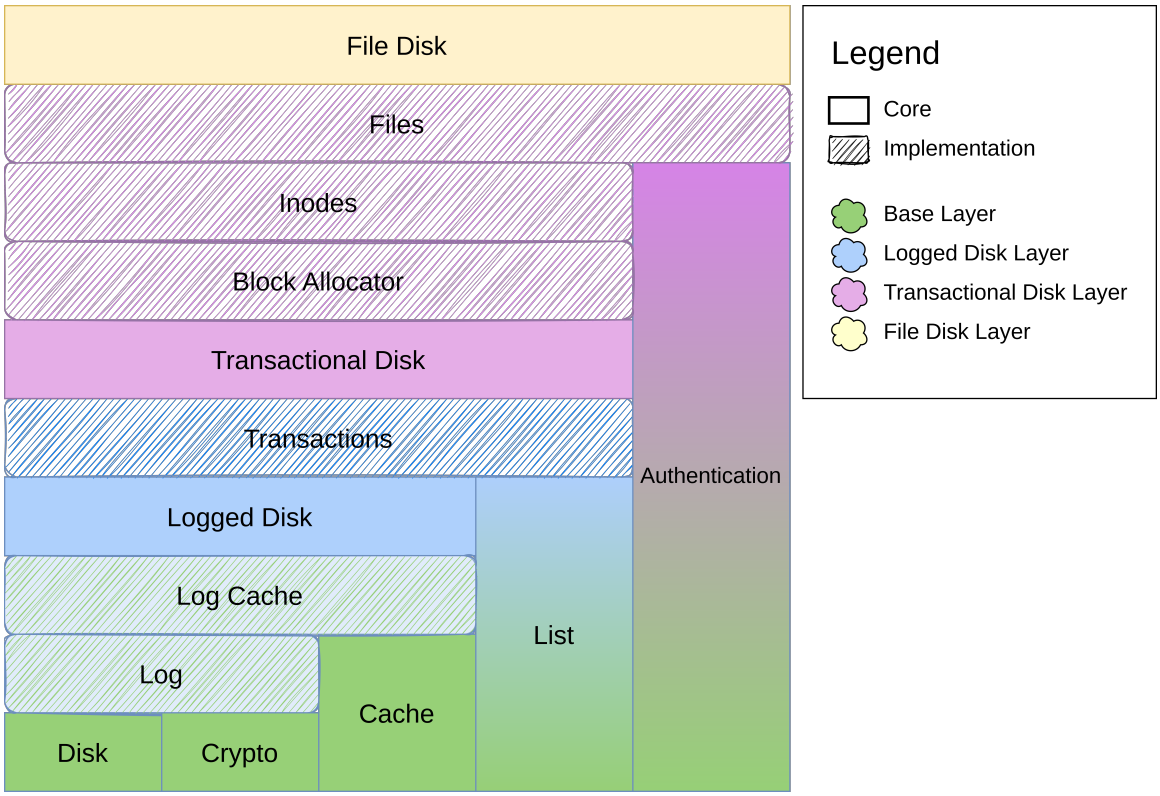

Figure 6-1: Structure of ConFs.

6.1.1 Base Layer and the Log Component

Base Layer The base layer is the one where we model disk, cache and some in-memory data structures for transactions and cryptographic operations. It provides all the basic operations that can be used in the system, which each file-system operation is compiled into. The list of operations can be seen in Figure 6-2.

Disk	Cache	Auth
`read`: addr → block `write`: addr → block → unit `sync`: unit	`read`: addr → option block `write`: addr → block → unit `flush`: unit	`auth`: user → bool
Crypto	List	
`hash`: hash → block → hash `generate key`: key `encrypt`: key → block → block `decrypt`: key → block → block	`get`: list (addr * block) `put`: (addr * block) → unit `delete`: unit	

Figure 6-2: Operations in the base layer.

Log and Log Cache ConFs contains a checksum-based write-ahead log similar to DFSCQ's. Our design differs from DFSCQ's log in that the log blocks are encrypted. Encryption of the log is necessary in a checksum-based log to avoid leaking previous transactions' contents. Figure 6-3 demonstrates the problem with a simple example.

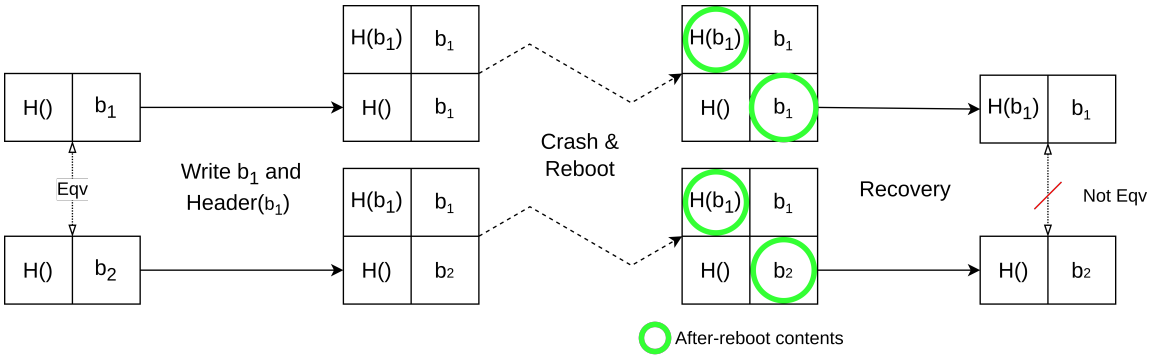

Figure 6-3: A sequence of events that leads to leakage of the confidential data.

In the above example, there are two logs with the length of one block. Both logs are initially empty, but there are leftover blocks b_1 and b_2 from a previously applied transaction. Now a user commits a new transaction with b_1 as its content.

Then both logs crash after the data and the header are written but before the disk is synced. In both cases, the new header manages to persist on the disk, but the data doesn't. After reboot, the recovery procedure of the first system will keep the latest transaction since the hash of the log and the hash in the header match. However, the second system will discard the transaction because the hashes won't match. Now the user who committed the last transaction can infer the contents of the previous transaction based on the state of the system after recovery by looking if his write is present on the disk.

We use encryption to fix the above problem. Encryption provides protection at two levels: (1) it makes collision between a block that is already on the disk and the block that is written on it extremely unlikely, and (2) even in the case of it happening, it prevents the user from inferring the contents of the previous transaction. Case 1 is due to the fact that it is extremely unlikely to produce the same ciphertext from two blocks that are encrypted with two different random keys. Case 2 is ensured by the usage of a fresh key for each transaction, since having the same ciphertext will not reveal any information about the plaintexts if two different keys are used. Figure 6-4 shows how using encryption fixes the problem.

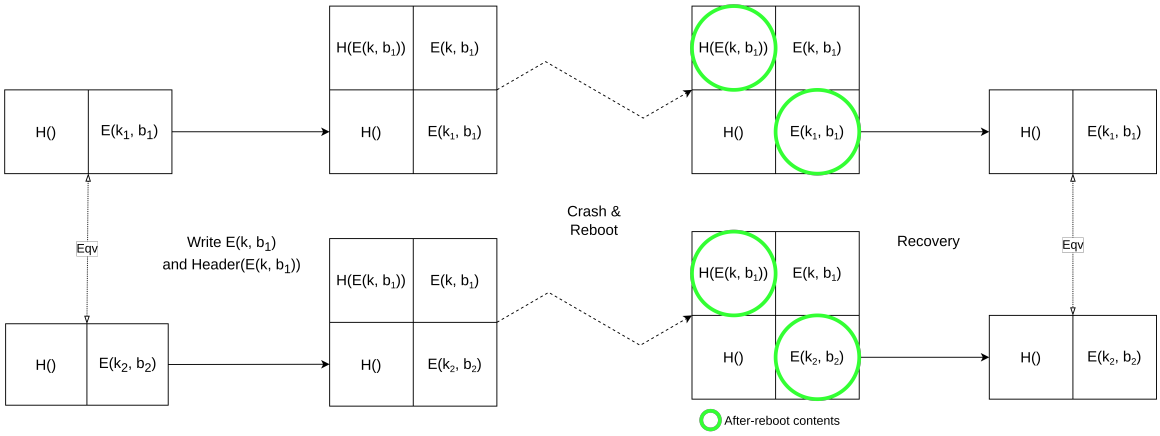

Figure 6-4: Encryption fixes the leakage.

The initial setting in the encrypted example is similar to the unencrypted version, except the leftover blocks are encrypted with keys k_1 and k_2. Same as before, a user commits a new transaction with b_1. Before writing b_1 to the log, it gets encrypted with a freshly generated key k. The crucial observation is that, if k is different than k_1 and k_2, then $E(k, b_1)$ is different than $E(k_1, b_1)$ and $E(k_2, b_2)$ with high probability. This difference implies that their hashes are different with high probability as well. Since the new hash is different than the hashes of leftover blocks, these is no after-reboot state where one transaction is kept but the other is rolled back. Therefore, all possible after-recovery states are equivalent.

Since the log contents are encrypted and encryption/decryption is computationally expensive, we implemented a write-through log cache to speed up the read requests. The cache contains unencrypted versions of the data stored in the log.

A checksum-based log raises the challenge of operating on confidential data as well as secret-dependent outcome probabilities. A hash of the log indirectly contains information from the data stored in the log. Any branching that is made based on hash values, (e.g., recovery checking the hash to restore the log), has the potential to leak confidential data. Secret-dependent outcome probabilities arise from the crash of the asynchronous disk. A crash of an asynchronous disk leads to multiple possible reboot states, which are dependent on the log contents.

6.1.2 Logged-Disk Layer and Transactions

Logged-Disk Layer We abstract the log-cache API to a new core called logged-disk core. Logged-disk core provides two improvements over directly using the implementation: (1) it simplifies the disk model, and (2) it simplifies the operational semantics.

The logged-disk layer's disk model is a total function from addresses to latest

blocks. Each address is shifted by the log length (e.g., the first block after the log in the base-layer disk corresponds to address 0 in the logged-disk-layer disk). Figure 6-5 illustrates how a base-layer disk transforms into a logged-disk-layer disk. Dashed arrows show the final addresses for the data stored in the log blocks. Solid arrows show where each block's content comes from.

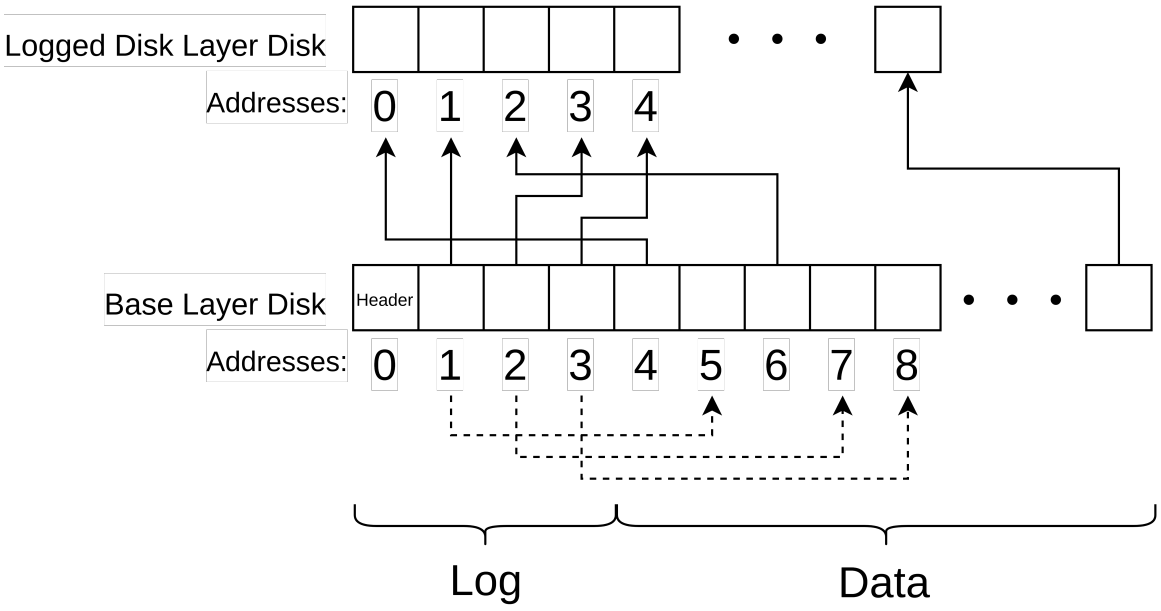

Figure 6-5: Base-layer disk's transformation to logged-disk-layer disk.

This model hides the existence of the log and the cache as well as previously written values in the base-layer disk.

Transactions Transactions don't pose any new confidentiality challenges apart from increasing the system complexity. However, they are required to make the file-system operations atomic.

6.1.3 File-System Structures

File-system structures include inode and data allocators, inodes, and files.

Inodes We designed inodes to be as simple as possible while retaining the required functionality. Each inode contains an owner and a list of direct block numbers. We decided to only use direct blocks to avoid the complexity of indirect addressing, since it doesn't pose any interesting confidentiality issues.

Files Our file-system API provides basic file operations that are relevant to the challenges we are trying to address. Figure 6-6 shows the file-system API. To keep the system simple, all operations are designed at a block granularity, i.e. they read or write entire blocks, because byte granularity adds extra complexity without presenting any challenges regarding confidentiality. Similarly, we chose to use inode numbers as file handles to focus on confidentiality without the complexity of managing a directory structure.

Operation	Type Signature
`read`	inum → addr → option block
`write`	inum → addr → block → option unit
`extend`	inum → block → option unit
`create`	user → option inum
`delete`	inum → option unit
`change_owner`	inum → user → option unit

Figure 6-6: File-system API.

File-system structures contain a notion of ownership, discretionary access control, and nondeterministic specifications for `create`. This leads to possible indirect disclosure and the challenge of implementing ownership changes.

File-Disk Layer The file-disk layer is the abstraction of the file-system API, where each system call is an operation in its language. The layer presents a simple disk model, a map from addresses to files. This simplification provides an intuitive model for how a file system is perceived and also simplifies the confidentiality specifications. Since there are no directories or naming, `create` nondeterministically chooses an empty inode number to create a file and returns that number upon successful creation.

6.2 Implementation

We implemented ConFs as a stack of layers and components to simplify the operational semantics as we implement higher-level functionality. Each component is written in a ConFrm layer and also abstracted into a higher layer via refinement. For example, the write-ahead log is written in the *base layer*, the layer that models the raw disk and provides disk operations as its language. Then the write-ahead log, combined with the log cache, is abstracted into the *logged-disk layer*. The logged-disk layer completely hides the existence of the log and the cache in the implementation, by presenting a crash-safe disk as its state.

6.2.1 Base Layer and the Log Component

Base Layer Two parts of the base layer's implementation are worth explaining further: (1) an asynchronous disk, and (2) cryptographic operations.

Asynchronous disk. We implemented an asynchronous disk as a total function from a natural number to a (block * list block) tuple, similar to DFSCQ. The first component of the tuple represents the latest block written on the disk. The second component represents the blocks that were previously written to the disk after the latest sync. This list is used to model the effect of buffering and reordering of writes

and used to enumerate possible crash states of the disk, where each disk block after the reboot may be either the latest value or one of the values in the list. Disk size is taken as a parameter and enforced by the operational semantics.

Cryptographic operations. To implement a checksum-based, encrypted log, we need encryption/decryption, key generation and hashing. Encryption and decryption are implemented as abstract functions with their properties stated as axioms. Figure 6-7 shows our encryption model.

```
encrypt: key → block → block.
decrypt: key → block → block.

encrypt_decrypt:
    ∀ k v,
    decrypt k (encrypt k v) = v.

decrypt_encrypt:
    ∀ k ev,
    encrypt k (decrypt k ev) = ev.
```

Figure 6-7: Encryption model.

We use operational semantics to implement the key generation and hashing, similar to DFSCQ. Using operational semantics allows us to add collision-freedom and key freshness as premises to the theorems without introducing inconsistency to the system (e.g., if there are no hash collisions, then the operation is confidential). If the premise doesn't hold for the executing program (e.g, if a previously generated key is returned during an execution), the theorem doesn't apply.

We do so by defining validity of an execution only if the desired property is satisfied. For example, the execution rule for key generation states that the returned key is not generated by the system before. As a consequence, any theorem that has an

execution as a premise, like RDNI, implicitly contains the assumption as a premise, since an execution that violates the premise cannot be constructed in the model.

Log and Log Cache

Since ConFs' log contents are encrypted, we modified header blocks to store the transaction keys. Similar to DFSCQ's header, ConFs' header contains two parts: *an active part* for the current state of the log and *an old part* for the log before the latest transaction. Each part contains a hash value, number of used log blocks, and list of transaction records. Each transaction record contains its encryption key, start index of the transaction, and number of address and data blocks in the transaction. Figure 6-8 shows the formal definition of the header and its related structures.

```
Record txn_record := {
    key : key;
    start : nat;
    addr_count : nat;
    data_count : nat;
}.

Record header_part := {
    hash : hash;
    count : nat;
    records : list txn_record;
}.

Record header := {
    old_part : header_part;
    active_part : header_part;
}.
```

Figure 6-8: Header structures.

Every time a transaction is committed, the active part replaces the old part before

the new transaction is added to the active part. This ensures that if a crash happens during the commit, a previous state of the log can be restored during recovery.

We implemented a simple log cache to eliminate the overhead of finding the latest block from the log. The cache is updated every time a new write is issued for the log, to ensure it contains exactly the data stored in the log. Similarly, the cache is flushed every time the log is applied to disk, to ensure the cache is consistent with the log. After a reboot, the cache is repopulated with the data stored in the log.

6.2.2 Logged-Disk Layer and Transactions

Logged-Disk Layer The logged-disk layer is the first place where a nontrivial crash behavior is implemented. When new data is being written to the log, two outcomes are possible based on when the crash happens. If the new header manages to persist on the disk, the write becomes successful, and the new data is visible on the disk. Otherwise, the write gets rolled back during recovery. This behavior of a crash during a write is determined by the oracle by having different crash tokens that dictate if the write is successful or not. The crash semantics of the logged-disk is depicted in Figure 6-9.

```
| ExecCrashBefore :
  ∀ disk user op,
    exec user CrashBefore disk op (Crashed disk)

| ExecCrashWriteAfter :
  ∀ disk user address_list block_list,
    preconditions for a successful write →
    ...
    exec user CrashAfter disk (Write address_list block_list)
        (Crashed (batch_update disk address_list block_list)).
```

Figure 6-9: Crash and recovery semantics of logged disk.

The `ExecCrashBefore` rule indicates that, when the token is `CrashBefore`, any operation can crash without any effect. The `ExecCrashWriteAfter` rule describes a crash after a successful write. If all the preconditions of a successful write are satisfied and the token is `CrashAfter`, then the disk crashes to an after-write state.

6.2.3 Transactions and Transactional-Disk Layer

Transactions To ensure atomicity of each top-level file-system call, we implemented a transaction system on top of the log. Each transaction consists of a list of address and block pairs. Upon commit, the function first deduplicates the transaction list based on the addresses, before writing it to the log. Figure 6-10 shows all the transaction functions.

Operation	Type Signature
`read`	addr → option block
`write`	addr → block → option unit
`commit`	unit
`abort`	unit

Figure 6-10: Transaction operations.

Transactional-Disk Layer The transactional-disk layer abstracts the transaction API to hide the list-based implementation. It transforms the disk and the transaction list into two disks: one that contains the most recent blocks and one that is the disk before the active transaction is started. It also keeps track of whether the transaction is empty or not.

There is no mechanism in the transactional-disk layer to keep track of the transaction size. Whenever a write operation executes, a token determines if the write is successful or it fails due to the transaction being full. Because of this, any write

operation may fail nondeterministically. Figure 6-11 shows the execution rules for `Write` operations.

```
| ExecWriteInbound :
  ∀ s a v u,
    a < disk_size →
    exec u  Cont s (Write a v)
    (Finished (NotEmpty, ((upd (fst (snd s)) a v), snd (snd s))) (Some tt))

| ExecWriteOutbound :
  ∀ s a v u,
    a ≥ disk_size →
    exec u  Cont s (Write a v) (Finished s None)

| ExecWriteInboundFull :
  ∀ s a v u,
    a < disk_size →
    exec u  TxnFull s (Write a v) (Finished s None)
```

Figure 6-11: Write semantics of transactional disk.

Whenever a `Write` operation needs to be executed, if the input address is in-bounds, there are two possible execution paths based on the token: (1) the write goes through successfully and the first disk is updated, or (2) the write fails without any changes.

6.2.4 File-System Structures and File-Disk Layer

Allocators. We implement a generic allocator that we use to manage inodes and data blocks. A generic implementation halves the proof effort required for allocators and demonstrates ConFrm's capability of handling generic implementations.

Inodes. The inode component is a wrapper around its allocator. Inodes are encoded into and decoded from blocks with abstract `encode` and `decode` functions. Using

abstract functions allows us to implement these functions with high performance in the extracted code.

Files. The implementation of files uses a data-block allocator and inodes to implement our file-system API. Each function except `create` has a similar implementation. They first make an access-control check to ensure that the current user is the owner of the accessed file. After a successful authentication, they perform the operation and return an optional value to signify a success or a failure.

The `change_owner` implementation is straightforward because file ownership is recorded in inodes, and changing the owner in the inode changes the owner of the entire file. This is in contrast to DiskSec, where each block has an owner and each block's owner must be changed to change the owner of a file.

6.3 Specifiying security

Security of ConFs is defined as an RDNI specification for each of the compiled file disk operations. The core of these specifications is the equivalence relation between two states. Since ConFs consists of four distinct layers, we have four different but related equivalence relations. We will explain these relations starting from the file disk layer and progressing towards the base layer.

For our confidentiality definition, we chose to treat user data as confidential but the file system metadata as public. Our choice is based on the fact that current file systems expose metadata (e.g., size of directory shows the number of files in it, amount of free space, number of free inodes). ConFs doesn't treat some metadata that can be confidential in a file system as confidential (e.g, private directory contents) since it doesn't implement all the functionalities of widely-used file systems.

6.3.1 File Disk Layer Security

The equivalence relation for file disk layer is the formalization of the idea of *two states are equivalent for a user they have the same structure, and the data owned by that user is identical in both states.* Figure 6-12 shows the formal definition of the relation.

```
Definition same_for_user_except (u: user)
  (exclude: option addr) (d1 d2: file_disk.state) :=

  addrs_match_exactly d1 d2 ∧

  (∀ inum file1 file2,
     exclude ≠ Some inum →
     d1 inum = Some file1 →
     d2 inum = Some file2 →
     (file1.owner = u ∨
      file2.owner = u) →
     file1 = file2) ∧

  (∀ inum file1 file2,
     d1 inum = Some file1 →
     d2 inum = Some file2 →
     file1.owner = file2.owner ∧
     length file1.blocks = length file2.blocks).
```

Figure 6-12: Equivalence relation for two file disk states.

The relation has two parts: (1) files with the same inode number have the same owner and length, and (2) if those files belong to the specified user, then their contents are the same. By requiring files to be identical only for the specified user, `same_for_user_except` captures the intuition of differing in confidential data belonging to other users, whose connection to confidentiality is explained in Chapter 1.

To make `same_for_user_except` usable in the `change_owner` specification,

same_for_user_except takes an optional inode number of the file whose owner is being changed for the following reason.

If the new owner is the user whose the states are equivalent for, then the resulting states would be equivalent only if the files whose owner is being changed are identical in both states. However, the files that belong to the old owner in the starting states are not guaranteed to be identical, since equivalence relation does not require other users' file contents to be the same between equivalent states. Therefore, the relation excludes the file being operated on and ensures other files of the user stay identical.

Since the equivalence relation used in change_owner's confidentiality specification excludes the file that is being operated on, it only provides half of the required security: it restricts the leakage from the changed file to the outside, but not the other way around. The fact that no information leaks from outside into the changed file is covered by change_owner's functional correctness, which states that changed file's contents stay unchanged after the operation.

```
Theorem write_RDNI:
  ∀ n inum adr blk current_user adversary,
    RDNI current_user
         (Write inum adr blk)
         (Write inum adr blk)
         Recover
         (same_for_user_except adversary None)
         (eq adversary) (repeat id n).
```

Figure 6-13: Confidentiality specification for Write operation.

Figure 6-13 shows a typical confidentiality specification for a file disk operation. This particular example states the noninterference of the Write operation. current_user is the user who is executing the Write operation. Two starting states are equivalent with respect to an adversary, which may be identical to current_user, and no file is excluded since we expect the effect of the operation to be identical on

both states. Return value equality is required only if `current_user` is identical to `adversary` because we are only concerned if an adversary can distinguish equivalent or not. Since file disk's disk model is crash-safe, list of reboot functions is a list of identity functions.

Above specification does not ensure confidentiality of the input block. Since input blocks are the same in both programs, even an obviously insecure implementation that directly writes the block to an adversaries file satisfies the specification. To ensure the confidentiality of the input, another specification where inputs are different is needed. Figure 6-14 shows the specification for the input data. The specification excludes the operated file from the equivalence because if the current user is the adversary, then the files after the write will not be the same. We have a similar specification for `extend`, which also takes confidential data as an input.

```
Theorem write_input_RDNI:
  ∀ n inum adr blk1 blk2 current_user adversary,
    RDNI current_user
         (Write inum adr blk1)
         (Write inum adr blk2)
         Recover
         (same_for_user_except adversary (Some inum))
         (eq adversary) (repeat id n).
```

Figure 6-14: Confidentiality specification for `Write` operation with different inputs.

`write_RDNI` and `write_input_RDNI` correspond to nonleakage and noninterference, respectively. They together ensure the confidentiality of `Write` operation, `write_RDNI` guarantees that confidential data on the disk doesn't affect the behavior, and `write_input_RDNI` guarantees the same for the input data.

6.3.2 Transactional Disk Layer Security

ConFrm provides a function that derives an equivalence over implementation states from an equivalence over abstract states, given that a refinement between the two exists. Informally, *the derived equivalence relates two implementation states if there exists two equivalent abstract states that are refined by those implementation states*. Figure 6-15 shows the definition of the derivation function.

```
Definition refines_equivalent
    (equivalent_abs:  abs.state → abs.state → Prop)
    (si1 si2: imp.state) : Prop :=
    ∃ (sa1 sa2: abs.state),
      refines si1 sa1 ∧
      refines si2 sa2 ∧
      equivalent_abs sa1 sa2.
```

Figure 6-15: Equivalence derivation function from ConFrm.

This function is sufficient to derive the equivalence for the transactional disk layer. Figure 6-16 depicts an example of a specification with a derived equivalence for a compiled `Write` operation. In the definition, each program is replaced with their compiled versions and equivalence relation is replaced with the derived relation. Since the state of the transactional disk is different from the state of the file disk, list of reboot functions is replaced with the transactional disk layer's reboot function. One other change is that the theorem weakens the specification to its termination insensitive variant. This was done due to time constraints and will be explained further in section 6.4.

```
Theorem transactional_disk_write_RDNI:
  ∀ n inum a v current_user adversary,
    Termination_Insensitive_RDNI
    current_user
    (compile (Write inum a v))
    (compile (Write inum a v))
    (compile Recover)
    (refines_equivalent (same_for_user_except adversary None))
    (eq adversary) (repeat transactional_disk_reboot_function n).
```

Figure 6-16: Derived confidentiality specification for compiled `Write` operation.

`refines_equivalent` is not always sufficient to derive a suitable equivalence relation. Sometimes, extra conditions are needed to establish the relation between the parts of the implementation state that is abstracted away. The main reason for requiring extra conditions in our case is oracles in ConFrm implicitly dictating the number of execution steps and the types of operations a program takes. Semantics of a program requires consumption of exactly one token per operation executed. This requirement generally implies that two noninterfering programs have to follow the same execution path as explained next.

6.3.3 Logged Disk and Base Layer Security

Both logged disk layer and base layer require extra conditions regarding the structure of the log and the transaction list. Following simple example shows why such conditions are necessary:

```
Definition read a :=
  mv <- transaction_read a;
  if mv = Some v then
    Ret v
  else
    disk_read a
```

Above is a standard `read` implementation to get latest value for an address when there is an active transaction in the system. Proving noninterference of this function requires showing that there exists two executions from related states with the same oracle. Since having the same oracle dictates that programs have to follow the same execution paths, two related states have to contain exactly same addresses in their transactions, although corresponding data could be different.

Since the transactional disk abstraction hides the existence of a separate transaction list, an equivalence relation between two abstract states cannot capture this requirement. In this instance, equivalence relation for implementation needs to be supplemented to make it finer-grained.

This particular example and some other more complicated variants are present in log functions as well. Therefore, we supplemented the equivalence relation with the following extra requirements:

- same addresses are present in the transactions,
- same addresses are present in the log caches,
- there are equal number of transactions in both logs, and
- corresponding transactions in both logs have the same number of address and data blocks

All of the above requirements can be summarized as *equivalent states having the same structure*, which is in line with the intuition behind our file disk equivalence relation.

6.4 Proving Security

As explained in section 5.2, proving confidentiality of our implementations in base layer requires three groups of theorems for each file disk operation: (1) an RDNI confidentiality specification proof, (2) a simulation proof between operations and their implementations, and (3) an oracle refinement independence from confidential data proof. Vertically composable nature of each type allowed us to prove the properties for each layer separately, by deriving intermediate RDNI specifications, instead of a one monolithic proof from file disk layer to base layer.

Confidentiality specification proofs. Confidentiality specification proofs of file disk operations directly follow from the operational semantics of the operations. Since each operation is executed in a single step and file disk is crash-safe, these proofs are straightforward and follow the same pattern.

Simulation proofs. We split simulation proofs to two parts to keep proofs shorter an more manageable: (1) existence of a refined abstract oracle given an implementation execution, and (2) existence of an abstract execution given a refined abstract oracle and an implementation execution. Both of these proofs took advantage of the functional correctness specifications of implementation programs. Biggest challenge regarding simulation proofs is establishing token refinement relations. As explained in 5.2.1, these relations depend on multiple parameters like program itself and reboot functions. Finding the correct relations required multiple iterations and corresponding changes in definitions and proof scripts.

Oracle independence proofs. Oracle independence proofs were the hardest due to them being two-execution proofs and requiring reasoning about two abstraction levels. We split oracle independence theorems into two smaller theorems: (1) two

programs follow the same execution path from related states with the same oracle, and (2) if two programs follow the same execution path from related states with the same oracle, then those oracles refine the same abstract oracle.

One interesting case appeared regarding a write operation that overwrites some data with the same data on the disk, making the operation effectively a 'noop'. The possibility of such a noop write operation makes it impossible to determine if a write succeeds or not after a crash by just examining the disk's final state. This causes a problem in the oracle refinement independence proof, where only one execution of the same write from equivalent states being noop.

If only the final states are considered in oracle refinement, a noop write's oracle can refine both `CrashBefore` and `CrashAfter` tokens. However, a normal write's oracle can refine only one of them, depending on the resulting state. Because of this, The problem arises when the noop write's oracle refines a token that is different from the normal write's oracle can refine, since it cannot be proven that normal write's oracle refines the other token.

To resolve this problem, we included the precise number of steps a write operation runs as well as the required conditions on the crash and reboot states of the disk to oracle refinement relation. This extra information prevents above problem from arising by making conditions of refining two tokens mutually exclusive, establishing that the same oracle cannot refine different tokens from equivalent states.

However, such a precise and low level reasoning was tedious and required significant proof effort. Despite our best efforts, question of whether this can be solved at meta-theoretic level or with another proof strategy that requires less effort remains open.

Termination sensitivity. One concession we had to make was using the termination insensitive RDNI as our final confidentiality specification. We discovered that

termination sensitivity is orthogonal to the other properties and needs an entirely new set of theorems to prove. This orthogonality is fundamental and comes from the definitions themselves. All the theorems in ConFs are about the properties of existing executions. In other words, the reasoning starts with an existing execution and derives required facts from it. However, termination sensitivity requires reasoning in the opposite direction: starting from some facts to establish the existence of an execution.

6.5 Extraction and Trusted Computing Base

ConFs extracts to Haskell using Coq's built-in extraction functionality. To obtain a functional file system, we implemented three unverified components: (1) an interpreter for base layer operations, (2) a directory structure, and (3) FUSE bindings for each system call. The Coq kernel, Haskell base library, implemented components, and the external libraries used in the components are all a part of the trusted computing base.

Chapter 7

Evaluation

This section experimentally answers the following questions:

- Do the confidentiality specifications prevent bugs?
- How much effort was required to develop the frameworks and to use them to prove the security of the file systems?
- How does performance of the systems compare to performance of the existing file systems?

7.1 Specification Trustworthiness

To evaluate the trustworthiness of our specifications, we performed several analyses.

Bug case study. To evaluate whether our confidentiality specifications would eliminate security bugs in SFSCQ and ConFs, we qualitatively analyzed the bugs presented in Section 1.1. Figure 7-1 shows the results. Functional-correctness theorems of both file systems preclude the possibility of integrity bugs. For SFSCQ, `ret_nonleakage`

as defined in Figure 3-2, prevents bugs that are caused by accessing normally inaccessible data due to incorrect permission handling or reading the residual data from newly allocated blocks, and `state_nonleakage` as defined in Figure 3-3, prevents bugs that leads to adding or changing data that is accessible to another user. For ConFs, `RDNI` as defined in 5-5 prevents all the bugs prevented by either `ret_nonleakage` or `state_nonleakage`, since it encompasses both definitions. Overall, the results demonstrate that our theorems preclude the possibility of all studied bugs in both SFSCQ and ConFs.

Description	**Theorem**
Access to deleted files' data [8]	FC
Data leak via unaligned file lengths [7]	ret NL & RDNI
Can set incorrect permissions on new filesystem objects [4]	FC
Data leak through uninitialized memory [2]	FC
A local user may create files that belongs to another user [6]	state NL & RDNI
A local user may be able to read arbitrary files [5]	ret NL & RDNI
Information leak due to permission bypass [3]	ret NL & RDNI

Figure 7-1: Security bugs in various file systems and which theorems preclude each one.

Trusted computing base. Both the frameworks and the file systems assume the correctness of several components. All of the implementations assume that Coq's proof-checking kernel is correct, because it verifies the proofs. SFSCQ and ConFs assume that the Haskell runtime and support libraries (and the underlying Linux kernel) do not have bugs, since they generate executable code through extraction to Haskell. Each file system assumes that its respective framework's model of the disk is accurate.

7.2 Effort

DiskSec and SFSCQ. 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 7-2 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.

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 7-2: Lines-of-code change required to implement DiskSec and apply it to build SFSCQ. Counts measure the diff between DFSCQ and SFSCQ.

ConFrm and ConFs. Since ConFrm and ConFs were built from scratch, we used lines of code as the effort estimate. We broke down the numbers to show how much effort went into each component. Figure 7-3 displays the results. According to data, functional correctness has 5.6x and confidentiality has 9.2x proof overhead compared to the implementation.

Component	Lines of Code
ConFrm	3610
ConFs implementation	2270
ConFs refinements and simulations	4594
Functional correctness	12691
Top-level RDNI proofs	1950
RDNI Transfer proofs	18887
Grand Total	44002

Figure 7-3: Lines-of-code required to implement ConFrm and apply it to build ConFs.

7.3 Performance

We used five benchmarks to measure the performance of our file systems compared to existing file systems: FSCQ and ext4. We used FSCQ as a representative of verified file systems because it is similar to ConFs and SFSCQ, and ext4 as a representative of widely used file systems.

7.3.1 Experimental Setup

To test our file systems, we extracted Coq implementations to Haskell and connected them to the FUSE [21] library to provide POSIX API. We wrote an interpreter function in Haskell for each file system to implement the operations in their disk and memory model. Since ConFs doesn't have directories, we implemented a simple directory structure in Haskell. Directory blocks are written directly to the disk instead of the log. Downside of this is that it incurs an extra disk sync to ensure they are persisted correctly.

For our tests, we used two types of benchmarks: data-heavy and metadata-heavy

benchmarks. Data-heavy benchmarks consists of `makefile`, which creates a file and writes 1MB data on it, and `writefile`, which overwrites the file with 1MB data.

Metadata-heavy benchmarks are `smallfile`, `createdelete`, and `rename`. `smallfile` benchmark creates a file then writes 100B of data to it. `createdelete` creates a file then immediately deletes it. Finally, `rename` creates a file then renames it to an existing file's name.

We tested ext4 in two configurations: checksum-logging and checksum-logging with data bypass. These configurations are similar to ConFs' and SFSCQ's designs, respectively.

All tests are run on a machine with an 3.33GHz Intel Core i7-980X CPU, 6x Samsung 4GB DDR3 1333 MHz memory, and 256GB Samsung 850 EVO SSD disk. We ran each benchmark 25 times and took the average of the results. We didn't observe any outliers in our results.

7.3.2 Results

Figures 7-4 and 7-5 show that SFSCQ performed an order of magnitude better than ConFs on data-heavy workloads and performs comparably in metadata-heavy workloads. Both systems perform better than FSCQ and worse than ext4 in all benchmarks.

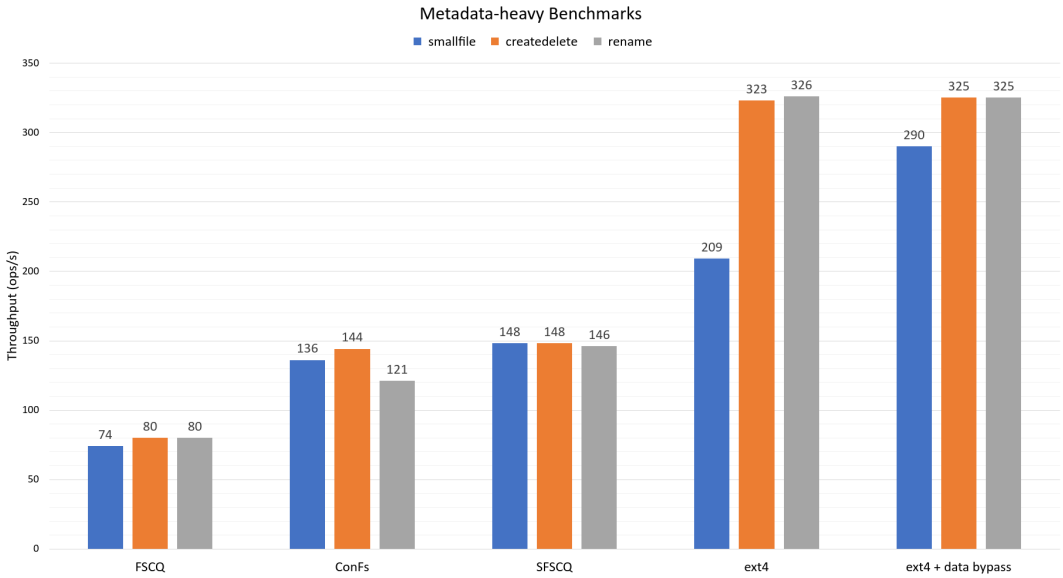

Figure 7-4: Performance comparison for metadata-heavy benchmarks.

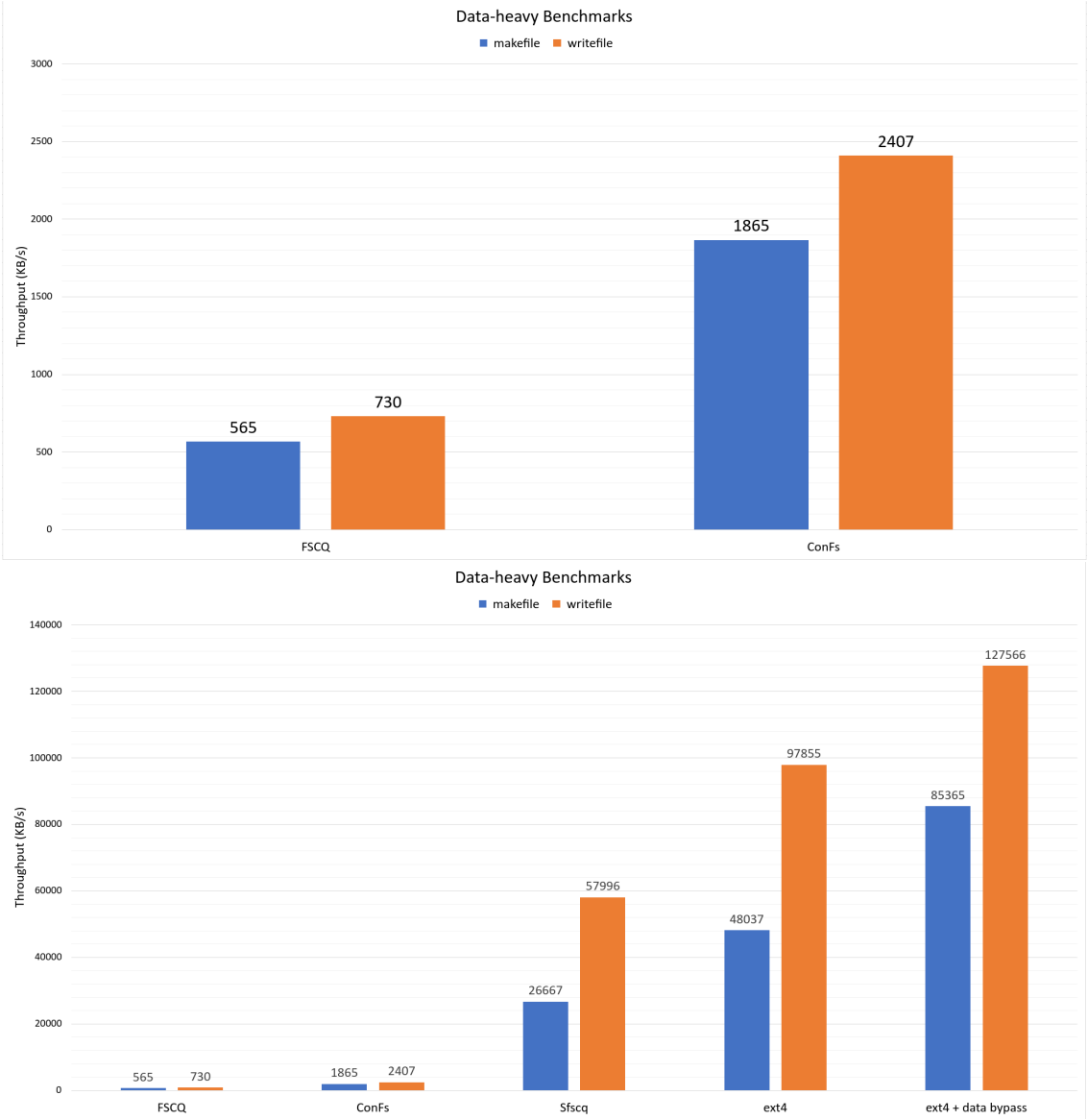

Figure 7-5: Performance comparison for data-heavy benchmarks.

SFSCQ's higher performance compared to ConFs in data-heavy benchmarks is due to fact that SFSCQ permits file data to be directly written on the disk, bypassing the log and overheads related to it. This is supported by the fact that ConFs and SFSCQ performed comparably in metadata-heavy benchmarks.

ConFs's performance speed-up over FSCQ can be attributed to using axiomatic

definitions for inner data structures that are implemented efficiently with native types after extraction. Our experiments with an earlier version of ConFs showed that converting native Haskell types to, and from, extracted Coq types for allocator bitmaps incurred a large performance overhead. We believe this to be true for other non-trivial extracted types. FSCQ and SFSCQ uses an extracted `Word` type to represent disk addresses, which is converted to, and from, native Haskell types when necessary. We believe this is the source of FSCQ's low performance.

Ext4 is written in C and cointains many optimizations that are not implemented in our research prototypes. Therefore it is natural that ext4 outperforms both SFSCQ and ConFs.

Chapter 8

Future Work

Program logics All the proofs in ConFs are done without employing any program logics. Although this was useful in the process of discovering good definitions and specifications, it is not practical to do so in a large-scale system. Proof length and complexity quickly spiral out of control if not meticulously maintained. We believe that a program logic that incorporates oracles would be a good solution to this scalability problem.

Termination sensitivity Although we define both termination-sensitive and insensitive version of RDNI, our experience with ConFs showed that termination sensitivity proofs are mainly orthogonal to the other required proofs. It is an open question whether a framework can be found to unify termination-sensitivity proofs with other proofs.

Relaxing all-possible-distributions requirement. In ConFrm, we solved the secret-dependent outcome probabilities challenge by introducing nondeterminism oracles and requiring the existence of an execution from a related state with the same oracle, as we describe in Chapter 5. Although this is a sufficient condition, it is not

necessary if there is a known distribution, or a set of distributions, over nondeterministic events. We believe that this approach may be adapted to such scenarios, however such adaptation is an open problem.

Oracles for other types of nondeterminism. Nondeterminism oracles allow reasoning about some properties with a probabilistic nature without modelling probability in the formal system. Using oracles to model the crash behavior of systems is one possible use of the nondeterminism oracles. The same idea can be used in other settings that have nondeterminism like concurrency. The exact power of nondeterminism oracles as a proof strategy for probabilistic reasoning remains open.

Chapter 9

Conclusion

This thesis investigates challenges surrounding confidentiality of nondeterministic crash-safe storage systems with rich sharing semantics and lays the groundwork for confidential storage systems with machine-checkable proofs. It introduces two confidentiality specifications, data nonleakage and relatively deterministic noninfluence, and two techniques that accompany them, sealed blocks and nondeterminism oracles, respectively. It also provides two formally verified frameworks, Disksec and ConFrm, as well as two confidential file systems with machine-checkable proofs, SFSCQ and ConFs.

Our confidentiality specifications support specifying a subset of the data stored in the system and allow the flow of nonconfidential information between users. This is in contrast to the traditional specifications, which prohibit any information flow between users. This flexibility makes our specifications suitable for a wider set of systems. Both specifications are supported by techniques that can be used in different contexts. Nondeterminism oracles can be implemented in any system that contains nondeterminism.

SFSCQ and ConFs are the first file systems with machine-checked confidentiality proofs. SFSCQ shows it is possible to obtain strong confidentiality guarantees for

existing file-system implementations with reasonable effort. ConFs demonstrates that the confidentiality of storage systems that manipulate confidential data in a safe manner can be proven even in the presence of nondeterminism.

Bibliography

[1] MITRE CVE List. `http://cve.mitre.org/`.

[2] CVE-2019-11833. Available from MITRE, CVE-ID CVE-2019-11833., May 9 2019.

[3] CVE-2019-9475. Available from MITRE, CVE-ID CVE-2019-9475., February 28 2019.

[4] CVE-2020-24394. Available from MITRE, CVE-ID CVE-2020-24394., August 19 2020.

[5] CVE-2021-1797. Available from MITRE, CVE-ID CVE-2021-1797., December 8 2020.

[6] CVE-2021-4037. Available from MITRE, CVE-ID CVE-2021-4037., December 1 2021.

[7] CVE-2021-4155. Available from MITRE, CVE-ID CVE-2021-4155., December 22 2021.

[8] CVE-2022-29973. Available from MITRE, CVE-ID CVE-2022-29973., May 2 2022.

[9] John Barkley. Introduction to POSIX security. `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-7.pdf`. Accessed: 2023-01-05.

[10] C. Baumann, M. Dam, R. Guanciale, and H. Nemati. On compositional information flow aware refinement. In *2021 2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 79–94, Los Alamitos, CA, USA, jun 2021. IEEE Computer Society.

[11] Bruno 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] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai 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, October 2017.

[13] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai 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, October 2015.

[14] Adam 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, October 2010.

[15] Adam Chlipala. Skipping the binder bureaucracy with mixed embeddings in a semantics course (functional pearl). *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.

[16] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 51–65, 2008.

[17] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-end verification of information-flow security for C and assembly programs. *SIGPLAN Not.*, 51(6):648–664, June 2016.

[18] 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, October 1981.

[19] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 287–305, 2017.

[20] Dan Frumin, Robbert Krebbers, and Lars Birkedal. Compositional non-interference for fine-grained concurrent programs. *arXiv preprint arXiv:1910.00905*, 2019.

[21] FUSE: Filesystem in userspace, 2013. `http://fuse.sourceforge.net/`.

[22] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, 1982.

[23] J. Graham-Cumming and J.W. Sanders. On the refinement of non-interference. In *Proceedings Computer Security Foundations Workshop IV*, pages 35–42, 1991.

[24] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, Broomfield, CO, October 2014. USENIX Association.

[25] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[26] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.*, 32(1), February 2014.

[27] John Mclean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1:37–58, 1992.

[28] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy*, pages 415–429, 2013.

[29] Andrew Myers and Barbara 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, October 1997.

[30] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Computer Systems*, 9(4):410–442, October 2000.

[31] Luke Nelson, James Bornholt, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Noninterference specifications for secure systems. *SIGOPS Oper. Syst. Rev.*, 54(1):31–39, aug 2020.

[32] Andrei Popescu, Johannes Hölzl, and Tobias Nipkow. Proving concurrent noninterference. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs*, pages 109–125, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[33] A.W. Roscoe. CSP and determinism in security modelling. In *Proceedings 1995 IEEE Symposium on Security and Privacy*, pages 114–127, 1995.

[34] J. M. Rushby. Proof of separability a verification technique for a class of security kernels. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, pages 352–367, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[35] Benedikt Schmidt, Simon Meier, Cas Cremers, and David 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.

[36] Cong Sun, Ning Xi, and Jianfeng Ma. Enforcing generalized refinement-based noninterference for secure interface composition. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 586–595, 2017.

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

[38] David von Oheimb. Information flow control revisited: Noninfluence = noninterference + nonleakage. In Pierangela Samarati, Peter Ryan, Dieter Gollmann, and Refik Molva, editors, *Computer Security – ESORICS 2004*, pages 225–243, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[39] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen 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.

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

[41] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans 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, October 2009.

[42] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In

Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS), Dallas, TX, October–November 2017.