ConFrm: Confidentiality Framework for Crash-Safe Storage Systems

Atalay Mert Ileri Frans Kaashoek Nickolai Zeldovich

1 Introduction

As computation and storage paradigms shift toward the cloud, the safety of computation and data is becoming increasingly important. Users expect their data stored in such systems to stay secret against malicious third parties. Precise descriptions of how such a system should behave to ensure confidentiality of the stored data 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.

Writing a confidentiality specification is generally more difficult than writing a correctness specification. To demonstrate the possible hardships of writing a specification that can prevent malicious implementations, we can look at a specification for 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 points to that file. A natural correctness specification for its return value could be "create() returns the index of the previously unused inode that now corresponds to the newly created file." If we were only interested in functional correctness, this could 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 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).

This problem, and others of the same nature, could be solved by writing a fully deterministic specification for create() that exactly pinpoints what it should return and what the file-system state should be after its completion. However, such a specification would be overly verbose, obscuring the important parts of the specification, making it difficult to read and comprehend and essentially defeating one of its key purposes. Therefore, a good confidentiality specification should address such vulnerabilities in its scope but should also be clear and concise so it can be read, understood and reviewed by humans. Finding the right balance between these conflicting requirements is what makes writing a confidentiality specification challenging.

Our proposed solution to the above problems is ConFrm, a framework for implementing and proving the confidentiality of storage systems in a modular fashion. ConFrm provides a noninterference definition with better guarantees for nondeterministic behavior, as well as the required tools and necessary conditions for safety-preserving abstractions. These two components of ConFrm enable us to overcome the limitations of previous works and provide a simple yet powerful tool for implementing safe storage systems.

To test the capabilities of ConFrm, we implemented ConFS, a confidential file system that is crash-resistant via checksum logging, transactional operations, and coarse-grained discretionary access control.

Both ConFrm and ConFS are implemented in Coq. All proofs are fully machine-checked to ensure their correctness.

2 Related Work

ConFrm and ConFs build on a diverse body of prior work. We will explain these works throughout this section.

Noninterference properties. There is a significant body of work about formalizing noninterference properties [19, 25, 26, 29, 30, 32, disksec]. ConFrm's relatively deterministic noninterference builds upon this existing work.

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 [23, 26], CertiKOS [15], Ironclad [20] and DiskSec [??]. 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 [17] 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 provides weaker guarantees for crashes.

This phrase is not grammatically parallel with the earlier ones in the list. Perhaps use "and sufficiently strong crash guarantees"?

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.

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

Huh? Aren't proofs like yours inherently static?

SeLoc[??] uses double weakest preconditions to prove noninterference for fine-grained concurrent programs. It is built on top of IRIS[??], 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, ConFrm is standalone and lightweight, but does not offer the full array of tools SeLoc offers.

Sequential composability and confidentiality-preserving refinements. Since it is known that noninterference is not preserved in the general case[??], there is a body of work that tries to identify the conditions that make refinements noninterference-preserving.

Sun et. al. [??] 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. [??] 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.

3 Definitions and Notation

Next we will present standard notations and definitions that are used in the rest of the text. All the definitions presented here are for the concepts that are commonly found in other works. Since there are multiple definitions for

those
noninterference, termination sensitivity, and simulation in the literature, ~~ones~~ that we used are also given below. We modified termination sensitivity and noninterference definitions to be parameterized over two programs, instead of one. The reason for this is that, in the context of this work, a program refers to a routine and its inputs. This ~~Aforementioned~~ modification allowed us to reason about the same routine with different inputs. Any definition or notation that is used for new contributions of the work will be introduced throughout the proposal.

We use a and i to denote abstraction and implementation states respectively, and use s when the distinction is irrelevant. $s = p \Rightarrow (s', v)$ denotes the execution of program p from state s that results in state s' and returns v. $\approx$ is used to represent an equivalence relation between two abstract states.

we

It's still super-weird to use "=" to mean something beside equality! Guaranteed to confuse readers.

The equivalence relation that is used in ConFs is parameterized by a user and states that:

- Each inode number is either used in both or free in both states.
- Files that are ~~pointed~~ referenced by the same inode number have the same owner and the same length.
- If the owner of those files is the provided user, then the file contents are the same as well.

OK, so no metadata of the file system may be influenced by secrets? That's *quite* restrictive considering realistic scenarios. It may be worth pointing out how much of this challenge comes from assuming a low-level application language like C, where inodes etc. can't be presented as an abstract type like in e.g. OCaml.

Informally, our relation relates two states that only differ in the contents of the files that are not owned by the provided user parameter.

R denotes a refinement relation between an abstraction and its implementation. $R(p)$ represents the refining implementation program, and $R(\approx)$ represents the derived equivalence relation between two implementation states. $R(\approx)$ is defined as

huh?

$$i_1 \; R(\approx) \; i_2 := \exists \; a_1 \; a_2, \; i_1 \; R \; a_1 \; \wedge \; i_2 \; R \; a_2 \; \wedge \; a_1 \approx a_2$$

Noninterference

$$\begin{aligned}
&\forall \; s_1 \; s_2 \; s_1' \; v, \\
&s_1 \approx s_2 \rightarrow \\
&s_1 = p_1 \Rightarrow (s_1', v) \rightarrow \\
&\exists s_2', \; s_2 = p_2 \Rightarrow (s_2', v) \; \wedge \; s_1' \approx s_2'.
\end{aligned}$$

Termination Sensitivity

$$\begin{aligned}
&\forall \; s_1 \; s_2 \; s_1' \; v, \\
&s_1 \approx s_2 \rightarrow \\
&s_1 = p_1 \Rightarrow (s_1', v) \rightarrow \\
&\exists s_2' \; v', \; s_2 = p_2 \Rightarrow (s_2', v').
\end{aligned}$$

Simulation

$\forall\ i\ i'\ a\ v,$
$i\ R\ a \rightarrow$
$i = R(p) \Rightarrow (i', v) \rightarrow$
$\exists a',\ a = p \Rightarrow (a', v)\ \wedge\ i'\ R\ a'.$

confusing free variable?

4 Problems and Motivation

4.1 Unsafety from Probability

We are trying to tackle two problems in this work. The first issue is the interaction between nondeterminism and noninterference. More specifically, standard noninterference definitions fail to address leakages that result from different frequencies of possible execution traces of the same program. A simple example of this weakness can be seen below:

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

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.

Let's say that two states are equivalent if they are the same for their non-secret parts. Since any pair of state and return value is a valid execution, it is possible to find another execution from a related state. This makes it satisfy the noninterference definition.

(though I would write "nonsecret")

Although the above program satisfies the noninterference definition, the secret bit can be determined via the observed frequency of repeated calls. When we look at the frequencies of output values, we can see that they correlate with the value of the secret bit.

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

Any adversary who can observe the output of the function sufficiently many times can infer the value of the secret bit with high confidence. These types of vulnerabilities are not limited to usage of randomization. They also manifest themselves when there are random events that can affect the behavior of the system.

The main example we will consider is crash and recovery of a storage system. If the recovery behavior of the system is dependent on the secret data stored in it, then the system may contain a vulnerability of this kind.

4.2 Unsafe Data Handling

The second problem observed involves handling the data in a potentially unsafe way. Many storage systems process the data provided to them before storing it. A potentially malicious developer can take advantage of this capability to leak information.

A classic example of this phenomenon is branching on confidential data. If not done carefully, it can result in leakage of confidential data. This does not mean that branching on confidential data should be avoided at all costs.

Data deduplication in a storage system in a good example to demonstrate this effect. For this example, we will treat data contents as secret but metadata as public. Therefore a user noticing extra space savings is fine as long as he cannot infer contents of the data from this knowledge.

Deduplication requires branching on confidential data, which can be abused if not handled carefully. Consider the following obviously unsafe implementation:

```
write_to_txn(v)
  if (v is in txn)
    return false
  else
    add_to_txn(v)
    return true
```

This implementation allows an adversary to query the contents of the transaction. A simple solution that can be considered to solve this problem is prohibiting branching control flow based on confidential data. This can be achieved by using a deeply embedded language that doesn't have an operation to compare secret data or treat secret data as an abstract object that cannot be compared. Such approaches have the advantage that any program written in such a language is noninterfering by construction.

It is quite a strong property but also a restrictive one which eliminates many possible efficient implementations.

However, it is possible to implement a safe deduplicator. One way would be performing the process after the transaction is finalized but before committing it. Such a deferred deduplication will still branch on confidential data but without revealing it.

Except through timing, etc.!

An ideal solution would prohibit unsafe implementations while allowing the safe ones with as little constraint as possible.

5 Solution Approach

To address the problems stated above, we used 3 approaches:

- Quantification of nondeterminism via tokens to solve unsafety from probability.
- Abstraction via a layer system to solve unsafe data handling.

- Property transfer via simulation proofs to enable usage of layers while maintaining safety.

5.1 Quantification of Nondeterminism

To be able to address unsafety coming from certain kinds of probabilistic behavior, we decided to quantize the nondeterminism. We enhanced our execution semantics with a sequence, such that, whenever a nondeterministic choice needs to be made during execution, the sequence will contain the option that will be chosen. For every possible execution trace a program has under general nondeterministic semantics, there should be a sequence that will guide the execution to that result.

Our assumption is that nondeterminism in the system is independent of the contents of secret data but can depend on the public metadata. For example, size of the data can affect which nondeterministic choices are available but actual data cannot. This assumption is enforced as a proof obligation to use our technique.

This approach has two benefits. First and most significant, it enables us to turn nondeterministic executions into relatively deterministic ones. Relatively deterministic execution is defined as an execution trace that is deterministic given a particular sequence of nondeterministic choices.

A simple example of this is execution of an operation that generates an encryption key.

A nondeterministic execution rule would be

$$\forall\ s\ key,\ \ s = GenerateKey \Rightarrow (s, key)$$

This rule states that when we run GenerateKey we can get any key as a result of the execution.

A relatively deterministic execution rule would be

$$\forall\ s\ key,\ \ s = [key]GenerateKey \Rightarrow (s, key)$$

The difference between this rule and the previous one is subtle but crucial. According to this rule, there is only one possible key you can get when you execute GenerateKey in these specific conditions.

Better to avoid this sense of "you" in formal writing.

In some sense, it can be understood that which key you will get is predetermined by the state of the world. Program execution is deterministic relative to the state of the world. In contrast, you could get any key in the nondeterministic execution regardless of the state of the world.

With this modified definition, we managed to state and prove a stronger version of noninterference: relatively deterministic noninterference (RDNI). RDNI requires that, for any sequence of nondeterministic choices, if there is a completed execution of a program from a state, then there exists another execution of a related program from a related state with the same choice sequence.

Strange to require exactly the same sequence! Couldn't a reasonable optimization lead to new usage of nondeterminism?
Oh, but then later you *do* allow different choice sequences! Better to be clearer here, so folks don't get worked up over nothing.
I think I got confused comparing different executions of one program vs. looking at implementation against spec.

$$
\begin{aligned}
&\forall\ o\ s_1\ s_2\ s_1'\ v,\\
&s_1 \approx s_2 \rightarrow\\
&s_1 = [o]p_1 \Rightarrow (s_1', v) \rightarrow\\
&\exists s_2',\ s_2 = [o]p_2 \Rightarrow (s_2', v)\ \wedge\ s_1' \approx s_2'.
\end{aligned}
$$

Fig. 1: Relatively Deterministic Noninterference

RDNI's requirement for matching execution from any choice sequence ensures that no matter which nondeterministic branch is followed during the execution, there will be an indistinguishable execution from an equivalent state. This one-to-one correspondence implies that a state cannot be distinguished based on repeated observation of the system.

One observation that can be made is that having an existing execution with the exact same oracle is a sufficient but not necessary condition to prevent information leakage caused by relative frequency of outcomes. However, a stronger definition is required to ensure confidentiality in the traditional sense.

Since ConFrm's model of nondeterminism contains events whose outcome is not directly observable (e.g. generation of a new key) and events whose outcome is observable (e.g. a system crash), contents of the confidential data shouldn't be inferable from the knowledge of what nondeterministic events occurred.

By enforcing same-oracle executions, ConFrm treats all events as observable and ensures that, even with the full knowledge of nondeterministic events, an adversary cannot infer the contents of confidential data.

If we revisit the random bit example, we can show that it indeed does not satisfy RDNI. For the choice sequence that chooses the first random bit as 1, there is no indistinguishable execution between the state where the secret bit is 0 and the state where the secret bit is 1.

We also modified the termination sensitivity and simulation definitions to integrate relative determinism. Modification of termination sensitivity was straightforward: executions are parameterized by the same oracle. Modifying simulation definition required a more complicated approach. We extended traditional refinements with an oracle refinement relation.

An oracle refinement relation is parameterized by a user, an implementation state and an abstract program and will be denoted with R_o. We omit the user, state and program parameters in notation when it is clear from the context. Our modified relation is

$$
\begin{aligned}
&\forall\ o_i\ o_a\ i\ i'\ a\ v,\\
&i\ R\ a \rightarrow\\
&o_i\ R_o\ o_a \rightarrow\\
&i = [o_i]R(p) \Rightarrow (i', v) \rightarrow\\
&\exists a',\ a = [o_a]p \Rightarrow (a', v)\ \wedge\ i'\ R\ a'.
\end{aligned}
$$

What makes for a sound choice of oracle relation? The technique looks too easy / widely applicable as stated.

Fig. 2: Simulation with Oracles

5.2 Abstraction via layer system

Abstraction is a useful tool when dealing with large and complex software systems. It allows developers to hide implementation details of different modules of a software program and think purely in terms of its abstract behavior.

ConFrm's layer system streamlines the abstraction of implementations by providing tools that ~~make~~ simplify defining new DSLs and proving an implementation exhibits the same behavior with the defined abstraction ~~easier~~. Once an implementation is abstracted away with a layer, it becomes completely opaque to the rest of the system. ConFrm guarantees that an implementation that simulates a confidential layer is also confidential.

Layering enables decoupling of system implementation and the proof technique that will be used to prove confidentiality of the system. It prevents the constraints associated with any particular proof technique from affecting the entire system. For example, DiskSec's sealed-block approach prohibit branching on confidential data to provide substantial help in proof effort. However, this makes optimizations like checksum logging impossible to implement. Layering allows developers to write such implementations, abstract them away to create an interface that conforms to the restrictions of a proof approach then use said proof approach to prove confidentiality of their system.

Additionally, the operations and state of an abstraction are simpler than its underlying implementation, so proving their confidentiality becomes less complex as well. Also, implementations under a layer can be changed without affecting other proofs in the system.

5.3 NI Transfer via Simulation Proofs

It is known that, in the general nondeterministic case, noninterference is not preserved under simulation but is preserved under bisimulation. This creates a problem for abstraction techniques, which rely on simulation proofs to show the behavior of the implementation is a subset of the abstraction's. Since bisimulation is a stricter requirement than simulation, the amount of detail an abstraction can omit is significantly reduced. Furthermore, it cannot introduce any new behavior or remove any existing behavior in an abstraction.

This restriction eliminates one of the best advantages of abstraction techniques, namely simplifying a system's model. Additionally, bisimulation proofs are much more cumbersome compared to simulation proofs.

However, we show that RDNI is preserved under simulations combined with some auxiliary properties. This relieves proof burden ~~of~~ the developer significantly as well as increases the power of abstraction layers. This preservation makes it possible to use traditional abstraction techniques to construct complex systems as isolated, self-contained pieces.

Our core theorem states that, if an abstract layer is RDNI w.r.t. $\approx$, then an implementation layer that refines the abstract layer is RDNI w.r.t. $R(\approx)$.

To be able to employ the theorem in termination-sensitive way, one needs to prove that programs he is reasoning about are termination-sensitive with

respect to $R(\approx)$.

6 Contributions

The contributions of my thesis, subject to this proposal, will be:

- A new definition for confidentiality that addresses unsafety due to nondeterminism. (Completed)
- Identification of required properties for preservation of confidentiality across implementation layers. (Completed)
- A framework for proving a new confidentiality property of multi-layer system implementations. (Completed)
- A file-system implementation with machine-checkable proofs of confidentiality.

7 Proposed Timeline

- **July 1st, 2021**: Finish ConFS implementation.
- **August 1st, 2021**: Finish testing extracted artifact.
- **June 1st, 2021**: Finish background, definitions and addressed problems chapters.
- **July 1st, 2021**: Finish proposed solutions, ConFrm and ConFs chapters.
- **August 1st, 2021**: Finish related work and discussion chapters.