Probability from Possibility: Probabilistic Confidentiality for Storage Systems Under Nondeterminism

Atalay Mert Ileri
Computer Science
Kansas State University
Manhattan, USA
atalay@ksu.edu

Nickolai Zeldovich
CSAIL
MIT
Cambridge, USA
nickolai@csail.mit.edu

Adam Chlipala
CSAIL
MIT
Cambridge, USA
adamc@csail.mit.edu

Frans Kaashoek
CSAIL
MIT
Cambridge, USA
kaashoek@mit.edu

***Abstract*—Nondeterminism, such as system crashes, poses an important challenge to the security of storage systems by making leakages possible through secret-dependent result probabilities. This paper proposes a new possibilistic confidentiality specification prohibiting such probabilistic leakages. Our specification is preserved under simulation to enable modularity and is sequentially compositional. We implemented our specification in a framework that contains structures to implement storage systems and prove their confidentiality in a modular fashion. On top of our framework, we implemented the first crash-safe file system with a termination-insensitive version of our specification and machine-checkable confidentiality proofs. Our evaluation shows that proving confidentiality incurs 9.2x proof overhead per line of implementation code. Both our framework and file system are implemented in Coq and extracted to Haskell to obtain an executable artifact.**

***Index Terms*—formal methods, formal verification, nondeterminism, confidentiality, security, hyperproperties, storage systems, probabilistic security, Coq**

I. Introduction

Storage systems are an integral part of many software systems we use daily. Users expect their data stored in such systems to stay secret. This paper investigates a key confidentiality challenge surrounding storage-system verification under nondeterminism and presents a solution. We will present the challenge, describe our threat model, and explain our contributions.

A. Probabilistic leakage

In addition to the standard challenges like confidentiality being a hyperproperty and the complexity of storage systems, nondeterminism poses unique challenges in specifying and proving confidentiality. This section will focus on a challenge arising from nondeterminism in implementation.

The challenge arises because an adversary may infer the secret information stored in the system if the distribution of the result of a function is dependent on a secret. We call this behavior *a probabilistic leakage*. Such a vulnerability may exist even when the possibility of observing a particular result is independent of the secret. Figure 1 shows a simple example of this challenge.

This research was supported by NSF awards CNS-1563763 and CNS-1812522, and by Google.

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

Fig. 2: Distributions of return values for each state.

```
maybe-leak() :=
    if (get_random_bit() == 1)
      return secret_bit
    else
      return get_random_bit()
```

Fig. 1: Example program that leaks information via result 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. Therefore, by observing 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 Figure 2 show that they correlate with the value of the secret bit. Any adversary aware of this behavior can infer the value of the secret bit with a certain confidence.

These types of vulnerabilities are not limited to the usage of randomization. They also manifest themselves when other random events can affect the system's behavior. For example, in storage systems, a source of randomness comes from the possibility of a system crash at any execution point. In the real world, at any point in time, there is a certain probability of the system crashing. Moreover, this probability depends on many complex factors that make it hard to estimate precisely. Modeling crashes as nondeterministic events instead of random ones can circumvent this hardship. However, such a modeling

choice does not change the fact that an unknown probability is associated with each nondeterministic event's materialization. Since the probabilities are unknown, a technique that addresses these vulnerabilities should work regardless of the distribution.

B. Threat Model

Our goal in verifying a storage system is to ensure the absence of bugs in its implementation that can compromise the confidentiality of user data. This includes a range of potential issues: for example, a file system with incorrect or missing permission checks might allow an adversary to directly access confidential data; a bug in crash-recovery code could result in one user's files being corrupted with another user's data after a crash, as was the case in ext4 [17], or could preferentially commit or abort in-progress operations depending on the contents of confidential files, when using checksum logging such as in ext4 [19], [32]; a bug in a de-duplicating file system could leak data through exposed reference counts to shared data blocks, etc.

We would like to consider the developer well-meaning but error-prone and capable of inadvertently introducing exploitable vulnerabilities if left to their own devices. To this end, we would like to have confidence that the storage system is secure purely based on the storage system's confidentiality specification if it is proven that it satisfies the specification. In other words, our goal is to show that an adversary with the capabilities within our threat model cannot obtain confidential data from any implementation that satisfies our specifications.

We adopt a threat model where the adversary can examine the source code and run an application on top of the storage system as one of its users. The application is limited to interacting with the storage system through its specified storage API. As a result, the adversary cannot interact with the system after a crash until recovery is completed.

Our threat model focuses on proving that the storage-system implementation has no confidentiality vulnerabilities rather than proving the absence of vulnerabilities in the environment outside the storage system. Thus, we assume that our model of the storage hardware's operation is correct and the adversary does not have physical access to the hardware. We are not concerned with bugs in unverified software or hardware outside the storage system or users mounting malicious disk images. We prove that an initialization process produces a correct image.

One of the limitations of our formalization is that it does not model execution time; as a result, our threat model assumes that the adversary does not exploit timing channels. We believe this is a reasonable trade-off, given that our actual goal is dealing with unintentional mistakes in the storage system implementation. We leave extending the execution model to be timing-sensitive to future work.

C. Contributions

This paper has four main contributions listed below:

- **RDNI**, a possibilistic confidentiality specification incorporating crash-reboot-recovery processes that provides probabilistic confidentiality guarantees.
- **A metatheory** for transferring RDNI through abstractions, including modified refinement and simulation definitions.
- **ConFrm**, a framework for specifying and proving confidentiality of storage systems with RDNI specifications. ConFrm implements RDNI and its metatheory and supports implementing systems as layers of abstractions, defining refinements, and proving simulations.
- **ConFs**, the first crash-safe file system with a termination-insensitive RDNI specification accompanied with a machine-checked confidentiality proof. ConFs is implemented in ConFrm.

The source code of ConFrm and ConFs is publicly available at https://github.com/Atalay-Ileri/ConFrm.

II. Preliminaries

Our computational model is called *an oraclized system*. An oraclized system is a tuple $\langle \mathbf{O}, \mathbf{U}, \mathbf{S}, \mathbf{P}, \mathbf{R}, exec \rangle$ where $\mathbf{O}$ is a set of oracles, $\mathbf{U}$ is a set of users, $\mathbf{S}$ is a set of states, $\mathbf{P}$ is a set of programs, $\mathbf{R}$ is a set of results, and $exec$ is a relation defined over $\mathbf{O} \times \mathbf{U} \times \mathbf{S} \times \mathbf{P} \times \mathbf{S} \times \mathbf{R}$ which represents operational semantics. $(o, u, s, p, s', r) \in exec$ means that when user u runs program p from state s with oracle o, the end state is s' and the result is r. We say that an oracle o *leads to an execution* of program p from a start state s for user u if there is a final state s' and a result r such that $(o, u, s, p, s', r) \in exec$.

Throughout the paper, we will treat relations as sets of tuples and write $S(i_1, \ldots, i_n)$ to indicate $(i_1, \ldots, i_n) \in S$ for a set S. We will also leave the oraclized system implicit in our definitions to improve readability.

One crucial requirement for our oraclized systems is *relative determinism*. Relative determinism states that the oracle must capture all the nondeterminism in the system. This is ensured by having each oracle lead to at most one execution for any program executed by any user from any starting state. Formally, our models must satisfy the following condition to achieve this:

$$\forall\ o\ u\ s\ p.\ |\{(s', r)|\ exec(o, u, s, p, s', r)\}| \leq 1$$

We want to clarify that programs in our model are procedures with their input arguments. For example, in this formalism, the program that writes 0 to the disk is considered different from the program that writes 1 to the disk, even though they may be using the same implementation. One side effect of this choice is that our definitions must be over two programs to reason about the same implementation with different input arguments. We decided to use this formalism to align with our Coq formalization.

We define confidentiality as state indistinguishability. The intuition behind this treatment is that if a user cannot access confidential data, they can't distinguish two states that differ only in what is confidential to them. Another way to phrase

this view is that two states should look and behave the same to a user if they differ only in the confidential data.

We formalize state indistinguishability as a family of equivalence relations parameterized by a user, denoted as eqv_u. The parameterization allows us to model systems where confidential information differs for different users. These equivalence relations implicitly determine what is confidential. Any part of the state that can change without breaking the equivalence is considered confidential for that user.

We model the probability of a nondeterministic event happening in real life, which we call *materializing*, as a family of probability distributions over oracles. However, some oracles may not be able to materialize for every program or every user. For example, a system may model crashes during privileged execution differently from the crashes from ordinary user executions by using different oracles. In that case, the oracle for a crash during privileged execution cannot materialize during the execution for an ordinary user. To accommodate these distinctions, our family of distributions is parameterized by a (user, state, program) tuple. We denote such distributions with $X_{(u,s,p)}$.

Let $X_{(u,s,p)}$ be a family of probability distributions over $\mathbf{O}$. We say X is *compatible* with an oraclised system S if it assigns nonzero probability to an oracle whenever that oracle leads to an execution of p from s for u, and vice versa. Compatibility captures the fact that only the oracles that lead to an execution have a positive probability of materializing and expressed formally as

$$\texttt{Compatible}\ (X) :=$$
$$\forall\ o\ u\ s\ p.$$
$$X_{(u,s,p)}(o) > 0 \leftrightarrow \exists\ s'\ r,\ exec(o, u, s, p, s', r)$$

We say that X is *invariant* for user u and programs p_1 and p_2 under a family of equivalence relations eqv if the distributions don't change between equivalent states. Formally,

$$\texttt{Invariant}\ (X, u, p_1, p_2, eqv) :=$$
$$\forall\ s_1\ s_2.\ eqv(u, s_1, s_2) \rightarrow X_{(u,s_1,p_1)} = X_{(u,s_2,p_2)}$$

An invariant distribution under a family of equivalence relations can be interpreted as materialization probability being independent of confidential data since equivalence determines confidentiality.

We define the probability of observing a particular result from a (u, s, p) tuple as the total probability of materialization of oracles that lead to executions with that result. Formally, given an oraclised system S and a compatible distribution X, we can define a family of result distributions $X^{\mathbf{R}}_{(u,s,p)}(r)$ as

$$X^{\mathbf{R}}_{(u,s,p)}(r) = \sum_{o \in \{o' \mid \exists\ s',\ exec(o',u,s,p,s',r)\}} X_{(u,s,p)}(o)$$

Note that X's compatibility combined with the relative-determinism condition ensures that each $X^{\mathbf{R}}_{(u,s,p)}$ is a probability distribution over results.

Finally, we define *probabilistic leakage* in oraclized system S for programs p_1 and p_2 under a family of equivalence relations eqv as the existence of a compatible invariant distribution and two equivalent states for a user where result distributions are different. We formalized probabilistic leakage for an oraclized system S as follows:

$$\texttt{Probabilistic-leakage}\ (u, p_1, p_2, eqv) :=$$
$$\exists X\ s_1\ s_2.$$
$$\texttt{Compatible}\ (X) \wedge$$
$$\texttt{Invariant}\ (X, u, p_1, p_2, eqv) \wedge$$
$$eqv(u, s_1, s_2) \wedge X^{\mathbf{R}}_{(u,s_1,p_1)} \neq X^{\mathbf{R}}_{(u,s_2,p_2)}$$

	Return Values	
Generated Bits	Secret = 0	Secret = 1
0, 0	0	0
0, 1	1	1
1	0	1

Fig. 3: Possible executions of `maybe-leak`

$$\texttt{Matching-execs}\ (u, p_1, p_2, eqv) :=$$
$$\forall\ o\ s_1\ s_2\ s_1'\ r.$$
$$exec(o, u, s_1, p_1, s_1', r) \rightarrow$$
$$eqv(u, s_1, s_2) \rightarrow$$
$$\exists\ s_2'.\ exec(o, u, s_2, p_2, s_2', r)$$

Fig. 4: Formalization of matching-executions property.

III. Relatively Deterministic Noninfluence

Section I shows that the probability of observing a result depending on a secret leads to confidential data leakage. To address this challenge, this paper introduces a new possibilistic confidentiality definition that we call *Relatively Deterministic Noninfluence* (RDNI) that implies probabilistic confidentiality.

A. Probability preservation

Possibilistic definitions can be interpreted as showing two sets of executions from equivalent states having the same set of possible results. Since nondeterminism is the reason behind multiple possible results for the same program from a state, we can think that each nondeterministic event corresponds to a result.

One way to relate the result probabilities is by ensuring that if a nondeterministic event leads to a result from a state, it also leads to the same result from all equivalent states. We call this property *matching executions* since it implies that one can match each execution from a state with another execution from an equivalent state. Figure 4 displays the formal definition of the matching-executions property, and Figure 5 visualizes it. Throughout the paper, dotted lines represent things that are posited to exist.

Figure 3 illustrates how the example in Figure 1 does not satisfy `Matching-execs`, although it satisfies conventional noninfluence. The last row shows that when the generated bit is 1, the equivalent states have different return values.

One important property of `Matching-execs` is that, due to the relative determinism of the system, the matching is unique and 1-to-1 if it exists.

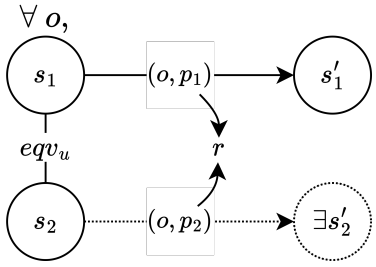


Fig. 5: Visualization of matching executions property.

Our main theorem shows that matching executions is a necessary and sufficient condition for the absence of probabilistic leakages. The intuition is that if the nondeterministic events lead to the same result from equivalent states, then no matter what event ends up happening in the real world, users will observe the same result from equivalent states. Since the probability of observing a result is the total probability of nondeterministic events that lead to that result, and each nondeterministic event leads to the same result from equivalent states, this will imply that the probability of observing a result is the same between equivalent states. Theorem III.1 states our probability-preservation result. Proof of the theorem can be found in the Appendix.

Theorem III.1 (Probability preservation)**.** *Let S be an oraclized system,*

$\forall\ u\ p_1\ p_2\ eqv.$
$\quad$ `Matching-execs`$(u, p_1, p_2, eqv) \leftrightarrow$
$\quad \neg$ `Probabilistic-leakage`(u, p_1, p_2, eqv)

B. Relatively Deterministic Noninfluence

Even though `Matching-execs` implies probabilistic confidentiality, it is insufficient to be a comprehensive confidentiality specification for a multi user system. There are two aspects it falls short of: it doesn't specify how an execution should impact other users' perception of the system and doesn't ensure the confidentiality of the stored data after the execution.

In a confidential multi user system, a user's activity shouldn't break the confidentiality for himself and other users. If equivalent states become distinguishable after the execution of the same program, a user can infer the initial state by observing the final state and reasoning backward. So, a desired property for confidential multi user systems is that equivalent states should stay equivalent if a user takes the same actions in both states. Our RDNI specification enhances `Matching-execs` with this requirement by requiring equivalent states to stay equivalent if two program arguments are the same. Below is the formal definition of our specification.

`RDNI` $(u, p_1, p_2, eqv) :=$
$\quad \forall\ u'\ o\ s_1\ s_2\ s'_1\ r_1.$
$\quad\quad exec(o, u, s_1, p_1, s'_1, r_1) \rightarrow$
$\quad\quad eqv(u', s_1, s_2) \rightarrow$
$\quad\quad \exists\ s'_2\ r_2.\ exec(o, u, s_2, p_2, s'_2, r_2) \wedge$
$\quad\quad (p_1 = p_2 \rightarrow eqv(u', s'_1, s'_2)) \wedge$
$\quad\quad (u = u' \rightarrow r_1 = r_2)$

Termination Sensitivity: The RDNI definition requires, for each execution, an execution to exist from any equivalent state. This requirement is called *termination sensitivity*. In its essence, termination sensitivity implies that an adversary cannot learn confidential information by observing if the program terminates. In the nondeterministic case, termination insensitivity requires adding nontermination as a possible result.

The relatively deterministic nature of RDNI allows us to define a termination-insensitive variant without needing to add nontermination as a possible result. In this variant, pairs of executions with the same oracles are required to have the same result, but there are no restrictions for a pair of executions with different oracles. The formal description is below.

`TI-RDNI` $(u, p_1, p_2, eqv) :=$
$\quad \forall\ o\ u'\ s_1\ s_2\ s'_1\ s'_2\ r_1\ r_2.$
$\quad\quad exec(o, u, s_1, p_1, s'_1, r_1) \rightarrow$
$\quad\quad exec(o, u, s_2, p_2, s'_2, r_2) \rightarrow$
$\quad\quad eqv(u', s_1, s_2) \rightarrow$
$\quad\quad (p_1 = p_2 \rightarrow eqv(u', s'_1, s'_2)) \wedge$
$\quad\quad (u = u' \rightarrow r_1 = r_2)$

An important limitation of the termination-insensitive definition is that it does not imply `Matching-execs`. As a result, systems that only satisfy `TI-RDNI` do not benefit from the probability-preservation theorem.

IV. ConFrm

ConFrm is a framework for implementing and proving the confidentiality of storage systems. It contains the RDNI implementation, structures to implement storage systems, definitions for defining abstractions, and relevant metatheory to prove the confidentiality of implementations.

A. System Structures

ConFrm splits defining a system into two components that build on each other: a core and a layer. Cores capture what is unique in a system, like its states and its operations. Layers augment cores with what is common in all system definitions, like sequencing of operations and recovery semantics. ConFrm adds the recurring parts to defined cores to create layers, reducing the implementations' clutter. This separation makes implementing systems easier and avoids redundant work on the developer's side.

1) Cores: ConFrm introduces cores as the main way to model a system. A core has six components following the definition in section II. It consists of a set of users, a set of states, a set of operations, a set of nondeterminism tokens, the execution semantics, and a relative-determinism proof. The set of operations corresponds to programs from section II. Nondeterminism tokens are our implementation of oracles from section II, where each token corresponds to a nondeterministic event. An example of a core can be found in the appendix.

$exec^r$:=
| Exec-finished :
  $\forall\ p\ rec\ u\ o\ s\ s_f\ r.$
  $exec(o, u, s, p, s_f, (\texttt{Finished}\ r)) \rightarrow$
  $exec^r([o], [\,], u, s, p, rec, s_f.(\texttt{Finished}\ r))$
| Exec-recovered :
  $\forall\ p\ rec\ u\ o\ l_o\ s\ s_c\ s_r\ rf\ lrf\ r.$
  $exec(o, u, s, p, s_c, \texttt{Crashed}) \rightarrow$
  $exec^r(l_o, lrf, u, rf(s_c), rec, rec, s_r, r) \rightarrow$
  $exec^r((o :: l_o), (rf :: lrf), u, s,$
  $p, rec, s_r, \texttt{Recovered})$

Fig. 6: Provided recovery semantics.

Crashes: ConFrm supports crash semantics by defining two different execution results: `Finished` and `Crashed`. A `Finished` result means the program has completed and contains a return value. A `Crashed` result means that the program crashed during its execution, and the end state is the system's state after the crash but before rebooting. Developers define the crash semantics of the system by defining execution rules that lead to a `Crashed` result.

2) Layers: Layers are ConFrm's model of full systems. They augment the cores with the ability to create programs, i.e., a sequence of core operations, and provide semantics to programs using the semantics of individual operations. In addition to the ability to define programs, layers contain semantics for the crash-reboot-recovery process. We will explain each of these parts in order.

Sequencing: Layers equip a core with `Bind` and `Return` operations. A `Bind` operation allows sequencing of operations. A `Return` operation passes values to the consequent operations. We call a sequence of operations *a program*.

The semantics of programs are derived from the semantics of the core. Since a program consists of a sequence of operations, layer semantics take a list of tokens, which we call an oracle, and consume exactly one token at each execution step.

On top of eliminating the clutter from an implementation, separating sequencing from core operations allows ConFrm to provide core-agnostic theorems and tactics.

Recovery semantics: Layers provide predefined recovery semantics to model repeated crash-reboot-recovery processes. To distinguish recovery semantics from the semantics of the execution of a single program, we will refer to recovery semantics as *executing-with-recovery* and denote it with $exec^r$ when the distinction is important. Figure 6 illustrates the provided recovery semantics.

Recovery semantics differs from layer semantics in three ways. Firstly, recovery semantics take two program arguments: a program to run and a recovery program to run in case of a crash.

`Refinement` $L_i\ L_a$:=
`compile` : $prog_a \rightarrow prog_i$
`refines` : $state_i \rightarrow state_a \rightarrow Prop$
`refines-reboot` : $state_i \rightarrow state_a \rightarrow Prop$
`oracle-refines` : $user \rightarrow state_i \rightarrow prog_a \rightarrow$
$(state_i \rightarrow state_i) \rightarrow oracle_i \rightarrow oracle_a \rightarrow Prop$

Fig. 7: Refinement definition.

Secondly, the effects of a reboot on a system may be nondeterministic. One example is an asynchronous disk. When a system crashes and reboots, the disk can be nondeterministically in one of the multiple possible states due to buffered and reordered writes. To capture and quantify this source of nondeterminism, we introduce *reboot functions*. A reboot function takes a state after a crash and returns the system's state after the reboot. Reboot functions serve as oracles for the reboot process. Similar to the tokens, different outcomes of a nondeterministic reboot are represented by different reboot functions. Although reboot functions and tokens are conceptually the same and can be combined, we decided to separate them in our implementation to simplify the refinement definitions and make incorporating the assumptions on the reboot process easier.

Thirdly, the execution semantics of a crash-reboot-recovery process capture multiple crash and recovery attempts with the `Exec-recovered` rule by providing an inductive rule that refers to the future recovery attempts after a crash. Since each execution requires an oracle and each crash requires a reboot function to determine the after-reboot state, recovery semantics take a list of oracles and a list of reboot functions. The provided semantics implicitly assume that recovery will eventually succeed.

B. Abstraction and Metatheory

Modularity is essential to manage the complexity of a system and necessary to build large-scale verified systems. ConFrm's support for modularity consists of support for abstraction and the metatheory to prove confidentiality through abstraction. We will first present the infrastructure for defining abstractions and then explain the metatheory. Throughout the section, we will use subscripts i and a to refer to implementation and abstraction, respectively.

1) Abstraction Structures: ConFrm uses refinements and simulations for defining abstractions. We will explain them in that order.

a) Refinements: ConFrm's primary mechanism for relating abstractions and implementations is refinement. ConFrm defines refinement in four parts, extending the standard refinement definition: a compile function that turns abstract programs into implementation programs, a state-refinement relation for the normal states, a state-refinement relation for after-reboot states, and an oracle-refinement relation that relates implementation oracles to abstraction oracles. Figure 7 displays the formal refinement definition.

$$
\begin{aligned}
&\texttt{Recovery-oracles-refine}\ (u, s_i, p_a, rec_a,\\
&\quad lrf_i, lo_i, lo_a) :=\\
&(\exists\ o_i\ o_a\ s_i'\ v.\\
&lo_i = [o_i] \wedge lo_a = [o_a] \wedge lrf_i = [\,] \wedge\\
&exec_i(o_i, u, s_i, \texttt{compile}(p_a), s_i', \texttt{Finished}\ v) \wedge\\
&\forall\ rf_i.\ \texttt{Oracle-refines}\ (u, s_i, p_a, rf_i, o_i, o_a))\\
&\vee\\
&(\exists\ o_i\ o_a\ rf_i\ lo_i'\ lo_a\ lrf_i'\ s_i'.\\
&lo_i = o_i :: lo_i' \wedge lo_a = o_a :: lo_a' \wedge\\
&lrf_i = rf_i :: lrf_i' \wedge\\
&exec_i(o_i, u, s_i, \texttt{compile}(p_a), s_i', \texttt{Crashed}) \wedge\\
&\texttt{Oracle-refines}(u, s_i, p_a, rf_i, o_i, o_a) \wedge\\
&\texttt{Recovery-oracles-refine}(u, rf_i(s_i'), rec_a,\\
&\quad rec_a, lrf_i', lo_i', lo_a'))
\end{aligned}
$$

Fig. 8: `Recovery-oracles-refine` definition.

$$
\begin{aligned}
&\texttt{Simulation}\ (u, p_a, rec_a, lrf_i, lrf_a,\\
&\quad ref_{begin}, ref_{end}) :=\\
&\forall\ lo_i\ s_i\ s_i'\ r\ s_a.\\
&ref_{begin}\ (s_i, s_a) \rightarrow\\
&exec_i^r(lo_i, lrf_i, u, s_i, \texttt{compile}(p_a),\\
&\quad \texttt{compile}(rec_a), s_i', r) \rightarrow\\
&\exists\ lo_a\ s_a'.\\
&\texttt{Recovery-oracles-refine}\ (u, s_i, p_a, rec_a,\\
&\quad lrf_i, lo_i, lo_a) \wedge\\
&exec_a^r\ (lo_a, lrf_a, u, s_a, p_a, rec_a, s_a', r) \wedge\\
&ref_{end}\ (s_i', s_a')
\end{aligned}
$$

Fig. 9: Simulation definition.

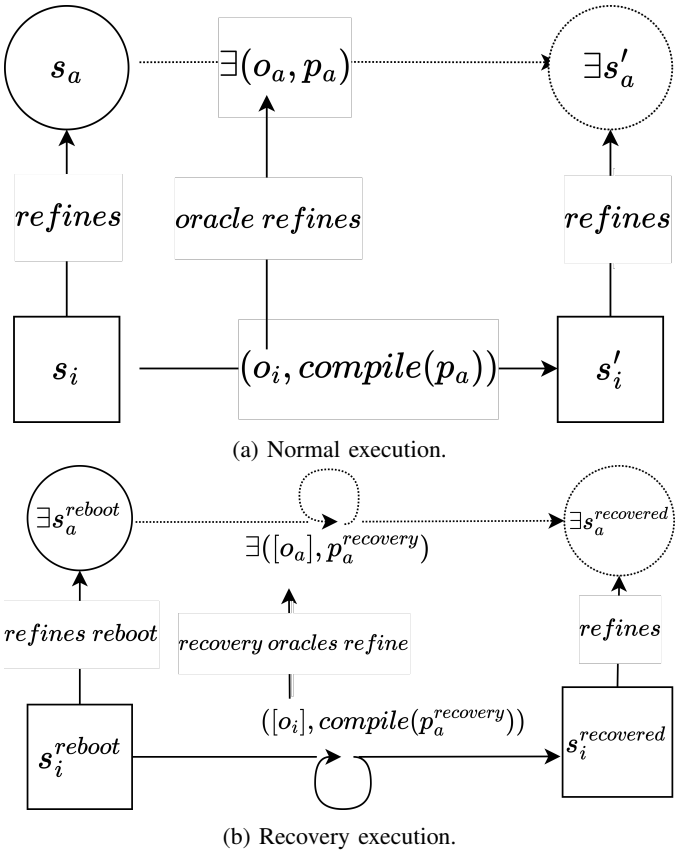
The refinement definition in the literature consists of `compile` and `refines` [8]. We added `refines-reboot` and `oracle-refines` relations to accommodate crash-reboot-recovery and oracles, respectively.

We separate `refines-reboot` from `refines` because, in general, a `refines` relation is too strong to hold for after-reboot states. For example, a file system may have an in-memory, write-through cache that contains exactly its log data to speed up disk reads. In this case, the `refines` relation, in addition to the facts about the disk, would include the fact that cache contents exactly match the log. This fact wouldn't be true after a reboot since memory contents will be arbitrary. However, we also need some information about after-reboot states to ensure recovery restores the original `refines` relation and `refines-reboot` contains such information.

`oracle-refines` relates the implementation oracles that lead to the same abstract representation with an abstraction oracle. Intuitively, an abstraction oracle concisely represents multiple nondeterministic implementation events that lead to the same abstract representation.

ConFrm adapts a provided `oracle-refines` relation to lists of oracles used in execution-with-recovery by deriving a `Recovery-oracles-refine` relation. `Recovery-oracles-refine` recursively relates oracles in the lists via `oracle-refines` through multiple iterations of the crash-reboot-recovery process. Figure 8 shows the definition of `Recovery-oracles-refine`.

b) Simulations: ConFrm uses simulation proofs to ensure abstractions capture implementation behavior correctly. We modified the standard simulation definition to accommodate our refinement definition. Our simulation definition is parameterized over implementation and abstraction layers and a refinement between them. We left those parameters implicit to improve readability. Figure 9 displays the formal definition.



(a) Normal execution.

(b) Recovery execution.

Fig. 10: Visualization of simulation for different cases.

$$
\begin{aligned}
&\texttt{RDNI-transfer} := \\
&\forall\ u\ p1_a\ p2_a\ rec_a\ lrf_i\ lrf_a\ eqv_a. \\
&\texttt{RDNI}\ (u, p1_a, p2_a, rec_a, eqv_a, lrf_a) \rightarrow \\
&\texttt{Simulation}\ (u, p1_a, rec_a, lrf_i, lrf_a, \\
&\quad \texttt{refines}, \texttt{refines}) \rightarrow \\
&\texttt{Simulation}\ (u, p2_a, rec_a, lrf_i, lrf_a, \\
&\quad \texttt{refines}, \texttt{refines}) \rightarrow \\
&\texttt{Oracles-refine-same}\ (u, p1_a, p2_a, rec_a, \\
&\quad lrf_i, eqv_a) \rightarrow \\
&\texttt{RDNI}\ (u, \texttt{compile}(p1_a), \texttt{compile}(p2_a), \\
&\ \texttt{compile}(rec_a), \texttt{Refines-eqv}(eqv_a), lrf_i)
\end{aligned}
$$

Fig. 11: RDNI-Transfer theorem.

The first change is that the modified simulation definition has two relations: one for the starting and one for the end states. We separate the relations for starting and end states to be able to reason about recovery, where the relations that hold at the beginning and the end are different.

The second change is that a simulation is defined over an entire execution-with-recovery. This allows the relation to be broken temporarily after a crash as long as the recovery process restores it. This change is necessary because crashes may leave the system in an implementation state that doesn't refine an abstract state but is also not visible to users. Since after-crash states are not visible to users until recovery is completed, they do not compromise confidentiality. Figure 10 displays the simulation definition in normal and recovery execution cases.

2) Metatheory: At the heart of ConFrm's metatheory lies the RDNI transfer theorem, which derives a compiled program's confidentiality from its abstraction. The theorem reveals sufficient conditions for preserving RDNI through refinement. The two conditions are that there should be a simulation between implementation and abstraction for the refinement relations and that oracle refinement is independent of confidential data. Figure 11 illustrates our theorem.

Simulation ensures that the abstraction captures all the behavior of the implementation. `Oracles-refine-same` captures the necessity that oracle refinement is independent of confidential data. Abstractions modeling some deterministic behaviors of an implementation as nondeterminism is a typical pattern. This property ensures the developer does not abstract the behavior that depends on confidential data as nondeterminism. The formal definition is displayed in Figure 12.

Finally, using the abstraction-equivalence relation, ConFrm generates an equivalence relation for implementation states. The new relation states that two implementation states are equivalent if they refine two equivalent abstract states. Below is the formal definition of the derivation, and Figure 13 shows its visualization. The user is not displayed in the visualization for clarity.

$$
\begin{aligned}
&\texttt{Oracles-refine-same}\ (u, s_i, p1_a, p2_a, \\
&\quad rec_a, lrf_i, eqv_a) := \\
&\forall\ lo_i\ lo1_a\ lo2_a\ s1_i\ s2_i. \\
&\texttt{Refines-eqv}(eqv_a, s1_i, s2_i) \rightarrow \\
&\texttt{Recovery-oracles-refine}(u, s1_i, p1_a, \\
&\quad rec_a, lrf_i, lo_i, lo1_a) \rightarrow \\
&\texttt{Recovery-oracles-refine}(u, s2_i, p2_a, \\
&\quad rec_a, lrf_i, lo_i, lo2_a) \rightarrow \\
&lo1_a = lo2_a
\end{aligned}
$$

Fig. 12: `Oracles-refine-same` definition.

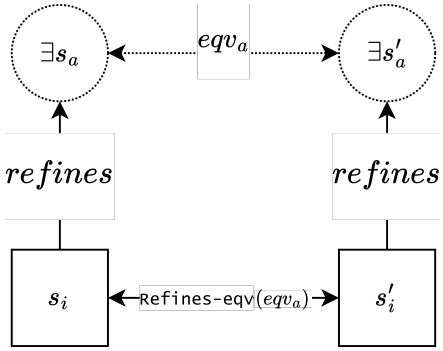


Fig. 13: Visualization of equivalence derivation.

$$
\begin{aligned}
&\texttt{Refines-eqv}(eqv_a, u, s_i, s'_i) := \\
&\quad \exists\ s_a\ s'_a. \\
&\qquad \texttt{refines}\ (s_i, s_a) \wedge \\
&\qquad \texttt{refines}\ (s'_i, s'_a) \wedge \\
&\qquad eqv_a(u, s_a, s'_a)
\end{aligned}
$$

3) Proper Initialization: ConFrm provides a proper-initialization definition to ensure the system is initialized correctly. The proper-initialization relation states that a successful initialization should put the system into an initial state that refines an abstract state, regardless of the starting implementation state. The formal definition is illustrated below.

$$
\begin{aligned}
&\texttt{Proper-initialization}\ (p_{init}) := \\
&\quad \forall\ u\ o\ s\ s_{init}\ r. \\
&\qquad exec_i\ (o, u, s, \texttt{compile}(p_{init}), \\
&\qquad\quad s_{init}, \texttt{Finished}\ r) \rightarrow \\
&\qquad \exists\ s_a.\ \texttt{refines}(s_{init},\ s_a)
\end{aligned}
$$

In this section, we covered the formalization of system components and support for modularity. Next, we will present ConFs, our confidential file system implemented and proved confidential in ConFrm.

V. ConFs File System

ConFs is the first crash-safe file system with a termination-insensitive RDNI specification and machine-checkable proofs. The first half of the section will explain its design and im-

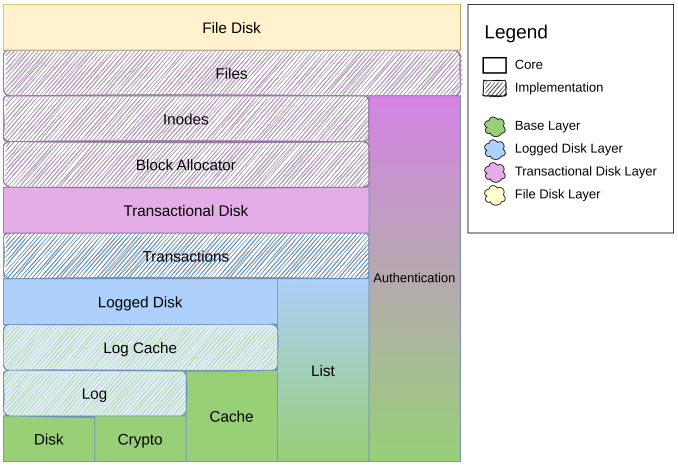


Fig. 14: Structure of ConFs.

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

Fig. 15: File-system API.

plementation. The second half will explain its confidentiality specifications.

A. Design

ConFs consists of three components: a checksum-based write-ahead log with a log cache, a transaction system, and file-system structures like block allocators and inodes.

The entire design of ConFs can be seen in Figure 14. 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., the transactional disk abstracts the functions of transactions into operations).

Now, we will explain the essential parts of our design.

a) Base Layer: We model disk, cache, and in-memory data structures for transactions and cryptographic operations in the base layer. It provides all the basic operations that can be used in the system, which each file-system operation is compiled into. The appendix shows the list of operations in the base layer.

b) Log and Log Cache: ConFs contain a checksum-based, encrypted write-ahead log. Encryption of the log is necessary for a checksum-based log to avoid leaking the contents of previous transactions. An example of a possible leakage and how encryption solves it can be found in the Appendix.

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.

c) Logged-Disk Layer: We abstract the log-cache API to a new core called the logged-disk core. The logged-disk core provides two improvements over directly using the implementation: it simplifies the disk model and the operational semantics. The logged-disk layer simplifies the disk model by hiding the existence of the log, the cache, and previously written values in the base-layer disk. Logged-disk's crash semantics are also significantly simpler because of the crash safety the log provides.

d) 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 only to 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 relevant to the challenge we are trying to address. Figure 15 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 disk blocks because byte granularity adds extra complexity without presenting any confidentiality challenges. Similarly, we chose to use inode numbers as file handles to focus on confidentiality without the complexity of managing a directory structure. File-system structures contain a notion of ownership, discretionary access control, and a nondeterministic functional specification for `create`. We also implemented a `change_owner` call that transfers the ownership of a file to another user to demonstrate that ConFrm and RDNI can handle systems with dynamic data ownership. The existence of the `change_owner` call poses challenges for providing a uniform confidentiality specification for arbitrary sequences of system calls. In later sections, we will explain how we accommodated the `change_owner` specification.

e) 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. This layer presents a simple model, a map from inode numbers to files. The simplification provides an intuitive model for how a file system is perceived and also simplifies the confidentiality specifications.

B. Specifying Security

Security of ConFs is defined as an RDNI specification for each compiled file-disk operation. The heart of these specifications is the equivalence relation between two states. For our definition of confidentiality, we treated 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., the size of a directory shows the number of files in it, the

amount of free space, and the 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.

The equivalence relation for ConFs captures the idea that *two states are equivalent for a user if they have the same structure and the data owned by that user is identical in both states*. Following is the formalization of the idea.

$$
\begin{aligned}
&\texttt{same-for-user-except}\ (exclude, u, s_1, s_2) := \\
&\quad (\forall\ inum.\ s_1(inum) = None \leftrightarrow \\
&\qquad s_2(inum) = None) \\
&\quad \wedge \\
&\quad (\forall\ inum\ file_1\ file_2. \\
&\qquad s_1(inum) = file_1 \rightarrow \\
&\qquad s_2(inum) = file_2 \rightarrow \\
&\qquad file_1.owner = file_2.owner \wedge \\
&\qquad len(file_1.blocks) = len(file_2.blocks) \wedge \\
&\qquad (exclude \neq inum \rightarrow \\
&\qquad\quad file_1.owner = u \rightarrow file_1 = file_2))
\end{aligned}
$$

The relation formalizes the following three properties: the same inode numbers are in use, files with the same inode number have the same owner and length, and if those files belong to the specified user, their contents are the same. By requiring files to be identical only for the specified user, our relation captures the intuition of differing in confidential data belonging to other users, whose connection to confidentiality is explained in section I.

To make the relation usable in the `change_owner` specification, it takes an optional inode number of the file whose owner is being changed. This is because if the new owner is the user whose states are equivalent, then the resulting states would be equivalent only if the files whose owner is being changed are identical. However, the files that belong to the old owner in the starting states are not guaranteed to be identical since the 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 the user's other files stay identical.

Since the equivalence relation used in `change_owner`'s confidentiality specification excludes the file 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 modified file is covered by `change_owner`'s functional correctness, which states that the changed file's contents stay unchanged after the operation. However, this prevents us from providing a succinct specification for arbitrary sequences of system calls since such a specification needs to account for each instance of `change_owner` appearing in the sequence. This requirement makes an equivalence that will provide such uniformity dependent on the sequence, which is counter-intuitive. The existence of a succinct and uniform specification for storage systems with dynamic ownership left as future work.

$$
\begin{aligned}
&\texttt{Write-RDNI} := \\
&\forall\ n\ u\ inum\ adr\ blk_1\ blk_2. \\
&\texttt{TI-RDNI}(u, \\
&\quad \texttt{Write}(inum, adr, blk_1), \\
&\quad \texttt{Write}(inum, adr, blk_2), \\
&\quad \texttt{Recover}, \\
&\quad \texttt{same-for-user-except}(None), \\
&\quad repeat(identity, n))
\end{aligned}
$$

Fig. 16: Specification for `Write` operation

Figure 16 shows the confidentiality specification for the `Write` operation. Two cases where input blocks are the same and different correspond to the confidentiality of stored and input data. The first case guarantees that the confidential data on the disk doesn't affect the behavior, and the second case guarantees the same for the input data.

1) Transactional Disk Layer Security: The derived equivalence 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 are abstracted away. In our case, the main reason for requiring additional conditions is that oracles in ConFs implicitly dictate the number of execution steps and the types of operations a program takes. The semantics of a program require the consumption of exactly one token per operation executed. This requirement implies that two programs with the same oracle must follow the same execution path. An example can be found in the Appendix.

2) Logged Disk and Base Layer Security: The logged disk and base layers require extra conditions regarding the log structure and the transaction list. 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 an 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 aligns with the intuition behind our file-disk-equivalence relation.

C. Proving Security

As explained in subsection IV-B, proving confidentiality of our implementations in the base layer requires three groups of theorems for each file-disk operation: an RDNI confidentiality specification proof, a simulation proof between operations and their implementations, and an oracle-refinement-independence-from-confidential-data proof. The vertically composable nature of each type allowed us to prove the properties for each layer separately by deriving

intermediate RDNI specifications instead of one monolithic proof from the file-disk layer to the base layer.

a) 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 the file disk is crash-safe, these proofs are straightforward and follow the same pattern.

b) Simulation proofs: We split simulation proofs into two parts to keep proofs shorter and more manageable: the existence of a refined abstract oracle given an implementation execution, and the 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. The biggest challenge regarding simulations is establishing `oracle-refines` relations. Finding the correct relations required multiple iterations and corresponding changes in definitions and proof scripts.

c) Oracle-independence proofs: Oracle-independence proofs were the hardest due to their being two-execution proofs. We split oracle-independence theorems into two smaller theorems: two programs follow the same execution path from related states with the same oracle, and 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 log-write operation that overwrites some data with itself, making the operation effectively a 'noop.' The possibility of such a noop write operation makes it impossible to determine whether a write succeeds after a crash by 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 is a noop, which can refine two different oracles.

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

However, such precise, low-level reasoning was tedious and required significant proof effort to finalize—the discovery of a proof strategy that requires less effort is left as future work.

d) Termination sensitivity: One concession we had to make was using the termination-insensitive RDNI as our final confidentiality specification for ConFs due to time constraints. We discovered that termination sensitivity is orthogonal to the other properties and needs a 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.

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

Fig. 17: Lines of code required to implement ConFrm and apply it to build ConFs.

D. Extraction and Trusted Computing Base

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

E. Evaluation

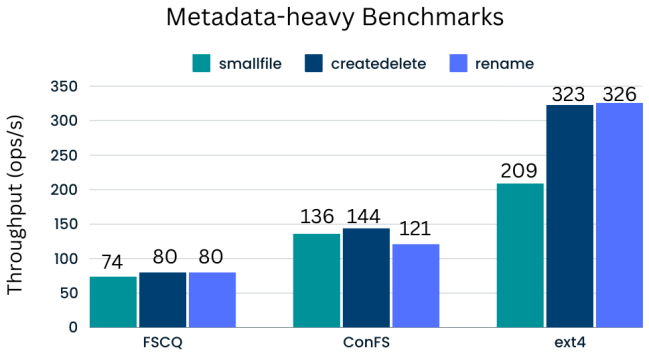
Since ConFrm and ConFs were built from scratch, we used lines of code as the effort estimate. We broke down the numbers to show how much effort went into each component. Figure 17 displays the results. According to the data, functional correctness has 5.6x and confidentiality has 9.2x proof overhead per line of implementation.

Performance evaluation: We used five benchmarks to measure the performance of ConFs compared to existing file systems: FSCQ [4] and ext4. We used FSCQ as a representative of verified file systems because it is similar to ConFs, and ext4 as a representative of widely used file systems.

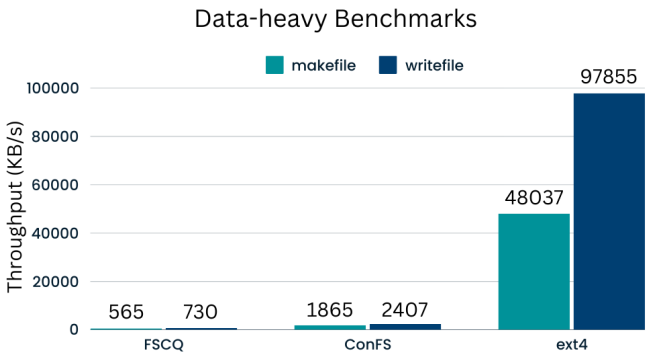
Experimental Setup: We extracted the Coq implementation to Haskell to test our file systems and connected them to the FUSE [11] library to provide the POSIX API. We wrote an interpreter function in Haskell to implement the operations in the disk and memory model. Since ConFs doesn't have directories, we implemented a simple directory structure in Haskell. Directory blocks are written directly to the disk instead of the log. The downside 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 consist of `makefile`, which creates a file and writes 1MB data, and `writefile`, which overwrites the file with 1MB data.

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



(a) Metadata-heavy benchmarks



(b) Data-heavy benchmarks

Fig. 18: Performance comparison benchmarks

We tested ext4 in the checksum-logging configuration since it is similar to ConFs' design.

All tests are run on a machine with a 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.

Results: Figure 18 show that ConFs perform better than FSCQ and worse than ext4 in all benchmarks. In metadata-heavy benchmarks, ConFs performed 1.5x to 1.8x of FSCQ and 0.37x to 0.47x of ext4. In data-heavy benchmarks, ConFs performed 3.3x of FSCQ and 0.02x to 0.04x of ext4.

ConFs's performance speed-up over FSCQ can be attributed to using axiomatic definitions for inner data structures 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 nontrivial extracted types. FSCQ 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 contains many optimizations that were not implemented in our research prototype. Therefore, it is natural that ext4 outperforms ConFs.

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

Fig. 19: Example of a secure program if the generated bit is uniformly random.

VI. Limitations and Future Work

This section will discuss various limitations associated with our specifications and implementations and will provide some future directions to pursue.

A. Limitations

1) Specification Limitations: There are three important limitations regarding the RDNI approach. First, RDNI prohibits some implementations that are secure for a specific distribution but may be insecure for others. Figure 19 shows a simple implementation that is secure if and only if the generated bit is uniformly random but doesn't satisfy RDNI. An important group of excluded systems is the ones whose security relies on pseudorandom generators. Such systems' security derives from the fact that the number generated is indistinguishable from a truly random number. We expect storage systems that use per-user or per-file keys, such as APFS, EFS, and ext4, to encounter verification challenges with our specifications. Even though some of these challenges can be circumvented, as we demonstrated in ConFs, we acknowledge that such solutions may not be suitable for other systems due to the underlying assumptions. We are unaware of any existing storage system implementation that our specifications have explicitly ruled out. However, this is due to the infeasibility of examining their implementations in-depth, which is required to identify such cases.

Second, our theorem requires 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 ours is not a fitting model for that system.

Third, our model does not address side-channel security like timing or power. This concession was made to keep the scope manageable. However, we acknowledge the importance of side channels in system security.

2) Implementation Limitations:

ConFrm Limitations: ConFrm has two important limitations. The first one is that it doesn't contain formalization of probabilities and doesn't implement our probability-preservation theorem. Therefore, ConFrm does not support reasoning about a formalized stochastic system.

The second limitation is that ConFrm 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.

ConFs Limitations: The first limitation is due to our design choices when implementing ConFs. ConFs cores instantiate ConFrm with a mixed embedding where disk and memory operations are deeply embedded, and the rest is shallowly embedded [6]. Because of this embedding, ConFrm can only apply nondeterminism oracles to memory operations. This is not a limitation of ConFrm and can be overcome using a fully deep embedded language for the implementations.

Secondly, due to time constraints, ConFs uses termination-insensitive RDNI as its confidentiality specification. Therefore, our probability-preservation theorem doesn't hold for ConFs. However, ConFrm supports termination-sensitive and termination-insensitive RDNI and provides theorems to derive one from the other.

B. Future Work

Our work can be extended in both theoretical and applied fronts. On the theoretical side, we believe that our approach can be extended to the stochastic systems where there is an assumed prior by systematic oraclization of the stochastic system. One promising direction to achieve this is by "duplicating" execution paths using oracles proportional to their probabilities, such that each path has an equal probability of being taken. An appropriate version of `Matching-execs` would be sufficient to establish return-value equality in such a system.

Similarly, given the flexibility of oraclization and the matching-executions definition, we believe integrating side-channel security into our definitions is possible by designing oracles to enforce the desired properties. For example, if oracles encode the runtime of each execution step, then properties about timing can be proved through the oracles. Accommodating both systems with priors and side-channel security is very important for implementing various systems with comprehensive security guarantees.

On the implementation side, ConFrm can be enhanced by formalizing our probability-preservation theorem. We do not see any major hardships in adding this support to the framework using a well-established probability library such as Polaris [31]. On top of enabling the formal reasoning about stochastic systems, such integration would allow future results to be incorporated into the framework and increase its usability in diverse settings.

Another interesting direction is the generalization of our framework and results to other systems that contain nondeterminism. Some immediate targets are concurrent and distributed systems due to the nondeterminism in execution order and network reordering.

VII. Related Work

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

a) Confidentiality properties: There is a significant body of work formalizing noninterference properties [12], [15], [21], [22], [26]–[28]. ConFrm's definitions build upon this existing work. One of particular interest is Oheimb's *noninfluence* [34]. Noninfluence is introduced as a comprehensive specification to ensure the confidentiality of a system that processes and stores confidential data. Noninfluence achieves this goal via two separate properties, one for the stored data and one for the newly introduced data: *nonleakage* and *noninterference*, respectively.

RDNI differs from its predecessors in how it treats nondeterminism in its formalism. RDNI takes a more fine-grained approach in relating nondeterministic executions by requiring a strong coupling between executions for each nondeterministic execution branch.

b) Resolution of nondeterminism: One similar technique to our oracle approach is called resolution of nondeterminism for Markov Decision Procedures (MDPs) and probabilistic labeled transition systems [1], [3], [13], [25], [29], [33]. Resolution of nondeterminism reduces a nondeterministic and probabilistic system into a fully probabilistic one via a scheduler. A scheduler is a structure that picks a nondeterministic choice whenever such a choice is necessary. The choice can be made deterministically or probabilistically with a predetermined distribution. Our nondeterminism oracles can be thought of as schedulers that reduce a system to a deterministic one. Our main difference is that we do not assume a prior distribution for our oracles.

c) Machine-checked security in systems: Several prior projects have proven security (and specifically confidentiality) properties about their system implementations: seL4 [18], [22], CertiKOS [7], Ironclad [14], and DiskSec [15]. 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 secrets (e.g., a private key), formulated as noninterference. Also using noninterference, Komodo [9] 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 the properties of 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, abstraction layers, and probabilistic confidentiality 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 definition of confidentiality. Even though

the definition can be customized by defining different state-equivalence relations, it may not express an arbitrary confidentiality specification.

d) 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 [23], [24] or the UrFlow analysis [5], have been developed to reason about information-flow properties of application code. UrFlow is specialized to database-backed web applications and uses querying language to define policies. Jif's analyzer would be hard to use for reasoning about dynamic data structures inside a file system (such as a write-ahead log or a buffer cache) containing data from different users.

Dynamic tools, such as Jeeves and Jacqueline [35], [36] and Resin [37], deal with dynamic data structures but require sophisticated and expensive runtime enforcement mechanisms. ConFrm avoids the overhead of runtime enforcement and an additional trusted runtime checker.

SeLoc [10] uses double weakest preconditions to prove noninterference for fine-grained concurrent programs. It is built on top of Iris [16], a separation logic-based framework that proves the correctness of fine-grained concurrent programs. It provides developers with a confidentiality-ensuring type system and a wide array of tools. However, employing SeLoc requires Iris, which adds a substantial entry barrier. Conversely, ConFrm is standalone and lightweight but does not offer the full array of tools SeLoc offers.

e) Sequential composability and confidentiality-preserving refinements: Since it is known that traditional noninterference is not preserved in simulation-based refinements [20], a body of work tries to identify the conditions that make refinements noninterference-preserving.

Sun et al. [30] propose 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. [2] formulate 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.

RDNI is also sequentially compositional but differs from existing work in two points. First, RDNI provides probabilistic guarantees for nondeterministic executions, while prior work only provides possibilistic guarantees. Second, RDNI supports reasoning about crashes and recovery of the system.

VIII. Conclusion

This paper investigates challenges related to probabilistic leakages surrounding the confidentiality of nondeterministic storage systems with rich sharing semantics. It lays the groundwork for probabilistically confidential crash-safe storage systems with machine-checkable proofs. It introduces a possibilistic confidentiality specification, relatively deterministic noninfluence, that implies probabilistic confidentiality. It also provides a formally verified framework, ConFrm, and a confidential file system built using the framework with machine-checkable proofs, ConFs.

Our confidentiality specification supports specifying a subset of the data stored in the system as confidential and allows obtaining probabilistic guarantees without modeling probabilities. It is also preserved under simulations, which allows them to be used with a wide variety of abstractions, which is not the case with traditional definitions. The specification is supported by techniques that can be used in different contexts.

ConFs is the first file system with termination-insensitive RDNI specifications and machine-checked confidentiality proofs. ConFs demonstrates that the confidentiality of storage systems that safely manipulate confidential data can be proven even with crashes and nondeterminism.

References

[1] Erika Ábrahám, Ezio Bartocci, Borzoo Bonakdarpour, and Oyendrila Dobe. Probabilistic hyperproperties with nondeterminism. *ArXiv*, abs/2005.06115, 2020.

[2] 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.

[3] Marco Bernardo. Coherent resolutions of nondeterminism. In *European Performance Engineering Workshop*, 2019.

[4] 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.

[5] 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.

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

[7] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-end verification of information-flow security for C and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 648–664, New York, NY, USA, 2016. Association for Computing Machinery.

[8] John Derrick and Eerke Boiten. *Labeled Transition Systems and Their Refinement*, pages 3–26. Springer International Publishing, Cham, 2018.

[9] 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.

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

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

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

[13] Hans A. Hansson. Time and probability in formal design of distributed systems. In *DoCS*, 1994.

[14] 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.

[15] Atalay Ileri, Tej Chajed, Adam Chlipala, Frans Kaashoek, and Nickolai Zeldovich. Proving confidentiality in a file system using DiskSec. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 323–338, Carlsbad, CA, October 2018. USENIX Association.

[16] 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.

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

[18] 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.

[19] Linux Kernel Developers. Ext4 filesystem, 2017. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

[20] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 79–93, 1994.

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

[22] 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.

[23] 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.

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

[25] Anna Philippou, Insup Lee, and Oleg Sokolsky. Weak bisimulation for probabilistic systems. In *International Conference on Concurrency Theory*, 2000.

[26] 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.

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

[28] 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.

[29] Roberto Segala and Nancy A. Lynch. Probabilistic simulations for probabilistic processes. In *Nordic Journal of Computing*, 1994.

[30] 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.

[31] Joseph Tassarotti and Robert Harper. A separation logic for concurrent randomized programs. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[32] Stephen C. Tweedie. Journaling the Linux ext2fs filesystem. In *Proceedings of the 4th Annual LinuxExpo*, Durham, NC, May 1998.

[33] Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite state programs. *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 327–338, 1985.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

APPENDIX

A. Proof of Theorem III.1

Before proving our theorem, we will prove a helper lemma where the existence of a compatible invariant distribution implies a symmetry in `Matching-execs`.

Lemma A.1. *Let S be an oraclized system.*

$$\begin{aligned}&\forall\ u\ p_1\ p_2\ eqv.\\ &(\exists\ X,\\ &compatible(S, X)\ \wedge\\ &invariant(X, u, p_1, p_2, eqv)) \rightarrow\\ &\texttt{Matching-execs}(u, p_1, p_2, eqv) \rightarrow\\ &\texttt{Matching-execs}(u, p_2, p_1, eqv)\end{aligned}$$

Proof. Pick an arbitrary oracle o, states s_1, s_2 and s_2', and a result r_2 such that $exec(o, u, s_2, p_2, s_2', r_2)$ and $eqv(u, s_2, s_1)$ holds. Using these two facts, we will show that $\exists\ s_1'.\ exec(o, u, s_1, p_1, s_1', r_2)$.

By compatibility of X, we know that $exec(o, u, s_2, p_2, s_2', r)$ implies $X_{(u,s_2,p_2)}(o) > 0$. By invariance of X and symmetry of eqv, $X_{(u,s_2,p_2)}(o) > 0$ implies $X_{(u,s_1,p_1)}(o) > 0$. By compatibility, $X_{(u,s_1,p_1)}(o) > 0$ implies that there exists s_1' and r_1 such that $exec(o, u, s_1, p_1, s_1', r_1)$. By `Matching-execs`, we know that there exists s_2'' such that $exec(o, u, s_2, p_2, s_2'', r_1)$. By relative determinism, we know that $r_1 = r_2$. Therefore, $\exists\ s_1'.\ exec(o, u, s_1, p_1, s_1', r_2)$ holds. □

Proof of Theorem III.1

Proof. $(\rightarrow)$: Pick an arbitrary result r. By definition, we know that $X^{\mathbf{R}}_{(u,s_1,p_1)}(r)$ is equal to

$$\sum_{o \in \{o'\ \mid\ \exists\ s_1',\ exec(o', u, s_1, p_1, s_1', r)\}} X_{(u,s_1,p_1)}(o)$$

.

By `Matching-execs` and Lemma A.1, we can conclude that set

$$\{o'\ \mid\ \exists\ s',\ exec(o', u, s_1, p_1, s', r)\}$$

is equal to the set

$$\{o'\ \mid\ \exists\ s',\ exec(o', u, s_2, p_2, s', r)\}$$

.

Since X is invariant for p_1 and p_2 under eqv, we know that $X_{(u,s_1,p_1)}(o)$ is equal to $X_{(u,s_2,p_2)}(o)$ for each oracle o. If we rewrite these two equalities in our definition we obtain

$$\sum_{o \in \{o'\ \mid\ \exists\ s',\ exec(o', u, s_2, p_2, s', r)\}} X_{(u,s_2,p_2)}(o)$$

which is equal to $X^{\mathbf{R}}_{(u,s_2,p_2)}(r)$.

$(\leftarrow)$: We will show this direction by proving its contrapositive. Assume that there exists a compatible invariant distribution X, two states s_1 and s_2 such that $eqv(u, s_1, s_2)$ and a result r such that $X^{\mathbf{R}}_{(u,s_1,p_1)}(r) \neq X^{\mathbf{R}}_{(u,s_2,p_2)}(r)$. Since X is compatible and invariant, this is only possible if, w.l.o.g. due to Lemma A.1, there exists an oracle o and state s_1' such that $exec(o, u, s_1, p_1, s_1', r)$ holds but $exec(o, u, s_2, p_2, s_2', r)$ does not hold for any s_2'. Therefore `Matching-execs`(u, p_1, p_2, eqv) does not hold for $exec(o, u, s_1, p_1, s_1', r)$. □

B. Example Core

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

Fig. 20: An example core for an in-memory 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 successful execution and `Crash` for crashing on that operation. Execution semantics and the proof of determinism are omitted for brevity.

C. Necessity of Extra Conditions

The 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 the latest value for an address when there is an active transaction in the system. Proving noninterference of this function requires showing that there exist two executions from related states with the same oracle. Since having the same oracle dictates that programs follow the same execution paths, two related states must contain the same addresses in their transactions, although corresponding data could differ.

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, the equivalence relation for implementation needs to be supplemented to make it finer-grained.

This particular example and some other more complicated variants are also present in log functions.

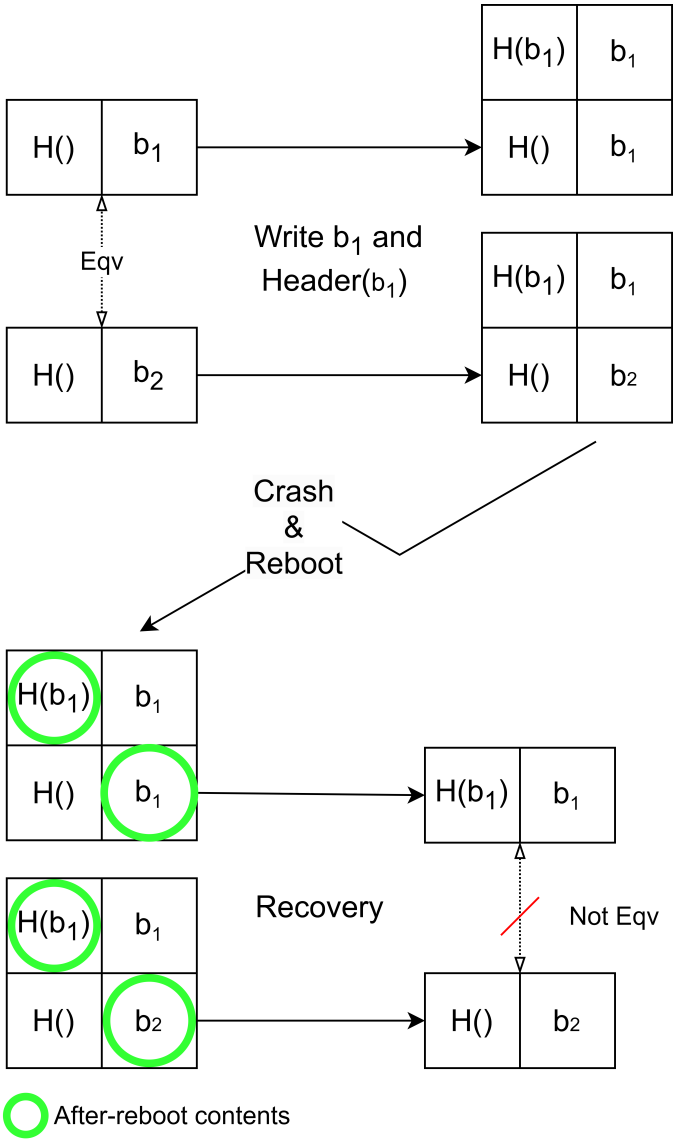


Fig. 22: A sequence of events that leads to confidential data leakage.

D. Our encryption model

```
encrypt: key -> block -> block.
decrypt: key -> block -> block.

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

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

Fig. 21: Encryption model.

E. Example Leakage from an Unencrypted Log

In the example depicted in Figure 22, 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 persists 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 system's state after recovery by looking if his write is on the disk.

We use encryption to fix the above problem. Encryption protects 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 because it is extremely unlikely to produce the same ciphertext from two encrypted blocks with two different random keys. Case 2 is ensured by using 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 23 shows how encryption fixes the problem.

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 differs from the hashes of leftover blocks, there is no after-reboot state where one transaction is kept, but the other is rolled back. Therefore, all possible after-recovery states are equivalent.

F. ConFs Base Layer Operations

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	

Fig. 24: Operations in the base layer.

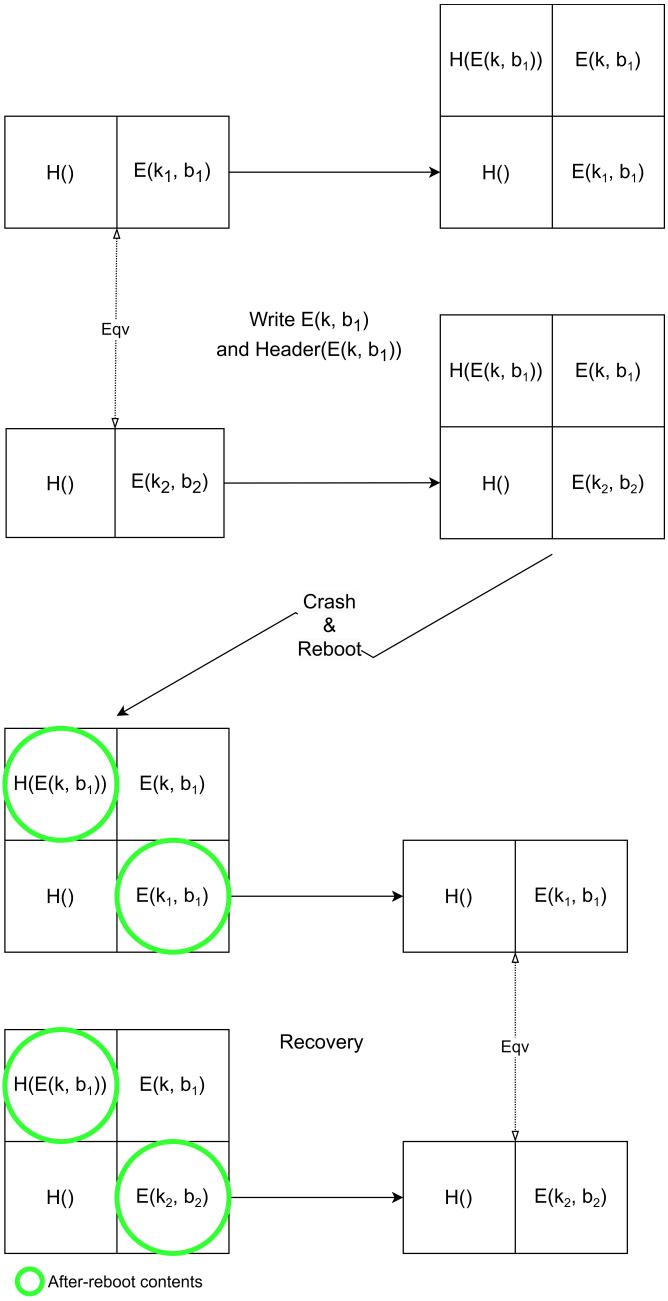


Fig. 23: Encryption fixes the leakage.