Noninterference specifications for secure systems

Luke Nelson, James Bornholt*, Arvind Krishnamurthy, Emina Torlak, Xi Wang
*University of Washington and *University of Texas at Austin*

Abstract

This paper presents an analysis of noninterference specifications used in a range of formally verified systems. The main findings are that these systems use distinct specifications and that they often employ small variations, both complicating their security implications. We categorize these specifications and discuss their trade-offs for reasoning about information flows in systems.

1 Introduction

Noninterference is a family of security properties that constrain information flows in a system [16]. It was originally proposed by Goguen and Meseguer [10]; many extensions [8, 37] have since been introduced. The basic idea is to ensure that the execution of users with low security levels is independent of the execution of users with high security levels; this restriction prevents, for example, an adversary from inferring secrets from legitimate users or influencing their decisions.

Noninterference provides a general approach for preventing subtle bugs that can violate information-flow policies. It is used to formally specify and verify confidentiality and integrity properties of a wide variety of systems, from OS kernels and storage systems to applications. This paper examines noninterference specifications used in such verified systems (Figure 1). We focus on system *implementations* that satisfy noninterference. Readers may refer to studies by van der Meyden and Zhang [37] and by Sabelfeld and Myers [29] for noninterference in abstract models and programming languages, respectively.

A verified system is guaranteed, through formal proof, to adhere to its noninterference specification. The proof that the implementation satisfies the specification is mechanically checked by a *theorem prover* (such as Isabelle/HOL [24], Coq [34], Dafny [17], or Z3 [6]). The implementation may be written in C and assembly, or *extracted* from a higher-level language that is designed for verification. The verification process rules out flaws in the design and implementation that would cause the system to violate the specification. Understanding the guarantees these systems provide thus boils down to understanding their specifications.

This paper makes two main contributions. First, we observe that verified systems often introduce their own notion of noninterference, complicating understanding of their guarantees [23: §6.5]. We identify *four distinct categories* of specifications under the umbrella name of "noninterference." These specifications differ in their origins, verification conditions, and the types of bugs they prevent. We hope our catalog can serve as a common vocabulary for distilling and contrasting noninterference specifications.

Second, using this catalog, we conduct a systematic analysis of specifications of the systems listed in Figure 1. For instance, our analysis shows that the specifications of the Komodo monitor [9] and the SFSCQ file system [14] deviate from the original `noninterference` definition (rendered in monospace in this paper to avoid ambiguity) and are closer to a variant called `nonleakage` used in the seL4 kernel [20].

We also find that verified systems often employ small variations of noninterference, creating more flavors of specifications. We hope that this paper helps researchers and practitioners determine which specifications are suitable for their systems, and motivates more research on verification practice using noninterference.

2 Motivation

This section illustrates common types of security policies that can be expressed using noninterference. As a representative example, consider a system consisting of a set of mutually distrustful processes. We will show how noninterference restricts the ways in which information can flow in the system. All the systems analyzed in this paper are single-threaded. Extending noninterference to verify concurrent systems is challenging [21, 32, 35], and it is a promising direction for future work.

Strict isolation. A strict form of isolation simply forbids information flows across security domains. For example, the system may statically partition resources (e.g., memory) among processes and disallow any inter-process communication. The system behaves as if each process were running on a physically isolated machine.

Using noninterference, one may specify that the execution of a process is independent of that of any other process. This specification precludes a process from accidentally leaking the content of its memory to another process. It also rules out, for instance, a system call using a shared memory allocator among processes; such a

	theorem prover	implementation	extraction	specification category	intransitive $\rightsquigarrow$	state-dependent `dom`	merged `observe`
Block Access Controller [11]	Isabelle/HOL	Isabelle/HOL	C	`noninterference`	○	○	–
PROSPER kernel [5, 30]	Isabelle/HOL	assembly	–	`noninterference`	○	○	–
Reflex browser kernel [25, 33]	Coq	Coq	OCaml	`noninterference`	○	○	–
NiStar kernel [31]	Z3 & Coq	C & assembly	–	`noninterference`	●	●	–
seL4 kernel [15, 20]	Isabelle/HOL	C & assembly	–	`nonleakage`	●	●	●
Ironclad Apps [13]	Dafny & SymDiff	Dafny	assembly	`nonleakage`	○	○	○
CertiKOS kernel [3, 4]	Coq	C & assembly	–	OC+SC	○	○	○
SFSCQ file system [14]	Coq	Coq	Haskell	OC+SC	○	○	○
Komodo monitor [9]	Dafny	Vale	assembly	OC+SC	○	○	●
CertiKOS^s kernel [23]	Z3	C & assembly	–	OC+WSC+LR	●	●	●
Komodo^s monitor [23]	Z3	C & assembly	–	OC+WSC+LR	●	●	●

Figure 1: Characterization of systems surveyed and their specifications: ● provides property; ○ does not provide property; – not applicable.

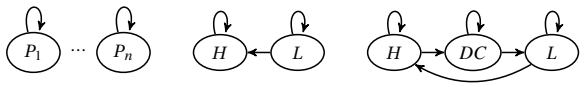
system call would enable a process to infer the memory usage of other processes [16], violating the specification.

High-low policy. Strict isolation can be too strong because it prohibits sharing. To enable sharing in specific ways, one may introduce a hierarchy among security domains [7]. For example, the system may consider some processes *high* as they are owned by a root user, and some *low* as they are owned by ordinary users. The system permits high processes to receive data from low processes, but forbids flows in the opposite direction. Using noninterference, one can specify that the execution of a low process is independent from that of any high process, which precludes high processes leaking information to low ones.

Downgrading. Many practical systems require information flows in both directions: not only from low to high, but also from high to low in a *controlled* manner. For example, a `sudo` process, which is considered high and has access to user passwords, needs to disclose to a low process the result of running the command. Another example is that a root user may invoke `chown` to change the ownership of a file in order to share it with an ordinary user.

This type of controlled declassification is an example of *downgrading* [18]. Using noninterference, one can specify that the execution of a low process is independent of that of any high process except through designated downgrading operations (e.g., declassifiers).

Fine-grained sharing. A system may wish to impose fine-grained policies. For example, the system may forbid processes from directly accessing the content of each other's memory, but consider associated metadata (e.g., how much memory each process consumes) public. Another example is SGX-like enclave systems [2], where an untrusted OS performs memory management for enclaves but cannot access their contents. For such systems, one can use noninterference to specify that the execution of a process is independent of a specific *subset* of other processes' state such as memory contents, while still allowing for influence by their metadata.



(a) strict isolation (b) high-low (c) assured pipeline

Figure 2: Examples of policies. Nodes and arrows denote domains and permitted flows, respectively.

3 Specifications of noninterference

This section describes common approaches to specifying noninterference. We first review two *trace-based* specifications, `noninterference` and `nonleakage`, followed by *unwinding* specifications that examine individual actions, and a summary of variations of these specifications.

Systems. We model a system as a state machine, which transitions from state to state in response to an action (e.g., a system call or external input). Formally, a system is represented as a tuple $\langle A, S, s_0, \texttt{step}\rangle$, consisting of a set of actions A, a set of states S, an initial state s_0, and a function $\texttt{step} : S \times A \to S$ that maps a state and an action to the next state. A sequence of actions is called a *trace*. To represent the state produced by executing the trace tr starting from the state s, we inductively define $\texttt{run} : S \times A^* \to S$ by $\texttt{run}(s, \epsilon) := s$ and $\texttt{run}(s, a \circ tr) := \texttt{run}(\texttt{step}(s, a), tr)$, where ϵ denotes the empty trace and $a \circ tr$ denotes the concatenation of the action a and trace tr. For simplicity, we assume a deterministic system; readers may refer to Eggert [8: §7.2] for extensions to nondeterministic systems.

Policies. A noninterference specification defines allowed information flows using a *policy*. To reason about information flows, we assume a set of *domains* D and a function $\texttt{dom} : A \to D$ that specifies the domain of each action. A domain is an abstract notion representing the authority of an action (e.g., a calling user). An information-flow *policy* $\rightsquigarrow \subseteq D \times D$ describes how information is permitted

to flow among these domains; $(u, v) \in \rightsquigarrow$ and $(u, v) \notin \rightsquigarrow$ are denoted by $u \rightsquigarrow v$ and $u \not\rightsquigarrow v$, respectively.

Figure 2 depicts three example policies: (a) a strict isolation policy that forbids flows across domains; (b) a high-low policy that permits flows from a low domain L to a low domain H but not from H to L; and (c) an assured pipeline policy that permits flows from H to L *only* indirectly through a declassifier DC [1]. We say that the first two policies are *transitive*, meaning $u \rightsquigarrow v$ and $v \rightsquigarrow w$ together imply $u \rightsquigarrow w$; and that the last policy is *intransitive*, as it permits $H \rightsquigarrow D$ and $D \rightsquigarrow L$, but not $H \rightsquigarrow L$. Intransitive policies are more general and can be used to specify systems that support downgrading [28].

To describe the externally visible behavior of a system, we assume a set of observations O and a function $\mathsf{observe} : S \times D \to O$ that specifies the values a domain can observe in a given state.[1] For example, given a system consisting of multiple processes, O may be defined as the subset of the system state each process can observe, such as its own memory and return values from system calls.

3.1 Trace-based specifications

Goguen and Meseguer [10] proposed the original formulation of noninterference, which was later generalized by Haigh and Young [12] and by Rushby [28]. Informally, noninterference says that if two traces are indistinguishable from the perspective of a domain u (because they differ only in actions that u is forbidden from seeing), then executing those traces should produce final states that are also indistinguishable to u, i.e., $\mathsf{observe}(s, u) = \mathsf{observe}(t, u)$. We call this formulation *trace-based* because it is expressed in terms of the allowed behavior of traces in the system.

Following Rushby [28], we formalize noninterference in terms of two auxiliary functions, $\mathsf{sources} : A^* \times D \to \mathcal{P}(D)$ and $\mathsf{purge} : A^* \times D \to A^*$. The function $\mathsf{sources}(tr, u)$ computes the set of domains that are permitted to transfer information to a domain u while executing a trace tr, either directly (i.e., $\mathsf{dom}(a) \rightsquigarrow u$) or indirectly (i.e., tr contains actions $a_1 \ldots a_n$ such that $\mathsf{dom}(a) \rightsquigarrow \mathsf{dom}(a_1) \rightsquigarrow \ldots \rightsquigarrow \mathsf{dom}(a_n) \rightsquigarrow u$):

$$\mathsf{sources}(\epsilon, u) := \{u\}$$

$$\mathsf{sources}(a \circ tr, u) := \begin{cases} \mathsf{sources}(tr, u) \cup \{\mathsf{dom}(a)\} \\ \quad \text{if } \exists v \in \mathsf{sources}(tr, u).\ \mathsf{dom}(a) \rightsquigarrow v \\ \mathsf{sources}(tr, u) \qquad \text{otherwise.} \end{cases}$$

The function $\mathsf{purge}(tr, u)$ filters the trace tr to keep only those actions that are permitted to transfer information to the domain u:

$$\mathsf{purge}(\epsilon, u) := \epsilon$$

$$\mathsf{purge}(a \circ tr, u) := \begin{cases} a \circ \mathsf{purge}(tr, u) \\ \quad \text{if } \mathsf{dom}(a) \in \mathsf{sources}(a \circ tr, u) \\ \mathsf{purge}(tr, u) \qquad \text{otherwise.} \end{cases}$$

To satisfy noninterference, for any trace tr and domain u, executing $\mathsf{purge}(tr, u)$ and executing tr should produce states indistinguishable to the domain u.

Definition 1 (noninterference). A system with an initial state s_0 satisfies noninterference for a policy $\rightsquigarrow$ iff the following holds for any domain u and trace tr:

$$\mathsf{observe}(\mathsf{run}(s_0, \mathsf{purge}(tr, u)), u) = \mathsf{observe}(\mathsf{run}(s_0, tr), u).$$

Another trace-based specification is nonleakage [38], originating from language-based security [29]. Unlike noninterference, which uses two traces over the same (initial) state, nonleakage uses the same trace over two states. Informally, nonleakage says that if two states are *equivalent* with respect to a trace and a domain, executing the same trace from the two states should result in indistinguishable states with respect to that domain. The definition of nonleakage therefore requires an additional *view-partitioning relation* $\sim \subseteq D \times S \times S$ that describes which system states appear equivalent to a given domain. We write $s \overset{u}{\sim} t$ to mean $(u, s, t) \in \sim$; we write $s \overset{C}{\approx} t$ for $C \subseteq D$ to mean $s \overset{u}{\sim} t$ for every domain $u \in C$.

Definition 2 (nonleakage). A system satisfies nonleakage for a policy $\rightsquigarrow$ and a view-partitioning relation $\sim$ iff the following holds for any trace tr, domain u, and states s, t:

$$s \overset{\mathsf{sources}(tr,u)}{\approx} t \Rightarrow$$
$$\mathsf{observe}(\mathsf{run}(s, tr), u) = \mathsf{observe}(\mathsf{run}(t, tr), u).$$

At a high level, noninterference restricts how a domain can infer *actions* in other domains, while nonleakage restricts which parts of system *state* a domain is allowed to infer. Which specification to use depends on the intended property for a given system. Recall the examples in §2. If the system intends to prevent an adversary from being able to infer what actions other processes have performed (e.g., whether they have allocated memory), noninterference is a better fit. If the system aims to protect the data of a process but not the metadata (e.g., the content of its memory, but not the occurrence of memory allocation), nonleakage is more natural as one can specify fine-grained sharing through the view-partitioning relation.

3.2 Unwinding-based specifications

The specifications of noninterference and nonleakage reason about traces. *Unwinding* is a common proof strategy for reducing such specifications to a set of conditions that must hold for individual actions, obviating the need

[1] Equivalently, one may use the function $\mathsf{output} : S \times A \to O$ [37].

OC (output consistency): equivalent states are indistinguishable.
$s \overset{u}{\sim} t \Rightarrow \texttt{observe}(s,u) = \texttt{observe}(t,u)$.[2]

SC (step consistency): executing an action on two states preserves equivalence for all domains.
$s \overset{u}{\sim} t \Rightarrow \texttt{step}(s,a) \overset{u}{\sim} \texttt{step}(t,a)$.

WSC (weak step consistency): executing an action on two states preserves equivalence for all domains if those states are equivalent from the perspective of the action's domain
$s \overset{\texttt{dom}(a)}{\sim} t \wedge s \overset{u}{\sim} t \Rightarrow \texttt{step}(s,a) \overset{u}{\sim} \texttt{step}(t,a)$.

SR (step respect): executing an action on two states preserves equivalence for all domains that the domain of the action is not permitted to flow to.
$\texttt{dom}(a) \not\rightsquigarrow u \wedge s \overset{u}{\sim} t \Rightarrow \texttt{step}(s,a) \overset{u}{\sim} \texttt{step}(t,a)$.

LR (local respect): the states before and after the execution of an action are equivalent to all domains that the domain of the action is not permitted to flow to.
$\texttt{dom}(a) \not\rightsquigarrow u \Rightarrow s \overset{u}{\sim} \texttt{step}(s,a)$.

Figure 3: Unwinding conditions. Each formula is universally quantified over its free variables, such as domain u, action a, and states s and t.

to reason about entire traces. Unwinding conditions can also be used as specifications in their own right. Figure 3 shows a list of unwinding conditions, which we briefly describe below.

OC (output consistency) specifies that a domain has the same observations given two equivalent states. This prevents, for example, an adversarial process from being able to infer secrets based on the return values of system calls from an OS kernel, which it can observe.

SC (step consistency) specifies that an action preserves state equivalence for a domain; for example, a domain's action does not leak information to part of the state accessible by the adversary. SC is usually too strong for systems with downgrading [28: §3]; relaxed forms, such as WSC (weak step consistency) and SR (step respect), are weaker than SC and suitable for such systems.

LR (local respect) specifies that an action does not affect a distrustful domain. For example, an action by the adversary has no influence on the state of other domains.

One may combine these conditions to yield unwinding specifications, as summarized in Figure 4. OC+SC and OC+WSC+LR are often used as the specifications for systems with transitive and intransitive policies, respectively, as §4 will show. OC+SC+LR and OC+WSC+SR, which imply `noninterference` [28: §4] and `nonleakage` [38: §5], respectively, are used as proof strategies rather than specifications in the analyzed systems.

Which unwinding specification to use boils down to (1) choosing between SC and WSC, which depends on whether the system's policy is transitive or intransitive; and (2) whether the system guarantees LR; for instance, whether an adversary can infer the executions of actions performed by other domains (or can influence other domains through the execution of its actions). The latter is similar to the discussion about choosing between `noninterference` and `nonleakage` (§3.1).

[2] Another form is: $s \overset{\texttt{dom}(a)}{\sim} t \Rightarrow \texttt{output}(s,a) = \texttt{output}(t,a)$ [37].

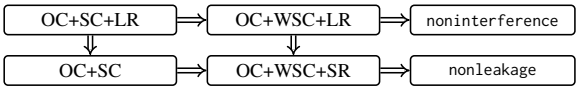


Figure 4: Logical implications among noninterference specifications.

In general, unwinding specifications are simpler and easier to understand compared to trace-based specifications, as one can think about individual actions rather than traces of actions. But there are some trade-offs. First, unwinding specifications, like `nonleakage`, require a view-partitioning relation $\sim$, which can be non-trivial to write and audit; `noninterference` does not require a view-partitioning relation. Second, both CertiKOS and Komodo use unwinding specifications with "big-step" actions, where each action can contain an unbounded number of system calls [23: §6]; such specifications blur the line between traces and individual actions.

3.3 Variations

We note a few variations of the noninterference specifications used by the systems in Figure 1.

Transitivity. For systems without downgrading, a transitive policy suffices. For systems that support downgrading, one may use a general, intransitive policy [28], which specifies that information may flow from high to low only indirectly through a downgrader.

State-dependent `dom`. OS kernels usually need to reason about the security of the scheduler, for instance, to ensure that scheduling decisions do not leak information [19]. In such cases, the domain of an action depends on which process is currently running, but that information is part of the system state rather than a static relationship. One variation of the specifications is to augment the `dom` function to take a state as an additional parameter, $A \times S \to D$, and change the auxiliary functions `sources` and `purge` accordingly [20, 31].

Merged `observe`. Recall that the `observe` function describes what a domain can observe in a state and the view-partitioning relation $\sim$ describes the subset of two states that appear equivalent to a domain. Sometimes it leads to a simpler specification by letting `observe` return the "equivalent" subset of the states, such that $\texttt{observe}(s,u) = \texttt{observe}(t,u)$ is simplified to $s \overset{u}{\sim} t$ for any states s,t and domain u. In doing so, one can remove `observe` from an unwinding specification, as OC trivially holds; or from `nonleakage`, which reduces to the following:

$$s \overset{\texttt{sources}(tr,u)}{\approx} t \Rightarrow \texttt{run}(s,tr) \overset{u}{\sim} \texttt{run}(t,tr).$$

4 Analysis

This section presents an analysis of noninterference specifications used by the systems listed in Figure 1. Based on the catalog from §3, we manually inspect and categorize their specifications. For systems that introduce their own notions of noninterference, we also attempt to interpret the specifications using the catalog.

Category 1: `noninterference`. As shown in Definition 1, a `noninterference` specification is parameterized by a policy. We briefly review examples below, from less to more general policies.

The PROSPER kernel [5, 30] supports multiplexing of two partitions on a single machine. The main theorem is that the behavior of the system is equivalent to that of an ideal system running two partitions on separate machines [27]. As a sanity check on the main theorem, given two partitions that do not communicate, it proves `noninterference` with strict isolation. This rules out information flows through memory or scheduling.

The Reflex browser kernel [25, 33] mediates access to system resources and messages among tabs and cookie processes; the rest of the browser is built on top of (untrusted) WebKit. It proves `noninterference` for a high-low policy. This policy rules out, for instance, communication between either two tabs or a tab and a cookie process of different website domains.

Similarly, the Block Access Controller [11] is a file-server component that mediates between user requests and disks. Both user requests and disk data are assigned different levels of labels. The system proves `noninterference` with a policy that permits flows from low to high labels only; this precludes a low-level user request from accessing high-level data.

Systems with rich, controlled sharing may use general, intransitive policies to support downgrading. One example is the NiStar kernel [31], which enforces decentralized information flow control [22] through a small set of object types, such as threads and pages. NiStar associates each object in the system with a triple of ⟨secrecy, integrity, ownership⟩ labels, where each label is a set of tags (opaque integers), and exposes a set of system calls to check and manipulate such labels. The policy for two such triples $L_1 = \langle S_1, I_1, O_1 \rangle$ to $L_2 = \langle S_2, I_2, O_2 \rangle$ is an extension of a classical lattice-based one [7]: information can flow from an object with L_1 to another with L_2 iff $(S_1 - O_1 \subseteq S_2 \cup O_2) \wedge (I_2 - O_2 \subseteq I_1 \cup O_1)$. NiStar proves `noninterference` for this policy, by showing that the OC+WSC+LR unwinding conditions hold.

Category 2: `nonleakage`. As shown in Definition 2, a `nonleakage` specification is parameterized by both a policy and a view-partitioning relation. One example is the seL4 kernel [15]. The system consists of a number of partitions P_i, as well as a static scheduler `PSched` that follows a pre-configured static schedule among partitions. Each partition contains a set of objects, such as threads and pages.

The security goal of seL4 is to enforce a general, intransitive policy among partitions and the scheduler:

1. $P_i \rightsquigarrow P_j$ according to a configuration of the system about any two partitions P_i and P_j;
2. $\texttt{PSched} \rightsquigarrow P_i$: the scheduler needs to be able to schedule (and thus influence) each partition P_i; and
3. $P_i \not\rightsquigarrow \texttt{PSched}$: no flow is permitted from any partition P_i to the scheduler, which would otherwise allow the scheduler to leak information between partitions when combined with (2).

Specifically, seL4 proves `nonleakage` for this policy. The proof strategy is by showing that an unwinding condition called "`confidentiality-u`" [20] holds:

$$(\texttt{dom}(a) \rightsquigarrow u \Rightarrow s \overset{\texttt{dom}(a)}{\sim} t) \wedge s \overset{u}{\sim} t \Rightarrow \texttt{step}(s,a) \overset{u}{\sim} \texttt{step}(t,a).$$

This unwinding condition is a variant of OC+WSC+SR. One may obtain it by combining WSC and SR; OC trivially holds, as seL4's specification merges the `observe` function with the view-partitioning relation (§3.3).

Next, consider the Ironclad system [13] with end-to-end security guarantees. An Ironclad application reads input from the network, performs computation using the input and secret keys, and writes output to the network. The main security goal is to ensure that the application's output depends only on its input, while still being able to declassify values derived from the secret keys in a controlled way. For example, a declassifier may produce a signature that is allowed to be written to output even though its value depends on the secret key.

Ironclad breaks this security goal into "input noninterference" and "output noninterference" for each application. The former specifies that the input to a declassifier depends only on (low) input from the network; and the latter specifies that the output to the network depends only on (low) input from the network and output of a declassifier. The noninterference specifications do not cover the declassifier, for which Ironclad proves a separate functional correctness specification.

One can interpret both noninterference specifications using `nonleakage` with a high-low policy, $L \rightsquigarrow H$ but $H \not\rightsquigarrow L$, where L represents public inputs and outputs, and H represents the entire system state (including secret keys). With this policy, `nonleakage` reduces to:

$$s \overset{L}{\sim} t \Rightarrow \texttt{observe}(\texttt{run}(s,tr),L) = \texttt{observe}(\texttt{run}(t,tr),L).$$

That is, low output depends only on low data, independent of (high) secret keys.

Category 3: OC+SC. Consider the CertiKOS kernel that proves noninterference [3, 4]. The system statically par-

titions memory and process identifiers among processes, with no inter-process communication. A process may print to the local output, spawn a child using its own memory quota, or yield to another process.

At the core of the noninterference specification of CertiKOS is the notion of "state indistinguishability" in a *per-process* view: a process behaves as if it were the only process in the system. Particularly, for a given process, it considers an action to be either a system call that does not change the current process, or a yield from the current process, followed by a number of system calls performed by other processes, and eventually a yield back to the original process. "State indistinguishability" specifies that the execution of such an action preserves state equivalence for a given process. One can interpret this specification using OC+SC with strict isolation: SC corresponds to "state indistinguishability"; OC can be viewed as "state indistinguishability" applied to the print call, where the `observe` function returns the local output.

The noninterference specification of CertiKOS applies to processes that already exist in the system; it does not cover information flows from a process to a newly created child during spawn, which does not satisfy strict isolation. An alternative OC+WSC+LR specification restricts parent-to-child flows during spawn [23: §6.2].

The SFSCQ file system [14] serves files owned by mutually distrustful users. The key specification is "data noninterference," which precludes an adversary from inferring the contents of files owned by other users via file-system operations. It does not, however, aim to preclude the adversary from inferring metadata (e.g., the existence of files or their lengths) or overwriting files owned by other users; these cases are covered separately by a functional correctness specification.

To achieve this goal, SFSCQ proves "return-value noninterference" and "state noninterference" for file-system operations. Upon the invocations of an operation on two states equivalent to a given user, the former specifies that both invocations produce the same return value and the latter specifies that the resulting states remain equivalent to the user. One can interpret the specification of SFSCQ using OC+SC with strict isolation: OC and SC correspond to "return-value noninterference" and "state noninterference," respectively.

One tricky case is `chown`, which changes the ownership of a file. It does not satisfy "state noninterference" (or SC). To see why, consider two states that differ only in the contents of one file owned by Alice. Suppose Alice changes the ownership of the file to Bob. Before `chown`, the two states appear equivalent to Bob, since he has no access to the file; however, after `chown`, the resulting two states no longer appear equivalent to Bob as he gains access to the file. To work around the issue, SFSCQ relaxes "state noninterference" for `chown` by adding an extra precondition: either the contents of the file are the same in the two states before `chown`, or `chown` is invoked to transfer the file ownership to a user other than Bob; in doing so, the resulting states remain equivalent to Bob [14: §7.2].

Another example is the Komodo monitor [9], which implements SGX-like enclaves using ARM TrustZone. Komodo exposes monitor calls for the OS to create, execute, and destroy enclaves, and for enclaves to request services and communicate with the OS. The security goal is to provide isolation for enclaves in the face of an adversary consisting of the OS and a colluding enclave, to preclude the adversary from inferring or influencing the execution of enclaves.

Komodo considers an action to be either a monitor call by the OS that does not involve context switching, or a series of monitor calls that start the execution of an enclave by the OS and eventually exit from the enclave to the OS; that is, each action both starts and ends with the OS. Given an enclave, Komodo proves "enclave confidentiality," that each action preserves state equivalence for the adversary (i.e., adversary cannot infer the enclave's secrets from the action); and "enclave integrity," that each action preserves state equivalence for the enclave. One can interpret both using OC+SC with strict isolation: SC specifies the preservation of state equivalence for the adversary and for a given enclave (OC is merged with the view-partitioning relation).

This interpretation helps explain a subtlety in the specification. "Enclave integrity," which can be interpreted as SC for a given enclave, precludes the OS from overwriting the enclave's memory using data the enclave cannot observe; however, it does not preclude the OS from doing so using public data such as zeros—such an operation would not violate SC, as the resulting states would remain equivalent to the enclave. An alternative OC+WSC+LR specification rules out such cases [23: §6.3].

Another subtlety is that enclaves need to declassify data (e.g., exit values) to the OS to be useful. Similar to `chown` in SFSCQ, such a downgrading operation does not satisfy "enclave confidentiality" (or SC). To work around this issue, Komodo relaxes the specification by adding axioms on the states upon an enclave exit.

Category 4: OC+WSC+LR. As a proof strategy for `noninterference` [28], OC+WSC+LR can also be used as a specification due to the fine-grained sharing afforded by the view-partitioning relation. Two examples of such use are Komodo^s and CertiKOS^s [23], ports of Komodo and CertiKOS, respectively. Their specifications differ from those of the original systems in two key aspects.

First, these specifications use "small-step" actions, where each action is an individual system or monitor call, enabling automated verification using Z3; those of the original systems use "big-step" actions, where each

action can contain an unbounded number of operations. Second, these specifications use intransitive policies for downgrading operations; as described earlier, those of the original systems use transitive policies (strict isolation), and either sidestep specifying downgrading (e.g., for parent-to-child flows in spawn in CertiKOS) or introduce axioms (e.g., for enclave exit in Komodo).

Discussion. Practical systems usually contain downgrading operations, which can manifest themselves as communications among partitions or processes, change of file ownership, or declassification of results of computation. A key challenge in specifying noninterference is deciding how to handle downgrading. Below we summarize the approaches used by the systems analyzed in this section.

The first approach is to apply noninterference only to the non-downgrading part of a system, such as in Ironclad or CertiKOS, and prove a separate functional correctness specification for the downgrading part (e.g., the declassifier). Doing so keeps the noninterference specification simple, but limits the scope of applying noninterference and requires care to separate the downgrading part from the non-downgrading one.

The second approach is to choose an unwinding specification with a transitive policy, such as OC+SC with strict isolation, and relax the specification by adding preconditions or axioms only for downgrading operations (e.g., `chown`), which would otherwise violate the specification. Two examples are SFSCQ and Komodo. This has the advantage of keeping the policy simple, but complicates the implications of the specification—the extra preconditions or axioms must be audited; and relaxing certain operations but not others means that the unwinding specification no longer implies a trace-based one.

The third approach is to use a specification with an intransitive policy, such as in NiStar, seL4, CertiKOS^s, or Komodo^s, making downgrading explicit in the noninterference specification. But it can be difficult to come up with a desired policy, and the specification is less intuitive compared to one with a transitive policy [26, 36].

5 Conclusion

Information-flow properties are key to the security of systems. It is crucial to understand the implications and trade-offs among the various forms of noninterference specifications and the types of bugs that can be prevented. We hope that this paper will help readers better understand different flavors of noninterference and motivate research for improving the practice of reasoning about information flows in systems.

References

[1] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, Gaithersburg, MD, Sept.–Oct. 1985.

[2] V. Costan and S. Devadas. Intel SGX explained. Report 2016/086, Cryptology ePrint Archive, Feb. 2016.

[3] D. Costanzo. *Formal End-to-End Verification of Information-Flow Security for Complex Systems*. PhD thesis, Yale University, Dec. 2016.

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

[5] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pages 223–234, Berlin, Germany, Nov. 2013.

[6] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, Budapest, Hungary, Mar.–Apr. 2008.

[7] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5): 236–243, May 1976.

[8] S. Eggert. *Security via Noninterference: Analyzing Information Flows*. PhD thesis, Kiel University, July 2014.

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

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

[11] P. Graunke. Verified safety and information flow of a block device. *Electronic Notes in Theoretical Computer Science*, 217:187–202, July 2008.

[12] J. T. Haigh and W. D. Young. Extending the noninterference version of MLS for SAT. *IEEE Transactions on Software Engineering*, 13(2):141–150, Feb. 1987.

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

[14] A. İleri, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Proving confidentiality in a file system using DiskSec. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 323–338, Carlsbad, CA, Oct. 2018.

[15] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–70, Feb. 2014.

[16] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, Oct. 1973.

[17] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 348–370, Dakar, Senegal, Apr.–May 2010.

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

[19] T. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein. Noninterference for operating system kernels. In *Proceedings of the 2nd International Conference on Certified Programs and Proofs*, pages 126–142, Kyoto, Japan, Dec. 2012.

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

[21] T. Murray, R. Sison, and K. Engelhardt. COVERN: A logic for compositional verification of information flow control. In *Proceedings of the 3rd IEEE European Symposium on Security and Privacy*, pages 16–30, London, United Kingdom, Apr. 2018.

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

[23] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 225–242, Huntsville, Ontario, Canada, Oct. 2019.

[24] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Feb. 2016.

[25] D. Ricketts, V. Robert, D. Jang, Z. Tatlock, and S. Lerner. Automating formal proofs for reactive systems. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 452–462, Edinburgh, United Kingdom, June 2014.

[26] A. Roscoe and M. Goldsmith. What is intransitive noninterference? In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW)*, pages 228–238, Mordano, Italy, June 1999.

[27] J. Rushby. Design and verification of secure systems. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)*, pages 12–21, Pacific Grove, CA, Dec. 1981.

[28] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, Dec. 1992.

[29] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[30] O. Schwarz. *No Hypervisor Is an Island: System-wide Isolation Guarantees for Low Level Code*. PhD thesis, KTH, sep 2016.

[31] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. Wang. Nickel: A framework for design and verification of information flow control systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 287–306, Carlsbad, CA, Oct. 2018.

[32] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proceedings of the*

17th ACM SIGPLAN International Conference on Functional Programming (ICFP), pages 201–214, Copenhagen, Denmark, Sept. 2012.

[33] Z. Tatlock. *Reducing the Costs of Proof Assistant Based Formal Verification or: Conviction without the Burden of Proof*. PhD thesis, University of California, San Diego, 2014.

[34] The Coq Development Team. *The Coq Proof Assistant, version 8.9.0*, Jan. 2019. URL https://doi.org/10.5281/zenodo.2554024.

[35] T. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*, pages 187–202, Venice, Italy, July 2007.

[36] R. van der Meyden. What, indeed, is intransitive noninterference? In *Proceedings of the 12th European Symposium on Research in Computer Security*, 2007.

[37] R. van der Meyden and C. Zhang. A comparison of semantic models for noninterference. *Theoretical Computer Science*, 411(47):4123–4147, Oct. 2010.

[38] D. von Oheimb. Information flow control revisited: Noninfluence = noninterference + nonleakage. In *Proceedings of the 9th European Symposium on Research in Computer Security*, pages 225–243, Sophia Antipolis, France, Sept. 2004.