Nickel: A Framework for Design and Verification of Information Flow Control Systems

Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney,
James Bornholt, Emina Torlak, Xi Wang
University of Washington

Abstract

Nickel is a framework that helps developers design and verify information flow control systems by systematically eliminating *covert channels* inherent in the interface, which can be exploited to circumvent the enforcement of information flow policies. Nickel provides a formulation of noninterference amenable to automated verification, allowing developers to specify an intended policy of permitted information flows. It invokes the Z3 SMT solver to verify that both an interface specification and an implementation satisfy noninterference with respect to the policy; if verification fails, it generates counterexamples to illustrate covert channels that cause the violation.

Using Nickel, we have designed, implemented, and verified NiStar, the first OS kernel for decentralized information flow control that provides (1) a precise specification for its interface, (2) a formal proof that the interface specification is free of covert channels, and (3) a formal proof that the implementation preserves noninterference. We have also applied Nickel to verify isolation in a small OS kernel, NiKOS, and reproduce known covert channels in the ARINC 653 avionics standard. Our experience shows that Nickel is effective in identifying and ruling out covert channels, and that it can verify noninterference for systems with a low proof burden.

1 Introduction

Operating systems often provide information flow control mechanisms to improve application security. These mechanisms enforce policies ranging from strict isolation to more flexible models using labels [12, 60]. By tracking and mediating data access, they aim to regulate the propagation of information among applications to provide secrecy and integrity guarantees.

Malicious applications can circumvent information flow control systems by encoding and transferring information indirectly, such as through temporary files, process names, or CPU and memory usage [47]. Many such *covert channels* exist not only in the POSIX interface but also in specialized information flow control systems (see §2 for a survey). For example, Krohn et al. [45] have described covert channels in Asbestos [15] that allow applications to leak data at a high bandwidth. Covert channels in the interface are critical flaws as *no* secure implementation of such an interface can exist [44]. Eliminating these channels at the interface level is thus a key challenge in the design of information flow control systems.

Even if an interface specification is free of covert channels, it remains challenging to correctly implement the system—incorrect or missing checks will invalidate the guarantees of information flow control. For instance, both KLEE [6: §5.3] and STACK [78: §6.1] have found such bugs in HiStar [82]. As another example, the implementation of Flume [45] relies on the Linux kernel, which is likely to contain bugs given its complexity [8, 51, 63].

This paper presents Nickel, a framework for systematically eliminating covert channels from such systems through formal verification of *noninterference*. Noninterference is a general security criterion that has been extensively studied in prior work [24, 53, 67]. Intuitively, given two mutually distrustful threads between which information flow is prohibited, noninterference requires the output of operations in one thread to be independent of operations in the other thread. This restriction ensures that a malicious thread can neither infer secrets nor influence the execution path of another thread via operations defined in the interface; any violation indicates a covert channel. However, applying noninterference to reason about an interface requires considering the precise behavior of each operation as well as the interaction of all pairs of operations [38, 39], which is non-trivial. Nickel helps automate this reasoning using an SMT solver such as Z3 [11].

Nickel introduces both a formulation of noninterference and new proof strategies that are amenable to automated verification. It asks developers to specify a concise and intuitive policy that describes permitted flows in a system, and checks whether a given interface specification satisfies noninterference for that policy. Furthermore, it extends our previous work on push-button verification [62, 70] to check whether a given implementation preserves noninterference through refinement. Verifying both an interface and an implementation this way incurs a low proof burden (see §8). An additional advantage of automated reasoning is that Nickel will provide a *counterexample* when it finds a covert channel in either the interface or the implementation, which is valuable for debugging and revising the design.

We have applied Nickel to three systems. The foremost is NiStar, a new OS kernel with provably secure decentralized information flow control (DIFC) [60]. DIFC is a flexible mechanism that allows applications to express powerful policies, but this flexibility makes it challenging to analyze covert channels and security implications of DIFC systems [44]. Inspired by HiStar [82], NiStar provides DIFC support through a small number of kernel object types. Unlike HiStar, however, NiStar provides a formal proof that both its interface and implementation satisfy noninterference, ruling out covert channels in the design. To the best of our knowledge, NiStar is the first formally verified DIFC OS kernel.

To demonstrate Nickel's applicability to a broader set of systems, we have used Nickel to verify NiKOS, an OS kernel that mirrors mCertiKOS [10] to enforce process isolation. We have also applied Nickel to formalize and analyze the specification of the communication interface of ARINC 653 [1], an industrial avionics standard. Nickel was able to reproduce the three covert channels in ARINC 653 previously reported by Zhao et al. [86].

Nickel reasons about sequential (uniprocessor) systems and provides no guarantees in concurrent settings. It focuses on eliminating covert channels inherent in the interface; physical effects (e.g., timing, sound, and energy) that are not captured by the interface specification are beyond the scope of this paper. We discuss these limitations further in §3.5.

In summary, this paper makes three contributions: (1) a formulation of noninterference and proof strategies amenable to automated reasoning; (2) the Nickel framework for verifying noninterference for the interface and implementation of information flow control systems; and (3) the formal specifications of three systems, including the first formally verified DIFC OS kernel.

The rest of this paper is organized as follows. §2 surveys common patterns of covert channels in interfaces. §3 formalizes noninterference and introduces theorems for proving noninterference. §4 gives an overview of the development workflow using Nickel. §5 presents guidelines for interface design. §6 describes the design, implementation, and verification of DIFC in NiStar. §7 describes the verification of isolation in NiKOS and ARINC 653. §8 reports our experience with using Nickel. §9 relates Nickel to prior work. §10 concludes.

2 Covert channels in interfaces

Nickel's main goal is to help developers identify and eliminate covert channels in the interface of an information flow control system. This section surveys common examples of covert channels and shows how to apply noninterference to understand them.

Consider two threads T_1 and T_2 that are prohibited from communicating as per the information flow policy. What kinds of interface operations can be exploited by the two threads to collude and bypass the policy (or equivalently, allow an adversarial T_2 to infer secret information from an uncooperative T_1)? As a simple example, if an operation introduces shared memory locations that both threads can read and write, then the two threads can use these memory locations as covert channels to transfer information. Unintended covert channels, however, are often subtle and difficult to spot, as detailed next.

Resource names. Resource names, such as thread identifiers, page numbers, and port numbers, can be used to encode information. Consider a system call `spawn` that creates new threads with sequential identifiers. Thread T_2 first spawns a thread with an identifier, say, 3; the other thread T_1 then spawns x times, creating threads with identifiers from 4 to $x+3$; and thread T_2 spawns another thread, whose identifier will be $x + 3 + 1$. In doing so, thread T_2 learns the secret x from T_1 through the difference of the identifiers of the two threads it has created [10: §5].

Resource exhaustion. Suppose that the system has a total of N pages shared by all threads. Thread T_1 first allocates $N-1$ pages, and encodes a one-bit secret by either allocating the last page or not. The other thread T_2 then tries to allocate one page and learns the secret based on whether the allocation succeeds [82: §3]. This covert channel is effective especially when a resource is limited and can be easily exhausted.

Statistical information. A thread's world-readable information, such as its name, number of open file descriptors, and CPU and memory usage, can be used to encode secret data or by adversarial threads to learn secrets [35, 85]. For example, if thread T_1's memory usage is accessible to another thread T_2 through `procfs` or system calls, T_1 could leak a secret x by allocating x pages.

Error handling. Error handling is known to be prone to information leakage [54], such as the TENEX password-guessing attack using page faults [48] and the POODLE attack against TLS [55]. As an example, consider a system call for querying the status of a page, which returns `-ENOENT` if the given page is free and `-EACCES` if the page is in use but not accessible by the calling thread. Thread T_1 encodes a one-bit secret by allocating a particular page or not; the other thread T_2 queries the status of that page and learns the secret based on whether the error code is `-ENOENT` or `-EACCES`.

Scheduling. Suppose an OS kernel uses a round-robin scheduler that distributes time slices evenly among threads. Thread T_1 encodes a secret by forking a number of threads (e.g., a fork bomb), which causes the other thread T_2 to observe the reduction of time allocated for itself and learn the secret from T_1; alternatively, T_2 can continuously ping a remote server, which will learn the secret from the time between pings [82: §9]. Access to only logical time suffices for such covert channels.

External devices and services. Suppose the system allows threads to communicate with external devices and services. Thread T_1 can write secret data to the registers of a device, or encode the secret as the frequency of accessing a device or even through a service bill [47]; the other thread T_2 can then retrieve the secret at a later time from the same device or service.

Mutable labels. Many information flow control systems express security policies by assigning *labels* to objects. Label changes complicate such systems and can lead to covert channels [12]. As an example, consider a system where each thread can be labeled as either tainted or untainted. The system enforces a tainting policy: a tainted thread cannot transfer information to an untainted thread without tainting it. To enforce this policy, the system raises the label of an untainted thread to tainted when another tainted thread sends data to it. Suppose thread T_1 is tainted and thread T_2 is untainted. To bypass the policy, T_2 first spawns an untainted helper thread H. T_1 encodes a one-bit secret by choosing whether to send data to taint H, which in turn chooses to send data to T_2 *only* if it is untainted and do nothing otherwise. In this way, T_2 learns the secret from T_1 by whether it receives data from H, without becoming tainted itself [44: §3].

2.1 Applying noninterference

Given two threads T_1 and T_2 that are prohibited from communicating with each other, noninterference states that the output of operations in one thread should not be affected by whether operations in the other thread occur. Now we will show how to apply noninterference to uncover covert channels.

Take the spawn system call as an example, which returns sequential thread identifiers and introduces a covert channel due to resource names. Figure 1 illustrates this channel. We denote an action of invoking a system call as a left half-circle spawn and its return value as a right half-circle 3. We use different colors to distinguish system calls from different threads: spawn_1 in T_1; spawn_0 and spawn_2 in T_2.

We apply noninterference to uncover the covert channel introduced by spawn in three steps. First, construct a trace of actions from both threads, for instance, spawn_0 spawn_1 spawn_2. Assume that the corresponding return values (i.e., outputs) are 3 4 5, as spawn sequentially allocates identifiers. Second, to examine possible effects of T_1 on T_2, construct a new trace that *purges* the actions from T_1 and retains the actions only from T_2, resulting in spawn_0 spawn_2. Third, replay this purged trace to the system, obtaining a new sequence of outputs 3 4. This

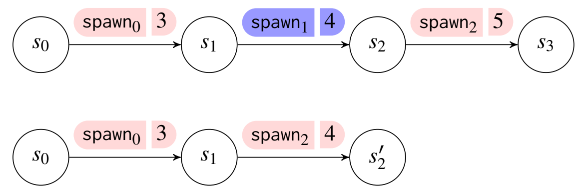

Figure 1: The output of spawn_2 changes from 5 in the original trace (first row) to 4 in the purged trace (second row), indicating a covert channel. Circles denote states, arrows denote state transitions, left half-circles denote actions, and right half-circles denote outputs.

sequence differs from the original output of the same actions, which is 3 5. The change of output in T_2 (in particular, the return value of spawn_2) caused by an action in T_1 violates noninterference, indicating a covert channel with which T_1 may transfer information to T_2. On the other hand, with a version of spawn that does not introduce a covert channel, the outputs of T_2's actions in the purged and original traces would be the same.

One can similarly apply noninterference to uncover the other covert channels described in this section. The challenge is to find a trace of actions that manifests the covert channel, and if there are no such channels, to exhaustively show that no trace violates noninterference. Nickel automates this task using formal verification techniques, as we will describe next.

3 Proving noninterference

This section formalizes the notion of noninterference used in Nickel and presents the main theorems that enable Nickel to prove noninterference for systems.

First, we address how to specify the intended policy of an information flow control system. The policy is trusted as the top-level specification of the system, which will be used to catch and fix potential covert channels in both the interface specification and the implementation (§3.1).

Next, we give a formal definition of noninterference in terms of traces of actions, which precisely captures whether an interface specification satisfies a given policy (§3.2).

To prove noninterference for an interface specification, Nickel introduces an unwinding verification strategy that requires reasoning only about individual actions, rather than traces of actions (§3.3). To extend the guarantee of noninterference to an implementation, Nickel introduces a restricted form of refinement that preserves noninterference (§3.4). Both strategies are amenable to automated verification using an SMT solver.

We end this section with a discussion of the limitations of the Nickel approach (§3.5).

3.1 Policy

We model the execution of a system as a state machine in a standard way [67]. A system $\mathcal{M}$ is defined as a tuple

$\langle A, O, S, \texttt{init}, \texttt{step}, \texttt{output}\rangle$, where A is the set of actions, O is the set of output values, S is the set of states, $\texttt{init}$ is the initial state, $\texttt{step} : S \times A \to S$ is the state-transition function, and $\texttt{output} : S \times A \to O$ is the output function.

An action transitions the system from state to state. In the context of an OS, an action can be either a user-space operation (e.g., memory access), or the handling of a trap due to system calls, exceptions, or scheduling. Each action consists of an operation identifier (e.g., the system call number) and arguments. We write $\texttt{output}(s, a)$ and $\texttt{step}(s, a)$ to denote the output value (e.g., the return value of a system call) and the next state, respectively, for the state s and action a. Actions are considered to be atomic; for instance, we assume that an OS kernel executes each trap handler with interrupts disabled on a uniprocessor system [40, 62].

A *trace* is a sequence of actions. We use $\texttt{run}(s, tr)$ to denote the state produced by executing each action in trace tr starting from state s. The $\texttt{run}$ function is defined as follows:

$$\begin{aligned} \texttt{run}(s, \epsilon) &:= s \\ \texttt{run}(s, a \circ tr) &:= \texttt{run}(\texttt{step}(s, a), tr). \end{aligned}$$

Here, ϵ denotes the empty trace, and $a \circ tr$ denotes the concatenation of action a and trace tr.

Definition 1 (Information Flow Policy)**.** A policy $\mathcal{P}$ for system $\mathcal{M}$ is defined as a tuple $\langle D, \rightsquigarrow, \texttt{dom}\rangle$, where D is the set of *domains*, $\rightsquigarrow \subseteq (D \times D)$ is the can-flow-to relation between two domains, and the function $\texttt{dom} : A \times S \to D$ maps an action with a state to a domain.

Intuitively, a domain is an abstract representation of the exercised authority of an action. A policy associates each action a performed from state s with a domain, denoted by $\texttt{dom}(a, s)$; the can-flow-to relation $\rightsquigarrow$ defines permitted information flows among these domains. The goal of a policy is to explicitly specify permitted flows and ensure that any trace of actions, given their specifications, will *not* lead to covert channels that enable unintended flows and violate the policy.

Below we show the policies for two example systems. We write $u \rightsquigarrow v$ and $u \not\rightsquigarrow v$ to mean $(u, v) \in \rightsquigarrow$ and $(u, v) \notin \rightsquigarrow$, respectively.

Example (Tainting)**.** Consider the label-based system mentioned in §2: it has a number of threads, where the label of each thread is either tainted or untainted. The system enforces a tainting policy as depicted in Figure 2. The policy permits information flow from untainted threads to either untainted or tainted threads, and between two tainted threads, but it prohibits untainted threads from directly communicating with tainted ones.

For this policy, we designate $\{\textit{tainted}, \textit{untainted}\}$ as the set of domains. The can-flow-to relation consists of

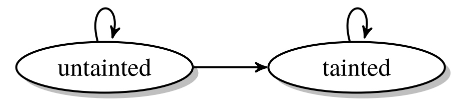

Figure 2: The tainting policy: information cannot flow from tainted threads to untainted threads.

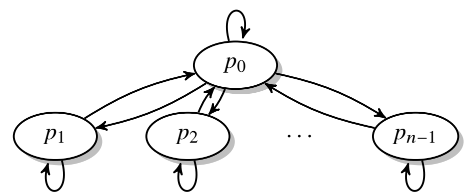

Figure 3: The isolation policy of NiKOS: information cannot flow between any two of the regular processes $p_1, p_2, \ldots, p_{n-1}$ (except through the scheduler p_0 indirectly).

the following three permitted flows: $\textit{tainted} \rightsquigarrow \textit{tainted}$, $\textit{untainted} \rightsquigarrow \textit{untainted}$, and $\textit{untainted} \rightsquigarrow \textit{tainted}$. The $\texttt{dom}$ function returns the label of the thread currently running. NiStar employs a more sophisticated version of this policy using DIFC (see §6).

Example (Isolation)**.** Consider a Unix-like kernel with n processes: a special scheduler process p_0, and regular processes $p_1, p_2, \ldots, p_{n-1}$. The system enforces a process isolation policy as depicted in Figure 3, which permits information flows from a process to itself, from the scheduler to any process, and from any process to the scheduler; no information flow is permitted between any two regular processes except indirectly through the scheduler [10].

To specify this isolation policy, we designate the processes $\{p_0, p_1, \ldots, p_{n-1}\}$ as the set of domains, where p_0 is the scheduler. The can-flow-to relation consists of the permitted flows $p_0 \rightsquigarrow p_i$, $p_i \rightsquigarrow p_0$, and $p_i \rightsquigarrow p_i$, for all $i \in [0, n-1]$. The $\texttt{dom}$ function returns the currently running process as the domain for system call actions, and returns the scheduler p_0 as the domain for context switching actions. NiKOS employs this policy (see §7).

We highlight two features in our policy definition (Definition 1). First, it allows the can-flow-to relation $\rightsquigarrow$ to be *intransitive* [67]. For instance, the isolation policy permits processes p_1 and p_2 to communicate through the scheduler, but prohibits them from communicating directly with each other. In other words, $p_1 \rightsquigarrow p_0$ and $p_0 \rightsquigarrow p_2$ do *not* have to imply $p_1 \rightsquigarrow p_2$, though that would also be accepted by Nickel if it were the intended policy.

This generality enables Nickel to support a broad range of policies, as practical systems often need *downgrading* operations (e.g., intentional declassification and endorsement) [49]. As a simple example, a system may prefer to have an untrusted application send data to an encryption program, which in turn is permitted to reach the network, while the application itself is prohibited from sending

$$\mathtt{sources}(\epsilon, u, s) \coloneqq \{u\}$$

$$\mathtt{sources}(a \circ tr, u, s) \coloneqq \mathtt{sources}(tr, u, \mathtt{step}(s, a)) \cup \begin{cases} \{\mathtt{dom}(a, s)\} & \text{if } \exists v \in \mathtt{sources}(tr, u, \mathtt{step}(s, a)).\ \mathtt{dom}(a, s) \rightsquigarrow v \\ \varnothing & \text{otherwise.} \end{cases}$$

Figure 4: $\mathtt{sources}(tr, u, s)$ is the set of domains that are allowed to influence domain u over a trace tr, starting from state s.

$$\mathtt{purge}(\epsilon, u, s) \coloneqq \{\epsilon\}$$

$$\mathtt{purge}(a \circ tr, u, s) \coloneqq \{a \circ tr' \mid tr' \in \mathtt{purge}(tr, u, \mathtt{step}(s, a))\} \cup \begin{cases} \varnothing & \text{if } \mathtt{dom}(a, s) \in \mathtt{sources}(a \circ tr, u, s) \\ \mathtt{purge}(tr, u, s) & \text{otherwise.} \end{cases}$$

Figure 5: $\mathtt{purge}(tr, u, s)$ is the set of all sub-traces of tr that retain the actions that are allowed to influence domain u, starting from state s.

data directly over the network. Such policies require intransitive can-flow-to relations [67, 80].

Second, in classical noninterference [24, 67], the dom function is state-independent ($A \to D$). The definition of dom used in Nickel is *state-dependent* ($A \times S \to D$). This extension is necessary for reasoning about many systems in which the domain (i.e., authority) of an action depends on the currently running thread or process [56, 68]. As we will show next, we have developed a definition of noninterference and theorems for proving noninterference that accommodate this extension.

3.2 Noninterference

Given a system and a policy for the system, what kind of action can violate the policy and introduce covert channels? As described in §2, to check for noninterference, one can construct a trace of actions, obtain a purged trace by removing actions from the original trace as per the policy, and compare the output of the corresponding actions in both traces—any change of output indicates a covert channel. Below we give a precise definition of noninterference that captures this intuition, in three steps.

First, suppose that a system has executed a trace tr to reach the state $\hat{s} = \mathtt{run}(\mathtt{init}, tr)$, and is about to perform action $\hat{a}$ next. To construct a purged trace of tr, we need to identify the actions that the policy permits to influence a domain u and therefore should be retained in the trace. This set is defined using the $\mathtt{sources}(tr, u, s)$ function shown in Figure 4, which returns the set of domains that can transfer information to domain u over trace tr from state s, either directly specified by the can-flow-to relation or indirectly through the domain of another intermediate action in the trace.

Second, to obtain a purged trace that retains the actions identified by sources, we define the $\mathtt{purge}(tr, u, s)$ function as shown in Figure 5. It returns the set of all sub-traces of tr where each action in the sources of u from state s has been retained; the actions whose domains are not identified by sources are optionally removed.

Third, let tr' denote a purged trace in the set $\mathtt{purge}(tr, \mathtt{dom}(\hat{a}, \hat{s}), \mathtt{init})$; like other traces in this set, tr' is obtained by retaining actions in trace tr that can transfer information to action $\hat{a}$. Now let's replay the purged trace tr' from the start, resulting in a new state $\hat{s}' = \mathtt{run}(\mathtt{init}, tr')$. If the system satisfies noninterference for the policy, then invoking $\hat{a}$ from state $\hat{s}$ should produce the same output as invoking $\hat{a}$ from state $\hat{s}'$.

Formally, we define noninterference as follows:

Definition 2 (Noninterference)**.** Given a system $\mathcal{M} = \langle A, O, S, \mathtt{init}, \mathtt{step}, \mathtt{output}\rangle$ and a policy $\mathcal{P} = \langle D, \rightsquigarrow, \mathtt{dom}\rangle$, $\mathcal{M}$ satisfies noninterference for $\mathcal{P}$ if and only if the following holds for any trace tr, action a, and purged trace $tr' \in \mathtt{purge}(tr, \mathtt{dom}(a, \mathtt{run}(\mathtt{init}, tr)), \mathtt{init})$:

$$\mathtt{output}(\mathtt{run}(\mathtt{init}, tr), a) = \mathtt{output}(\mathtt{run}(\mathtt{init}, tr'), a).$$

To ensure that our definition of noninterference is reasonable, we show two properties of this definition. First, recall that we use a state-dependent dom function; if dom is restricted to be state-independent, that is, $\mathtt{dom}(a, s) = \mathtt{dom}(a)$ holds for any a and s, then our definition reduces to classical noninterference [67], suggesting that our definition is a natural extension.

Second, a reasonable definition of noninterference should be *monotonic* [17]: a system satisfying noninterference for some policy should also satisfy noninterference for a more relaxed policy in which more flows are permitted. More formally, given two policies $\mathcal{P} = \langle D, \rightsquigarrow, \mathtt{dom}\rangle$ and $\mathcal{P}' = \langle D, \rightsquigarrow', \mathtt{dom}\rangle$, we say $\mathcal{P}'$ *contains* $\mathcal{P}$ to mean that any flow permitted by $\mathcal{P}$ is also permitted by $\mathcal{P}'$ (i.e., $\rightsquigarrow \subseteq \rightsquigarrow'$). We have proved the following monotonicity property as a sanity check on our definition of noninterference: if a system $\mathcal{M}$ satisfies noninterference for a policy $\mathcal{P}$, then it also satisfies noninterference for any policy $\mathcal{P}'$ that contains $\mathcal{P}$.

3.3 Unwinding

It is difficult to directly apply Definition 2 to prove noninterference for a given system and policy, as it requires

$\mathcal{I}$ **is a state invariant:**

$$\mathcal{I}(\texttt{init}) \wedge (\mathcal{I}(s) \Rightarrow \mathcal{I}(\texttt{step}(s, a)))$$

$\overset{u}{\approx}$ **is an equivalence relation:**

$\overset{u}{\approx}$ is reflexive, symmetric, and transitive

$\overset{u}{\approx}$ **is consistent with** $\texttt{dom}$**:**

$$\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{\texttt{dom}(a,s)}{\approx} t \Rightarrow \texttt{dom}(a, s) = \texttt{dom}(a, t)$$

$\overset{u}{\approx}$ **is consistent with** $\rightsquigarrow$**:**

$$\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{u}{\approx} t \Rightarrow (\texttt{dom}(a, s) \rightsquigarrow u \Leftrightarrow \texttt{dom}(a, t) \rightsquigarrow u)$$

output consistency:

$$\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{\texttt{dom}(a,s)}{\approx} t \Rightarrow \texttt{output}(s, a) = \texttt{output}(t, a)$$

local respect:

$$\mathcal{I}(s) \wedge \texttt{dom}(a, s) \not\rightsquigarrow u \Rightarrow s \overset{u}{\approx} \texttt{step}(s, a)$$

weak step consistency:

$$\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{u}{\approx} t \wedge s \overset{\texttt{dom}(a,s)}{\approx} t \Rightarrow \texttt{step}(s, a) \overset{u}{\approx} \texttt{step}(t, a)$$

Figure 6: Unwinding conditions. Each formula is universally quantified over its free variables, such as domain u, action a, and states s and t.

reasoning about all possible traces. A standard approach is to define a set of *unwinding conditions*, which together imply noninterference but require reasoning only about individual actions. We generalize the classical unwinding conditions given by Rushby [67] to obtain an unwinding theorem that accommodates our state-dependent `dom` function and is amenable to automated verification. Proving noninterference using the unwinding theorem requires two extra inputs from developers: a *state invariant* and an *observational equivalence* relation, as described next.

A state invariant $\mathcal{I}$ [46] is a state predicate that must hold on all *reachable* states (i.e., the set of states produced by running any trace starting from the `init` state). The state invariant overapproximates the set of reachable states, as it may also hold for unreachable states. If the unwinding theorem holds for states satisfying $\mathcal{I}$, then it holds for all reachable states of the system. We use this overapproximation to enable automation: in contrast to reachability, which cannot be expressed in first-order logic, the state invariant can be both expressed and effectively checked with an SMT solver.

The next input required for the unwinding theorem is an observational equivalence relation $\approx \subseteq (D \times S \times S)$. The observational equivalence describes, for each domain, the set of states that appear to that domain to be indistinguishable. We write $\overset{u}{\approx}$ to mean the binary relation $\{(s, t) \mid (u, s, t) \in \approx\}$ relating all equivalent states for domain u, and $s \overset{u}{\approx} t$ to mean $(u, s, t) \in \approx$.

We then define the unwinding conditions of system $\mathcal{M}$ for policy $\mathcal{P}$, shown in Figure 6, and prove the following unwinding theorem:

Theorem 1 (Unwinding)**.** A system $\mathcal{M}$ satisfies noninterference for a policy $\mathcal{P}$ if there exists a state invariant $\mathcal{I}$ and an observational equivalence relation $\approx$ for which the unwinding conditions in Figure 6 hold.

The unwinding theorem obviates the need to reason about traces to prove noninterference; instead, it suffices to show that the unwinding conditions hold for each action. This theorem enables Nickel to automate the checking using the Z3 SMT solver (see §4). Both the state invariant $\mathcal{I}$ and the observational equivalence relation $\approx$ are *untrusted*: any instances that satisfy the conditions are sufficient to establish noninterference.

We give some intuition behind the unwinding theorem. The first four conditions are natural: they ask for a reasonable state variant $\mathcal{I}$ and observational equivalence relation $\approx$ (i.e., $\overset{u}{\approx}$ should be an equivalence relation and be consistent with the policy). The remaining three conditions, *output consistency*, *local respect*, and *weak step consistency*, provide more hints to interface design, as follows. As a shorthand, we say "objects" to mean individual storage locations in the system state.

First, the output of an action should depend only on objects that the domain of the action can read. Restricting the output prevents an adversarial application from inferring information about system state via return values, such as the error-handling channel described in §2.

Second, if an action attempts to modify an object, the domain of the action should be able to write to that object, and its new value should depend only on the old value and objects that the domain of the action can read. This requirement prevents unintended flows while updating the system state, such as the resource-name channel introduced by `spawn` sequentially allocating identifiers.

Third, if an action attempts to create a new object, that new object should have equal or less authority than the domain of the action; similarly, if an object becomes newly readable after an action, then the domain of the action should have been able to read that object before the call. These restrictions preclude "runaway" authority—no action can arbitrarily increase the authority of its domain, or create an object more powerful than itself.

3.4 Refinement

Refinement is widely used for verifying systems: developers describe the intended system behavior as a high level, abstract specification and check that any behavior exhibited by a low level, concrete implementation is allowed by the specification. Refinement allows developers to reason about many properties of the system at the specification level, which is often simpler than reasoning about the implementation directly.

In our case, it would be ideal to prove noninterference (using the unwinding theorem) for an interface specification, and extend that guarantee to an implementation that refines the specification. However, it is well known that noninterference is generally *not* preserved under refinement [25, 52]; for example, the implementation may introduce extra stuttering steps that leak information. Nickel supports a restricted form of refinement over state machines and policies. We show here that this refinement preserves noninterference as defined in §3.2.

Let's consider the following systems:

- $\mathcal{M}_1 = \langle A, O, S_1, \texttt{init}_1, \texttt{step}_1, \texttt{output}_1 \rangle$, and
- $\mathcal{M}_2 = \langle A, O, S_2, \texttt{init}_2, \texttt{step}_2, \texttt{output}_2 \rangle$.

These two systems share the set of actions A and the set of outputs O, but differ in the state spaces, as well as the state-transition and output functions. One may consider $\mathcal{M}_1$ as the specification and $\mathcal{M}_2$ as the implementation. We say that $\mathcal{M}_2$ is a *data refinement* of $\mathcal{M}_1$ to mean that they produce the same output for any trace [33, 46]. Data refinement is particularly useful for verifying systems with a well-defined interface, such as OS kernels [41, 62].

A standard way to prove data refinement of $\mathcal{M}_1$ by $\mathcal{M}_2$ is to ask developers to identify a data refinement relation $\propto \subseteq (S_2 \times S_1)$; we write $s_2 \propto s_1$ to mean $(s_2, s_1) \in \propto$. Let $\mathcal{I}_2$ denote a state invariant for $\mathcal{M}_2$. To prove that $\mathcal{M}_2$ is a data refinement of $\mathcal{M}_1$, it suffices to show that the following *refinement conditions* hold:

- $\texttt{init}_2 \propto \texttt{init}_1$.
- $\mathcal{I}_2(s_2) \wedge s_2 \propto s_1 \Rightarrow \texttt{step}_2(s_2, a) \propto \texttt{step}_1(s_1, a)$.
- $\mathcal{I}_2(s_2) \wedge s_2 \propto s_1 \Rightarrow \texttt{output}_2(s_2, a) = \texttt{output}_1(s_1, a)$.

Each formula is universally quantified over s_1, s_2, and a.

Given policies $\mathcal{P}_1 = \langle D, \rightsquigarrow, \texttt{dom}_1 \rangle$ and $\mathcal{P}_2 = \langle D, \rightsquigarrow, \texttt{dom}_2 \rangle$ for systems $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively, we say that $\mathcal{P}_2$ is a *policy refinement* of $\mathcal{P}_1$ with respect to $\mathcal{M}_1$ and $\mathcal{M}_2$ if and only if the following holds for any action a and trace tr: $\texttt{dom}_1(a, \texttt{run}_1(\texttt{init}_1, tr)) = \texttt{dom}_2(a, \texttt{run}_2(\texttt{init}_2, tr))$. Here $\texttt{run}_1$ and $\texttt{run}_2$ apply a trace starting from a given state for $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively (§3.2).

With these notions of data refinement and policy refinement, we have proved the following refinement theorem for noninterference:

Theorem 2 (Refinement)**.** Given two systems $\mathcal{M}_1$ and $\mathcal{M}_2$ and policy $\mathcal{P}$ for $\mathcal{M}_1$, $\mathcal{M}_2$ satisfies noninterference for any policy refinement of $\mathcal{P}$ with respect to $\mathcal{M}_1$ and $\mathcal{M}_2$ if:

- there exists a state invariant $\mathcal{I}_1$ of system $\mathcal{M}_1$ and an observational equivalence relation $\approx$ for which the unwinding conditions of $\mathcal{M}_1$ for $\mathcal{P}$ hold; and
- there exists a state invariant $\mathcal{I}_2$ of system $\mathcal{M}_2$ and a data refinement relation $\propto$ for which the refinement conditions of $\mathcal{M}_1$ by $\mathcal{M}_2$ hold.

The refinement theorem enables Nickel to check noninterference for an implementation by checking the unwinding conditions for the interface specification and the refinement conditions (see §4). As with the unwinding theorem, the state invariants $\mathcal{I}_1$ and $\mathcal{I}_2$, the observational equivalence relation $\approx$, and the data refinement relation $\propto$ are *untrusted* for establishing noninterference.

3.5 Discussion and limitations

Nickel's formulation of noninterference falls into the category of *intransitive noninterference* [67]; in other words, it allows the can-flow-to relation of a policy to be either transitive or intransitive. As explained in §3.1, this flexibility is particularly useful for verifying practical systems, which often require downgrading operations. In addition, unlike classical noninterference, Nickel uses a state-dependent `dom` function, inspired by the formulation used to verify multiapplicative smart cards [68] and the seL4 kernel [57].

Nickel extends previous work in the following ways: the formulation supports a general set of policies and systems, which enables us to verify DIFC in NiStar (§6) and isolation in NiKOS and ARINC 653 (§7); all of its verification conditions for unwinding and refinement are expressible using an SMT solver, enabling automated verification to minimize the proof burden; and it provides a restricted form of refinement that preserves noninterference from an interface specification to an implementation.

Nickel's formulation of noninterference has the following limitations. It cannot uncover covert channels based on resources that are not captured in the interface specification, such as timing, sound, and energy. Modeling the effects of these resources is an orthogonal problem. Recent microarchitectural attacks [5, 42, 50] suggest the need for new hardware designs and primitives in order to eliminate such channels [21, 22].

Nickel does not support reasoning about concurrent systems. Concurrency is challenging not just for verification in general, but also for its implications on noninterference [71, 75]. In addition, Nickel models systems as deterministic state machines and requires developers to eliminate nondeterminism from the interface design (see §5). This requirement enables better proof automation and simplifies noninterference under refinement, but it restricts the types of interfaces that Nickel can verify [77].

Nickel's can-flow-to relation $\rightsquigarrow$ is state-independent, which means that Nickel cannot reason about dynamic, state-dependent policies [17] (though state-dependent `dom` functions partially compensate for this limitation). Moreover, Nickel's notion of refinement requires the interface specification and the implementation to use the same sets of actions and domains; this equality is sufficient for verifying systems like NiStar and NiKOS. Extending Nickel to support dynamic policies and more flexible refinements [76] would be useful future work.

4 Using Nickel

This section explains how the Nickel framework works and describes the steps needed to design and verify information flow control systems using Nickel.

Figure 7 depicts an overview of the Nickel framework and the required inputs from system developers (shaded

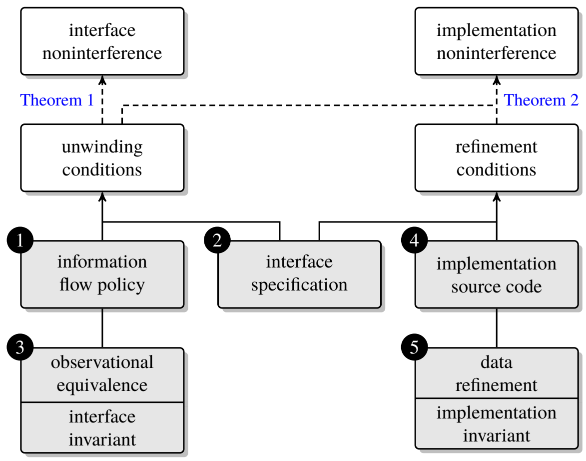

Figure 7: An overview of development flow using Nickel. Shaded boxes denote files written by system developers and the rest are provided by the framework. Circled numbers denote the steps. Solid and dashed arrows denote proof flows in SMT and Coq, respectively.

boxes with circled numbers). As part of the framework, the unwinding and refinement theorems (Theorem 1 and Theorem 2) serve as the metatheory for Nickel. We have formalized and proved both theorems using the Coq interactive theorem prover [74].

Developers write the system implementation in C and specify the rest of the inputs in Python. In particular, the development flow of using Nickel is the following:

1. Write the intended information flow policy to serve as the top-level specification of the system.
2. Model the system as a state machine and write a precise specification of each operation in the interface.
3. Construct a state invariant and observational equivalence for the interface specification, and invoke Nickel to check the unwinding conditions.
4. Implement each operation in the interface.
5. Construct a state invariant for the implementation and data refinement between the interface specification and the implementation, and invoke Nickel to check the refinement conditions.

Nickel extends the specification and verification infrastructure from Hyperkernel [62] to support reasoning about noninterference. It reduces all the inputs to SMT constraints—for instance, by performing symbolic execution on the LLVM intermediate representation of the implementation—and invokes Z3 to verify noninterference by checking the unwinding and refinement conditions. As with Hyperkernel, the initialization and glue code of the implementation is unverified. Interested readers can refer to Nelson et al. [62] for more information.

For verifying noninterference for an interface specification, the trusted computing base includes the information flow policy, the checker of unwinding conditions from Nickel, and Z3. For verifying noninterference for an implementation, it further includes the checker of refinement conditions from Nickel and the unverified initialization and glue code of the implementation.

Below we highlight two features of the development flow using Nickel.

A simple API for specifying the policy. As described in §3.1, a policy consists of a set of domains, a can-flow-to relation over domains, and a dom function associating each action in a state with a domain. Nickel provides a simple and intuitive API for specifying policies.

As an example, recall the isolation policy in Figure 3: each process p_i is a domain; the permitted flows in the system are: $p_0 \rightsquigarrow p_i$, $p_i \rightsquigarrow p_0$, and $p_i \rightsquigarrow p_i$ for $i \in [0, n-1]$. In Nickel, this policy is written as follows:

```
class ProcessDomain:
    def __init__(self, pid):
        self.pid = pid

    def can_flow_to(self, other):
        # Or is a built-in logical operator
        return Or(
            self.pid == 0,          # p0 ~> pi
            other.pid == 0,         # pi ~> p0
            self.pid == other.pid, # pi ~> pi
        )
```

In addition, the dom function of this policy returns the process currently running by default, or the scheduler p_0 for context switching actions (say, the yield system call):

```
class State:
    current = PidT() # PidT is an integer type
    ...

def dom(action, state):
    if action.name == 'yield':
        return ProcessDomain(0)
    else:
        return ProcessDomain(state.current)
```

This is all Nickel needs for the policy of NiKOS (§7).

Since a policy is the top-level specification of a system and must be trusted, developers should carefully audit the policy and ensure that it captures the design intention. We hope that the simple API for policies provided by Nickel makes auditing easier.

Debugging through counterexamples. To verify noninterference for an interface specification, Nickel checks the unwinding conditions from Theorem 1. If verification fails, Nickel produces a counterexample that illustrates the violation, including the operation name, an assignment of the operation arguments and system state(s), and the offending unwinding conditions.

Counterexamples provide useful information for debugging two types of failures. First, the violation may be in the interface specification, indicating a covert channel. Developers can use the counterexample to understand the violation and iterate on the interface design (see §5 for guidelines) until verification passes. Second, the state invariant or the observational equivalence may be insufficient to establish noninterference. Developers can consult the counterexample to fix these inputs. Debugging the verification of an implementation follows similar steps.

5 Designing interfaces for noninterference

We have applied Nickel to verify noninterference in three systems: NiStar (§6), NiKOS (§7), and ARINC 653 (§7). While they have different information flow policies, our experience with these systems suggests several common guidelines for interface design.

Perform flow checks early. In general, operations need to validate parameters, especially those from untrusted sources (e.g., user-specified values in system calls), and return error codes indicating the cause of failure. As described in §2, returning error codes requires care to avoid covert channels. One simple way to avoid such channels is to use fewer error codes (or drop error codes altogether), but doing so makes debugging applications difficult.

NiStar addresses this issue by performing flow checks as early as possible. For example, many system calls need to check whether the current thread has permission to access specified data. After such a flow check succeeds, the system call has more liberty to validate parameters and return more specific error codes without violating noninterference.

Limit resource usage with quotas. Shared resources can lead to covert channels due to resource exhaustion. Systems may impose a quota on shared resources for each domain to avoid such channels. There are several quota schemes. One simple scheme is to statically assign predetermined quotas to domains; for instance, allowing processes to allocate only a predetermined number of identifiers for child processes [10]. However, this scheme limits the functionality of the system if the quota is too low, and wastes resources if the quota is set too high.

A more flexible and explicit quota scheme is to organize resources into a hierarchy of *containers* [4, 69, 82], where each container has a quota for resources such as memory and CPU time. A thread can allocate objects from a container, including creating subcontainers, if the container has sufficient quota and the policy allows the thread to access the container. A thread can also transfer quotas between two containers if the policy allows the thread to access both containers. NiStar uses containers to manage resources.

Partition names among domains. Resource names in a shared namespace, such as thread identifiers and page numbers, can lead to covert channels. A per-domain naming scheme partitions names among domains to eliminate such channels. A classical example is using ⟨process identifier, virtual page number⟩ pairs to refer to memory pages, effectively partitioning page numbers among processes. As another example, a system with container-based resource management may use ⟨container identifier, resource identifier⟩ pairs to refer to resources [82]; a thread may access the resource only if the policy permits it to access the container. Both NiStar and NiKOS employ per-domain naming schemes.

Encrypt names from a large space. Using encrypted names is an alternative way to address covert channels due to resource names. Many DIFC systems allocate sequential identifiers for resources, but return *encrypted* values to make them unpredictable [15, 45, 82]. This design technically violates noninterference, but since the identifier space is sufficiently large (e.g., 64 bits), the amount of information that can be leaked through this channel is negligible in practice. However, verifying noninterference for this design would require probabilistic reasoning [44] and complicate the semantics of noninterference [17: §6.4]. We therefore do not use encrypted names for the systems verified using Nickel.

Expose or enclose nondeterminism. As mentioned in §3.5, Nickel does not allow nondeterministic behavior in the interface specification (for instance, a system call that allocates an unspecified physical page), since doing so would complicate refinement for noninterference.

There are several options for revising the semantics of such system calls to eliminate nondeterminism. The first option is to make the (nondeterministic) decision explicit as a system call parameter, for example, asking user space to decide which page to allocate, similarly to exokernels [18, 37, 62]. The second option is to ask developers to explicitly describe the behavior (e.g., the allocation algorithm) as part of the interface specification. This makes the interface specification less abstract but simplifies the verification of noninterference under refinement; NiStar uses this option for memory management. The third option is to enclose the source of nondeterminism below the interface [28], for example, using virtual addresses to refer to memory pages and removing the use of physical pages from the interface. NiKOS uses this option.

Reduce flows to the scheduler. An OS scheduler is generally associated with a powerful domain, such as in Figure 3. The scheduler decides and updates which process to run, and other domains usually need to access this information (e.g., to look up the process currently running), creating inherent flows from the scheduler to other domains. Many scheduling approaches access information about processes to make scheduling decisions, creating flows from other domains to the scheduler. The combination of these flows makes the scheduler a powerful domain that two processes might exploit to communicate.

One way to control this risk is to enforce a stricter policy that prohibits flows *to* the scheduler. This policy restricts the power of the scheduler, since it can no longer query state that belongs to other domains. One simple design that satisfies this policy is to use a static, predetermined schedule [1, 57] that does not need to query the system state for scheduling decisions. NiStar instead satisfies this policy with a more flexible design: like exokernels [18, 37], it allows applications to allocate time slices to implement dynamic scheduling policies. Unlike exokernels, NiStar performs flow checks at run time to prevent these allocations introducing covert channels (see §6.2).

6 DIFC in NiStar

NiStar is a new OS kernel that supports decentralized information flow control (DIFC). NiStar's design is inspired by HiStar [82]: the kernel tracks information flow using labels and enforces DIFC through seven object types, and a user-space library implements POSIX abstractions on top of these kernel object types. Unlike HiStar, however, we have formalized NiStar's information flow policy and verified that both its interface specification and implementation satisfy noninterference for this policy. This section describes how we designed the NiStar interface to eliminate covert channels and used Nickel to achieve automated verification.

6.1 Labels

Like other DIFC systems [23, 45, 65], NiStar uses tags and labels to track information flow across the system. It follows a scheme used in DStar [83] and a revised version of HiStar [84]. A tag is an opaque integer, which has no inherent meaning. For instance, Alice uses tags t_S and t_I to represent the secrecy and integrity of her data, respectively. A label is a set of tags. Every object in the system is associated with a triple of ⟨secrecy, integrity, ownership⟩ labels, which we designate as the domain of the object. For instance, Alice labels her files with $\langle \{t_S\}, \{t_I\}, \varnothing \rangle$.

We use Figure 8 as an example to illustrate how Alice can constrain untrusted applications using labels. Suppose Alice launches a spellchecker to scan her files; the spellchecker consults a shared dictionary and prints the results (misspelled words) to her terminal. An updater periodically queries a server through the `netd` daemon and keeps the dictionary up to date. Alice trusts her `ttyd` daemon to declassify data only to her terminal. She trusts neither the spellchecker nor the updater, which may each be buggy, compromised, or malicious. Alice hopes to achieve the following security goals: (1) neither the spellchecker nor the updater can modify her files; and (2) her spellchecked files can not be leaked to the network.

Classical information flow control expresses policies using only secrecy and integrity labels (i.e., ignoring ownership). Given two objects with domains $L_1 = \langle S_1, I_1, O_1 \rangle$ and $L_2 = \langle S_2, I_2, O_2 \rangle$, respectively, it is safe in the classical model for information to flow from L_1 to L_2 if (1) the secrecy of S_1 is subsumed by that of S_2 and (2) the integrity of I_1 subsumes that of I_2: $S_1 \subseteq S_2 \wedge I_2 \subseteq I_1$. In other words, a flow is safe if it neither discloses secrets nor compromises the integrity of any object. For example, given the label assignment in Figure 8 and a system enforcing such flow checks, Alice can conclude that her files will not be modified by the spellchecker or the updater: her files have t_I in their integrity labels, but the spellchecker and updater do not, ruling out flows from them to her files.

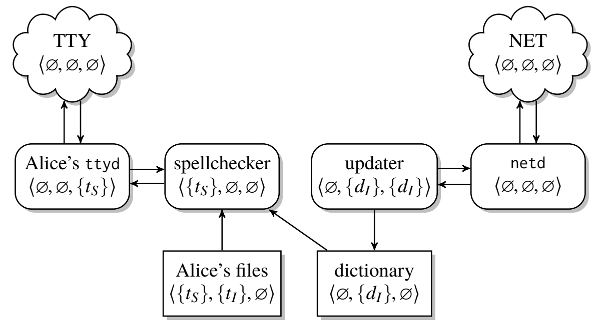

Figure 8: Information flow of a spellchecker and updater. Cloud boxes represent terminal (TTY) and network (NET); rounded boxes represent threads; and rectangular boxes represent data. Each object is associated with a triple of ⟨secrecy, integrity, ownership⟩ labels; arrows denote the flows of information allowed by these labels.

The classical model is often too restrictive for practical systems. For instance, a password checker needs to declassify whether login succeeds to untrusted users; as another example, to output misspelled words in Figure 8, the spellchecker (with t_S in secrecy) needs to communicate with to Alice's trusted `ttyd` (without t_S). Like other DIFC systems, NiStar supports such intentional downgrading without a centralized authority. It uses the ownership label to relax label checking for trusted threads, giving them the privilege to temporarily remove tags from secrecy labels (declassification) or add tags to integrity labels (endorsement), as follows:

Definition 3 (Safe Flow)**.** Information can flow from $L_1 = \langle S_1, I_1, O_1 \rangle$ to $L_2 = \langle S_2, I_2, O_2 \rangle$, denoted as $L_1 \rightsquigarrow L_2$, if and only if $(S_1 - O_1 \subseteq S_2 \cup O_2) \wedge (I_2 - O_2 \subseteq I_1 \cup O_1)$.

This can-flow-to relation is central to NiStar's information flow policy. $L_1 \rightsquigarrow L_2$ means that L_1 and L_2 can combine their ownership to allow the maximum flow from L_1 to L_2; that is, L_1 lowers its secrecy to $S_1 - O_1$ and raises its integrity to $I_1 \cup O_1$, while L_2 raises its secrecy to $S_2 \cup O_2$ and lowers its integrity to $I_2 - O_2$.

Referring to Figure 8, as information can flow from the spellchecker to Alice's `ttyd` given their label assignments, $\langle \{t_S\}, \varnothing, \varnothing \rangle \rightsquigarrow \langle \varnothing, \varnothing, \{t_S\} \rangle$, Alice's `ttyd` is able to print out misspelled words. In addition, Alice can conclude that her files will not be leaked to the network: the spellchecker cannot directly leak information to the network given its label assignment. The spellchecker can, however, indirectly write to Alice's terminal only through her `ttyd`, which she trusts to declassify data only to the terminal; no other threads in the system are trusted. This example shows how labels can minimize the amount of application code that must be trusted.

6.2 Kernel objects

NiStar provides seven object types:

- *labels* represent domains of objects;
- *containers* are basic units for managing resources;
- *threads* are basic execution units;
- *gates* provide protected control transfer;
- *page-table pages* organize virtual memory;
- *user pages* represent application data; and
- *quanta* represent time slices for scheduling.

Each object, other than labels, is associated with a domain of ⟨secrecy, integrity, ownership⟩ labels; only threads and gates can have non-empty ownership labels. The kernel interface consists of a total of 46 operations for manipulating these objects. Each operation performs flow checks among objects using their labels. NiStar's design goal is to ensure that the interface specification satisfies noninterference for the policy given by Definition 3.

NiStar largely follows HiStar's object types [82], with the following exceptions: it provides a new object type, quantum, for scheduling; and to make the interface finite and therefore amenable to automated verification, it uses fixed-sized page-table pages and user pages similar to Hyperkernel [62] and seL4 [40]. Interested readers can refer to Zeldovich et al. [84] for details of object types and label checks; below, we highlight three key differences in NiStar that close covert channels.

Given $L_1 = \langle S_1, I_1, O_1 \rangle$ and $L_2 = \langle S_2, I_2, O_2 \rangle$, we introduce the following notations for flow checks:

- $L_1 \sqsubseteq_{\mathsf{R}} L_2$ means that L_1 *can be read by* L_2: $(S_1 \subseteq S_2 \cup O_2) \wedge (I_2 - O_2 \subseteq I_1)$.
- $L_1 \sqsubseteq_{\mathsf{W}} L_2$ means that L_1 *can write to* L_2: $(S_1 - O_1 \subseteq S_2) \wedge (I_2 \subseteq I_1 \cup O_1)$.

As a shorthand, we write $L_2 \sqsubseteq_{\mathsf{R}} L_1 \sqsubseteq_{\mathsf{W}} L_2$ to mean that L_1 *can modify* L_2: $(L_2 \sqsubseteq_{\mathsf{R}} L_1) \wedge (L_1 \sqsubseteq_{\mathsf{W}} L_2)$. It is generally difficult for L_1 to modify L_2 without receiving any information in return (e.g., error code), and so this definition includes L_1 being able to read L_2. By definition, $L_1 \sqsubseteq_{\mathsf{W}} L_3$ and $L_3 \sqsubseteq_{\mathsf{R}} L_2$ together imply $L_1 \rightsquigarrow L_2$ for any L_1, L_2, and L_3; we will use this fact below to analyze covert channels. We denote $\mathcal{L}_x$ as the domain of object x.

Maintain accurate quotas in containers. Like HiStar, NiStar manages all system resources in a hierarchy of containers, starting from a root container created during kernel initialization. Each container maintains a set of quotas, indicating the amount of memory pages and time quanta it owns. A thread T may allocate an object O from a container C only if it can modify the container (i.e., $\mathcal{L}_C \sqsubseteq_{\mathsf{R}} \mathcal{L}_T \sqsubseteq_{\mathsf{W}} \mathcal{L}_C$), the new object does not exceed the authority of the thread (i.e., $\mathcal{L}_T \sqsubseteq_{\mathsf{W}} \mathcal{L}_O$), and the container has sufficient quota for the object.

NiStar maintains accurate quotas in containers, which differs from HiStar in two ways. First, NiStar sets the memory quota of the root container to be number of available physical pages upon booting, rather than infinity [82: §3.3], avoiding a potential covert channel due to resource exhaustion. Second, NiStar does not allow an object to be linked by multiple containers, which would require the kernel to conservatively charge each container as in HiStar. Instead, each object is uniquely owned by one container. This design leads to a simpler invariant: for each resource type, the sum of the quotas of each object in a container equals the total quota of the container.

Enforce can-write-to-object on deallocation. In HiStar, to deallocate an object O from a container C, a thread T must be able to write to the container, but not necessarily to the object itself. This relaxed check supports reclaiming *zombie* objects to which no one else can write (e.g., those with a unique integrity tag) [81]. However, it leads to a covert channel. Consider a thread T' whose domain permits it to read object O (i.e., $\mathcal{L}_O \sqsubseteq_{\mathsf{R}} \mathcal{L}_{T'}$) but prohibits it from receiving information from thread T (i.e., $\mathcal{L}_T \not\rightsquigarrow \mathcal{L}_{T'}$). To bypass DIFC, thread T encodes a one-bit secret by either deallocating object O from container C or not. T' learns the secret by observing whether object O still exists [82: §3.2], violating noninterference since the label assignment prohibits information flow from T to T'.

NiStar enforces a stricter flow check on deallocation by requiring that thread T can write to object O (i.e., $\mathcal{L}_T \sqsubseteq_{\mathsf{W}} \mathcal{L}_O$). With this stricter check, this covert channel is closed: if thread T' can read object O (i.e., $\mathcal{L}_O \sqsubseteq_{\mathsf{R}} \mathcal{L}_{T'}$), the new check implies that thread T' is permitted to receive information from thread T, since $\mathcal{L}_T \sqsubseteq_{\mathsf{W}} \mathcal{L}_O$ and $\mathcal{L}_O \sqsubseteq_{\mathsf{R}} \mathcal{L}_{T'}$ together imply $\mathcal{L}_T \rightsquigarrow \mathcal{L}_{T'}$.

NiStar considers reclaiming zombie objects an administrative decision and leaves it to user space. Some systems may consider it legitimate for a user to create objects that no one else can reclaim; since NiStar enforces accurate quotas, adversarial users cannot create "runaway" zombie objects that exceed their quotas. On the other hand, a system wishing to reclaim zombie objects can emulate the HiStar behavior by setting up a trusted garbage collector with a powerful domain during booting, without baking this requirement into flow checks in the kernel.

Remove flows to the scheduler using quanta. As noted in §5, two processes can exploit the scheduler to communicate in violation of information flow policy. To close this channel, NiStar borrows the design of the exokernel scheduler [18] and extends it with label checking. NiStar associates the scheduler with domain $\langle \varnothing, \mathbb{U}, \varnothing \rangle$, where $\mathbb{U}$

denotes the universal label of all tags. This domain allows the scheduler to switch to any thread (its universal integrity allows it to influence any thread it runs) while restricting it from leaking information (its empty secrecy and ownership prevent it receiving secrets). The resulting scheduler allows applications to implement more flexible scheduling schemes compared to static scheduling.

NiStar introduces *time quanta* to allow the scheduler to make decisions while respecting this label assignment. The system is configured with a fixed number of quanta, each associated with a thread identifier for scheduling. Like other resources, all quanta are initially owned by the root container; a thread can move quanta between two containers only if it can modify both containers. To schedule thread T' at quantum Q, thread T writes the identifier of T' to Q. Thread T can perform this write only if it can write to quantum Q (i.e., $\mathcal{L}_T \sqsubseteq_{\mathsf{W}} \mathcal{L}_Q$).

To schedule using time quanta, assume that the system delivers an infinite stream of timer interrupts. Upon the arrival of a timer interrupt, the scheduler cycles through all the quanta in a round-robin fashion and retrieves the thread identifier T' associated with the next quantum Q. If quantum Q can be read by thread T' (i.e., $\mathcal{L}_Q \sqsubseteq_{\mathsf{R}} \mathcal{L}_{T'}$), the scheduler switches to T'; otherwise, it idles.

To see why these flow checks suffice to close the channel, suppose T is able to schedule T' to execute at quantum Q. The checks ensure $\mathcal{L}_T \sqsubseteq_{\mathsf{W}} \mathcal{L}_Q$ and $\mathcal{L}_Q \sqsubseteq_{\mathsf{R}} \mathcal{L}_{T'}$, which together imply $\mathcal{L}_T \rightsquigarrow \mathcal{L}_{T'}$; in other words, the label assignment permits T to communicate with T'.

This design closes covert channels arising from logical time. As mentioned in §3.5, physical timing is beyond the scope of this paper, for which NiStar provides no guarantees of noninterference.

6.3 Implementation

To demonstrate that NiStar's interface is practical, we have built a prototype implementation for x86-64 processors, and have applied Nickel to verify that both the interface specification and the implementation satisfy noninterference for the policy given by Definition 3.

To simplify verification, NiStar borrows ideas from previous verified OS kernels. First, like Hyperkernel [62], NiStar uses separate page tables for the kernel and user space. It uses an identity mapping for the kernel address space, sidestepping the complication of reasoning about virtual memory for kernel code [43]. Second, like seL4 [40], NiStar enables timer interrupts only in user space and disables them in the kernel. This restriction ensures that the execution of system calls and exception handling is atomic, avoiding reasoning about interleaved executions. Third, NiStar disables all other interrupts and requires device drivers to use polling, a common practice in high-assurance systems [1, 57].

For user space, we have ported the *musl* C standard library [59] to NiStar, running on top of an emulation layer for Linux system calls. A library implements the abstraction of Unix-like processes on top of NiStar's kernel object types, similar to HiStar's emulation layer [82]. The file-system service is implemented as a thin wrapper over containers and user pages, and the network service is provided by lwIP [13]. Although our current user space implementation is incomplete, it is able to run programs such as a set of POSIX utilities from Toybox, a web server, and the TinyEMU emulator to boot Linux.

7 Verifying isolation

Nickel generalizes to information flow control systems beyond DIFC. This section describes applying Nickel to two such systems: NiKOS and ARINC 653.

Process isolation. NiKOS is a small OS that enforces an isolation policy among processes (Figure 3). The interface of NiKOS mirrors that of a version of mCertiKOS as described by Costanzo et al. [10]. It consists of seven operations, including spawning a process, querying process status, printing to console, yielding, and handling a page fault. Like mCertiKOS, NiKOS imposes a memory quota on each process and statically partitions identifiers among processes, avoiding covert channels due to resource names and exhaustion (§5). We implemented a prototype of NiKOS for x86-64 processors and ported user-space applications from mCertiKOS. We used Nickel to verify that both the interface and implementation satisfy noninterference for the isolation policy. This effort took one author a total of two weeks.

We made one change to the design in order to verify noninterference. In mCertiKOS, the `spawn` system call creates a new process and loads an executable file; the specification of `spawn` models file loading as a no-op, whereas the implementation allocates pages and consumes memory quota [26]. In NiKOS, to match the memory quota in the specification with that in the implementation, `spawn` creates an empty address space and the page-fault handler lazily loads each page of the executable file instead.

Partition isolation. ARINC 653 [1] is an industrial standard for safety-critical avionics operating systems. It models the system as a set of *partitions* and defines an inter-partition communication interface comprising 14 operations. Figure 9 depicts its isolation policy among partitions: information can flow to a partition only from the *transmitter*, the scheduler, and itself. The transmitter forwards messages among partitions as configured at boot time; each dashed arrow represents a flow that can be independently enabled in the configuration. The scheduler uses a pre-configured fixed schedule, and so does not require flows from other domains to the scheduler (§5).

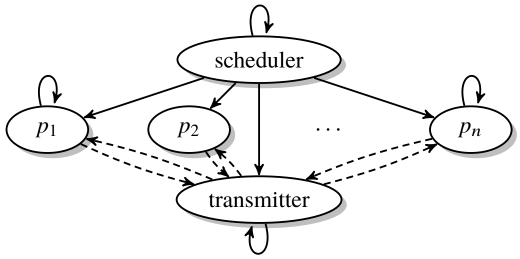

Figure 9: The isolation policy of ARINC 653: information can flow between the transmitter and each partition p_i for $i \in [1, n]$ as per a boot-time configuration (dashed arrows); it cannot flow between any two partitions, or from any partition or the transmitter to the scheduler.

Using Nickel, we formalized the specification of the communication interface based on the pseudocode provided by the ARINC 653 standard. Applying Nickel to verify noninterference for the partition isolation policy reproduced all three known covert channels first discovered by Zhao et al. [86], which were caused by missing partition permission checks, allocating identifiers in a shared namespace, and returning error codes that leak information; verification succeeded once we fixed these channels. This effort took one author a total of one week.

8 Experience

This section reports our experience with using Nickel and reflects lesson learned during development. Experiments ran on an Intel Core i7-7700K CPU at 4.5 GHz.

Covert channel discussion. To test the effectiveness of Nickel for detecting covert channels, we injected each of the examples in §2 into the NiStar interface specification. In each case, Nickel was able to find a counterexample pointing to the issue. As a concrete example, we switched NiStar's scheduler to a round-robin one. When verifying this round-robin scheduler, Nickel failed and produced a counterexample (§4).

Figure 10 shows empirical evidence of a covert channel by comparing the NiStar scheduler with the round-robin one. In this experiment, one process sampled the current (logical) time, while a background process repeatedly forked and then killed 30 child processes. The measuring process recorded the duration between scheduling points in terms of number of quanta. With the round-robin scheduler, the gaps observed by the measuring process vary as the background task forks and kills its children, creating patterns that indicate the covert channel. With the NiStar scheduler, which is verified using Nickel, the gaps between scheduling points remain constant regardless of the behavior of the background process. This result suggests that the Nickel is effective in identifying and proving the absence of covert channels.

Development effort using Nickel. Figure 11 shows the sizes of the three systems we verified using Nickel:

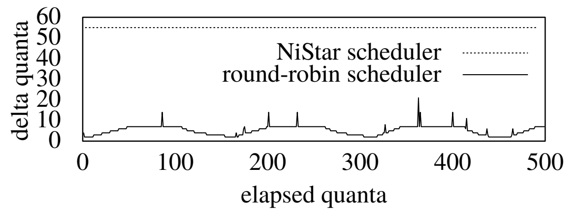

Figure 10: A round-robin scheduler leaks background thread behavior through patterns in logical time; no such pattern is observed in NiStar.

component	NiStar	NiKOS	ARINC 653
specification:			
information flow policy	26	14	33
interface specification	714	82	240
proof input:			
interface invariant	398	63	66
observational equivalence	127	56	80
implementation invariant	52	7	–
data refinement	139	30	–
implementation:			
interface implementation	3,155	343	–
user space implementation	9,348	389	–
common kernel infrastructure	4,829 (shared by NiStar/NiKOS)		

Figure 11: Lines of code for the three systems verified using Nickel.

NiStar, NiKOS, and ARINC 653. The lines of code for the interface implementations of both NiStar and NiKOS do not include common kernel infrastructure (C library functions and x86 initialization), and those of the user space implementations do not include third-party libraries (e.g., musl and lwIP). The implementation of the Nickel framework is split between the formalization of the metatheory (1,215 lines of Coq) and the verifier for the unwinding and refinement conditions (3,564 lines of C++ and Python).

The information flow policies for the three systems are concise compared to the rest of the specification and implementation, indicating the simplicity of creating policies ranging from DIFC to isolation using Nickel (§4).

In our experience, the most time-consuming part of the verification process was coming up with an appropriate observational equivalence relation—it was non-trivial to determine which part of the system state was observable by each domain, and the complexity increased as the size of the system state and the number of interface operations grew. We found the counterexamples produced by Nickel particularly useful for debugging and fixing observational equivalence. The specification and verification of NiStar, NiKOS, and ARINC 653 took one author six weeks, two weeks, and one week, respectively; as a comparison, implementing NiStar took several researchers roughly six months. This comparison shows that the proof effort required when using Nickel is low, thanks to its support for automated verification and counterexample generation.

Using Z3 4.6.0, verifying NiStar, NiKOS, and ARINC 653 on four cores took 72 minutes, 7 seconds, and 8 seconds, respectively.

Lessons learned. Our development of Nickel was guided by two motives. First, in our previous work on Hyperkernel [62], we proved memory isolation among processes, but this did not preclude covert channels through system calls; Nickel extends push-button verification to support proving stronger guarantees about noninterference. Second, we aimed to develop a general framework that can help analyze and design interfaces not only for isolation, but also for mechanisms as flexible as DIFC.

While designing Nickel, we spent a total of two months iterating through several formulations of noninterference before settling on the one described in §3. Among these alternatives were classical transitive noninterference [29] and intransitive noninterference [67], as well as variants such as nonleakage [56, 77]. As discussed in §3.5, Nickel's formulation has the advantage of supporting both a spectrum of policies and automated verification.

As Figure 7 shows, Nickel combines both automated and interactive theorem provers: Z3 automates proofs for individual systems, while the proofs in Coq improve confidence in Nickel's metatheory. Similar approaches have been used for the verification of compiler optimizations [72], static bug checkers [79], and Amazon's s2n TLS library [9]. We believe that this combination is an effective approach to developing verified systems.

9 Related work

Verifying noninterference in systems. Noninterference is a desirable security definition for operating systems looking to guarantee information flow properties [66]. For example, the seL4 microkernel [40] is proven to satisfy a variant of noninterference for a given access control policy [56, 57]; a version of mCertiKOS [27] includes a proof of process isolation [10]; Ironclad [32] proves end-to-end guarantees for applications using a form of input and output noninterference; and Komodo [19] proves noninterference for isolated execution of software-based enclaves. Noting the difficulty of extending noninterference proofs to concurrent systems, Covern [58] provides a logic for the shared memory setting. Noninterference also has applications in secure hardware [20, 21], programming languages [49, 73], as well as browsers and servers [36, 64]. Nickel takes inspiration from these efforts, focusing on formalizations and interface designs that are amenable to automated verification of noninterference.

DIFC operating systems. Information flow control was originally envisioned as a mechanism to enforce multi-level security in military systems [2, 3]. *Decentralized* information flow control (DIFC) additionally allows applications to declare new classifications [60, 61]. The design of NiStar was influenced by prior DIFC operating systems [7, 15, 45, 65, 82], particularly HiStar and Flume.

HiStar [82, 84] enforces DIFC with a small number of types of kernel objects. All label changes in HiStar are explicit, closing the covert channel in Asbestos due to implicit label changes [15]. NiStar's design draws from HiStar, using a similar set of kernel object types, but adapted to close remaining covert channels and enable automated verification.

Flume [45] is a DIFC system built on top of the Linux kernel. Building on top of an existing kernel makes porting easier, but expands Flume's TCB. Flume's design has a pen-and-paper proof [44] of noninterference for a single label assignment, modeled using Communicating Sequential Processes [34]; a more general formalization of Flume is given by Eggert [16]. NiStar takes this effort a step further, with the first noninterference proof of both the interface and implementation of a DIFC OS kernel.

Reasoning about information flows for applications. Assigning DIFC labels for applications is a non-trivial task. To help application developers, Asbestos offers a domain-specific language [14] for generating label assignments from high-level specifications. The SWIM tool [30] generates label assignments from lists of prohibited and allowed flows, and has been further extended using synthesis techniques [31]. These tools can benefit from a precise specification of the DIFC framework they use to implement policies for, such as the one provided by NiStar.

10 Conclusion

Nickel is a framework for designing and verifying information flow control systems through automated verification techniques. It focuses on helping developers eliminate covert channels from interface designs and provides a new formulation of noninterference to uncover covert channels or prove their absence using an SMT solver. We have applied Nickel to develop three systems, including NiStar, the first formally verified DIFC OS kernel. Our experience shows that the proof burden of using Nickel is low. We believe that Nickel offers a promising approach to the design and implementation of secure systems. All of Nickel's source code is publicly available at https://unsat.cs.washington.edu/projects/nickel/.

Acknowledgments

We thank Jon Howell, Kevin Zhao, the anonymous reviewers, and our shepherd, Raluca Ada Popa, for their feedback. We thank Nickolai Zeldovich and Ronghui Gu for answering our questions on HiStar and mCertiKOS, respectively. This work was supported in part by DARPA under contract FA8750-16-2-0032, by NSF under grant CCF-1651225, and by a Google Faculty Award.

References

[1] ARINC 653. Avionics application software standard interface: Part 1, required services, August 2015.

[2] David E. Bell and Leonard La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corp., Bedford, MA, March 1976.

[3] Kenneth J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corp., Bedford, MA, April 1977.

[4] Alan C. Bomberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, April 1992.

[5] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*, pages 991–1008, Baltimore, MD, August 2018.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, December 2008.

[7] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in Aeolus. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, June 2012.

[8] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 73–88, Chateau Lake Louise, Banff, Canada, October 2001.

[9] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. Continuous formal verification of Amazon s2n. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*, pages 430–446, Oxford, United Kingdom, July 2018.

[10] 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)*, pages 648–664, Santa Barbara, CA, June 2016.

[11] Leonardo de Moura and Nikolaj 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, March–April 2008.

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

[13] Adam Dunkels. Design and implementation of the lwIP TCP/IP stack. Swedish Institute of Computer Science, February 2001.

[14] Petros Efstathopoulos and Eddie Kohler. Manageable fine-grained information flow. In *Proceedings of the 3rd ACM EuroSys Conference*, pages 301–313, Glasgow, Scotland, April 2008.

[15] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, M. Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 17–30, Brighton, United Kingdom, October 2005.

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

[17] Sebastian Eggert and Ron van der Meyden. Dynamic intransitive noninterference revisited. *Formal Aspects of Computing*, 29(6):1087–1120, June 2017.

[18] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Copper Mountain, CO, December 1995.

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

[20] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Verification of a practical hardware security architecture through static information flow analysis. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 555–568, Xi'an, China, April 2017.

[21] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, October 2018.

[22] Qian Ge, Yuval Yarom, and Gernot Heiser. No security without time protection: We need a new hardware-software contract. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, Jeju Island, South Korea, August 2018.

[23] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, October 2012.

[24] 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, April 1982.

[25] John Graham-Cumming and J. W. Sanders. On the refinement of non-interference. In *Proceedings of the Computer Security Foundations Workshop VI*, pages 35–42, Franconia, NH, June 1991.

[26] Ronghui Gu. Private communication, August 2018.

[27] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 653–669, Savannah, GA, November 2016.

[28] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 193–208, San Diego, CA, December 2008.

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

[30] William R. Harris, Somesh Jha, and Thomas Reps. DIFC programs by automatic instrumentation. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 284–296, Chicago, IL, October 2010.

[31] William R. Harris, Somesh Jha, Thomas Reps, and Sanjit A. Seshia. Program synthesis for interactive-security systems. *Formal Methods in System Design*, 51(2):362–394, November 2017.

[32] 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 *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, Broomfield, CO, October 2014.

[33] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, December 1972.

[34] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[35] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pages 143–157, San Francisco, CA, May 2012.

[36] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21st USENIX Security Symposium*, pages 113–128, Bellevue, WA, August 2012.

[37] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In

Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP), pages 52–65, Saint-Malo, France, October 1997.

[38] Richard A. Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *ACM Transactions on Computer Systems*, 1(3):256–277, August 1983.

[39] Richard A. Kemmerer. A practical approach to identifying storage and timing channels: Twenty years later. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC)*, pages 109–118, Las Vegas, NV, December 2002.

[40] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, October 2009.

[41] Gerwin Klein, Thomas Sewell, and Simon Winwood. Refinement in the formal verification of the seL4 microkernel. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 323–339. Springer, January 2010.

[42] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2019.

[43] Rafal Kolanski. *Verification of Programs in Virtual Memory Using Separation Logic*. PhD thesis, University of New South Wales, July 2011.

[44] Maxwell Krohn and Eran Tromer. Noninterference for a practical DIFC-based operating system. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, pages 61–76, Oakland, CA, May 2009.

[45] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 321–334, Stevenson, WA, October 2007.

[46] Leslie Lamport. Computation and state machines, April 2008.

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

[48] Butler W. Lampson. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, pages 33–48, Bretton Woods, NH, October 1983.

[49] Peng Li and Steve 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, January 2005.

[50] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium*, pages 973–990, Baltimore, MD, August 2018.

[51] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. *ACM Transactions on Storage*, 10 (1):31–44, January 2014.

[52] Heiko Mantel. Preserving information flow properties under refinement. In *Proceedings of the 22nd IEEE Symposium on Security and Privacy*, pages 78–91, Oakland, CA, May 2001.

[53] John McLean. Security models and information flow. In *Proceedings of the 11th IEEE Symposium on Security and Privacy*, pages 180–187, Oakland, CA, May 1990.

[54] MITRE. CWE-209: Information exposure through an error message. https://cwe.mitre.org/data/definitions/209.html, January 2018.

[55] Bodo Möller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures. https://www.openssl.org/~bodo/tls-cbc.txt, September 2014.

[56] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In *Proceedings of the 2nd International Conference on Certified Programs and Proofs (CPP)*, pages 126–142, Kyoto, Japan, December 2012.

[57] 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 *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013.

[58] Toby Murray, Robert Sison, and Kai 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, April 2018.

[59] musl. https://www.musl-libc.org/, 2018.

[60] 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.

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

[62] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 252–269, Shanghai, China, October 2017.

[63] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. Faults in Linux: Ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–318, Newport Beach, CA, March 2011.

[64] Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, and Sorin 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.

[65] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[66] John 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, December 1981.

[67] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, December 1992.

[68] Gerhard Schellhorn, Wolfgang Reif, Axel Schairer, Paul Karger Vernon Austel, and David Toll. Verification of a formal security model for multiapplicative smart cards. In *Proceedings of the 6th European Symposium on Research in Computer Security*, pages 17–36, Toulouse, France, October 2000.

[69] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 170–185, Kiawah Island, SC, December 1999.

[70] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, November 2016.

[71] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David 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, September 2012.

[72] Zachary Tatlock and Sorin Lerner. Bringing extensibility to verified compilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 111–121, Toronto, Canada, June 2010.

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

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

[75] Ta-chung Tsai, Alejandro Russo, and John Hughes. A library for secure multi-threaded information flow

in Haskell. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 187–202, Venice, Italy, July 2007.

[76] Ron van der Meyden. Architectural refinement and notions of intransitive noninterference. *Formal Aspects of Computing*, 24(4–6):769–792, July 2012.

[77] David 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, September 2004.

[78] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 260–275, Farmington, PA, November 2013.

[79] Konstantin Weitz, Steven Lyubomirsky, Stefan Heule, Emina Torlak, Michael D. Ernst, and Zachary Tatlock. SpaceSearch: A library for building and verifying solver-aided tools. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Oxford, United Kingdom, September 2017.

[80] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations*, Cape Breton, Canada, June 2001.

[81] Nickolai Zeldovich. Private communication, April 2018.

[82] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, Seattle, WA, November 2006.

[83] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–308, San Francisco, CA, April 2008.

[84] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. *Communications of the ACM*, 54(11):93–101, November 2011.

[85] Kehuan Zhang and XiaoFeng Wang. Peeping Tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *Proceedings of the 18th USENIX Security Symposium*, pages 17–32, Montreal, Canada, August 2009.

[86] Yongwang Zhao, David Sanán, Fuyuan Zhang, and Yang Liu. Reasoning about information flow security of separation kernels with channel-based communication. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 791–810, Eindhoven, The Netherlands, April 2016.