A note on verifying information flow control systems with Nickel

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

October 2019

Abstract

This technical report provides supplementary material for the paper *Nickel: A Framework for Design and Verification of Information Flow Control Systems* [7]. It documents some design trade-offs we have considered during the development of both the Nickel framework and systems verified using Nickel.

1 Intransitivity

Practical systems generally need downgrading operations for intentional declassification and endorsement; otherwise, information continues to flow in one direction and there is no way to get results out of these systems. Transitive noninterference is too restrictive for expressing policies for such downgrading operations. Therefore, Nickel adopts a general form of *intransitive* noninterference [6]: information can flow from A to B and from B to C, but not necessarily from A to C directly; in other words, can-flow-to relations can be both transitive and intransitive. Below we detail this decision in the context of NiStar.

NiStar is an OS kernel for decentralized information flow control (DIFC) [5]. It provides a small number of kernel object types. Each object is associated with a triple of $\langle \text{secrecy}, \text{integrity}, \text{ownership} \rangle$ labels, where each label is a set of tags (i.e., opaque integers). At the core of NiStar's information flow policy is the following can-flow-to relation:

Definition 1. 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 intransitive: $L_1 \rightsquigarrow L_2$ and $L_2 \rightsquigarrow L_3$ together do *not* imply $L_1 \rightsquigarrow L_3$. Like in many other DIFC systems (e.g., HiStar [9] and Flume [4]), this flexibility enables users to declassify private data as they deem appropriate. For instance, Alice may remove secrecy tags from her files (based on ownership) and copy them to a public directory for sharing; she may also set up a gate, which provides additional ownership tags to threads that enter the gate and allows other users to access her data in a controlled way.

We initially considered using transitive noninterference with two workarounds to support NiStar's policy. The first workaround was to follow the approach taken by Ironclad [3]: carve out the downgrading part of each operation, prove transitive noninterference for the parts before and after downgrading, and prove state-machine refinement for the downgrading part. This workaround was not suitable for NiStar, as downgrading was an essential functionality of the OS kernel and it was unclear how to separate the downgrading part of each system call. The second workaround was to consider the universal set of ownership tags as the domain of each operation, which effectively made the policy transitive. This workaround was able to catch many covert channels, however it is too permissive: for instance, it was unable to catch a bogus system call that allowed Alice to declassify another user's data.

The formulation of noninterference used by Nickel supports intransitive policies such as the one in NiStar: it allows Alice to declassify her own data (whether her user-space declassifiers are correct is an orthogonal problem), and also prevents her from declassifying other users' data.

2 State-dependent `dom`

Let A, D, and S denote the set of actions, domains, and states, respectively. Classical noninterference uses a state-independent `dom` function ($A \to D$), which associates each action with a domain. However, as many system calls query

the system state to decide the currently running process or thread, the dom function also needs to look into the state to decide the resulting domain. As a workaround, we initially tried to fold the system state into an action, as an extra argument to every system call. This workaround complicated both unwinding and refinement proofs: if we added the extra argument to both the specification and implementation, this would introduce performance overhead; if we added the the extra argument to the specification only, extra axioms or trusted code were needed to match the specification with the implementation.

To avoid these issues, Nickel uses a state-dependent dom function ($A \times S \to D$). This change slightly complicates the definition of noninterference, which we find acceptable, and requires additional conditions for unwinding. We initially used the following extra unwinding condition:

$$\mathcal{I}(s) \wedge \mathcal{I}(t) \wedge s \overset{u}{\approx} t \wedge \mathsf{dom}(a, s) \rightsquigarrow u \Rightarrow \mathsf{dom}(a, s) = \mathsf{dom}(a, t).$$

We replaced it with two weaker conditions as described in the paper:

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

3 Formulations of noninterference

We have considered two formulations as the top-level definition: noninterference and a variant called *noninfluence* [8]. Noninterference is defined in Sigurbjarnarson et al. [7]. Noninfluence is defined as follows:

Definition 2 (Noninfluence)**.** Given a system $\mathcal{M} = \langle A, O, S, \mathtt{init}, \mathtt{step}, \mathtt{output} \rangle$, a policy $\mathcal{P} = \langle D, \rightsquigarrow, \mathsf{dom} \rangle$, and an observational equivalence $\approx$, $\mathcal{M}$ satisfies noninfluence for $\mathcal{P}$ and $\approx$ if and only if the following holds for any trace tr, domain u, reachable states s and t, and purged trace $tr' \in \mathsf{purge}(tr, u, t)$:

$$(\forall v \in \mathsf{sources}(tr, u, s).\ s \overset{v}{\approx} t) \Rightarrow \mathsf{run}(s, tr) \overset{u}{\approx} \mathsf{run}(t, tr').$$

Figure 1 shows the logical implications among these formulations: unwinding conditions together imply both noninfluence and noninterference; noninfluence plus output consistency implies noninterference. Note that noninterference depends on a policy only, while noninfluence depends on both a policy and an observational equivalence.

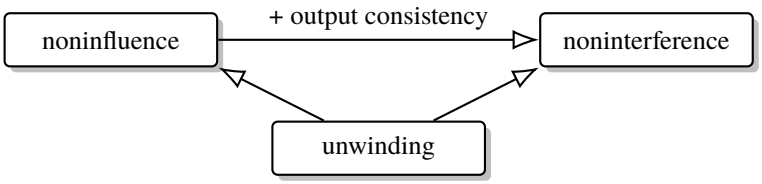

Figure 1: Logical implications among noninfluence, noninterference, and unwinding.

Since unwinding conditions imply both noninfluence and noninterference, developers may choose either one as the top-level specification of their systems and use Nickel in the same way, though these two formulations have different trusted components and interpretations. If developers trust the policy only (which is often much simpler than the observational equivalence), we consider noninterference as a reasonable definition, and the observational equivalence is an untrusted proof input. However, if developers choose to additionally trust the observational equivalence, noninfluence is a useful definition as it provides more detail on which subset of the system state is involved in certain flows.

As an example, consider $A \rightsquigarrow B$, $B \rightsquigarrow C$, but $A \not\rightsquigarrow C$, where B is the domain of a downgrading operation. One may use noninterference as the top-level definition and trust the state-machine specification of B that describes how B restricts the indirect influence from A to C. Alternatively, one may use noninfluence as the top-level definition and trust the observational equivalence that describes the subset of the system state B may access.

We considered more restrictive formulations of noninterference in terms of downgrading, such as ta-security [2: §3]; we also considered formulations with dynamic policies (i.e., state-dependent can-flow-to) [2: §4]. We decided to keep the current formulation as it was simpler for understanding and verification, and it was sufficient for the systems we verified; it would be interesting to further investigate these formulations.

4 Experience with using Nickel

NiStar. We initially followed HiStar's design to allow an object to be linked by multiple containers. Such links complicate the invariant on quotas, as the kernel would need to deduct the quota from multiple containers. Furthermore, they require reference counting and a more complex naming scheme—to avoid covert channels from reference counters, HiStar uses the pair ⟨container-ID, object-ID⟩ to refer to an object in most system calls, rather than an object-ID alone [9]. We decided to not support multiple links in NiStar; therefore, an object-ID was sufficient for naming an object.

In NiStar, a thread may free an object from a container only if it can write to both the object and the container. This closes a channel in HiStar but also allows users to create "zombie" objects that cannot be reclaimed by anyone in the system. We initially considered designing a garbage collector with domain $\langle \varnothing, \mathbb{U}, \varnothing \rangle$: it would be able to free any object in the system given its universal integrity, but would not be able to take any user input to decide which object to free given its empty secrecy and ownership. A usable garbage collector may need to be trusted with more power (e.g., universal ownership), or the system administrator may consider it legitimate to create "zombie" objects.

To avoid a channel in scheduling, the scheduler's decision on which thread to run next should be independent of other domains. We considered several scheduler designs: (1) a naïve scheduler that enumerates every object ID in the system for scheduling, (2) a cooperative `yield` system call that allows thread T to yield to thread T' only if $\mathcal{L}_T \rightsquigarrow \mathcal{L}_{T'}$, and (3) a hierarchical scheduler that walks the container hierarchy. We decided against these designs due to performance and flexibility concerns, and instead adapted the exokernel scheduler to DIFC.

NiKOS. We have implemented two versions of NiKOS. The first one closely follows mCertiKOS [1]. While the policy is simple (information can flow between the scheduler and any process, but not directly between two processes), a downside is that it can hide other covert channels, as two processes may exploit the scheduler to communicate information. One might use fine-grained domains rather than processes as the policy, or choose noninfluence as the top-level specification, additionally trusting the observational equivalence; both will complicate the (trusted) specification.

As an experiment, we have implemented a second version of NiKOS, referred to as NiKOS+, by removing flows to the scheduler. The policy is shown in Figure 2: information can flow from each process to its descendants and from the scheduler to each process, but *not* from a process to the scheduler or to a non-descendant process. This policy ensures isolation between any two sibling processes, and more generally, between each process and its non-descendants.

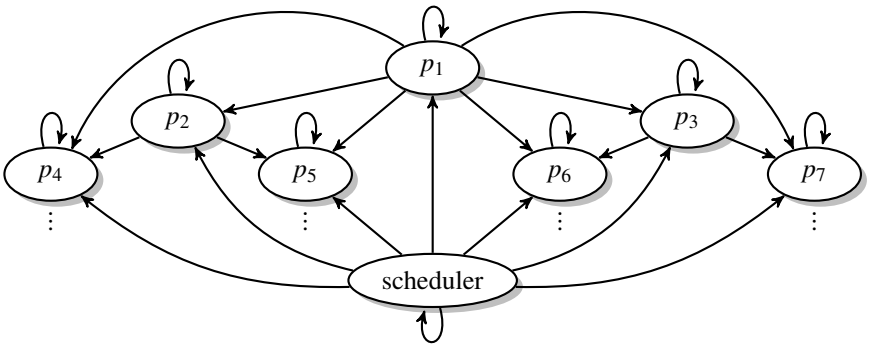

Figure 2: The isolation policy of NiKOS+. Each process can have up to two child processes in this case.

To implement this policy, NiKOS+ uses a hierarchical scheduler, which matches its PID allocation scheme. It maintains a global array of PIDs. Initially, the array contains PID 1 in every slot; when the `spawn` system call creates a child process p_i, it updates the "subspace" of the slots (which correspond to all its descendants) with PID i. For `yield`, the scheduler simply cycles through the array and schedules the process as specified in each slot.

ARINC 653. Following previous work [10], we use a single transmitter domain in the information flow policy for ARINC 653. This policy is unable to catch covert channels due to mixing data from different ports. A better policy would be to break the transmitter domain to smaller domains among communication ports.

Acknowledgments

We thank Nickolai Zeldovich for discussions on HiStar and noninterference.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

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

[7] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A framework for design and verification of information flow control systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 287–306, Carlsbad, CA, October 2018.

[8] 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.

[9] 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.

[10] 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.