Notary: A Device for Secure Transaction Approval

Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich
MIT CSAIL

Abstract

Notary is a new hardware and software architecture for running isolated *approval agents* in the form factor of a USB stick with a small display and buttons. Approval agents allow factoring out critical security decisions, such as getting the user's approval to sign a Bitcoin transaction or to delete a backup, to a secure environment. The key challenge addressed by Notary is to securely switch between agents on the same device. Prior systems either avoid the problem by building single-function devices like a USB U2F key, or they provide weak isolation that is susceptible to kernel bugs, side channels, or Rowhammer-like attacks. Notary achieves strong isolation using *reset-based switching*, along with the use of physically separate systems-on-a-chip for agent code and for the kernel, and a machine-checked proof of both the hardware's register-transfer-level design and software, showing that reset-based switching leaks no state. Notary also provides a trustworthy I/O path between the agent code and the user, which prevents an adversary from tampering with the user's screen or buttons.

We built a hardware/software prototype of Notary, using a combination of ARM and RISC-V processors. The prototype demonstrates that it is feasible to verify Notary's reset-based switching, and that Notary can support diverse agents, including cryptocurrencies and a transaction approval agent for traditional client-server applications such as websites. Measurements of reset-based switching show that it is fast enough for interactive use. We analyze security bugs in existing cryptocurrency hardware wallets, which aim to provide a similar form factor and feature set as Notary, and show that Notary's design avoids many bugs that affect them.

CCS Concepts • **Security and privacy → Systems security**; *Software and application security*; *Security in hardware*; *Logic and verification.*

Keywords Security, Verification, Cryptocurrency hardware wallet

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6873-5/19/10.
https://doi.org/10.1145/3341301.3359661

1 Introduction

Users routinely rely on computers such as workstations, laptops, and smartphones to approve security-critical operations. These include financial operations, such as bank transactions and cryptocurrency transfers, and system-administration operations, such as deleting backups, changing user permissions, and modifying DNS records. Today's systems do not provide strong guarantees of security for such operations. For instance, cryptocurrency theft is a major problem: it is estimated that cybercriminals stole nearly $1B in cryptocurrency in 2018 [19], a significant fraction of which was a result of private keys being stolen from individuals' computers.

This paper contributes Notary, a new design that provides trustworthy user approval of security-sensitive operations. Notary is a physical device that executes a subset of an application that is responsible for obtaining user approval. We use the word *agent* to refer to this critical component of a logical application that has been factored out, as in `ssh-agent`. The Notary device has a display and buttons to provide agent code with a trustworthy user I/O path, which ensures that an adversary cannot provide fake output to the display or fake the user's input. Notary supports multiple agents, but provides strong isolation between them by using a new technique called *reset-based switching*, which avoids confidential data leakage even across microarchitectural side channels [38, 40]. Notary also provides strong isolation between agent code and the trusted kernel by running them on physically separate systems-on-a-chip (SoCs), which avoids attacks like Rowhammer [59].

Notary's design is influenced by a number of prior systems that also factored out important operations for security, as discussed in more detail in §2. The key research contribution of Notary is providing secure task switching between different agents running on the device. Prior work either avoids the problem altogether with a fixed-function device like RSA SecurID, iPhone secure enclave, or a USB U2F key, or it provides weak isolation between agents, which allows confidential data to leak from one agent to another. For example, smartphones run multiple apps, but their kernel is complex, and they have had bugs that can give the adversary root access. Cryptocurrency hardware wallets like the Ledger wallet inherit the kernel-based isolation design and have similar bugs that compromise isolation between different agents (§2.1). Finally, verified operating systems can ensure correct behavior with respect to a specification, but current systems do not reason about microarchitectural side channels or Rowhammer-like attacks that can violate isolation.

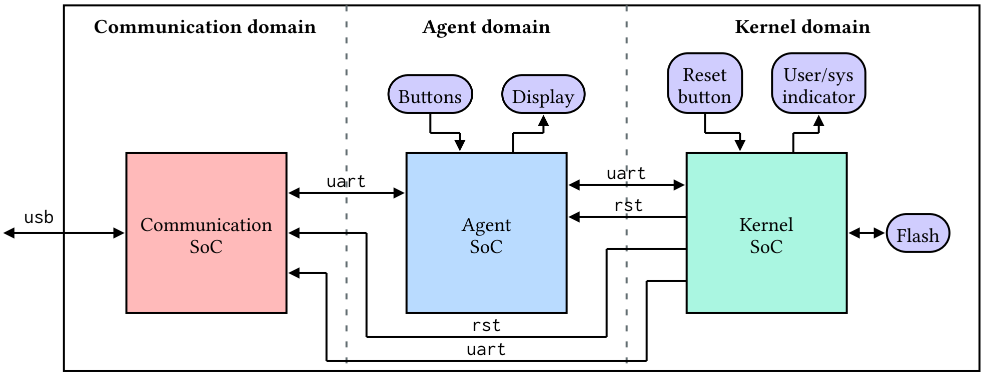


Figure 1. NOTARY's design physically separates trust domains with an SoC per domain and a simple interconnect between trust domains (reset wire and UART). NOTARY employs two such separations, the first between the kernel and the agent UI/signing code, and the second between the agent UI/signing code and the agent communication code.

Figure 1 shows NOTARY's overall architecture. NOTARY achieves strong isolation using its *separation architecture*, which implements privilege separation using separate *domains*. Each domain runs on its own system-on-a-chip, which contains its own CPU, ROM, RAM, and peripherals. The domains are connected by a limited interface (a serial UART link). NOTARY applies this separation twice. The first separation provides strong isolation between agents and the kernel: the kernel runs on its own dedicated domain, separate from third-party agent code. This protects the kernel from buggy or malicious agents, enabling the multi-agent support of the wallet. The second separation provides isolation within a single agent, between the sensitive UI and signing component of an agent and the complex and bug-prone communication code such as the USB driver, which is placed in its own domain. Physical separation eliminates the need to rely on complicated hardware protection mechanisms such as user/kernel mode and memory protection units, and it ensures that NOTARY's isolation cannot be subverted by Rowhammer-like attacks.

To avoid having one domain (i.e., dedicated SoC) per agent, NOTARY runs only one agent at a time and implements agent switching via a new technique we call *reset-based switching*, which uses a formally verified *deterministic start* primitive to fully reset a domain's SoC, encompassing its CPU, RAM, and peripherals, to clear all internal state before starting to execute new agent code. Just resetting or power-cycling the SoC is insufficient: for example, reset is not guaranteed to clear the CPU's architectural state such as registers [70] or microarchitectural state, and power-cycling leaves state in SRAM for minutes [51]. NOTARY's formal verification encompasses the register-transfer-level (RTL) description of the SoC's internals as well as the initialization code in boot ROM that assists in clearing state after reset. Reset-based switching eliminates by design many classes of bugs that affect traditional user/kernel co-resident designs, because no state is leaked by the hardware between executing one agent and another.

To demonstrate that NOTARY can support diverse agents, we developed two examples: a Bitcoin wallet and a general-purpose approval manager (§8). The contributions of this paper are:

- The reset-based switching technique for securely multiplexing agents on the same hardware while avoiding unintended side channels or leakage.
- The specification of deterministic start, the implementation strategy using software-assisted reset, and the formal verification of deterministic start for a RISC-V SoC.
- The separation architecture for supporting multiple agents with strong isolation and a secure user I/O path.
- A physical prototype of the NOTARY design.
- An implementation of two NOTARY agents.
- An evaluation of NOTARY's security and usability.

NOTARY's prototype has several limitations. The prototype uses development boards instead of a custom board that fits into a USB stick, and it doesn't provide resistance to physical attacks. We believe that NOTARY could be made in a production form factor similar to existing cryptocurrency hardware wallets and use tamper-proof hardware at a similar price as today's wallets. Like current hardware wallets, the prototype uses an LCD display to communicate with the user; an implementation with a refreshable braille display could support vision-impaired users. The prototype uses an open-source RISC-V processor which is not yet commercially available in silicon, so it uses a soft core instantiated on an FPGA.

2 Background and related work

NOTARY adopts many security ideas of previous systems. This section relates NOTARY's design to existing security devices as well as to research on supporting strong isolation.

2.1 Hardware wallets

Hardware wallets are designed to protect a user's private key for a cryptocurrency. Transactions proceed as follows: the user sets up a transaction on their computer, sends it to the device, reviews the transaction on the device's display, and presses a button to confirm. If confirmed, the device signs the transaction and sends it to the computer for broadcast.

Among other devices such as Trezor [4] and KeepKey [2], the Ledger [3] family of hardware wallets is one of the most secure. The Ledger Nano S is the only current device that makes use of a secure element with a custom OS, and it is the only certified hardware wallet on the market [6].

The Ledger supports many cryptocurrencies. It runs one agent for each cryptocurrency, which can sign transactions and store the private key using tamper-proof hardware. Agents share a screen for displaying transactions and an input device for approving the displayed transactions. The Ledger has 84 approved applications in its app store, 79 of which are wallets and 5 of which are other security-oriented agents such as FIDO U2F and OpenPGP. Third-party developers have published 55 of the apps.

The Ledger hardware has two SoCs: a secure element (ST31) that runs a custom closed-source OS kernel and multiplexes between agents in user/kernel co-resident style, and a fixed-function microcontroller (STM32) that acts as an input/output proxy for the secure element because of pin limitations [7].

Ledger's secure element is resistant to physical attacks, and it provides a hardware root of trust to ensure the device is running authentic firmware. Additionally, it can prove to the user's computer that it is genuine hardware, a measure to protect against malicious counterfeit wallets [7].

In the Ledger wallet, as well as other current hardware wallets, the user I/O path is handled by the same processor that runs the complex USB driver. Thus, a bug in the driver could compromise the I/O channel by allowing an adversary to forge button presses to authorize malicious transactions.

NOTARY inherits the form factor and many design ideas from existing hardware wallets. Most hardware wallets do not run third-party code, but Ledger supports multiple agents, with many written by third parties. Ledger's approach of using a standard user/kernel boundary to isolate agents from one another has led to vulnerabilities in the past (see below). In contrast, NOTARY leverages its separation architecture and reset-based agent switching to improve isolation between agents. NOTARY also places USB and user I/O on separate SoCs, so that the agent code has a trustworthy user I/O path.

The architecture of hardware wallets like the Ledger has a number of shortcomings that are illustrated by the range of vulnerabilities that have been discovered, which motivate NOTARY's design:

System call vulnerabilities. The Ledger kernel had a number of system calls that incorrectly validated pointer arguments, potentially allowing agents to read data belonging to the kernel or other agents [33, 52]. The system calls performed hardware-accelerated cryptographic operations; the kernel was involved in order to mediate access to the hardware. More broadly, Ledger must provide many system calls (and thus has a large attack surface) because the kernel has to mediate access to many hardware resources.

Memory protection errors. The Ledger kernel had a bug that caused it to misconfigure the memory protection unit (MPU) bounds, allowing agents to read 16K of memory belonging to another agent [33, 52]. Similarly, the Trezor hardware wallet suffered from a bug that erroneously allowed writes to flash memory [57]. These hardware wallets depend on correctly configured memory protection hardware because the kernel and agents share the same CPU and RAM.

USB software bugs. The Trezor wallet suffered from vulnerabilities in which data packets sent over the USB interface were able to trigger a buffer overflow [56], and USB leaked discarded memory [58]. These bugs are particularly threatening because the USB software runs on the same CPU as the agents and the kernel.

2.2 Security devices

Two-factor authentication devices. Hardware security devices such as RSA SecurID [53], smart cards [10], and U2F tokens [5, 24, 62] serve as a second factor for authentication. Such devices can prove physical possession and protect against phishing attacks. However, because they lack a display, they are more suitable for authenticating *logins* than *transactions*. These devices do not help if the user's computer has been compromised and is running malware. Before the user logs in, two-factor authentication prevents the malware from impersonating the user. However, as soon as the user logs in with their two-factor device, malware can take over the user's session and impersonate the user by submitting requests on the user's behalf. This is possible because the two-factor device does not participate in anything after login.

Transaction approval devices. Certain devices support explicit transaction approval by the user, such as the E.dentifier2 [11] or EMV-based card payment systems. However, these devices implement a single protocol for a single purpose. It would be impractical for the user to carry around separate physical devices for many agents. Supporting multiple agents on the same device securely is the key problem that NOTARY addresses.

Smartphones. Smartphones have design features that provide better security than computers running traditional desktop operating systems. For example, Apple iOS uses a hardware root of trust to ensure that it boots an unmodified kernel [13]. Furthermore, iOS lets a user launch an application unambiguously, and iOS enforces stronger isolation between applications than a desktop OS. However, the iOS kernel's complexity has made it vulnerable to a long history of jailbreaks [21, 49], often exploitable by malicious applications; such exploits would expose an agent's secrets to the attacker. Furthermore, the design suffers from hardware side channels as a result of sharing the CPU and RAM between running applications.

The secure element found in iPhone and Android smartphones is used to maintain cryptographic secrets in isolation from the general-purpose CPU. However, this secure element does not have a trustworthy output path to the user-visible display, and it does not allow execution of application-specific code. This makes it impossible to implement an agent that, for example, approves and signs Bitcoin transactions, because the code to sign a Bitcoin transaction could not run on the secure element, and because a compromised kernel on the general-purpose CPU could display a different transaction from the one that the user is about to sign.

Trusted execution environments. Trusted execution environments (TEEs) like Intel SGX [22], Komodo [26], and virtual machines [54] provide stronger isolation between applications on the same computer. The Trusted Platform Module (TPM) [66] can also be used to implement TEEs, such as in the Flicker architecture [45]. TEEs can rely on a kernel or hypervisor to mediate access to shared storage and user I/O [77, 78], or they can use TPMs to bootstrap a secure user I/O path, such as in the Cloud Terminal system [43] or Lockdown [68].

In contrast to Notary, TEEs typically do not provide strong isolation between protection domains [48], especially against microarchitectural side-channels or hardware defects like Rowhammer. Notary uses its separation architecture and reset-based task switching to defend against these attacks.

2.3 Strong isolation

Verification. Formal verification is a promising approach to provide strong correctness and isolation guarantees. Researchers have built verified operating systems [31, 32, 35, 36, 47, 60] that provide proofs of varying degrees of isolation, and some verified operating systems are used in real-world security-critical applications [37]. The proofs typically reason at the architectural level, on top of a model of the ISA specification. This does not take into account microarchitectural side channels, which are not captured by the specification, or hardware bugs, where the implementation does not satisfy the specification.

Traditional hardware verification efforts have focused on the correctness of the CPU implementation with respect to the ISA specification. This is insufficient to prevent data leaks, because the specifications are too weak and because they do not extend to microarchitectural state (see §5.1).

Some work focuses on verifying security properties of hardware implementations, such as verifying information flow in an implementation of ARM TrustZone [27]. HyperFlow [28] contributes a RISC-V processor verified to enforce secure information flow, taking into account timing side-channels. These systems prove properties at the hardware description language (HDL) level. Notary's reset-based task switching avoids having to prove strong information-flow properties about hardware and shows that it suffices to prove a simpler deterministic start property of the hardware and boot code. Furthermore, Notary's verified deterministic start encompasses the entire SoC, including RAM and peripherals, not just the CPU itself. Finally, Notary's separation architecture avoids Rowhammer-like attacks that could otherwise be used to subvert isolation.

Type-safe languages could be used to harden the implementation of the Notary TCB (which is implemented in C++ in our prototype). Prior work has shown that both Rust [39] and Go [23] can help prevent bugs in an OS kernel. Restricted languages and verification can also be used to ensure that one application's code cannot observe confidential state from another application [29, 42, 74].

CPU state cleansing. Several systems have proposed cleansing CPU state to mitigate side channel attacks. For example, Düppel periodically flushes portions of L1 caches to reduce the possibility of cache timing attacks [75]. MRT improves upon Düppel by executing a carefully crafted sequence of instructions to overwrite additional state, including the I-cache, D-cache, and branch predictor [67]. The MI6 processor introduces a purge instruction that flushes microarchitectural state [16]. However, ensuring that all CPU state has been exhaustively cleansed is a formidable task that depends on complex implementation details of the CPU. To our knowledge, Notary is the first system to employ verification to prove that the internal (microarchitectural) state of a CPU can be reset correctly. In CleanOS, sensitive data is managed at the language runtime level by evicting idle secrets from RAM, but it does not take into account microarchitectural state [63].

Partitioning hardware. Another approach to reducing side channel risks is to segregate shared CPU resources, either by eliminating sharing entirely or by reserving bandwidth. This strategy has been employed on various hardware layers, such as on-chip networks [69, 71], last-level caches [41], and memory controllers [34]. Previous systems have also relied on partitioning cores to improve performance [14, 15, 17, 50, 61]. Because we focus on hardware wallet security rather than general purpose computing, we can use a more aggressive approach of dividing physical resources into separate domains, each with a dedicated SoC.

3 Threat model and security goal

Notary is designed to defend against an adversary who wants to approve an operation contrary to the wishes of the user (e.g., to transfer Bitcoin to the attacker's address). Notary's design assumes a threat model similar to that of current cryptocurrency hardware wallets:

- The remote attacker has full control over the user's computer, including the ability to execute code as root. This includes the ability to tamper with network packets to and from the user's computer, manipulate the computer screen, spoof keyboard and mouse input, corrupt the computer's operating system and running processes, and attack the Notary over USB by sending arbitrary malicious packets. We believe this assumption is an accurate model of the real world where the user may have malware running on their computer, either because the adversary exploited some vulnerability in the OS or because the adversary tricked the user into installing the malware.
- The remote attacker can author malicious agents and trick the user into installing and running them on the Notary. Malicious agents may run arbitrary code. This includes trying to exploit bugs in system software, hardware vulnerabilities, or microarchitectural side-channels.
- The attacker's agent code can take advantage of Rowhammer-like bugs to violate the digital gate abstraction in the hardware of the agent domain (i.e., cause unintended bit flips). However, we assume that once the reset line is asserted, gates in the Notary start behaving according to the digital abstraction.
- The attacker cannot physically interact with the trusted user I/O of the Notary. This means either that the attacker does not have physical control of the Notary, or that the attacker cannot bypass the access control mechanism on the Notary (like a fingerprint reader or PIN code, combined with tamper-resistant hardware [7, 13]).
- The user will use the Notary correctly. This means only approving operations that the user intends to approve, and launching the correct agent. Similar assumptions have proven to be a significant problem in past designs (with issues such as phishing), so the Notary aims to make all of these interactions explicit and stateless to reduce the possibility of user confusion about state.

Notary's threat model is focused on the end user; security of server-side components is largely orthogonal (or not relevant, in the case of serverless systems like Bitcoin). We expect that servers are architected so that they can securely check signatures or other messages from agents running on Notary.

Notary's threat model is stronger than that of other security-related devices such as U2F tokens, which assume that the user's computer is trusted for I/O and which focus on phishing attacks and protecting a user's key. For example, U2F tokens typically trust the browser on the user's computer to display information about the site they are logging into.

For high-stakes applications like cryptocurrencies, where an adversary can steal millions of dollars from a single successful attack that is untraceable and irreversible, a strong threat model is appropriate. Past attacks have ranged in sophistication, from phishing schemes [64] to cryptocurrency-stealing malware injected into popular libraries [20].

Central to Notary's goal is to securely multiplex agents on the same device, which requires providing strong isolation between potentially buggy or malicious agents. Running agents on Notary should be as secure as using a separate physical device dedicated to running each agent. Providing security equivalent to a separate per-agent device is an ideal goal, because the overall application can get a strong end-to-end guarantee. For example, consider a Bitcoin wallet agent in the ideal world with per-agent devices. A transaction can be signed with the user's Bitcoin private key only if the user explicitly approved the transaction. This is because the signing key is only available on that user's agent device, only the agent code executes on that device, and the agent code only signs a transaction if it displays that transaction to the user and the user presses a button to approve. The assumptions in this argument are that (1) the agent code must be correct, which is outside of the scope of Notary itself; (2) the adversary must not have physical access to the user's devices; and (3) the user must correctly identify which device they are using, and interact with it properly.

This paper does not focus on physical attacks such as supply-chain attacks or physical extraction of secrets; existing solutions such as secure enclaves, hardware root of trust, and attestation, as deployed in devices such as Apple iPhones [13] and Ledger wallets [7] provide protection against such attacks and are compatible with Notary's design. Similarly, except for microarchitectural side channels, Notary's threat model does *not* include arbitrary side channels [76] such as electromagnetic radiation [12], power analysis [44], and acoustic analysis [30].

4 Overview

Notary consists of three security domains, each with its own separate SoC (which has a CPU, ROM, RAM, and peripherals such as UART). One domain runs the kernel and two domains run separate components of an agent, following the least-privilege design principle [55]. This section provides an overview of Notary. The next section, §5, describes deterministic start, the primitive that provides strong isolation in Notary. §6 and §7 describe the entire Notary design, including the hardware and software, in more detail.

Figure 1 illustrates Notary. The design is structured around physical separation. First, Notary separates the kernel, which is responsible for switching between agents and managing persistent state. Second, Notary separates agent UI and signing code from communication code that is exposed to the outside world (such as the USB subsystem). As a result, the system is split into the following three trust domains:

The *kernel domain* protects the user's master key, stores agent code and data, protects agents' persistent data from each other, and performs agent switching. This is the most security-critical processor because it has direct access to all private key material. Notary runs as little code on this processor as possible, and no third-party code is ever run here. With this design, Notary minimizes the amount of code that must be audited for security. We anticipate this code will be amenable to push-button verification [47, 60] due to its simplicity.

The *agent domain* contains code for the currently active third-party agent and has exclusive access to the display hardware and buttons. The agent uses the display to indicate when an action requires user consent. Because the agent's code is security critical, it must be isolated from the outside world.

The *communication domain* provides this necessary layer of defense in depth by handling all untrusted I/O; we focus on USB in our design, but the communication domain could easily be extended to handle Bluetooth, NFC, etc. We anticipate an attacker could try to attack Notary over these interfaces because they are complex and therefore error-prone. However, because the communication domain cannot access the user's private key and must communicate with the agent domain through a narrow interface, it can be excluded from the trusted computing base.

Notary supports multiple agents but only runs one at a time. To switch between agents, the system fully resets the state of the agent and communication domain hardware before loading a different agent's code. The communication and agent SoCs have no shared memory with one another or with the kernel domain. This simplifies isolation and deterministic start: all code is loaded from the kernel domain at reset time via UART.

Agents. Each Notary agent consists of two binaries. The code can be written in any language, as long as it compiles to native code. The code that runs on the agent domain is generally responsible for UI and signing operations, while the code that runs on the communication domain is responsible for USB communication and other functionality that is outside of the agent's TCB. Each agent, a bundle of the two binaries, is installed to the kernel domain's flash memory, which is also where the agent's persistent data is stored.

When launched, an agent's code and data is loaded onto the agent and communication SoCs. Agent code does not run on top of an operating system, but instead has direct access to raw hardware, with full privileges. The agent domain has direct access to the display and buttons, which provides the secure I/O path between the agent and the user. Notary can expose hardware to agents safely because of its reset-based switching: raw hardware can be shared without fear that sensitive data will remain latent in hardware or that agents will corrupt the hardware, because the hardware is reset before each agent is launched. To simplify interacting with raw hardware, agents can optionally statically link with Notary's library code, which includes display, GPIO, UART, and USB drivers. The library is structured similar to existing microcontroller SDKs' peripheral drivers.

5 Deterministic start

Notary relies on the ability to reset a domain to start executing an agent from a well-known state. We would like to have a noninterference property for agents A and B that run sequentially: informally, the execution of one agent on the SoC should be unable to influence the execution of the next. This implies that A cannot corrupt the execution of B, and B cannot learn secrets of A. Notary achieves this property with a *stronger* one that implies noninterference. Between runs of code from agents A and B on the agent and communication SoCs, Notary fully resets the SoC state, including all CPU architectural and microarchitectural state, as well as the RAM and peripherals, to a deterministic value independent of previous SoC state, i.e., a constant value. We refer to this notion as *deterministic start*. If all state in the domain is reset to a fixed value, then to the agent B, it is indistinguishable whether the SoC previously ran agent A or a different agent A', or even whether an agent had previously run at all, so B's execution must be independent of whatever happened on the SoC before B started running. This fully captures all CPU architectural and microarchitectural side-channel attacks, as well as attacks related to leftover state in RAM or peripherals.

5.1 Challenge

By definition, specifications of SoC components like the CPU describe behavior only at the architectural level, i.e., in terms of architectural registers and opcodes accessible to software. To reason about reset at the level of microarchitectural state, we must analyze a particular SoC *implementation* (and therefore a particular CPU implementation, etc.) at the register-transfer level. Achieving deterministic start is challenging for several related reasons.

In existing CPUs, merely asserting the reset line does not clear all internal state. CPU specifications acknowledge this: the RISC-V specification says that after reset most architectural CPU state is undefined [70, §3.3]. This means that after reset, even directly software-visible state could contain data from the code that was running before reset. This is true even if the CPU has been formally verified against the ISA specification, since the ISA does not require state to be cleared. Similarly, asserting the reset line does not clear RAM, and peripheral specifications do not provide guarantees about how asserting the reset line affects internal state.

Even power-cycling (as opposed to asserting the reset line) is insufficient, because state stored in SRAM can persist for minutes without power. For example, Rahmati et al. [51] showed that bitmap images stored in SRAM can be recovered even after a device has been without power for several minutes. This means that confidential state from one application could still be present in CPU registers, memory, or peripherals after the SoC has been powered off and then powered on to run a different application, unless the user waits for well over 3 minutes [51].

5.2 Software-assisted deterministic start

To overcome the above challenges, this paper introduces the notion of *software-assisted* deterministic start: with carefully crafted initialization code set up to run on the CPU after reset (e.g., as the first instructions in boot ROM), starting from any possible SoC state, a reset followed by executing for some number of cycles n causes the CPU to execute the initialization code, which clears all architectural state (such as the values in general-purpose registers), microarchitectural state (such as the instruction decoder and ALU state), RAM state, and peripheral state. For example, asserting the reset line might set the program counter to point to the start of the boot ROM, but it could leave the value of `r0` as it was pre-reset; a `mov r0, $0` instruction could clear the leftover value in the register. Similarly, the instruction decoder could have leftover state after reset, but executing a number of instructions could clear that internal state. RAM could have leftover contents after reset, but the initialization code could loop over the RAM and set it to zero. Peripherals could retain internal state after reset, but initialization code could ensure it is cleared properly. We use the approach of software assistance to achieve deterministic start for the entire SoC, ensuring that no matter the previous state, after n cycles of execution following reset, the SoC completes deterministic start, so all SoC-internal state is set to a well-known value. The particular value of n is important to know, because only after n cycles is it safe to run another agent on the SoC.

5.3 Formalization

Deterministic start requires reasoning about the implementation of the SoC at the register-transfer level (RTL). We have to reason about the behavior of asserting the reset line, as well as the result of letting the SoC run for a number of cycles, where its behavior could be affected by the leftover (potentially malicious) state, over all possible initial states. Missing a single register in the CPU for a single pre-reset state invalidates the property of deterministic start, and in turn, the noninterference property. It may even be the case that for a particular SoC, no configuration of boot ROM will achieve the deterministic start property. For example, consider an SoC that has a CPU with a "lifetime cycle counter" that can be read but not written to, that is preserved even when the reset line is asserted.

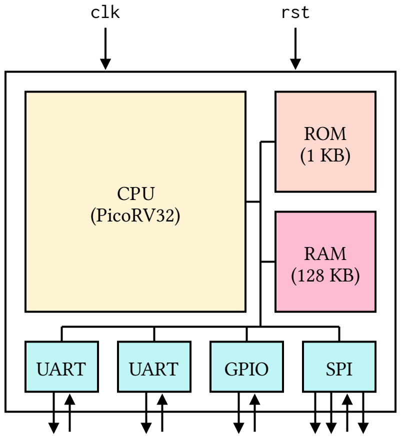


Figure 2. A schematic of Notary's agent domain SoC, which is formally verified to satisfy deterministic start.

For this reason, we use formal verification to prove that a particular SoC with some particular code in boot ROM satisfies deterministic start, no matter what the (potentially malicious) starting state. Figure 2 shows the SoC we verify.

The SoC is a stateful digital circuit, a collection of registers and combinatorial circuits. The SoC has some internal configuration, and it also has a number of input and output wires. Let S be the set of all possible internal SoC states (registers at the RTL level). Let $\hat{I}$ be the set of all possible inputs (with elements assigning values on all input wires, such as the reset line, GPIO pins, and UART RX wire). Because the SoC is deterministic at the RTL level, there is a function step: $S \longrightarrow \hat{I} \longrightarrow S$ that describes the behavior of the SoC, simulating it for a *single cycle*. The step function encapsulates the behavior of the entire SoC, and so it is dependent on the gate-level implementation of the CPU, memory controller, peripherals, and RAM, as well as the implementation and contents of the ROM.

Let $\hat{I} = \{\text{rst=0}, \text{rst=1}\} \times I$ be a decomposition of inputs, separating the reset wire from other inputs. We define two helper functions to describe the behavior of the SoC depending on the state of the reset line. The function reset: $S \longrightarrow I \longrightarrow S$ describes one cycle of execution with input i while the reset line is asserted:

$$\text{reset}(s, i) = \text{step}(s, (\text{rst=1}, i))$$

In practice, the reset line will end up being held for more than one cycle, but this is okay: if held for n cycles, we can think of the first $n-1$ cycles as having some unknown effect and the last cycle as having the reset effect as given above.

The function run: $\forall n, S \longrightarrow I^n \longrightarrow S$ describes n cycles of execution with a sequence of inputs while reset is not held:

$$\begin{aligned} \text{run}(s, []) &= s \\ \text{run}\Big(s, i :: \vec{i}\Big) &= \text{run}\Big(\text{step}(s, (\text{rst=0}, i)), \vec{i}\Big) \end{aligned}$$

The *deterministic start* property of an SoC, that the SoC will enter a well-known state after n cycles of execution following reset, is formalized as follows:

$$\forall i \in I, \vec{i} \in I^n.\ \exists s_f \in S.\ \forall s \in S.$$
$$\text{run}\left(\text{reset}(s, i), \vec{i}\right) = s_f$$

That is, for a given sequence of inputs, regardless of SoC starting state s, it much reach the same fixed SoC state s_f (that is independent of s) after executing for n cycles. The state s_f is not necessarily a "all zeros" state, but one where all state is some constant value. The state equivalence includes all registers in the SoC at the RTL level.

In practice, it is often the case that the input to the SoC during deterministic start is a constant i_0 known in advance (e.g., all GPIO inputs have value 0, and all UART RX lines have value 1), in which case the deterministic start property can be simplified to:

$$\exists s_f.\ \forall s.$$
$$\text{run}\left(\text{reset}\left(s, i_0\right), i_0^n\right) = s_f$$

For clarity, we use this simplified form of the deterministic start property in the rest of this section.

5.4 Verification

To avoid explicitly constructing s_f, which requires precisely specifying all internal state of the SoC at the end of deterministic start, we can prove the following property that is equivalent to deterministic start as specified above. We consider that the SoC could be in two possible states, s or s', that could differ arbitrarily. We show that executing the reset sequence starting from either state makes them indistinguishable from each other (converging to the state s_f).

$$\forall s, s'.$$
$$\text{run}\left(\text{reset}\left(s, i_0\right), i_0^n\right) = \text{run}\left(\text{reset}\left(s', i_0\right), i_0^n\right)$$

We verify the above property with a SMT solver by unrolling run, which effectively symbolically simulates the circuit for n cycles, and checking for satisfiability of the negation of state equivalence (and automatically trying increasing values of n). If the SMT solver proves that the negation is unsatisfiable, then the above property holds, and deterministic start is verified. Otherwise, the SMT solver finds that the formula is satisfiable[1], and it produces a concrete counterexample: two states s and s' that do not converge to the same CPU state after n cycles.

With this formulation, we can adopt a workflow that lets us *interactively* construct the initialization code (the contents of the boot ROM) instruction-by-instruction. We start with no initialization code, just nops encoded in the boot ROM, and attempt to run the verifier. If it fails, the SMT solver gives a concrete counterexample, showing two states that differ post-reset. From this, we determine one component of state that was not provably cleared, add initialization code to reset it (if necessary consulting the implementation of the relevant SoC component for guidance), and repeat the process.

6 Hardware architecture

Physical privilege separation. Notary supports running mutually distrustful agents, which requires some form of privilege separation between the firmware and agent code (e.g., to multiplex storage). Notary does not attempt to enforce isolation between code running on the same CPU: it does not use mechanisms such as hardware page tables or user/kernel mode. Such hardware mechanisms are complicated, so correctly programming them for privilege separation is an error-prone process; if the programmer forgets to set one bit in a crucial register, isolation will be broken. Even if the mechanism could be programmed correctly, CPUs have a history of microarchitectural attacks such as Meltdown [40] and Spectre [38], and systems have been shown to be susceptible to other types of hardware-based attacks like Rowhammer [59]. For this reason, Notary *physically* separates trust domains onto SoCs with their own separate CPU, memory, and peripherals.

Narrow interfaces. With code in different trust domains running on physically separate CPUs, Notary needs some mechanism for communication between trust domains, analogous to system calls in a traditional user/kernel co-resident design. To avoid complex drivers for protocols such as USB or Ethernet, inter-domain communication occurs using simple universal asynchronous receiver/transmitter (UART) peripherals. For supporting transactional agents, a more complex, higher-bandwidth interface is unnecessary.

Reset-based agent switching. Notary resets the SoC when switching between agents, in order to implement the deterministic start primitive. This approach ensures that every agent starts execution with a clean state, limiting the ability of a buggy or malicious agent to corrupt other agents. The user can restart the kernel domain by pressing a physical reset button. The kernel domain in turn resets the other domains by asserting the reset wire. On reset, each domain boots from a local ROM, which first completes the software-assisted deterministic start sequence and then executes a boot loader. The kernel domains loads the kernel from local flash, while the other domains load data provided over the UART connection to the kernel domain.

Display. Notary has a display controlled by agent software, connected only to the agent domain. This display shows descriptions of operations to the user for confirmation. The display protects against a malicious computer sending corrupt transactions to the device: if the user observes a bad

[1] Another possibility is that the SMT solver times out, and the result is inconclusive. We treat this case like the solver has found a counterexample, and we attempt to determine why the solver timed out.

transaction on NOTARY's display, the user should press "cancel" to reject signing the transaction.

Flash. Flash memory is attached to the kernel domain, storing the kernel code, the code for each installed agent, and each agent's saved state. Because only the kernel domain can access the flash memory, the kernel can employ wear leveling and other techniques to prolong the flash's lifetime. In contrast, Ledger exposes flash memory directly to agents, permitting malicious code to cause corruption or physical damage through repeated writes.

7 Software architecture

NOTARY splits software across the CPUs, with the goal of allowing the user to execute a number of mutually distrustful agent applications in a secure fashion.

Kernel domain. The most privileged domain runs the kernel, which is responsible for launching agents and multiplexing storage. Upon reset, the CPU starts running the kernel, which resets lower-privilege domains. Like other hardware wallets, the kernel maintains a master key that is used to derive per-agent cryptographic keys based on the agent developer's public key and the agent name. By deriving agent keys from a common master key, the kernel simplifies the backup problem, since it is sufficient to back up the master key. The kernel also stores agents and their state in flash—namely, for each agent, the kernel stores the agent code, as well as mutable data for that agent, and an audit log.

Table 1 lists system calls exported by the kernel to the agent code running on the agent domain. Unlike a traditional user/kernel boundary, each system call must be sent to the kernel domain through the UART connection. Only three system calls are available to all agents (`exit`, `exit_state`, and `log`), and these system calls do not return any response to the agent code. This limits the ability of malicious agents to learn any information by measuring the execution time of system calls.

The two most sensitive system calls, `launch` and `install`, are available only if the agent domain is running the built-in launcher or installer agents, respectively. The kernel tracks which agent is running at any given time (much like a traditional OS kernel tracks the current process), and uses that to determine which syscalls it will allow via the UART. This limits the risk of malicious or compromised agents taking advantage of those system calls.

At all times, the kernel indicates (using the user/sys indicator LED) whether the device is running system-supplied software (the launcher or the agent installer) or third-party agent software, so that agents can't spoof system apps.

Table 1. System calls supported by NOTARY's kernel.

Syscall	Apps	Description
`exit`	any	Exit without saving state
`exit_state`	any	Exit and save state
`log`	any	Append app log entry
`launch`	launcher	Start user-selected agent
`install`	installer	Install user-selected agent

Agent domain. The main logic of an agent runs on the agent domain. When the kernel launches an agent, the kernel resets the agent domain, which starts executing the boot loader after the deterministic start sequence. The kernel then sends the agent's code to the agent domain over the UART link, together with the cryptographic key and the mutable data for that agent. The boot loader receives all of this data, places it into RAM, and executes the agent.

Since the display and input buttons are directly attached to the agent domain, the agent code has a trusted I/O path to the user for displaying information and confirming operations. NOTARY assumes that the agent's mutable state is small, so the entire state is sent over to the agent domain at start time. If an agent wants to modify its state, it sends the modified state to the kernel on exit. Additionally, the kernel domain maintains an append-only log for each agent; an agent can add an entry to the log by sending a `log` system call to the kernel domain at any time.

The agent domain is also used to run the agent launcher (§7.2). When NOTARY is powered up (or reset), the kernel domain uses the same protocol described above to start a special launcher agent on the agent domain, whose job is to display the set of installed agents, and to allow the user to select which agent should execute. The choice is sent back to the kernel domain, which then resets the agent domain to execute the chosen agent. If the user wants to switch to a different agent, they must restart the entire NOTARY device to return to the agent launcher. Similarly, the agent domain is used to run the agent installer (§7.3).

Communication domain. NOTARY allows the agent developer to isolate potentially buggy code in the agent to the communication domain. For example, a developer might implement a complex USB protocol on this domain, or error-prone message parsing, so that bugs will be isolated from the trusted agent code responsible for displaying and signing transactions. The interaction for starting the necessary code on the communication domain follows the same protocol as the one used by the kernel domain to launch the agent code on the agent domain. Each agent application bundles code for the agent domain and the communication domain.

7.1 Initialization

On first use (and hard reset), NOTARY wipes all installed applications and data and re-initializes the master key using a hardware random number generator.

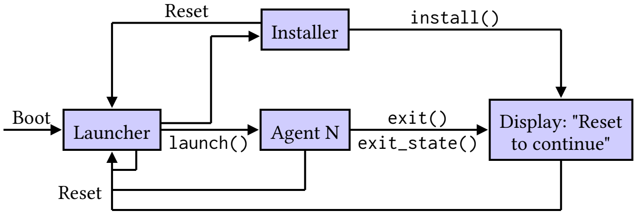


Figure 3. Notary's reset-based workflow.

7.2 Launching agents

Figure 3 shows the workflow for using Notary. Upon device boot or reset, the system spawns the launcher application, which is implemented as a system application that has the authority to call the `launch` system call. After launching an agent (or the installer), the only way to switch agents is to restart the device.

When an agent is launched, the kernel supplies the code to run on the agent domain and the communication domain, and at the same time, injects the application-specific cryptographic key as well as persistent data into the address space for the application. With this design, agents do not require system calls that read data: everything is provided at launch.

7.3 Installing new agents

The installer is implemented as an application that has special privileges to call the `install` system call. Installing agents when using a compromised computer is a challenge. The Notary kernel must obtain executable code from an untrusted source, because it comes from the internet and goes through the computer, and save it as an agent with a given name. Having the agent have the wrong name would be dangerous, as it would confuse the user and enable phishing attacks, so we must protect against this. Running malicious agents on the device is not an issue, however, as long as the user is not confused into believing one agent is another: Notary prevents agents from interacting in any way, so a malicious agent cannot compromise others. Installing agents is an infrequent process, so there is no requirement for the installation process to be particularly streamlined. Notary follows the same approach as existing hardware wallets for this challenge:

Curation. Notary relies on a curator, similar to Ledger Manager or Apple's App Store, where reviewers examine agents before accepting them to the store. While this doesn't prevent malicious agents, the reviewing phase can filter out deceitfully named agents, preventing phishing attacks. For example, reviewers may not accept an agent called "Ethereum" unless the Ethereum Foundation submitted it. In this approach, all agents are digitally signed by the curator, with signatures checked by the kernel prior to installation.

Out-of-band communication. Notary supports reliance on out-of-band communication. For example, a bank could provide their public key on a slip of paper, so that when installing the agent, Notary verifies a signature on the agent code and displays the public key for the user to confirm.

7.4 Upgrading the Notary kernel

To support upgrades of the kernel, Notary has a boot loader in ROM that loads kernel code from flash memory. During the upgrade process, Notary receives new kernel code from the computer over USB, checks the version number to prevent downgrades, and checks a digital signature by the vendor to confirm authenticity before writing new firmware to the flash.

7.5 Device loss

Notary follows the same approach as existing hardware wallets for handling device loss. The user backs up their master key so it can be restored to a new device. All agents that use keys derived from the master secret will have the same key on the new device. Other agent state is backed up in encrypted form, with a key derived from the master secret, so it can be restored to a new device.

To prevent an adversary from using the lost device, Notary requires the user to enter a PIN to access any functions, with retry limits and hard reset after sufficiently many failures. Standard tamper-resistance techniques ensure that an adversary cannot physically bypass PIN retry limits.

8 Applications and agents

Notary allows us to make applications more secure, where critical operations can be factored out and described to the user for approval by an agent running on the Notary. The agent running on the Notary maintains a private key that never leaves the device, and it allows the user to attest to operations through a signature with the private key that is only performed if the user approves of the operation that is displayed on the Notary screen.

This section examines two applications and discuss challenges in modifying the app and implementing the agent.

8.1 Bitcoin

A cryptocurrency is a natural fit for Notary because it already has a structure where critical operations must be signed with a private key: no changes to the application are required. The workflow for using a Bitcoin agent on Notary involves creating transactions on the computer, which only has the public key, and approving them on the device, which has the private key, and therefore the ability to sign the transaction. In practice, authenticating receive addresses may be challenging, requiring out-of-band communication if the computer is compromised, but verifying transactions and amounts provides added security even without recipient verification.

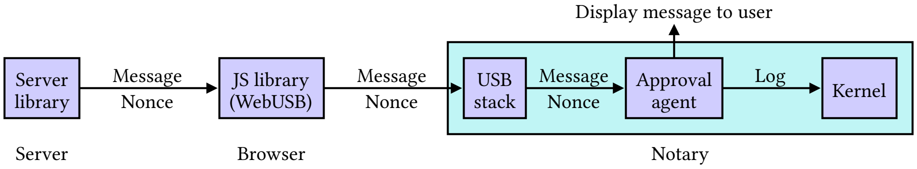


Figure 4. Approval agent framework: server asks the user to approve an operation by sending a message describing the operation through the browser to the approval agent on Notary, which displays it to the user. If the user approves, the approval agent signs the message and logs it. The signed response (not shown) is returned to the server through the same path.

Like all Notary agents, the Bitcoin agent bundles binaries for running on the agent and communication domains. The agent domain code receives Bitcoin transactions over UART from the communication domain, parses and displays them, and signs the transaction if approved by the user. The communication domain code implements USB-related functionality for compatibility with existing desktop wallet software, proxying requests over UART; this code is outside the TCB of the Bitcoin agent.

8.2 Approval Manager

Web applications can benefit from obtaining strong approval for sensitive operations from a user's Notary device. Examples of sensitive operations are sending money in a banking system, deleting backups in a storage system, and updating DNS records in a critical domain. Other operations in these applications may be less sensitive and could be performed without explicit approval from the Notary device.

To support this style of application, we developed a generic Approval Manager for Notary, as shown in Figure 4. The Approval Manager framework consists of an agent that has a private key (derived from the device master key), a JavaScript library for interacting with the approval agent via WebUSB [9], and a server-side library for checking approval messages for sensitive operations, similar to the transaction authorization extension of the Web Authentication API [8]. The Approval Manager agent has code for displaying and signing operations, which runs on the agent domain, and code that implements WebUSB-related functionality and proxies requests over UART, which runs on the communication domain.

When a user attempts to perform a sensitive operation in a web app, the server sends code to the web browser to request approval, together with a nonce. This code uses the client-side JavaScript library to send the request to the approval agent on the Notary. If the explicit approval flag is set, the agent displays the ASCII string provided by the server to the user; if the user approves, the agent signs the ASCII message, together with the nonce and sends the signature back to the JavaScript library, which relays it to the server. If the explicit approval flag is not set, the agent signs the message without asking the user. The server checks the signature with the registered public key for the user.

The agent logs every signed message, appending the message to the agent's log stored in the kernel domain's flash by calling the `log()` system call before sending out the signature. The log provides the rationale for "blindly" signing requests that do not have the explicit approval flag set, without displaying the request to the user: this increases convenience for requests of medium importance but still provides a strong audit log that allows for tracking down the effects of a compromise if one is discovered later.

Initial enrollment involves supplying the Approval Manager's public key to the server so it can be associated with a user account. This step can either be done at account creation or later, as is done with U2F enrollment. The computer must be uncompromised at the time of enrollment.

We discuss two web applications that could benefit from Notary to obtain strong approval for sensitive user-initiated operations.

8.2.1 Banking

Many bank websites simply require the user to log in with a username/password to make a transfer; a phishing site or a keylogger can compromise the application. Some banks support 2-factor authentication, but malware on the computer could still perform transfers after authentication succeeds.

A bank website can use the Approval Manager to improve security by requiring all transactions be signed by the Notary. If the transaction is for a small amount and there haven't been a large number of such small transactions issued recently, the bank could ask for a non-explicit approval, which simply logs the approval in the user's Notary. For large-value transactions or many small transactions, the bank can request explicit approval, which requires the user to physically confirm the transaction on the device. The message includes information about the transaction, such as the recipient's name and account number, and the amount being transferred, that the user can interpret on the Notary screen.

8.2.2 DNS

Domain registrar websites are susceptible to the same kinds of attacks that affect the bank application. In some ways, DNS is more important to protect with the Approval Manager, because some DNS operations are difficult to undo, unlike bank transfers. To secure DNS updates, a domain registrar could require approval for all changes to DNS data. Similar to the bank, this requires adapting the application to require signatures on these operations. The message signed by the approval agent contains a description of the operation, such as "update the A record for example.com to 1.2.3.4."

9 Security argument

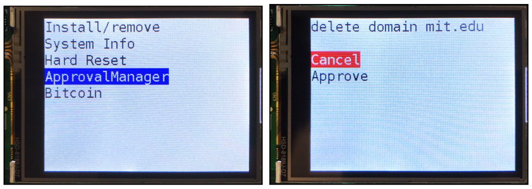
Notary runs multiple agents on the same physical device, but its design aims to ensure that for any agent *A*, running it on the Notary should be as secure as running it on a separate device. The security argument boils down to considering two cases: when the Notary is running agent *A*, and when the Notary is running other agents.

When the Notary is running agent *A*, the agent code was started correctly (by assumption that the kernel, which sends the initial agent code and state, is implemented correctly, and by our verified deterministic start, which ensures the agent SoC starts from a state that is not influenced by prior agent executions), no other code is executing on the agent SoC, and the agent SoC is directly wired to user I/O devices (display and buttons). Furthermore, because agents are started only by the launcher after explicit device reset by the user, the user must have interacted with the launcher to choose agent *A*, and thus knows which agent they are interacting with.

When the Notary is running a different agent *B*, it has no access to the secrets of agent *A*. In particular, the state of the agent SoC is independent of any prior executions by other agents (by deterministic start), and, by the assumption that the kernel is correct, the kernel will not send the secrets of agent *A* when launching agent *B*. Any state on the communication SoC should not reveal secrets of agent *A*, by the assumption that the agent was implemented correctly and did not send any secrets over the untrusted communication link.

10 Implementation

We have built a hardware/software prototype of Notary. The hardware uses three SoCs, two of which are STM32 microcontrollers with ARM Cortex-M4 processors (kernel and communication domains), and one of which is a RISC-V SoC based on the PicoRV32 CPU [72], instantiated on an FPGA (agent domain). The SoCs are connected as described in §6. All CPUs use UART peripherals for communication between domains. The kernel domain uses persistent flash memory and GPIO pins to drive the user/sys indicator LED and reset wires; the agent domain uses an SPI peripheral to drive the display and GPIO pins for the user input buttons; the communication domain uses a USB peripheral to communicate with the host computer. Figure 5 shows pictures of the device.



delete domain mit.edu

Figure 5. The launcher (*left*), showing the currently selected application. The Approval Manager (*right*), showing a prompt to confirm a DNS domain deletion.

Table 2. Size of Notary's system software implementation.

	Component	LOC
TCB	Kernel	870
	Launcher	110
	Installer	290
	Boot loader	210
	Drivers	2630
	Total	4110
Untrusted	Shared drivers	1310

The agent domain's RISC-V SoC is formally verified to satisfy the deterministic start property. For convenience, the communication domain uses an ARM-based SoC that is not verified. This does not impact the security of Notary, because the communication domain is outside the TCB of both the kernel and agent code.

We implemented the Notary kernel and system software (launcher and installer) in about 150 lines of ARM assembly, 100 lines of RISC-V assembly, and 5,500 lines of C/C++. Table 2 describes the breakdown between different components of the system and shows which components are inside the TCB. Although there is a significant amount of driver code in the TCB, much of it is not exposed to an adversary. The only device that is directly exposed to potentially adversarial input from third-party code is the UART (which accepts input from the agent domain). The UART driver is simple: it uses the simplest mode of operation (polling-based with no buffers), containing 2 lines of initialization code, 3 lines of read code, and 3 lines of write code.

Notary aims to provide the security of physically separate hardware for separate agents with a single physical device. As a part of this, we ensure that different agents appear as different USB devices to the host computer. When Notary switches agents, it briefly disconnects the pull-up resistor on the D+ line, which appears as a USB disconnect to the host machine. When the next agent runs, the host computer re-enumerates the USB device.

There are some differences between the prototype implementation of Notary and the design of §6. For ease of development, the ARM-based SoCs have flash memory rather than a ROM for the boot loader. To prevent modification of flash memory in the communication domain, the boot loader uses a feature of the flash controller that locks up the controller until reset, which prevents modification of flash (resetting would return control to the trusted boot loader code).

Verifier. We used the SMT backend of the Yosys synthesis framework [73] to flatten the Verilog RTL of the RISC-V SoC into a single file with a model of the SoC state and the step function. We implemented the verification strategy described in §5 in 250 lines of Racket code on top of the Rosette solver-aided programming library [65], which uses the Z3 theorem prover [25]. The Verilog code, as well as the contents of the boot ROM, are untrusted and verified to satisfy the deterministic start property. The verification tools, including the 250 lines of Racket code, as well as Rosette, Z3, Racket, and Yosys, are part of the TCB.

11 Evaluation

To evaluate Notary, we answer the following questions:

- Does Notary's design prevent vulnerabilities that have affected hardware cryptocurrency wallets? (§11.1)
- Is it feasible to verify deterministic start? (§11.2)
- What applications fit the agent model? How easy is writing agents and integrating Notary into applications? (§11.3)
- Is reset-based agent switching fast enough? (§11.4)
- Is Notary's hardware cost reasonable? (§11.5)

11.1 Notary's design prevents vulnerabilities

For the vulnerabilities discussed in §2.1, we analyze the possibility of similar bugs impacting Notary.

System call vulnerabilities. Existing secure device kernels have had bugs in system calls that let agents read kernel memory. Notary avoids the possibility of such problems: there are only a handful of system calls, and none of them read data (Table 1). One reason this design is possible is that agents have direct access to hardware for user I/O, UARTs, and (if our hardware had it) cryptographic acceleration, so the kernel does not need to mediate access to these resources. Another reason is that agents' persistent state is moved implicitly at launch and exit time, rather than explicitly via read and write system calls.

Memory protection errors. Existing secure devices have had bugs that cause memory protection hardware to be misconfigured, allowing agents to read sensitive data. Notary avoids the possibility of such bugs by not using memory protection hardware to isolate agents. Instead, it isolates with a combination of physically separate domains and reset-based switching, which is formally verified.

USB software bugs. Existing secure devices have had buffer overflow bugs in USB interface software. Notary could suffer from such bugs as well, but blunts their security impact by placing the USB software in its own physical domain, so that overflows or code injection cannot easily interact with the main agent or kernel.

Table 3. Size of agent, desktop, and server software.

Application	Component	LOC
Bitcoin	Agent	300
	Desktop software	—
Approval	Agent	150
	JavaScript library	100
	Bank	100
	DNS manager	100

11.2 Deterministic start can be verified

To test the feasibility of verifying deterministic start, we verified the RISC-V SoC of the agent domain (Figure 2). The SoC's CPU, the PicoRV32, is comparable to processors used in existing hardware wallets, such as the Cortex-M0 in Ledger wallets, which does not have a branch predictor or speculation of any kind. Using our interactive approach, we developed initialization code that performs full deterministic start of the processor and rest of the SoC, provably resetting the CPU, RAM, and peripherals to a well-known state before launching agents.

We first ran the verifier with no software reset code, relying only on hardware reset: this revealed leftover architectural state, such as in general-purpose registers, microarchitectural state, such as in the instruction decoder, and peripheral state, such as in GPIO, that was not cleared by the hardware reset. Next, we added initialization code to clear general-purpose registers (which also indirectly cleared some microarchitectural state such as the instruction decoder). Running the verifier again revealed that the initialization code successfully cleared architectural state, but there was still microarchitectural state in the memory write machinery that was not reset. To rectify this, we added code to issue a dummy write to ROM. Next, we wrote code to clear peripheral state: this required a write to the memory-mapped GPIO peripheral; UART and SPI were cleared by the initial reset. Finally, we wrote code to clear contents of RAM. With this final modification, the verifier could prove that our code correctly implements deterministic start on our RISC-V SoC: starting from any state, asserting the reset line for a single cycle and then letting the CPU run for 180342 cycles (3.6 ms) brings it to a deterministic starting state. Figure 6 shows the final code for reset.

11.3 Notary agents are easy to develop

Notary is suitable for running agents that run on existing hardware security devices, such as cryptocurrency wallets, U2F, and OpenPGP. We implemented two agents for Notary: a Bitcoin wallet and an Approval Manager. Table 3 shows the size of components involved, including agents that run on Notary (not including shared driver code), code that runs on the untrusted computer, and code that run on the server.

```
/* clear registers and
 * some microarchitectural CPU state */
  li ra, 0
  la sp, _stack_top  // 0x20000800
  li gp, 0
  /* ... */
  li t6, 0
/* clear state in
 * memory write machinery */
  sw zero, 0(zero)
/* clear gpio */
  la t0, _gpio       // 0x40000000
  sw zero, 0(t0)
/* clear sram */
  la t0, _sram_start // 0x20000000
  la t1, _sram_end   // 0x20020000
loop:
  sw zero, 0x00(t0)
  /* ... */
  sw zero, 0x3c(t0)
  addi t0, t0, 0x40
  bne t0, t1, loop
/* done with deterministic reset,
 * proceed to load code over UART */
```

Figure 6. Initialization code for verified software-assisted deterministic start of the RISC-V SoC. The code contains 58 RISC-V assembly instructions and executes in 180342 cycles (3.6 ms) on the PicoRV32, resetting all SoC state to a fixed value starting from any initial state.

Bitcoin wallet. We implemented a Bitcoin hardware wallet in 300 lines of code, not including the Bitcoin parsing/cryptography library and shared driver code. Our implementation works with existing desktop wallets like Electrum [1], requiring no new code to be written for the computer.

Approval Manager. We implemented the Approval Manager described in §8.2 in 150 lines of code. Integration into existing websites required minimal code changes. We wrote a JavaScript library on top of WebUSB for the Approval Manager in 100 lines of code. Modifying a toy banking application to require confirmation for transfers required changing less than 100 lines of code. Modifying a web-based DNS manager [18] required adding 100 lines of code, plus about 25 lines of code per form modified to require approval.

11.4 Reset-based agent switching is fast enough

Notary has a user interface similar to existing wallets, except it implements strong isolation between agents with reset-based agent switching. We measure the speed of reset-based agent switching to assess the impact on usability of this "heavy-handed" approach to isolation. We measure, in aggregate, the time to execute the following launch sequence:

1. Launcher sends selection to kernel
2. Kernel resets agent and communication SoCs
3. Agent and communication SoCs run deterministic start
4. Agent and communication SoCs run the boot loader
5. Kernel sends code to agent and communication SoCs
6. Execution starts on agent and communication SoCs

Table 4. Latency of reset-based task switching.

Step	Time (ms)
Reset & boot loader	7.4
Receiving program	127.0
Total	134.4

Table 4 shows the latency for reset-based task switching. We measure end-to-end latency to be **134 ms** in our prototype. Two steps consume the majority of this time. The reset sequence and boot loader (steps 3–4) take **7.4 ms** to execute. Loading a 40KB agent over the relatively slow UART (step 5) takes **127 ms**. The end-to-end latency is within the time scale required to create the illusion of direct manipulation [46], so switching is fast enough for interactive use.

11.5 Notary is competitive on cost

We estimate that the cost of a production version of Notary will be comparable to existing hardware wallets, which currently retail for $50–$150. Notary has essentially the same hardware, except it requires two extra SoCs that cost about $4 each, increasing the cost by about $8.

12 Conclusion

Notary is a case study in designing for security. Notary simplifies software (e.g., using reset-based agent switching) and wastes resources (e.g., using physical separation) in order to achieve strong isolation and defense in depth. This separation and reset-based switching eliminates by design classes of bugs that affect traditional user/kernel co-resident designs, including OS bugs, microarchitectural side-channels, and certain hardware bugs. Notary can improve the security of applications where the crucial transaction decision can be succinctly summarized and delegated to a strongly isolated agent running on Notary. Source code for Notary is available at https://github.com/anishathalye/notary.

Acknowledgments

Thanks to the anonymous reviewers and our shepherd, Haibo Chen, for feedback that improved this paper. This research was supported by NSF awards CNS-1413920 and CNS-1812522. Anish Athalye is supported by an SOSP 2019 student scholarship from the National Science Foundation.

References

[1] Electrum Bitcoin wallet. https://electrum.org/.

[2] KeepKey. https://shapeshift.io/keepkey/.

[3] Ledger hardware wallets. https://www.ledger.com/.

[4] Trezor. https://trezor.io/.

[5] Yubico. https://www.yubico.com/.

[6] Rapport de certification ANSSI-CSPN-2019/03. https://www.ssi.gouv.fr/uploads/2019/02/anssi-cspn-2019_03fr.pdf, Feb. 2019.

[7] Ledger documentation hub. https://buildmedia.readthedocs.org/media/pdf/ledger/latest/ledger.pdf, Feb. 2019.

[8] Web authentication: An API for accessing public key credentials. https://www.w3.org/TR/webauthn/, Mar. 2019.

[9] WebUSB API. https://wicg.github.io/webusb/, Apr. 2019.

[10] M. Abadi, M. Burrows, C. Kaufman, and B. Lampson. Authentication and delegation with smart-cards. *Science of Computer Programming*, 21(2):93–113, 1993.

[11] ABN AMRO. E.dentifier2. https://www.abnamro.nl/en/mobile/images/Generiek/PDFs/Overig/edentifier2_usermanual_english.pdf.

[12] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi. The EM side-channel(s). In *Proceedings of the 2002 IACR Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Redwood City, CA, Aug. 2002.

[13] Apple, Inc. iOS security. https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf, Nov. 2018.

[14] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–44, Big Sky, MT, Oct. 2009.

[15] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 49–65, Broomfield, CO, Oct. 2014.

[16] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas. MI6: Secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Columbus, OH, Oct. 2019.

[17] E. Bugnion, S. Devine, and M. Rosenblum. DISCO: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–156, Saint-Malo, France, Oct. 1997.

[18] J. Carr. NamedManager. https://github.com/jethrocarr/namedmanager.

[19] CipherTrace. Cryptocurrency anti-money laundering report. https://ciphertrace.com/wp-content/uploads/2018/10/crypto_aml_report_2018q3.pdf, Oct. 2018.

[20] T. Claburn. Check your repos... crypto-coin-stealing code sneaks into fairly popular NPM lib (2m downloads per week). https://www.theregister.co.uk/2018/11/26/npm_repo_bitcoin_stealer/, Nov. 2018.

[21] CoolStar. Electra. https://coolstar.org/electra/, Dec. 2018.

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

[23] C. Cutler, M. F. Kaashoek, and R. T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 89–105, Carlsbad, CA, Oct. 2018.

[24] E. Dauterman, H. Corrigan-Gibbs, D. Mazières, D. Boneh, and D. Rizzo. True2f: Backdoor-resistant authentication tokens. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 743–761, San Francisco, CA, May 2019.

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

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

[27] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. 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, Apr. 2017.

[28] A. Ferraiuolo, M. Zhao, A. C. Myers, and G. E. 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, Oct. 2018.

[29] M. Fleming. A thorough introduction to eBPF. https://lwn.net/Articles/740157/, Dec. 2017.

[30] D. Genkin, A. Shamir, and E. Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *Proceedings of the 34th Annual International Cryptology Conference (CRYPTO)*, pages 444–461, Santa Barbara, CA, Aug. 2014.

[31] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 595–608, Mumbai, India, Jan. 2015.

[32] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. 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, Nov. 2016.

[33] C. Guillemet. Firmware 1.4: deep dive into three vulnerabilities which have been fixed. https://www.ledger.com/2018/03/20/firmware-1-4-deep-dive-security-fixes/, Mar. 2018.

[34] A. Gundu, G. Sreekumar, A. Shafiee, S. H. Pugsley, H. Jain, R. Balasubramonian, and M. Tiwari. Memory bandwidth reservation in the cloud to avoid information leakage in the memory controller. In *Proceedings of the 3rd Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, pages 11:1–11:5, Minneapolis, MN, June 2014.

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

[36] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. 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, Oct. 2009.

[37] G. Klein, J. Andronick, M. Fernandez, I. Kuz, T. Murray, and G. Heiser. Formally verified software in the real world. *Communications of the ACM*, 61(10):68–77, Oct. 2018.

[38] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 19–37, San Francisco, CA, May 2019.

[39] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64kB computer safely and efficiently. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 234–251, Shanghai, China, Oct. 2017.

[40] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium*, pages 973–990, Baltimore, MD, Aug. 2018.

[41] F. Liu, Q. Ge, Y. Yarom, F. McKeen, C. V. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proceedings of the 22nd IEEE International Symposium On High Performance Computer Architecture (HPCA)*, pages 406–418, Barcelona, Spain, Mar. 2016.

[42] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster. p4v: Practical verification for programmable data planes. In *Proceedings of the 2018 ACM SIGCOMM Conference*, Budapest, Hungary, Aug. 2018.

[43] L. Martignoni, P. Poosankam, M. Zaharia, J. Han, S. McCamant, D. Song, V. Paxson, A. Perrig, S. Shenker, and I. Stoica. Cloud terminal: Secure access to sensitive applications from untrusted systems. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, June 2012.

[44] R. Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *Proceedings of the 2000 IACR Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 78–92, Worcester, MA, Aug. 2000.

[45] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM EuroSys Conference*, pages 315–328, Glasgow, Scotland, Apr. 2008.

[46] R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the AFIPS 1968 Fall Joint Computer Conference*, pages 267–277, San Francisco, CA, Dec. 1968.

[47] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. 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, Oct. 2017.

[48] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *Proceedings of the 2018 USENIX Annual Technical Conference*, pages 227–240, Boston, MA, July 2018.

[49] Pangu Team. Pangu jailbreak. http://en.pangu.io/, July 2016.

[50] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Broomfield, CO, Oct. 2014.

[51] A. Rahmati, M. Salajegheh, D. E. Holcomb, J. Sorber, W. P. Burleson, and K. Fu. TARDIS: Time and remanence decay in SRAM to implement secure protocols on embedded devices without clocks. In *Proceedings of the 21st USENIX Security Symposium*, pages 221–236, Bellevue, WA, Aug. 2012.

[52] Riscure Team. Hacking the ultra-secure hardware cryptowallet. https://www.riscure.com/blog/hacking-ultra-secure-hardware-cryptowallet/, Aug. 2018.

[53] RSA Security. RSA SecurID hardware tokens. https://www.rsa.com/content/dam/en/data-sheet/rsa-securid-hardware-tokens.pdf, Oct. 2015.

[54] J. Rutkowska and R. Wojtczuk. Qubes OS architecture. https://www.qubes-os.org/attachment/wiki/QubesArchitecture/arch-spec-0.3.pdf, Jan. 2010.

[55] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

[56] SatoshiLabs. Details about the security updates in Trezor One firmware 1.6.2. https://blog.trezor.io/details-about-the-security-updates-in-trezor-one-firmware-1-6-2-a3b25b668e98, June 2018.

[57] SatoshiLabs. Trezor one: Firmware update 1.6.3. https://blog.trezor.io/trezor-one-firmware-update-1-6-3-73894c0506d, Aug. 2018.

[58] SatoshiLabs. Details about the security updates in Trezor One firmware 1.7.2. https://blog.trezor.io/details-about-the-security-updates-in-trezor-one-firmware-1-7-2-3c97adbf121e, Dec. 2018.

[59] M. Seaborn and T. Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, Mar. 2015.

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

[61] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[62] S. Srinivas, D. Balfanz, E. Tiffany, and A. Czeskis. Universal 2nd Factor (U2F) overview. https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-u2f-overview-v1.1-id-20160915.pdf, Sept. 2016.

[63] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. CleanOS: Limiting mobile data exposure with idle eviction. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 77–91, Hollywood, CA, Oct. 2012.

[64] A. Thomas and J. Segura. Electrum Bitcoin wallets under siege. https://blog.malwarebytes.com/cybercrime/2019/04/electrum-bitcoin-wallets-under-siege/, Apr. 2019.

[65] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 530–541, Edinburgh, United Kingdom, June 2014.

[66] Trusted Computing Group. Trusted Platform Module. https://www.trustedcomputinggroup.org/groups/tpm/.

[67] V. Varadarajan, T. Ristenpart, and M. M. Swift. Scheduler-based defenses against cross-VM side-channels. In *Proceedings of the 23rd USENIX Security Symposium*, pages 687–702, San Diego, CA, Aug. 2014.

[68] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig. Lockdown: Towards a safe and practical architecture for security applications on commodity platforms. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST)*, pages 34–54, Vienna, Austria, June 2012.

[69] Y. Wang and G. E. Suh. Efficient timing channel protection for on-chip networks. In *Proceedings of the 6th IEEE/ACM International Symposium on Networks-on-Chip (NoCS)*, pages 142–151, Copenhagen, Denmark, May 2012.

[70] A. Waterman and K. Asanovic. The RISC-V instruction set manual, volume II: Privileged architecture. https://riscv.org/specifications/privileged-isa/, June 2019.

[71] D. Wentzlaff, C. J. Jackson, P. Griffin, and A. Agarwal. Configurable fine-grain protection for multicore processor virtualization. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 464–475, Portland, OR, June 2012.

[72] C. Wolf. PicoRV32 – a size-optimized RISC-V CPU. https://github.com/cliffordwolf/picorv32, 2019.

[73] C. Wolf. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/, 2019.

[74] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[75] Y. Zhang and M. K. Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pages 827–838, Berlin, Germany, Nov. 2013.

[76] Y. Zhou and D. Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. Cryptology ePrint Archive, Report 2005/388, Oct. 2005.

[77] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy*, pages 616–630, Oakland, CA, May 2002.

[78] Z. Zhou, M. Yu, and V. D. Gligor. Dancing with giants: Wimpy kernels for on-demand isolated I/O. In *Proceedings of the 25th IEEE Symposium on Security and Privacy*, pages 308–323, Oakland, CA, May 2004.