Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris and Nickolai Zeldovich *MIT, Cambridge, MA, USA*

Editors: Nic Lane and Xia Zhou

Notary: A DEVICE FOR SECURE TRANSACTION APPROVAL

Excerpted from "Notary: A Device for Secure Transaction Approval," from SOSP '19: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, https://dl.acm.org/doi/10.1145/3341301.3359661 and *;login; The USENIX Magazine*, with permission. © 2019

Notary is a new design for a hardware wallet, a device that is used to perform sensitive transactional operations like cryptocurrency transfers. Notary aims to be more secure than past hardware wallets by eliminating classes of bugs by design and by formally proving the correctness of the key operation used in its implementation. We built a physical prototype of Notary and showed that it achieves functionality similar to existing hardware wallets while avoiding many bugs that affect them.

Hardware wallets, small devices with a display, buttons, and the ability to run custom code, aim to provide a secure environment for running isolated *approval agents* for confirming operations such as bank transfers or cryptocurrency transactions. In the absence of such a device, these operations are often performed using an app running on a smartphone, which is relatively complex and bug-prone. Hardware wallets achieve overall application security without the need to trust the smartphone. Today's wallets run multiple applications, written by third-party developers, that need to be isolated from each other. Existing wallets do this using a traditional operating system design that relies on hardware mechanisms like CPU privilege levels and memory protection, but unfortunately existing wallets suffer from bugs similar to those that plague traditional operating systems.

Notary avoids such bugs with a simple design that does not rely on complicated hardware or OS protection mechanisms, instead using a physically separate system-on-a-chip for running third-party code, running only one agent at a time, and provably clearing secrets when switching agents.

THE HARDWARE WALLET PARADIGM

Users routinely rely on apps running on their smartphones to perform security-critical operations. These operations include financial operations, such as bank transfers and cryptocurrency transactions, and non-financial operations, such as system administration tasks like deleting backups or modifying DNS records. The security of these operations relies on the security of the application as well as the underlying software and hardware. Unfortunately, smartphones are inadequate for providing strong security and isolation for applications, because they have complicated software stacks that have a history of vulnerabilities. While smartphones are getting more secure over time, they are also getting more complex, and bugs continue to be found [1]. On a smartphone, a buggy or malicious app might be able to break into the operating system and tamper with another application, corrupting security-critical operations.

Modern smartphones have started including security-related hardware, such as the Secure Enclave chip in iPhones or the Titan M chip in Google's Pixel phones. This hardware enhances certain aspects of security by providing features such as secure boot. There are other aspects the enclave cannot help with, such as removing the operating system or main CPU from the trusted computing base (TCB) for third-party applications. This is because the enclave chip does not support running third-party code, and it does not have a way to directly communicate with the user because it is not directly connected to the display or touch input. Most functionality is still provided by the main CPU, an ARM A-series processor. In Android, the main processor provides a trusted execution environment (TEE), built on top of ARM TrustZone, and the operating system provides a service called Android Protected Confirmation [2], which removes the operating system from the TCB for obtaining confirmation from the user of a short prompt string (in the form of a signature on the string). This can support applications specially designed to work with Android Protected Confirmation (e.g., a new banking app), but it cannot provide strong security for applications relying on a specific transaction format or signature scheme, such as existing applications or cryptocurrencies. Furthermore, the relatively complex main processor, TrustZone, and TEE implementation remain in the TCB. Even though modern phones have secure enclaves and TEEs, approval agents that need to run custom code for parsing or signing transactions (necessary for supporting any pre-existing application or cryptocurrencies) must be run outside the secure hardware, so bugs in the operating system or main processor are still problematic for application isolation. Is it possible to achieve security for sensitive transactional operations even when a user's smartphone might be compromised?

Recently, we have seen an increase in the adoption of two-factor authentication (2FA) devices such as Universal 2nd Factor (U2F) tokens, which provide additional security for logins. Unfortunately, 2FA devices only authenticate the login process and not the actions that come afterward, so they do not help if the user's smartphone is compromised: malware on the smartphone can wait until the user logs in to a target service (using their U2F token), and then the malware can use the valid session to perform malicious actions.

In contrast, hardware wallets can provide security even when the user's smartphone is compromised. In the hardware wallet paradigm, an application is refactored to separate out security-critical approval decisions from the rest of the application. An untrusted part of the application runs on the user's smartphone, while a trusted security-critical agent runs on the hardware wallet and is used for approving transactions. The wallet has a display where it shows the user a transaction, and it has buttons to allow the user to confirm or deny the transaction. The approval is required to go through the hardware wallet, and this is enforced by requiring a signature on the transaction with a private key that's stored only in the wallet.

Cryptocurrencies already fit this paradigm where the approval decision is cleanly separated out, and in fact, hardware wallets are already popular with users of cryptocurrencies. For example, users run Bitcoin wallet software on their smartphone, where they can view their balance, view past incoming and outgoing transactions, and set up transfers, but they cannot actually transfer currency. To send Bitcoins, the user crafts a transaction on their smartphone and sends it to their hardware wallet, which parses the transaction and displays on its screen a human-readable description like "send 1.3 BTC to 1M3K…vUQ7." Only if the user presses a "confirm" button on the hardware wallet does the device sign the transaction, which enables it to be processed by the Bitcoin network.

The paradigm of authenticating transactions on a separate, secure device has gained traction among cryptocurrency users, perhaps due to the high-stakes nature of irreversible transactions. The idea has not yet caught on with more traditional client-server applications like web apps, but there has been some progress in that direction. For example, the Web Authentication API has an extension for transaction authorization, which allows for displaying a prompt string on an authenticator device and receiving confirmation from the user [3].

HARDWARE WALLETS CAN HAVE BUGS TOO

With hardware wallets, the smartphone is removed from the TCB: security depends only on the wallet, which is a big win. These devices are much simpler than smartphones, and the belief is that while the smartphone may be difficult to make secure, the simplicity of wallets allows for more secure designs.

Most hardware wallets today are fixed-function, in the sense that they don't run third-party code: they have built-in support for some fixed set of agents, for example a particular set of cryptocurrencies, and users depend on the firmware vendor to add support for specific applications. This has the obvious downside in terms of usability: when new applications come out, such as a new cryptocurrency, users have to hope that the device manufacturer implements support. The developer of the cryptocurrency has no power to add the support themselves. On the other hand, high-end wallets on the market, such as the Ledger wallet [4], support downloading and running multiple third-party agent applications on the device. This is great for usability, but it adds considerable complexity, requiring that the device be capable of isolating agents from each other, because these third-party agents could be buggy or malicious.

Current devices achieve this by multiplexing the shared hardware between mutually untrusting agents with a traditional operating system using hardware protection mechanisms like CPU privilege modes and memory protection. This leads to the potential for the same kinds of bugs that exist in smartphone operating systems. And indeed, existing hardware wallets have suffered from isolation bugs in memory protection configuration, system call implementations, and driver code [5, 6]. There is also potential for hardware-related bugs: any shared hardware state could potentially be used to infer information about other applications (this is what is happening in attacks like Spectre, for example).

NOTARY'S APPROACH

Notary is a hardware wallet that aims to avoid by design many of the security issues that affect past wallets. Notary doesn't rely on hardware protection mechanisms like CPU privilege modes or memory protection, and it doesn't have any system calls or even an operating system in the traditional sense. Instead, Notary is built around the idea of achieving isolation by using a dedicated system-on-a-chip (SoC), with its own CPU and memory, to run untrusted programs. Notary runs one program at a time on this chip, and it completely resets this chip (and all of its internal state) when switching between programs, a primitive that's formalized and proven correct in our prototype. Running untrusted code on the dedicated SoC is orchestrated by a separate chip that never runs third-party code.

Figure 1 illustrates Notary's design. The design is structured around physical separation. Notary consists of two security domains, each with its own separate SoC, which includes a CPU, ROM, RAM, and peripherals such as UART. One domain runs the kernel, and one domain runs third-party agent code. The Kernel SoC manages persistent storage and switching between agents; no third-party code ever runs on the Kernel SoC. The Agent SoC, which has no mutable non-volatile storage, runs agent applications one-at-a-time directly on raw hardware (with no OS to protect the hardware). The Agent SoC has direct access to the user I/O path, the buttons and display, as well as access to USB to communicate with the outside world.

In this architecture, after the user chooses an agent to run, it is launched as follows. First, the Kernel SoC resets the Agent SoC and clears all of its internal state. Next, the Kernel SoC reads an agent's code, keys, and data from persistent storage and sends it over the UART; on the other side of the UART, the Agent SoC's bootloader receives the code/data, saves it in RAM, and executes it. At this point, the agent runs directly on top of the hardware on the Agent SoC, not requiring further interaction with the Kernel SoC. The agent has access to everything it needs: its own code and data, the user I/O path, and communication to the outside world. It can do its job, such as displaying a Bitcoin transaction, receiving confirmation from the user, and sending

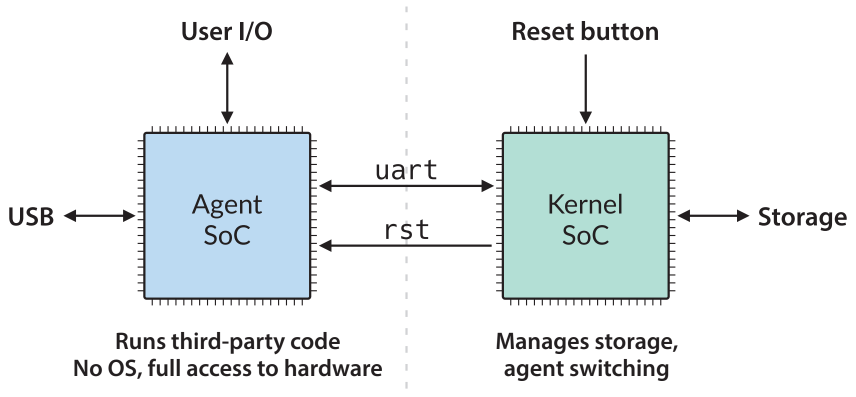

FIGURE 1. Notary's design physically separates trust domains with an SoC per domain and a simple interconnect between trust domains (reset wire and UART). Untrusted programs are run one-at-a-time on the Agent SoC, which has its state cleared between running agents.

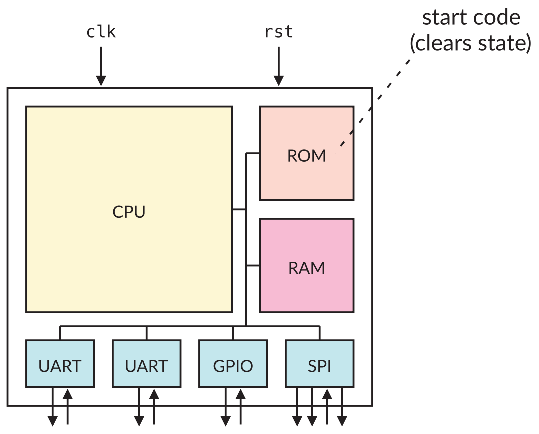

FIGURE 2. A schematic of Notary's Agent SoC. Carefully written code in boot ROM clears all internal state in the SoC after reset.

```
/* 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 start,
 * proceed to load code over UART */
```

FIGURE 3. Initialization code for Notary's SoC. The code is proven to clear all architectural and micro-architectural state in our SoC.

a signed transaction out via USB. Finally, when the agent is done, it has only one way of interacting with the Kernel SoC: a "save and exit" operation, where the agent requests termination, optionally supplying a new persistent state. After this, to run a different agent on the device, the process starts over, beginning with clearing state in the Agent SoC. Notary's separation architecture has analogs for all the operations that hardware wallets generally support: factory-resetting the device, installing/removing agents, and launching agents.

In Notary's design, the decision to connect user I/O and USB directly to the Agent SoC is important for security. An alternative design might connect these to the Kernel SoC, but that would be undesirable, because it would introduce the need to have communication between the Agent SoC and Kernel SoC during regular agent operation, adding complexity by requiring a large number of system calls beyond the single save/exit "system call" that Notary supports.

In Notary's design, it is safe to give untrusted code raw access to the user I/O and USB peripherals, because the state clearing operation covers peripherals: if a malicious or buggy agent puts the display or USB controller into a bad state, the reset and state clearing operation will fix it. Furthermore, having the display connected to the Agent SoC running potentially untrustworthy code does not introduce the possibility of confusing the user, due to Notary's reset-based workflow. The user switches applications by restarting the entire device, which makes the kernel start a special agent, the application launcher, on the Agent SoC. The user can unambiguously select an agent to run, and after that point, the chosen agent has exclusive control over user I/O until the device is restarted.

With this architecture, Notary achieves isolation between two agents running one after another on the same chip. Running agent code directly on top of raw hardware, using reset as a mechanism to switch agents, obviates the need for a traditional operating system and hardware protection mechanisms, which can be error-prone to program. Performing state clearing, wiping out all state in the Agent SoC between running different agents, ensures that one agent's secrets can't leak to another. Essentially, Notary boils isolation between agents down to state clearing.

STATE CLEARING

Clearing all internal state in a SoC turns out to be challenging, and simple approaches don't work.

At first, we thought that asserting the reset line of an SoC might be adequate. It turns out that ISAs don't guarantee that reset clears internal state; for example, the RISC-V ISA says that the program counter is set to an implementation-defined reset vector, and all other state is undefined [7]. In practice, many chips implement reset such that it only does the minimal work necessary to get the chip going again. For example, on our SoC, asserting the reset line did set the program counter to a well-known value, but it left much state inside the SoC untouched, including in registers, some CPU-internal caches, RAM, and peripherals.

Another approach we considered is power cycling the SoC to clear its internal state. However, research has shown that state inside these chips can persist for minutes without power [8]. Notary applies state clearing before every application switch, so a delay of several minutes to clear state would translate to a delay of several minutes when launching any application, making the device unusable. Furthermore, powering off the SoC for a few minutes provides no guarantees that state is actually cleared.

Provably correct software-based state clearing

Notary's approach is to use software to clear an SoC's state. The idea is that asserting the reset line resets the program counter, so it can return control to software in boot ROM that can complete the job of clearing all state in the chip, as shown in Figure 2. The idea of having initialization code run on startup is not new, but Notary's boot code is doing something unusual: it's aiming to clear every bit of state *internal* to the SoC, which includes details that don't even exist at the ISA level, such as microarchitectural state. Writing this boot code is a challenge; it's not immediately obvious that writing such code will even be possible. We normally think about code at the abstract machine level, consulting the ISA specification to understand its behavior, but in Notary's case, we need this code to affect internal state in just the right way.

To help develop this boot code and convince ourselves that it's correct, we built a tool that analyzes an SoC's implementation at the gate level to determine whether the boot code successfully clears all internal state in all situations. The tool takes the Verilog code for the SoC, converts it to a format compatible with SMT solvers, and then checks if boot code running on the chip satisfies our correctness property by simulating the circuit symbolically.

Figure 3 shows Notary's boot code for its simple RISC-V-based SoC, built on the PicoRV32 [9]. The code is formally verified to clear all SoC-internal state correctly using our tool. We are currently working on applying this technique to more complex SoCs.

PROTOTYPE

We built a hardware/software prototype of Notary, along with two agents that run on the device: a Bitcoin agent and a general-purpose web-app approval agent similar to Web Authentication. Figure 4 shows our prototype running the Bitcoin agent in the process of approving a transaction. In our prototype, the heavyweight reset-based approach for launching agents takes about 135ms, fast enough for interactive use. Of this, 7ms is spent running the formally verified state clearing code, with most of that time spent clearing RAM, and the rest of the time is spent copying the agent code/data to the Agent SoC over the relatively slow UART.

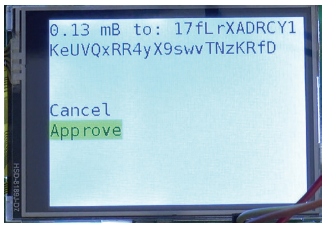

FIGURE 4. Notary prototype running a Bitcoin wallet agent.

CONCLUSION

Notary is a case study in designing for security. Notary simplifies software (e.g., using reset-based agent switching) and simplifies hardware (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.

Users have embraced hardware wallets for use with cryptocurrencies, a high-stakes operation where the overhead of using a hardware wallet is justified. In the future, we hope to see more secure and seamless support for transaction approval on smartphones, which already have much of the hardware needed to provide such functionality. ■

Anish Athalye is a PhD student in the PDOS group at MIT, working on systems, security, and formal verification.

Adam Belay is an assistant professor of computer science at MIT's EECS department, and a member of the Computer Science and Artificial Intelligence Lab. He received a PhD from Stanford for his work on high performance networking. Recent projects include Shenango, an operating system that improves datacenter efficiency, and Shinjuku, a system that uses fine-grained preemption to reduce tail latency. His current research focuses on the intersection of hardware and software, with an emphasis on improving security and performance.

M. Frans Kaashoek is the Charles Piper Professor in MIT's EECS department and a member of CSAIL, where he coleads the parallel and distributed operating systems group https://pdos.csail.mit.edu. He is a member of the National Academy of Engineering and the American Academy of Arts and Sciences, the recipient of the ACM SIGOPS Mark Weiser award and the 2010 ACM Prize in Computing, and a cofounder of Sightpath, Inc. and Mazu Networks, Inc. His current research focuses on verification of system software.

Robert Morris is a professor in MIT's EECS department and a member of the Computer Science and Artificial Intelligence Laboratory. He received a PhD from Harvard University for work on modeling and controlling networks with large numbers of competing connections. His interests include operating systems and distributed systems.

Nickolai Zeldovich is a professor of EECS at MIT and a member of the Computer Science and Artificial Intelligence Lab. He received his PhD from Stanford University in 2008. Recent projects include the CryptDB encrypted database, the STACK tool for finding undefined behavior bugs in C programs, the FSCQ formally verified file system, the Algorand cryptocurrency, and the Vuvuzela private messaging system. His current research lies in building practical verified systems.

REFERENCES

[1] B. Azad. A survey of recent iOS kernel exploits. June 2020. https://googleprojectzero.blogspot.com/2020/06/a-survey-of-recent-ios-kernel-exploits.html

[2] Android protected confirmation. https://source.android.com/security/protected-confirmation.

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

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

[5] Riscure Team. Hacking the ultra-secure hardware cryptowallet. Aug. 2018.

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

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

[8] 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. Aug. 2012. In *Proceedings of the 21st USENIX Security Symposium*, 221-236, Bellevue, WA.

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

[10] A. Athalye, A. Belay, M. F. Kaashoek, R. Morris, and N. Zeldovich. Notary: A device for secure transaction approval. Oct. 2019. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 97-113, Huntsville, ON, Canada. (The full Notary paper is available at https://pdos.csail.mit.edu/papers/notary:sosp19.pdf).