The K2 Architecture for Trustworthy Hardware Security Modules

Anish Athalye
M. Frans Kaashoek
Nickolai Zeldovich
MIT CSAIL

Joseph Tassarotti
New York University

Abstract

K2 is a new architecture and verification approach for hardware security modules (HSMs). The K2 architecture's rigid separation between I/O, storage, and computation over secret state enables modular proofs and allows for software development and verification independent of hardware development and verification while still providing correctness and security guarantees about the composed system. For a key step of verification, K2 introduces a new tool called Chroniton that automatically proves timing properties of software running on a particular hardware implementation, ensuring the lack of timing side channels at a cycle-accurate level.

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

Keywords: Security, Verification

1 Introduction

Hardware security modules (HSMs) are powerful tools for building secure systems. Applications can factor out key security-critical state and operations onto these devices, which have simple special-purpose software and hardware. For example, certificate authorities like Let's Encrypt use HSMs to store their signing key and sign certificates [10, 40]; cloud providers including Apple and Google use HSMs to enforce guess limits for cloud backup keys protected by a low-entropy PIN [9, 30, 33]; and end users use USB security keys to defend private keys used for authentication with schemes like U2F [46].

Building trustworthy HSMs that are correct, secure, and free of timing side-channel vulnerabilities, is a major challenge. Unfortunately, like any other hardware/software system, HSMs are susceptible to bugs, including logic bugs, memory corruption bugs, hardware bugs, and timing side channels [1–8, 17, 24, 32, 36, 48].

HSMs have a number of hardware/software components and subsystems, including the storage layer, I/O stack, and application logic, that not only must work correctly but also must interoperate correctly. These components interact in subtle ways, and the misbehavior of any component can break the security of the system or introduce side-channel vulnerabilities.

Formal verification is a promising approach for eliminating correctness and security bugs in individual components as well as their integration. Prior work has verified HSMs end-to-end using the Knox framework [16]; Knox requires directly reasoning about the combined hardware/software system as a monolith, limiting practicality and scalability.

This paper proposes K2, a new modular architecture for HSMs that introduces rigid separation between I/O, storage, and computation over secret state. The design enables modular formal verification, simplifying software and proof development while maintaining strong guarantees about the behavior of the entire hardware/software system, including its timing behavior. Furthermore, the architecture provides benefits even without end-to-end verification. Isolation of components limits the damage caused by bugs: for example, bugs in the I/O driver can't leak secrets of the application.

K2's architecture sidesteps a large class of bugs (e.g., bugs in the I/O driver cannot leak secrets) and simplifies verification. In K2's design, a small kernel runs a tiny amount of code in RISC-V M-mode (and the rest of its code in U-mode) and uses Physical Memory Protection (PMP) hardware to separate I/O with the external world from computation over secret state. With this setup, computation over sensitive data — where bugs or side-channel leakage are a concern — starts from a clean/constrained state, is unaffected by inputs from the external world, and is unable to produce observable intermediate outputs aside from the final result of the computation and the end-to-end running time.

Verifying HSM software for K2 is done in two steps. First, the developer proves functional correctness of their software, using existing tools for program verification like $F^\star$ and $Low^\star$. Then, the developer uses a tool we developed called Chroniton, which automatically proves the timing behavior of the software on the HSM's hardware, ensuring constant-time execution and the absence of timing side channels.

KISV '23, October 23, 2023, Koblenz, Germany
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0411-6/23/10
https://doi.org/10.1145/3625275.3625402

```
var signing_key = null

def load_key(key):
  signing_key = key

def sign_certificate(cert):
  return rsa_sign(signing_key, cert)
```

Figure 1. A functional specification for a certificate authority's certificate-signing HSM. The spec doesn't support reading out the signing key once it's installed into the device. The specification of the `rsa_sign` function is not shown.

Metatheory ties together hardware-related and software-related proofs carried out as part of K2 development, providing a guarantee that an HSM implementation's wire-level behavior correctly and securely implements a high-level functional specification.

2 Threat model and security goal

K2 considers an adversary that gains complete control over the wire-level interface of the HSM, with the ability to read and write digital-level signals on wires at every cycle. This captures realistic adversaries, such as a remote attacker who gains control over a host machine that is connected to the HSM.

K2 is concerned with remote attackers, a common adversary that HSMs aim to defend against. K2 does *not* consider attacks that require physical access to the HSM, including attacks that require physical tampering. Side channels other than digital-level timing side channels, such as electromagnetic radiation [11], temperature [31], and power [35] are also out of scope.

K2's security goals are captured by information-preserving refinement (IPR) [16]. IPR is a real/ideal-style definition that relates an HSM's wire-level behavior to a high-level RPC-style specification, such as that in Figure 1. Informally, IPR says that the HSM implementation (and it's wire-level cycle-precise behavior) should be indistinguishable from and reveal no more information than an oracle that implements the functional specification (where there is no notion of wires or timing, only input and output values). This ensures that the HSM implements the specification, but that an adversary attempting to exploit the HSM by manipulating wire-level inputs and observing the outputs at every cycle, learns no more information than the specification allows. This ensures the absence of exploitable bugs at the digital level, as well as the absence of timing side-channel vulnerabilities.

3 Design

K2's hardware and software architecture isolates computation over secret data and prevent bugs and timing side channels. A small kernel runs different functionality at different times in a strict phase-based operation, to maximize simplicity and ease verification. Application software is written as a state transition system: it receives the HSM state and a command as input, and it writes an updated state and output in response.

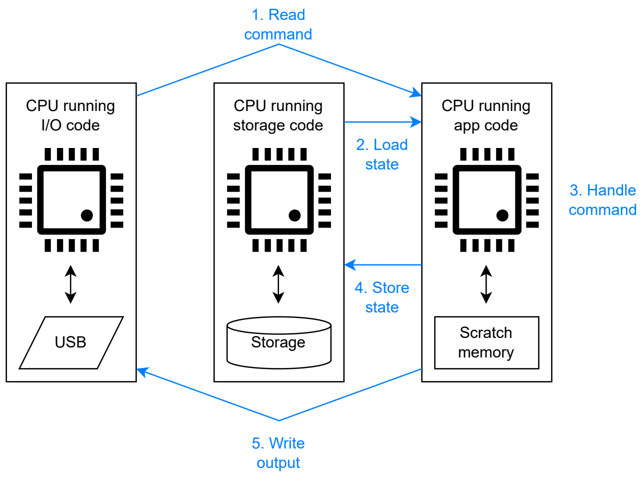

Figure 2. A logical view of K2's architecture contains three separate CPUs that communicate. The first performs I/O with the host machine; the second manages persistent storage; the third performs sensitive computation over sensitive state.

Figure 2 shows a logical view of the design: I/O, storage, and computation over secret state are split onto separate systems-on-a-chip (each with their own CPU, memory, peripherals, etc.) that communicate with each other. Commands from the host are processed as follows: the I/O CPU reads in a command; then, the storage CPU reads in the persistent state; both are sent to the app CPU, which takes the state and command and handles it to produce a new state and an output; the new state is sent back to the storage CPU, which persists it; finally, the output is sent to the I/O CPU, which forwards it to the host.

Figure 3 shows K2's hardware implementation, which realizes this logical design on a single CPU using RISC-V Physical Memory Protection (PMP) hardware and memory-mapped peripherals, along with a small kernel that runs some code in RISC-V M-mode to manage PMP configuration and run each phase of the logical design in sequence on the single CPU, clearing state between phases to ensure isolation.

3.1 Kernel

The software architecture's main goal is to separate and isolate the HSM application code that operates on sensitive data, such that computing over confidential data runs correctly and doesn't leak information via timing side channels. To accomplish this, the design uses a couple hundred lines of kernel code that is independent of the particular HSM application. Tens of lines of code run in RISC-V M-mode to program the PMP and orchestrate switching between phases.

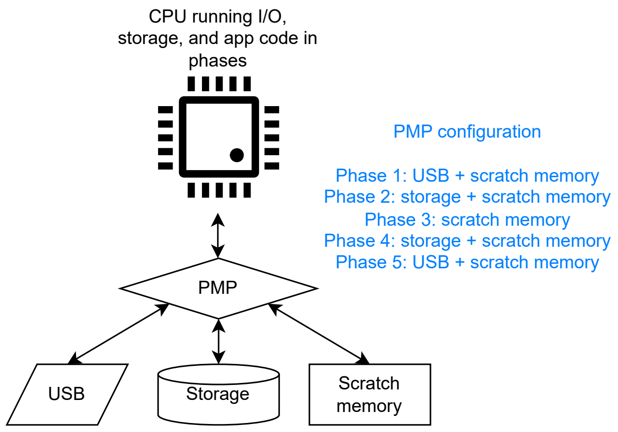
CPU running I/O, storage, and app code in phases
PMP configuration
Phase 1: USB + scratch memory
Phase 2: storage + scratch memory
Phase 3: scratch memory
Phase 4: storage + scratch memory
Phase 5: USB + scratch memory
PMP
USB
Storage
Scratch memory

Figure 3. K2 hardware uses the RISC-V PMP to isolate I/O, storage, and computing over sensitive data.

The phases allow the host to send a command to the HSM, then have the HSM process the command, and finally allow the host to read back the response from the HSM. Each phase runs in U-mode with the appropriate PMP permissions set; a tiny M-mode trampoline ensures appropriate PMP setup and control flow integrity between phases: a RISC-V `ecall` instruction invoked by a phase running in U-mode jumps back into the M-mode kernel, which switches to the next phase. Between each phase transition, the software clears all microarchitectural state in the processor, to avoid leaking secrets across phases. Figure 2 illustrates the different phases of operation in the logical design.

Phase 0: Boot and clear state. In case the device is power cycled when there is sensitive state in RAM or in CPU microarchitectural state, the software clears all volatile state on boot, so that any secrets that might have been present there aren't leaked.

Phase 1: Read command from host. The kernel programs the PMP to disable access to the persistent memory, which is the only place that secret state is stored, while in this phase. Then, the kernel runs an I/O peripheral driver in U-mode, which reads a serialized command from the host machine and writes it into the device's RAM. Once the host has written the command, the software transitions to the next phase. The behavior of the device in this phase is independent of secrets.

Phase 2: Load persistent state. The kernel programs the PMP to disable access to the I/O peripheral, so the device's behavior is no longer affected by inputs from the external world, and the device can't produce externally-observable outputs. It also programs the PMP to enable read-only access to persistent memory. Then, the kernel uses U-mode code to copy state from persistent memory into the RAM, so the RAM now stores both the device's state as well as the command from the host machine.

Phase 3: Run application. The kernel programs the PMP to disable access to both the I/O peripheral and the persistent memory. It then runs the application code (in RISC-V U-mode) that the application developer writes, which reads state and a command from RAM, does some computation, and writes an updated state and an output back to RAM.

Phase 4: Store persistent state. The kernel enables read/write access to the persistent memory, but not the I/O peripheral. It then uses U-mode code to write updated state from RAM back into the persistent memory, using a simple journaling strategy for crash atomicity. Finally, it clears all state, including both microarchitectural state in the CPU as well as the RAM, except the output from the application code (from phase 3).

Phase 5: Write output to host. The kernel configures the PMP to disable access to the persistent memory. It then runs the I/O peripheral driver again (in U-mode), which sends the output over the I/O interface to the host machine. This completes the execution of a single command, and the device returns to phase 0 to begin processing the next command.

3.2 Application code

The developer of an HSM application only writes the software that runs in phase 3, implementing a state transition function with the signature:

```
void main(char *state, char *cmd, char *out)
```

The code operates on a copy of the state in RAM, interprets a serialized command, and updates the state and produces an output in response to the command, both written back to the RAM and read by later phases.

With the phase-based operation of the HSM, this application code runs start-to-finish with no interruptions, with the only observables being the final output value and the end-to-end timing of execution.

The application developer is responsible for writing code that is correct, i.e., updates the state and produces the correct output value, according to the specification. Furthermore, the application developer is responsible for writing code that runs in constant time. K2 provides a tool that verifies this property and helps debug violations.

4 Formal verification

Verification of a K2 HSM relates the HSM implementation to a high-level functional specification such as that in Figure 1 using information-preserving refinement (IPR) [16]. IPR captures correctness and security of the HSM by requiring that an implementation implements the specification but leaks no more information than the specification allows.

K2 decomposes IPR into a number of properties, some that can be proved in a once-and-for-all manner by the developer of the hardware and kernel, and others that can be proved by the application developer on a per-application basis, that together imply IPR.

4.1 Hardware and kernel

The system developer proves functional correctness of all but phase 3, which is when the application software runs: this is mostly straightforward functional correctness verification. On the security and information leakage side, phases 0, 1, and 5 have no access to secrets; merely showing that the PMP is configured appropriately is enough. For phases 2 and 4, which handle secret state, the developer proves that the phases run in constant time, i.e., in a constant number of cycles.

4.2 Application code

The application developer only needs to prove properties about the execution of the application software (phase 3). The functional correctness aspect is standard and can leverage existing tools and libraries for formally-verified software; we build on top of HACL*. The challenge is achieving security and the absence of timing side-channel leakage. To enable this, K2 includes a tool called Chroniton that allows the application developer to verify that their software runs in constant time on the hardware implementation.

Unlike other approaches for verifying timing behavior, Chroniton verifies the timing behavior of software with respect to the particular HSM's hardware implementation (e.g., Verilog code), rather than a proxy like a leakage model. The benefit is that this approach does not require making assumptions about the hardware's timing behavior.

Tools like Icarus Verilog and Verilator are capable of cycle-accurate simulations of processors running such code. These simulators operate on concrete values: every bit is a 0 or a 1, so they are not directly useful for verifying timing properties, such as how execution time depends on the private key.

Chroniton implements a cycle-accurate *symbolic* hardware simulator to reason about processors executing code. Using a symbolic simulator, we can mark state elements in the circuit, e.g., locations in the processor's memory such as those corresponding to `state` and `cmd`, as symbolic variables, and then symbolically execute the entire circuit (as it's executing the application code) and determine whether the circuit's execution time depends on these symbolic variables.

Symbolic simulation can reason about the number of cycles a circuit, starting from the start of phase 3 (when the application code starts executing) to the start of the next phase (when the circuit is no longer computing on secret values).

5 Current status

We currently have a prototype implementation of an HSM following the K2 design, and we have verified some properties of the implementation, focusing on verifying constant-time execution using Chroniton.

The HSM uses mostly off-the-shelf hardware (the Ibex processor and OpenTitan SoC) and software (the cryptography code from HACL*), which is compatible with our approach and tooling.

5.1 Implementation

Our HSM's hardware is a stripped-down version of the OpenTitan open-source root of trust, which uses the Ibex RISC-V processor. On top of that, we have written a couple hundred lines of C and RISC-V assembly to implement the K2 architecture and phase-based design.

We have written a signature oracle (similar to a certificate-signing HSM) as an example application, leveraging the HACL* [50] formally-verified cryptographic library.

5.2 Verification

Our initial efforts on formal verification have been focused on proving timing properties of software running on hardware. To do this, we have implemented the Chroniton tool[1], written in approximately 100 lines of Rosette [47] code. It builds on the Verilog-to-Rosette toolchain from Notary [15, 37], which translates circuits written in Verilog into cycle-accurate Rosette models supporting symbolic execution.

We have applied Chroniton to verify constant-time execution of several software applications against several hardware implementations, including verifying that HACL*'s Ed25519 implementation runs in constant time on our processor. We have also applied Chroniton to other HSM-like devices like the OpenTitan Big Number Accelerator[2] (OTBN), verifying that its X25519 implementation runs in constant time.

6 Related Work

Hardware/software verification. Knox [16] performs end-to-end verification of HSM hardware and software with monolithic reasoning, while modularity is a central goal of K2. A long line of work performs end-to-end verification of functional correctness properties for hardware/software systems, with an emphasis on modular verification [12, 18, 25, 34]. Proving functional correctness does not prove security properties (enforcing correct behavior even if a host machine deviates from the wire protocol, for example) or rule out timing side channels, while these are central security goals of HSMs and K2.

Time protection and enclaves. Research on verified security-focused operating systems proves properties such as process isolation in systems like seL4 [39], mCertiKOS [23], Komodo [27], and Nickel [44]. Recent work proposes *time*

[1]https://github.com/anishathalye/chroniton
[2]https://opentitan.org/book/hw/ip/otbn/index.html

protection to address violations to process isolation from timing side channels [29, 45]. Process isolation (taking into account side channels) is different from K2's goals: a K2 HSM must leak no more information than its specification allows, which is not an isolation-style property. Mapped to the process isolation setting: a HSM software that leaks its private key through timing (or just directly sends it out over the I/O interface) could be strongly isolated from other processes, but is clearly insecure.

Leakage models. One approach for verifying constant-time execution uses leakage models [14]; these are commonly used in verifying constant-time cryptography [13, 19, 22, 26, 43, 50]. Recent work has proposed more sophisticated leakage models [21, 28, 38] and validated hardware against these models through fuzzing [20, 41, 42], sometimes revealing gaps between leakage models and hardware implementations. Today's verified cryptographic software is not verified against these sophisticated leakage models. In recent work, Wang et al. [49] formally verify simple open-source RISC-V processors against leakage contracts. Today's leakage models reason only about the CPU/ISA and do not support reasoning about hardware-level timing behavior of other parts of the stack, such as code that uses control and status registers or interacts with peripherals such as storage or I/O in a system-on-a-chip, whereas our approach with Chroniton does support this.

7 Discussion

What workloads fit the K2 model? K2 supports traditional HSM applications, where the state and functionality is relatively simple. The entire premise of HSMs is to factor out the key security-related functionality, so these applications are going to be simple by design. The current K2 design requires copying the entire state of the HSM for every operation, so it is not a good fit for functionalities that require large amounts of state such as table lookup into a large database.

Does the approach scale to more sophisticated CPUs? Our implementation currently uses a simple two-stage pipelined RISC-V CPU used in the OpenTitan, but the CPU could be replaced with a much more powerful one. One challenge might be that the verification approach doesn't scale to more sophisticated hardware. So far, we have successfully applied Chroniton to software running on the biRISC-V CPU, which is a 6-stage dual-issue processor that is more complex than the Ibex processor used in K2. We have not yet tried applying Chroniton to, e.g., out-of-order processors.

How much does performance matter in this context? For end-user devices like U2F tokens, as well as for certain server-side HSM applications where the request rate is low, the slowdown from K2's relatively underpowered CPU as well as architecture with strict time-partitioning of I/O, storage, and computation, is tolerable. In some applications, high performance is a requirement, such as CA certificate signing: Let's Encrypt performed approximately 450 signatures per second as of September 2019, split over 4 HSMs [10]. When using K2 HSMs, a greater number can be used to provide the necessary throughput.

Acknowledgements

This work was supported by NSF award CNS-2225441.

References

[1] 2004. CVE-2004-0320. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-0320.

[2] 2015. YSA-2015-1. https://developers.yubico.com/ykneo-openpgp/SecurityAdvisory%202015-04-14.html.

[3] 2018. CVE-2018-6875. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6875.

[4] 2018. YSA-2018-01. https://www.yubico.com/support/security-advisories/ysa-2018-01/.

[5] 2019. CVE-2019-18671. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-18671.

[6] 2019. CVE-2019-18672. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-18672.

[7] 2020. YSA-2020-04. https://www.yubico.com/support/security-advisories/ysa-2020-04/.

[8] 2021. CVE-2021-31616. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31616.

[9] 2021. WhatsApp Security Whitepaper: Security of End-to-End Encrypted Backups. https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf.

[10] Josh Aas, Richard Barnes, Benton Case, Zakir Durumeric, Peter Eckersley, Alan Flores-López, J. Alex Halderman, Jacob Hoffman-Andrews, James Kasten, Eric Rescorla, Seth Schoen, and Brad Warren. 2019. Let's Encrypt: An Automated Certificate Authority to Encrypt the Entire Web. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*. London, United Kingdom, 2473–2487.

[11] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. 2002. The EM Side-Channel(s). In *Proceedings of the 2002 IACR Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Redwood City, CA.

[12] Eyad Alkassar, Wolfgang J. Paul, Artem Starostin, and Alexandra Tsyban. 2010. Pervasive Verification of an OS Microkernel: Inline Assembly, Memory Consumption, Concurrent Devices. In *Proceedings of the 3rd Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*. Edinburgh, United Kingdom, 71–85.

[13] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *Proceedings of the 23rd International Conference on Fast Software Encryption (FSE)*. Bochum, Germany, 163–184.

[14] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *Proceedings of the 25th USENIX Security Symposium*. Austin, TX, 53–70.

[15] Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2019. Notary: A Device for Secure Transaction Approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. Huntsville, Ontario, Canada, 97–113.

[16] Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying Hardware Security Modules with Information-Preserving Refinement. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA, 503–519.

[17] Jean-Baptiste Bédrune and Gabriel Campana. 2019. Everybody be Cool, This is a Robbery! https://donjon.ledger.com/BlackHat2019-presentation/.

[18] William R. Bevier, Warran A. Hunt Jr., J. Strother Moore, and William D. Young. 1989. An Approach to Systems Verification. *Journal of Automated Reasoning* 5, 4 (Dec. 1989), 411–428.

[19] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *Proceedings of the 26th USENIX Security Symposium*. Vancouver, Canada, 917–934.

[20] Pablo Buiras, Hamed Nemati, Andreas Lindner, and Roberto Guanciale. 2021. Validation of Side-Channel Models via Observation Refinement. In *Proceedings of the 42th IEEE/ACM International Symposium on Microarchitecture*. Athens, Greece, 578–591.

[21] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. London, United Kingdom, 913–926.

[22] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: A DSL for Timing-Sensitive Computation. (June 2019), 174–189.

[23] David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-End Verification of Information-Flow Security for C and Assembly Programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Santa Barbara, CA, 648–664.

[24] Filippo Cremonese. 2020. Security Analysis of the Solo Firmware. https://blog.doyensec.com/2020/02/19/solokeys-audit.html.

[25] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification across Software and Hardware for a Simple Embedded System. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Virtual conference.

[26] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*. San Francisco, CA, 73–90.

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

[28] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*. Virtual conference, 1868–1883.

[29] Gernot Heiser, Toby Murray, and Gerwin Klein. 2020. Towards Provable Timing-Channel Prevention. *ACM SIGOPS Operating System Review* 54, 1 (aug 2020), 1–7.

[30] Mason Hemmel, Jason Meltzer, Thomas Pornin, Keegan Ryan, Javed Samuel, David Wong, Rob Wood, and Greg Worona. 2018. Android Cloud Backup/Restore. https://research.nccgroup.com/wp-content/uploads/2020/07/Final_Public_Report_NCC_Group_Google_EncryptedBackup_2018-10-10_v1.0.pdf.

[31] Michael Hutter and Jörn-Marc Schmidt. 2013. The Temperature Side Channel and Heating Fault Attacks. In *Proceedings of the 12th Smart Card Research and Advanced Application Conference (CARDIS)*. Berlin, Germany, 219–235.

[32] Jan Jancar, Vladimir Sedlacek, Petr Svenda, and Marek Sys. 2020. Minerva: The curse of ECDSA nonces (Systematic analysis of lattice attacks on noisy leakage of bit-length of ECDSA nonces). *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020, 4 (2020), 281–308.

[33] Ivan Krstić. 2016. Behind the Scenes with iOS Security. https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf.

[34] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. 2019. Verified Compilation on a Verified Processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Phoenix, AZ, 1041–1053.

[35] Rita Mayer-Sommer. 2000. 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)*. Worcester, MA, 78–92.

[36] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. 2020. TPM-FAIL: TPM meets Timing and Lattice Attacks. In *Proceedings of the 29th USENIX Security Symposium*. Virtual conference, 2057–2073.

[37] Noah Moroze, Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. rtlv: push-button verification of software on hardware. In *Proceedings of the 5th Workshop on Computer Architecture Research with RISC-V (CARRV)*. Virtual conference.

[38] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. 2022. Axiomatic Hardware-Software Contracts for Security. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*. New York, NY, 72–86.

[39] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: from General Purpose to a Proof of Information Flow Enforcement. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*. San Francisco, CA, 415–429.

[40] OASIS PKCS 11 Technical Committee. 2020. PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 3.0. https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/os/pkcs11-curr-v3.0-os.html.

[41] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2022. Revizor: Testing Black-Box CPUs against Speculation Contracts. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Lausanne, Switzerland, 226–239.

[42] Oleksii Oleksenko, Marco Guarnieri, Boris Kopf, and Mark Silberstein. 2023. Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing. In *Proceedings of the 44th IEEE Symposium on Security and Privacy*. San Francisco, CA, 1737–1752.

[43] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph Wintersteiger, and Santiago Zanella-Beguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*. San Francisco, CA, 983–1002.

[44] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. 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)*. Carlsbad, CA, 287–306.

[45] Robert Sison, Scott Buckley, Toby Murray, Gerwin Klein, and Gernot Heiser. 2023. Formalising the Prevention of Microarchitectural Timing Channels by Operating Systems. In *Proceedings of the 25th International Symposium on Formal Methods (FM)*. Lübeck, Germany, 103–121.

[46] Sampath Srinivas, Dirk Balfanz, Eric Tiffany, and Alexei Czeskis. 2016. Universal 2nd Factor (U2F) Overview. https://fidoalliance.org/specs/fido-u2f-v1.1-id-20160915/fido-u2f-overview-v1.1-id-20160915.pdf.

[47] Emina Torlak and Rastislav Bodik. 2014. 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)*. Edinburgh, United Kingdom, 530–541.

[48] Florian Uekermann. 2016. Buggy OTP slot range check. https://github.com/Nitrokey/nitrokey-pro-firmware/issues/4.

[49] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. 2023. Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts. https://arxiv.org/abs/2305.06979.

[50] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.