Mechanized Proofs That Hardware Is Safe From Timing Attacks

Faye Duxovni (dukhovni@mit.edu)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

June 2018

© 2018 the Massachusetts Institute of Technology.
All rights reserved.

Faye Duxovni
Department of Electrical Engineering and Computer Science
May 25, 2018

certified by

Adam Chlipala
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor
May 25, 2018

accepted by

Katrina LaCurts
Chair, Masters of Engineering Thesis Committee
May 25, 2018

Abstract

A recurring problem in cryptography engineering is the potential for secret data to be leaked through aspects of software and hardware that are orthogonal to functional correctness. In particular, much effort is put into writing cryptography code whose timing behavior — how many CPU clock cycles it takes to complete a given cryptographic operation — is independent of any secret inputs to that operation. This is a difficult problem because it depends not only on the code itself, but also on various optimizations such as branch prediction and memory caching implemented by the underlying hardware the program runs on. We make use of Kami, a domain-specific language for describing and formally verifying hardware modules, to build a system for constructing machine-checked proofs that a given piece of code running on a given RISC-V CPU design will not leak secret inputs through timing behavior. Our system allows software and hardware to be analyzed and verified independently, and we prove that any combination of software and hardware that meet our validation criteria will be safe from timing-based side channels. We demonstrate an example of validating a real cryptographic program and a concrete RISC-V CPU using our system, illustrating the applicability of our tools and laying the groundwork for validating more complex programs and CPUs.

Thesis Supervisor: Adam Chlipala
Associate Professor of Electrical Engineering and Computer Science
Department of Electrical Engineering and Computer Science

Acknowledgements

I'd like to thank the following people for their invaluable help over the course of this project:

- Adam Chlipala, for giving me interesting problems to chase, and for being patient and supportive when I got lost in the weeds
- Joonwon Choi and Muralidaran Vijayaraghavan, for answering all my questions about Kami and helping me spend less time fighting with my tools
- Andres Erbsen, for helping me perform a much more interesting case study, brainstorming with me about the project as a whole, and sharing his thesis template
- ikdc, for suggestions and sympathy during my countless struggles with Coq
- Cassandra McClure, for indispensible productivity advice and generous emotional support during a very complicated year

Contents

Chapter 1

Introduction

1.1 Formal Verification

When one is developing software, it is important to ensure that the software is actually correct — that is, that for all possible inputs it may receive, it functions in accordance with a given specification of its desired behavior. Developers often attempt to check correctness through manual code review, hand-written test cases, or automatically generated tests. However, these methods do not comprehensively guarantee correctness and are prone to oversights. A code review is only as reliable as the person doing the reviewing, and hand-written or automatically generated test cases merely show that code behaves correctly on a hopefully representative sample of possible inputs.

In order to provide stronger assurances of correctness, much work is being done today on formally verifying the correctness of software. In formal verification, a programmer constructs mathematical proofs that their code obeys a desired specification, guaranteeing that the code is free from bugs and always produces correct results. Rather than writing these proofs with pencil and paper and checking manually that the proofs are logically sound, formal verification uses machine-checked proofs written in and validated by a computer proof assistant such as Coq,

Isabelle, or Agda. Thus, as long as the proof assistant itself is correct, the system of formal logic it implements is consistent, and specifications are expressed properly within the proof assistant, any code that has been proved to obey a specification is guaranteed to satisfy that specification in all cases. Formal verification is already being used to verify code with real-world applications, from file systems[1] to operating-system kernels[2] to C compilers[3].

1.2 Side Channels

Unfortunately, even writing proper specifications for code is deceptively complex. There are many properties that a program needs to satisfy to be safe for actual production use, and a developer can easily overlook important requirements. Sometimes, a specification may fail to address important edge cases, allowing programs to behave in unexpected ways. Specifications may also fail by relying on faulty abstractions and making incorrect assumptions about how programs interact with their environments. This problem of faulty abstractions is quite common with regard to physical hardware; a specification may attempt to enforce functional correctness by fully specifying the relationship between inputs and outputs of a program, but even if a program is functionally correct, it may not be safe to run on an actual computer.

In particular, many implementations of cryptographic algorithms have been found to be vulnerable to "side-channel attacks," which exploit properties of computer hardware to leak secret data in ways that aren't accounted for at the level of software. Side-channel attacks may rely on analysis of complex physical properties such as power consumption[4], electromagnetic emissions[5], or even sounds produced by computers running cryptography code[6]. While programmers tend to model computers as abstract systems that execute logical rules to transition between states, only

1. Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. "Using Crash Hoare Logic for Certifying the FSCQ File System". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: ACM, 2015, pp. 18–37. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815402. URL: http://doi.acm.org/10.1145/2815400.2815402

2. Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels". In: *In Proc. 2016 USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA, Nov. 2016, pp. 653–669

3. Xavier Leroy. "A Formally Verified Compiler Back-end". In: *J. Autom. Reason.* 43.4 (Dec. 2009), pp. 363–446. ISSN: 0168-7433. DOI: 10.1007/s10817-009-9155-4. URL: http://dx.doi.org/10.1007/s10817-009-9155-4

4. Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. "Introduction to differential power analysis". In: *Journal of Cryptographic Engineering* 1.1 (Apr. 2011), pp. 5–27. ISSN: 2190-8516. DOI: 10.1007/s13389-011-0006-y. URL: https://doi.org/10.1007/s13389-011-0006-y

interacting with the outside world through predefined channels, real hardware made of actual electrical components can have much more complex behaviors that leak information in unforeseen ways.

In this project, we will not concern ourselves with side channels based on physical properties of hardware, as these are difficult to formally reason about without accurate formal models of the relevant physical systems. Instead, we will focus on timing-based side channels and cache-based side channels.

Timing-based side channels leak secrets through the number of CPU clock cycles a program takes to execute, which may vary depending on secret inputs[7]. For example, consider a modular exponentiation algorithm based on repeated squaring. As the algorithm goes through each bit of the exponent, if it only performs a multiplication by the base when the current bit of the exponent is 1, then the presence or absence of this multiplication instruction may affect the number of CPU clock cycles the algorithm takes, thus leaking the value of the exponent.

Related to timing-based side channels are cache-based side channels, which leak secrets through the pattern of a program's memory accesses[8]. When programs access the computer's memory, the segments of memory they access are stored in a cache which operates more quickly than regular memory, allowing for faster repeated lookups. However, when two processes running on the same computer share a cache, some of one process' cached data may be evicted when the other process loads data into the cache. This eviction can then be observed by measuring how much time it takes to read the same memory again, as the read will be faster if the data is still in the cache. Thus, one program can observe which memory addresses another program accesses, by observing which parts of the cache are evicted at what times. This pattern of memory accesses may depend on the values of cryptographic secrets, allowing a

5. Vincent Carlier, Hervé Chabanne, Emmanuelle Dottax, and Hervé Pelletier. *Electromagnetic Side Channels of an FPGA Implementation of AES*. Cryptology ePrint Archive, Report 2004/145. `https://eprint.iacr.org/2004/145`. 2004

6. Daniel Genkin, Adi Shamir, and Eran Tromer. *RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis*. Cryptology ePrint Archive, Report 2013/857. `https://eprint.iacr.org/2013/857`. 2013

7. Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances In Cryptology - CRYPTO '96, LNCS 1109*. 1996, pp. 104–113

8. Colin Percival. "Cache missing for fun and profit". In: *In Proc of BSDCan 2005*

malicious program to steal secrets from programs running on the same computer.

1.3 Goals

Developers of cryptography software use various techniques to prevent timing-based side channels and cache-based side channels. For example, to avoid leaking information through delays in a CPU's pipeline caused by branch-prediction misses, it is considered good practice to avoid branching on secret data. To avoid leaking information through memory accesses, a program may deliberately re-read an entire region of memory whenever it needs to look up a single address in that region. However, it is important that we be able to verify that these techniques are being used correctly and comprehensively, fully preventing any information leakage through timing behavior or memory-access patterns.

Unlike side channels involving physical properties of hardware, timing-based side channels and cache-based side channels are relatively amenable to formal reasoning, as they only depend on the logical behavior of the CPU and memory. This is a much simpler and more discrete system that is easier to model precisely. Thus, by proving theorems about the scheduling of a program's instructions by the CPU and the pattern of the program's memory accesses, one can formally prove that the program is safe from timing-based and cache-based side channels.

Our goal, then, is to create a framework for formally proving that code and CPU designs are safe from timing and cache side channels, and to use this framework to construct safety proofs for realistic example systems. We would like this framework to apply to software and hardware in an independent fashion, so that introducing a new implementation of a CPU architecture does not necessitate re-verifying already-verified code, and vice versa.

Chapter 2

Kami

Reasoning in this way about how code runs on real hardware requires formal models of CPUs and memory. Fortunately, such models already exist: the Kami project, created by the Programming Languages and Verification research group at CSAIL, provides a framework for designing and verifying hardware circuits within the Coq computer proof assistant[1].

1. Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. "Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification". In: *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP'17)*. Sept. 2017

2.1 Motivation

The goal of Kami is to allow reasoning about the behavior of hardware at a higher level of abstraction than the level of circuits and gates, and facilitate a style of verification similar to what is commonly done for verifying software. The desired specification for a hardware module can be written as a functional program within the proof assistant, and a more concrete implementation can be proved equivalent to the specification. The verified module can then be extracted into a format suitable for fabrication.

2.2 Structure and Semantics

Kami provides a domain-specific language within the Coq proof assistant for describing hardware as a collection of modules. Each module consists of a set of registers which

store data; rules which may fire if specific guard conditions are satisfied; and methods which may be called by that module or other modules. The syntax and semantics of Kami closely follow that of the Bluespec hardware design language[2]; Kami modules can be extracted into Bluespec source code and compiled with the Bluespec compiler, which produces a low-level description of a physical circuit in a register-transfer language (RTL). This physical circuit implements the logic of the modules' rules and methods, and decides which rules fire on which clock cycles based on the guard conditions governing when each rule may fire. This circuit may then be simulated on an FPGA or laid out with actual physical components.

A key feature of Kami is that it allows modules to be verified against specifications, allowing similar sorts of formal correctness proofs for hardware as for software. These specifications take the form of special Kami modules whose rules make use of the full expressive power of the Gallina functional language used within Coq, rather than the much simpler language that ordinary Kami modules are constrained to. Such modules cannot be extracted to Bluespec code but can provide a clear and concise description of a module's desired functionality, and an extractable module can be proved to only behave in ways the specification allows.

Furthermore, in a hardware system composed of multiple interconnected modules, Kami allows for easy replacement of one module with little added verification burden. As long as the new submodule has been proved to behave only in ways the old module behaved, Kami provides a theorem stating that the entire new system of modules behaves only in ways the old system behaved. This modularity allows developers to redesign complex systems on a module-by-module basis without needing to re-verify the entire system each time a single module is replaced.

2. N. Dave, Arvind, and M. Pellauer. "Scheduling as Rule Composition". In: *2007 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007)*. May 2007, pp. 51–60. DOI: 10.1109/MEMCOD.2007.371249

Example

To illustrate how hardware modules are represented in Kami, figure 2.1 shows an example Kami module implementing a producer-consumer pattern.

The module consists of three submodules, `producer`, `consumer`, and `queue`, which are concatenated to form the complete `prodQCons` module.

`producer` contains one register holding a word of size `cSz` bits, and one rule `produce` which reads the value from the register, calls the `enq` method with that value as an argument, and writes the incremented value back to the register. Note that `producer` is parameterized over the value of `cSz`; for any value n, `producer` n will be a module which operates on n-bit words.

`consumer` contains only a single rule `consume`, which calls the `deq` method and then calls `output` with the value returned by `deq`.

`queue` contains three registers: `elts`, which holds an array of length $2^{\texttt{qSz}}$ containing elements of type `dataType`; and `head` and `tail`, which hold `qSz` $+ 1$-bit words. `queue` also has methods `enq` and `deq`, which use `head` and `tail` as indices into `queue` and implement a circular queue.

Because `producer` calls a method named `enq`, and `queue` defines a method named `enq`, when these two modules are concatenated together, a firing of rule `produce` will result in the operations defined in `enq` being performed in the concatenated module. (To be well-formed, a module must not define multiple methods with the same name.) In the concatenated module `prodQCons`, the only method that is called but not defined is `output`; therefore, calls to `output` will be the only externally visible behavior of the module under the Kami semantics.

Note also that the `enq` method *asserts* that the circular queue is not full, and the `deq` method asserts that the queue is not empty. Under the Kami semantics, these asserts provide guard conditions which must be satisfied in

order for the method to be called. Therefore, if the guard condition of `enq` is not met, a rule which calls `enq` cannot fire.

To summarize, the behavior of `prodQCons` is as follows: On every clock cycle, rule `produce` may fire if the queue is not full, and rule `consume` may fire if the queue is not empty. The Kami semantics place no constraints on whether these rules will fire together, separately, in any given order, or at all. All that is required is that any simultaneous execution of rules be consistent with some serial execution.

```
Definition producer cSz := MODULE {
  Register ”counterReg” : Bit cSz <- Default
  with Rule ”produce” :=
    Read val <- ”counterReg”;
    Call ”enq”(#val);
    Write ”counterReg” <- #val + $1;
    Retv
}.
Definition consumer cSz := MODULE {
  Rule ”consume” :=
    Call val: Bit cSz <- ”deq”();
    Call ”output”(#val);
    Retv
}.
Definition queue dataType qSz := MODULE {
 Register ”elts” : Vector dataType qSz <- Default
  with Register ”head”: Bit (qSz+1) <- Default
  with Register ”tail”: Bit (qSz+1) <- Default
  with Method ”enq”(d : dataType) : Unit :=
    Read elts <- ”elts”;
    Read head <- ”head”;
    Read tail <- ”tail”;
    Assert (#tail + $(1<<qSz) != #head);
    Write ”elts” <- #elts@[#head <- #d];
    Write ”head” <- #head + $1;
    Retv
  with Method ”deq”() : dType :=
    Read elts <- ”elts”;
    Read head <- ”head”;
    Read tail <- ”tail”;
    Assert (#tail != #head);
    Write ”tail” <- #tail + $1;
    Return #elts@[#tail]
}.
Definition prodQCons cSz qSz :=
 producer cSz + queue (Bit cSz) qSz + consumer cSz.
```

Figure 2.1: Example producer-consumer module in Kami, taken from Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. “Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification”. In: *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP’17)*. Sept. 2017

2.3 CPU Designs in Kami

To demonstrate the applicability of Kami for real-world hardware design, and to illustrate how complex hardware modules are verified in Kami, several implementations of the RISC-V CPU architecture[3] have been designed and verified in Kami as case studies. The simplest of these is an unpipelined CPU where each instruction is completed in a single cycle, and memory requests receive instant responses. There are also more complex RISC-V implementations with three-stage and four-stage pipelines, which have been proved to behave consistently with the unpipelined version. Our project makes heavy use of these Kami modules and is focused on the RISC-V ISA and RISC-V CPU implementations.

3. Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. RISC-V Foundation, May 2017. URL: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

Chapter 3

Reasoning About Timing Behavior in Kami

The relatively high level at which the Kami language describes hardware, while useful for simplifying the work of hardware design, poses significant challenges to this project. In particular, reasoning about the timing behavior of code running on a Kami CPU is difficult, because a Kami module has no well-defined timing behavior in and of itself. As described above, a Kami module simply provides a set of rules, which include guard conditions governing when the rules may fire. The actual timing of rule firings is determined at the RTL level by the scheduler circuit produced by the Bluespec compiler, which has not yet been formalized at the time of this writing.

3.1 Proof Structure

To work around this difficulty, and also to maximize the usefulness of the security proofs we create, we establish a clear set of boundaries in our reasoning between machine code, RISC-V CPU implementations, and Bluespec compiler implementations. In particular, we define a condition that machine code should be expected to satisfy, a condition that Kami RISC-V processor implementations should be

expected to satisfy, a condition that Kami memory implementations should be expected to satisfy, and a condition that Bluespec compilers should be expected to satisfy. This allows reasoning about the system of code, CPU, memory, and Bluespec compiler in a modular fashion, verifying each component independently of the others. We then prove that if each component satisfies its requirements, the given machine code running on the given CPU+memory module compiled with the given compiler will not leak secret data through timing behavior.

For the sake of simplicity, our model systems read in all secret data through a special FROMHOST CPU instruction, which receives a word of data from some external source and writes it to a register. Thus, the goal is that the timing behavior of the full processor+memory system should be completely independent of the values returned by FROMHOST calls. There is an analogous TOHOST method which sends data back to the external source; this is the only channel by which we will allow secret data to be sent.

Our ultimate goal is to prove that the externally visible timing behavior of a compiled RTL hardware module is independent of values received through FROMHOST calls. We express this proposition in the following way: under the semantics of the register-transfer language, an RTL module can be said to produce a trace of operations it performs — updating registers, sending signals to external modules, and so forth. At the RTL level, this trace is fully determined by the initial register contents and the values received from external method calls. We provide a censorship function, which takes a trace and hides various parts of the trace that would not be observable externally. Our safety condition, then, is that given any trace which receives a given sequence of values in FROMHOST calls, we can choose another arbitrary sequence of FROMHOST values, and find a second trace which has the same initial register values and receives our chosen sequence of FROMHOST values, such that the

censored first trace is identical to the censored second trace. Thus, the externally visible parts of the trace do not depend on the FROMHOST values, since changing the FROMHOST values yields a trace which is identical to the original under censorship.

The theorems we prove about machine code, and about CPU implementations, follow the same pattern of trace censorship. Each layer has its own type of trace, its own semantics, and its own censorship function. The exact details of what is censored, and what is not, at each level are chosen to facilitate the proofs at adjacent levels. To prove that a Kami CPU implementation is safe within our framework, we prove that given *any* code which satisfies the code-level censorship theorem, our chosen CPU running that code satisfies the CPU-level censorship theorem. Once the RTL layer is formally specified in future work, we will prove that given any Kami CPU which satisfies the CPU-level censorship theorem, the compiled RTL version of that Kami CPU will satisfy the RTL-level censorship theorem. Thus, we can reason about code, Kami CPU implementations, and Kami compilers independently.

Essentially, our proofs for each level have the following structure:

$$
\begin{aligned}
\text{Instructions} &\in \text{Set} \\
\text{State} &\in \text{Set} \\
\text{TraceElement} &\in \text{Set} \\
\text{hasTrace} &\in \text{Instructions} \to \text{State} \to \text{list}(\text{TraceElement}) \to \text{Prop} \\
\text{censorTrace} &\in \text{list}(\text{TraceElement}) \to \text{list}(\text{TraceElement}) \\
\text{extractFhTrace} &\in \text{list}(\text{TraceElement}) \to \text{list}(\text{word32})
\end{aligned}
$$

We have instructions and state (the distinction is fairly arbitrary); a datatype to represent execution traces; a predicate for when a given instruction/state pair produces a

given trace; and functions to censor a trace and to extract a sequence of FROMHOST values from a trace.

Our safety property that an instruction/state pair needs to satisfy is then

$$
\begin{aligned}
&\mathsf{hiding}(i, s) := \\
&\quad \forall t\, f, \\
&\qquad \mathtt{hasTrace}(i, s, t) \rightarrow \\
&\qquad \mathsf{extractFhTrace}(t) = f \rightarrow \\
&\qquad \forall f', \\
&\qquad\quad \mathsf{length}(f) = \mathsf{length}(f') \rightarrow \\
&\qquad\quad \exists t', \\
&\qquad\qquad \mathtt{hasTrace}(i, s, t') \wedge \\
&\qquad\qquad \mathsf{censorTrace}(t) = \mathsf{censorTrace}(t') \wedge \\
&\qquad\qquad \mathsf{extractFhTrace}(t') = f'
\end{aligned}
$$

3.2 The Machine Code Layer

We begin by defining a notion of a behavior trace for a piece of RISC-V machine code, considered independently of any specific hardware. Our "instructions" are the contents of a program memory, an array of RISC-V machine instructions. Our "state" consists of a register file, a program counter indicating our current location in program memory, and a data memory. We define a trace as a list of elements of a TraceEvent datatype, representing events that can happen on a given clock cycle:

$$
\begin{aligned}
&c \in \mathsf{ProgramCounter} \\
&a \in \mathsf{Address} \\
&v \in \mathsf{Data} \\
&b \in \mathsf{Bool}
\end{aligned}
$$

$$\begin{aligned}\text{TraceEvent} := \ & \text{Rd}(c, a, v) \mid \text{RdZ}(c, a) \mid \text{Wr}(c, a, v) \\ & \mid \text{FromHost}(c, v) \mid \text{ToHost}(c, v) \mid \text{Branch}(c, b) \\ & \mid \text{Nm}(c) \mid \text{Nop}(c)\end{aligned}$$

- **Rd** `(pc : address) (laddr : address) (val : data)` (At program location `pc`, value `val` was read from memory address `laddr`.)
- **RdZ** `(pc : address) (laddr : address)` (A read was made from memory address `laddr` to register `0`. The RISC-V spec designates register `0` as special: reads from register `0` always return `0` and writes are silently discarded. Since the value loaded from memory is never used, some CPU designs may not actually perform the memory access, so this case must be treated as different from the general case of reading a value from memory, and the actual value at the memory address is completely irrelevant.)
- **Wr** `(pc : address) (saddr : address) (val : data)` (Value `val` was written to memory address `saddr`.)
- **FromHost** `(pc : address) (val : data)` (Value `val` was received via a `FROMHOST` call.)
- **ToHost** `(pc : address) (val : data)` (Value `val` was sent via a `TOHOST` call.)
- **Branch** `(pc : address) (taken : bool)` (There was a conditional branch at program location `pc`. The value of `taken` indicated whether the branch was taken or not.)
- **Nm** `(pc : address)` (A "normal" instruction with no externally visible effects; for instance, performing an arithmetic operation on register contents.)
- **Nop** `(pc : address)` (A cycle where no instruction is executed. This does not correspond to anything in

the RISC-V semantics; it exists to make it easier to draw a correspondence later on between machine-code traces and Kami traces, as the Kami semantics allow dead cycles where no rule fires.)

We then define an inductive predicate for what it means for a particular program memory, starting from a particular register file, program counter, and data memory, to have a particular trace:

$$\frac{r \in \mathsf{Regfile} \quad p \in \mathsf{ProgMem} \quad c \in \mathsf{ProgramCounter} \quad m \in \mathsf{Memory}}{\mathtt{hasTrace}(r, p, c, m, [])}$$

This rule means that after we've built a trace up to some point, we can always end the trace at that point. This is important because our current semantics for RISC-V have no halt condition, other than encountering a malformed instruction that doesn't have any opcode we know how to handle. Thus, we conceptualize a program as executing forever, and any finite prefix of its execution trace is a valid trace produced by the program.

$$\frac{\mathtt{hasTrace}(r, p, c, m, t)}{\mathtt{hasTrace}(r, p, c, m, \mathsf{Nop}(p) :: t)}$$

$$\frac{\mathtt{hasTrace}(r[d \to m[a]], p, c+4, m, t) \quad \mathsf{OpType}(p[c]) = \mathsf{opLd} \quad a = \mathsf{LdAddr}(p[c], r) \quad d = \mathsf{Dest}(p[c]) \quad d \neq 0}{\mathtt{hasTrace}(r, p, c, m, \mathsf{Rd}(p, a, m[a]) :: t)}$$

$$\frac{\mathtt{hasTrace}(r, p, c+4, m, t) \quad \mathsf{OpType}(p[c]) = \mathsf{opLd} \quad a = \mathsf{LdAddr}(p[c], r) \quad \mathsf{Dest}(p[c]) = 0}{\mathtt{hasTrace}(r, p, c, m, \mathsf{RdZ}(p, a) :: t)}$$

$$\frac{\mathtt{hasTrace}(r, p, c+4, m[a \to r[s]], t) \quad \mathsf{OpType}(p[c]) = \mathsf{opSt} \quad a = \mathsf{StAddr}(p[c], r) \quad s = \mathsf{Src}(p[c])}{\mathtt{hasTrace}(r, p, c, m, \mathtt{Wr}(p, a, r[s]) :: t)}$$

$$\frac{\mathtt{hasTrace}(r[d \to v], p, c+4, m, t) \quad \mathsf{OpType}(p[c]) = \mathsf{opFh} \quad d = \mathsf{Dest}(p[c]) \quad v \in \mathsf{Data}}{\mathtt{hasTrace}(r, p, c, m, \mathtt{FromHost}(p, v) :: t)}$$

$$\frac{\mathtt{hasTrace}(r, p, c+4, m, t) \quad \mathsf{OpType}(p[c]) = \mathsf{opTh} \quad s = \mathsf{Src}(p[c])}{\mathtt{hasTrace}(r, p, c, m, \mathtt{ToHost}(p, r[s]) :: t)}$$

$$\frac{\mathtt{hasTrace}(r, p, \mathsf{NextPc}(c, p[c], r), m, t) \quad \mathsf{Opcode}(p[c]) = \mathsf{opBRANCH}}{\mathtt{hasTrace}(r, p, c, m, \mathtt{Branch}(p, \mathsf{BranchTaken}(p[c], r)) :: t)}$$

$$\frac{\mathtt{hasTrace}(r[d \to \mathsf{ExecVal}(p[c], r)], p, \mathsf{NextPc}(c, p[c], r), m, t) \quad \mathsf{OpType}(p[c]) = \mathsf{opNm} \quad \mathsf{Opcode}(p[c]) \neq \mathsf{opBRANCH} \quad d = \mathsf{Dest}(p[c])}{\mathtt{hasTrace}(r, p, c, m, \mathsf{Nm}(p) :: t)}$$

Note that under this definition, the trace produced by a given program is fully determined (aside from inserting/removing `Nop` events at any point) by the starting conditions and the values received via `FromHost` events.

Following the structure laid out in Section 3.1, we then need to define a censorship function and a function to extract FROMHOST values. To censor a trace element, we simply set to zero any values that are not visible to adversaries and might reasonably hold secret data — i.e., the values (but not the addresses) in `Rd` and `Wr` events, and

the values in FromHost and ToHost events. (We assume that FROMHOST and TOHOST are our communication channel with a trusted entity which supplies us with secrets and receives the results of our computations on those secrets.) Our extractFhTrace function simply compiles a list of the data values from all FromHost events in a trace.

To illustrate our definitions of traces and censorship, consider the following example code:

```
x = fromHost();
y = *x;
z = fromHost();
y += z;
toHost(y);
```

If this code were compiled and loaded into program memory, it might look as follows:

```
0x1234  FROMHOST x1
0x1238  LW x1 x2 $0
0x123c  FROMHOST x3
0x1240  ADD x2 x3 x2
0x1244  TOHOST x2
```

Now, under the semantics we've defined, consider possible traces that might be produced by this program, starting at instruction address 0x1234 with a zeroed register-file and the following memory contents:

```
0x0badcafe  0x11111111
...
0xdeadbeef  0x20202020
```

Potential traces include

- []
- [Nop(0x1234), Nop(0x1234), FromHost(0x1234, 0xfeedbacc), Nop(0x1238)]

- [FromHost(0x1234, 0x0badcafe),
  Rd(0x1238, 0x0badcafe, 0x11111111),
  FromHost(0x123c, 0x12345678), Nm(0x1240),
  ToHost(0x1244, 0x23456789)]
- [FromHost(0x1234, 0xdeadbeef),
  Rd(0x1238, 0xdeadbeef, 0x20202020),
  FromHost(0x123c, 0x04040404), Nm(0x1240),
  ToHost(0x1244, 0x24242424)]

The censored versions of these traces are as follows:

- []
- [Nop(0x1234), Nop(0x1234),
  FromHost(0x1234, 0x0), Nop(0x1238)]
- [FromHost(0x1234, 0x0),
  Rd(0x1238, 0x0badcafe, 0x0),
  FromHost(0x123c, 0x0), Nm(0x1240),
  ToHost(0x1244, 0x0)]
- [FromHost(0x1234, 0x0),
  Rd(0x1238, 0xdeadbeef, 0x0),
  FromHost(0x123c, 0x0), Nm(0x1240),
  ToHost(0x1244, 0x0)]

As discussed above, our safety condition for machine code is as follows: given any trace such as the ones above, and any sequence of FROMHOST values, there must exist another trace produced from the same initial state, which receives our chosen FROMHOST values, and whose trace is equal to the original trace when both are censored.

Consider the trace

$$
\begin{aligned}
&[\texttt{FromHost}(\texttt{0x1234}, \texttt{0x0badcafe}),\\
&\texttt{Rd}(\texttt{0x1238}, \texttt{0x0badcafe}, \texttt{0x11111111}),\\
&\texttt{FromHost}(\texttt{0x123c}, \texttt{0x12345678}),
\end{aligned}
$$

$$\texttt{Nm}(\texttt{0x1240}),$$
$$\texttt{ToHost}(\texttt{0x1244},\ \texttt{0x23456789})]$$

The sequence of FROMHOST values it receives is

$$[\texttt{0x0badcafe}, \texttt{0x012345678}]$$

If we now consider the set of possible traces which receive values

$$[\texttt{0xdeadbeef}, \texttt{0x04040404}]$$

instead, we see that traces produced with these values must resemble the fourth trace presented above (aside from truncation of the trace and insertion of Nops). As we saw, the censored form of this trace is not equal to the censored form of our original trace; the memory addresses in the Rd events are not the same. Thus, this program does not satisfy our safety condition.

Taint Tracking

To simplify the process of proving that machine code satisfies our safety condition, we also provide a simple taint-analysis tool, with proofs that passing this analysis implies the safety condition holds.

The taint-analysis function takes in a program memory, an initial register file, an initial program-counter value, and initial memory contents. It also takes a goal program-counter value, goal register file, and goal memory contents. It then simulates an execution of the program, using the same semantics as above, and determines whether the program reaches the goal state within some bounded number of execution steps.

However, the register file and memory contents provided to the taint tracker hold values of type `option data` rather than `data`. A None value indicates that the contents of this register or memory address have been tainted by

dependencies on secret data. In the simulated execution, calls to FROMHOST always return None. None values are read, written, and copied like normal data; if a computation is performed that uses a None value, the result is also None. If a None value is ever used as the address of a memory access, used to determine a branch condition, or used to calculate the target of a jump, the taint-analysis function stops its execution and returns `false`.

In summary, this taint-analysis function decides whether a program goes from a start state to a goal state without ever using data derived from FROMHOST calls in forbidden ways. We then prove the following theorems:

- If the taint-analysis function returns `true` for some initial state and some goal state, and the goal state satisfies our safety property, then the initial state satisfies the property as well.
- If the taint-analysis function returns `true` when the initial state and goal state are the same, then the state satisfies the safety property.

With these two theorems, we can now prove any program safe as follows: We consider the program as an initial setup routine followed by an infinite loop. We run our taint-analysis function to check the program from the initial state to the start of the loop, and run it again to check the program from the start of the loop to the start of the loop. If both of these checks return `true`, we have a proof that the program satisfies our safety property starting from the given initial state.

Implementation Effort

Most of the effort in implementing the machine-code definitions and proofs was in defining the taint-analysis function and in showing that the semantics of the taint-analysis function matched the semantics of `hasTrace`.

The `hasTrace` semantics calculates opcodes, branches, and instruction results using the same functions that the unpipelined RISC-V processor uses. However, because the taint-analysis function operates on a different datatype, some of these functions needed to be reimplemented.

Furthermore, we discovered that several of the low-level bit-manipulation functions used by Kami were defined using opaque proofs that various bit-lengths were equal, in order for uses of our dependently typed variable-length word datatype to typecheck. These opaque equality proofs made it drastically slower to perform actual computations using these operations, but by rewriting the relevant functions using transparent lemmas, we were able to remove the slowdowns.

Having completed the one-time effort of proving the taint-analysis function correct, proving that any specific piece of code satisfies the machine-code-safety condition is now substantially easier. A proof that the code satisfies the safety condition can be reduced to proving that the taint-analysis function returns true on the code's initial setup routine and on the code's infinite loop, and these can be proved directly by evaluating the function.

3.3 The Hardware Layer

At the level of Kami, our "instructions" as considered in Section 3.1 are now the rules and methods of our module, and our "state" is module register contents, which include the register file, program memory, program counter, and data memory from the previous section. When proving the processor safety theorems, we assume that the code running on the processor satisfies the machine-code safety theorems.

We define the behavior trace of a Kami module according to the Kami semantics. A Kami trace is a list of labels; each label contains the name of a rule that fired on that clock

cycle (if any rule did), a list of all method calls to external modules, and a list of all method calls from external modules.

For example, a trace of just the `producer` module from figure 2.1 might look as follows:

$$\begin{aligned}&[(\text{"}\texttt{produce}\text{"}, [\text{"}\texttt{enq}\text{"} \to (0,0)], [])\\ &\ (\text{"}\texttt{produce}\text{"}, [\text{"}\texttt{enq}\text{"} \to (1,0)], [])\\ &\ (\text{"}\texttt{produce}\text{"}, [\text{"}\texttt{enq}\text{"} \to (2,0)], [])]\end{aligned}$$

A trace of the complete `prodQCons` module might look like this:

$$\begin{aligned}&[(\text{"}\texttt{produce}\text{"}, [], [])\\ &\ (\text{"}\texttt{produce}\text{"}, [], [])\\ &\ (\text{"}\texttt{consume}\text{"}, [\text{"}\texttt{output}\text{"} \to (0,0)], [])\\ &\ (\text{"}\texttt{consume}\text{"}, [\text{"}\texttt{output}\text{"} \to (1,0)], [])]\end{aligned}$$

Note that the internal method calls among `producer`, `consumer`, and `queue` are hidden from the trace of the concatenated module.

Our highest-level theorems for Kami modules consider a module that includes both a processor and a memory. For such a module, we censor its trace by finding any calls to the external `toHost` and `fromHost` methods in the trace and setting their arguments and return values respectively to 0. We extract `FROMHOST` values in the obvious way, by finding all calls to `fromHost` and extracting the return values.

To make it easier to prove the safety property for a combined processor/memory module, we introduce safety properties for a processor alone and for a memory module alone, and we prove that when these properties are satisfied, the concatenated module will satisfy our overall Kami safety property.

The Processor

Our processor-only safety property closely resembles the overall Kami safety property, but with some small changes. Instead of censoring only calls to `fromHost` and `toHost`, we also censor calls to the methods that read and write data in the memory module, zeroing the values (but not the addresses) passed in these calls. This is essentially the same type of censorship that was used at the machine-code layer; it was used there precisely so that this processor property would be provable.

Note that while the overall Kami safety property makes no assumptions about the details of how the processor and the memory interact, in order to define this processor censorship function, we need to know the specific interface between processor and memory. In the unpipelined RISC-V processor implemented in Kami, this interface consists of just a method `exec` in the memory module, which takes an address, an operation (read or write), and a value, and returns a value accordingly. The pipelined processors implement a more realistic interface where memory accesses do not yield results instantly; instead, there is one method to enqueue a memory request and another method to dequeue a result when the result is ready. The censorship function must be designed with a specific processor/memory interface in mind, and a new interface necessitates new proofs connecting the processor-only and memory-only safety properties to the overall Kami safety property.

Our processor-only safety property adds another requirement: the results of memory accesses as reflected in the trace must be *consistent* with some initial memory state. That is, whenever we read a value from memory, it must match the last value we wrote. (We consider shared memory in Section 5.2.) This constraint is necessary for our processor safety property to be meaningful. Consider the following program:

```
x = fromHost();
y = *(0x10101010);
*(x + y) = 0xdeadbeef;
```

This program clearly accesses a memory address which depends on data from FROMHOST and is therefore not constant-time. However, if our processor safety property does not require memory accesses to return consistent values, then consider the following trace:

$$[(\text{"execFh"}, [\text{"fromHost"} \to (0, 0x22222222)], [])$$
$$(\text{"execLd"}, [\text{"exec"} \to ((0x10101010, \text{false}, 0), 0x33333333)], [])$$
$$(\text{"execSt"}, [\text{"exec"} \to ((0x55555555, \text{true}, 0xdeadbeef), 0)], [])]$$

If we consider possible traces of this program receiving FROMHOST value 0x5050505050 instead of 0x22222222, the same initial state can yield the following trace under the Kami semantics:

$$[(\text{"execFh"}, [\text{"fromHost"} \to (0, 0x50505050)], [])$$
$$(\text{"execLd"}, [\text{"exec"} \to ((0x10101010, \text{false}, 0), 0x05050505)], [])$$
$$(\text{"execSt"}, [\text{"exec"} \to ((0x55555555, \text{true}, 0xdeadbeef), 0)], [])]$$

Note that the value loaded from memory is also different; because the memory is an external module, the Kami semantics place no restrictions on it, so it may return arbitrary values. This means that both of these traces censor to

$$[(\text{"execFh"}, [\text{"fromHost"} \to (0, 0)], [])$$

$$(\text{"execLd"}, [\text{"exec"} \to ((\texttt{0x10101010}, \texttt{false}, \texttt{0}), \texttt{0})], [])$$
$$(\text{"execSt"}, [\text{"exec"} \to ((\texttt{0x55555555}, \texttt{true}, \texttt{0}), \texttt{0})], [])]$$

Even though the FROMHOST values were different, the nondeterminism of the Kami semantics allowed us to find a censorship-equivalent trace. To prevent this, we define our memory-consistency predicate and modify our processor safety condition to say that for all *consistent* traces, and for all FROMHOST values, there exists another *consistent* trace which receives those values and is equivalent to the first trace under censorship.

There is another subtle detail which must be considered: the Kami RISC-V processors in general use a single method for both reads and writes to memory. Thus, this method includes a parameter for a value to be written, even when we're performing a read; and includes a return value for the result of a read, even when we're performing a write. Therefore, it is important for correctness' sake to enforce that these spare parameters are not used to leak information. We do not censor these unused parameters in our censorship function, forcing them to remain identical no matter what FROMHOST values we receive. We also prove that the processor always sets the unused write value to 0.

The Memory

The properties that must be proved for the memory module are relatively simple. We must show that all interactions with the memory module do in fact obey the consistency property the processor relies on, and that the memory sets the unused return value on writes to 0. We must also show that the behavior of the memory module does not depend on what *values* are written to memory. This is a simple and obvious-seeming property, but it must be proved rigorously in order for our reasoning to be sound. One can also imagine well-intentioned memory designs that violate this property: for instance, a deduplicating memory module that

transparently maps blocks of identical data onto a single location internally. Such a module would then need to analyze the contents of memory in order to deduplicate effectively. Note also that what we refer to as "memory" really means "everything other than our processor running trusted code". In real-world instances, the memory module could include other untrusted processors in a multiprocessor system; we are then proving that the memory properly implements access controls and does not give the other untrusted processors access to our private data.

The proof that memory behavior is independent of values written follows the same censorship-equivalence pattern as the other proofs. We define a censorship function that removes read and write values from memory accesses in the memory's behavior trace, and we prove that for any trace, there exists another trace with *write values* of our choosing which is censorship-equivalent to the first trace.

Implementation Effort

In proving the theorems to connect the Kami safety condition with the processor-only and memory-only safety conditions for the synchronous memory interface, we encountered some difficulties in reasoning about the behavior of our censorship functions. Method calls are represented in Kami traces in a dependently typed way which includes both the datatypes of the parameters/return values and the values themselves. Thus, in order to selectively modify specific parts of a method's parameters/return values, we needed to perform a case-match on the datatypes of these values. For methods whose parameters are complicated data structures, this required very complex match terms which slowed down proofs. There were also a large number of simple results about finite maps that needed to be proved manually, perhaps suggesting the need for a proof tactic that implements a more comprehensive decision procedure for

propositions about finite maps.

Figuring out the exact details of what needed to be censored, and what needed to remain uncensored, was also surprisingly tricky. In particular, the issues with the unused parameters in the memory access method were not immediately obvious and became an obstacle somewhat late in the proof process, necessitating modifications to earlier proofs. However, the lessons learned from this case can be reused for defining censorship functions and writing proofs for other processor/memory interfaces in future work.

Chapter 4

Concrete Case Study

In this section, we discuss a specific machine-code program and hardware module that we have proved satisfy their respective safety conditions, giving us an end-to-end proof that this software running on this processor/memory module has a Kami behavior pattern which is independent of secret inputs.

4.1 Machine Code: Salsa20

The program we chose to validate was an implementation by Andres Erbsen of the Salsa20/20 stream cipher[1]. Starting from C source code, we modified the code to use our FROMHOST/TOHOST interface for input and output; our modified code is shown in figure 4.1. We then compiled this code using the RISC-V compiler webapp at `https://cx.rv8.io/`, translated the RISC-V assembly into the format used within Kami, identified a setup routine and infinite loop, and validated each of these stages with the taint-analysis function. The result is a proof that this compiled Salsa20 code satisfies the machine-code-safety condition; this proof can be composed with our proof described below for an unpipelined RISC-V processor to show that the combined software/hardware system is safe (at least down to the level of Kami semantics, as semantics

1. Daniel J. Bernstein. "The Salsa20 family of stream ciphers". In: *New stream cipher designs: the eSTREAM finalists*. Ed. by Matthew Robshaw and Olivier Billet. 2007, pp. 84–97. URL: `https://cr.yp.to/papers.html#salsafamily`

for RTL have not yet been formalized in Coq).

Implementation Effort

While the actual theorem-proving was made fairly easy by the taint-analysis function, the work required to produce machine code suitable for Kami was highly nontrivial, due in part to unfortunate limitations of the current Kami RISC-V implementations. One source of difficulty, which we were not previously aware of, was the fact that current Kami processor implementations have a hard-coded size limit of 256 instructions in their program memory. This number appears to be relied on by other parts of the codebase, and changing it might require substantial refactoring. However, 256 instructions is a frustratingly small amount of space in which to write secure constant-time crypto code. Our first compiled version, before we noticed the limit, was roughly 350 instructions, but by replacing a frequently-used macro with a function and by passing the `-Os` flag to the compiler to optimize for small program size, we were able to reduce it to 209 instructions.

Another difficulty arose due to the small number of instructions that appeared to actually be implemented in Kami's version of the RISC-V ISA. In particular, at first glance the only instructions with immediate variants appeared to be `ADDI` and `LI`, requiring other instructions such as `ANDI`, `SLLI`, and so on to be replaced with an `LI` followed by a non-immediate instruction. Eventually, we noticed that several more instructions had in fact been implemented, but had not been included in the inductive datatype intended to simplify writing RISC-V machine code.

We also needed to essentially link our compiled code ourselves, using a combination of Python scripts and manual inspection of the code. The compiled assembly code used labels to refer to jump targets, rather than address offsets, and included pseudo-instructions such as `call` and `ret` that needed to be replaced by jump instructions. In

Figure 4.1: Salsa20/20 implementation, modified from https://github.com/andres-erbsen/salsa20/blob/master/salsa20.c

```
// rotate x to left by n bits, the bits that go over
// the left edge reappear on the right
#define R(x,n) (((x) << (n)) | ((x) >> (32-(n))))

// addition wraps modulo 2^32
// the choice of 7,9,13,18 "doesn't seem very important" (spec)
static void quarter(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t *d) {
  *b ^= R(*d+*a, 7);
  *c ^= R(*a+*b, 9);
  *d ^= R(*b+*c, 13);
  *a ^= R(*c+*d, 18);
}

void salsa20_words(uint32_t *out, uint32_t in[16]) {
  uint32_t x[4][4];
  int i;
  for (i=0; i<16; ++i) x[i/4][i%4] = in[i];
  for (i=0; i<10; ++i) { // 10 double rounds = 20 rounds
    // column round: quarter round on each column; start at ith element and wrap
    quarter(&x[0][0], &x[1][0],&x[2][0], &x[3][0]);
    quarter(&x[1][1], &x[2][1], &x[3][1], &x[0][1]);
    quarter(&x[2][2], &x[3][2], &x[0][2], &x[1][2]);
    quarter(&x[3][3], &x[0][3], &x[1][3], &x[2][3]);
    // row round: quarter round on each row; start at ith element and wrap around
    quarter(&x[0][0], &x[0][1], &x[0][2], &x[0][3]);
    quarter(&x[1][1], &x[1][2], &x[1][3], &x[1][0]);
    quarter(&x[2][2], &x[2][3], &x[2][0], &x[2][1]);
    quarter(&x[3][3], &x[3][0], &x[3][1], &x[3][2]);
  }
  for (i=0; i<16; ++i) out[i] = x[i/4][i%4] + in[i];
}

// inputting a key, message nonce, keystream index and constants to that transormation
void salsa20_block(uint32_t *out, uint32_t key[8], uint64_t nonce, uint64_t index) {
  static const char c[17] = "expand 32-byte k"; // arbitrary constant
  #define LE(p) ( (p)[0] | ((p)[1]<<8) | ((p)[2]<<16) | ((p)[3]<<24) )
  uint32_t in[16] = {LE(c), key[0], key[1], key[2],
                     key[3], LE(c+4), nonce&0xffffffff, nonce>>32,
                     index&0xffffffff, index>>32, LE(c+8), key[4],
                     key[5], key[6], key[7], LE(c+12)};
  salsa20_words(out, in);
}

// enc/dec: xor a message with transformations of key, a per-message nonce and block index
void salsa20(uint64_t nonce) {
  int i, j;
  uint32_t msgword;
  uint32_t block[16];
    uint32_t key[8];
    for (i = 0; i < 8; i++) {
        key[i] = fromhost();
    }
  for (i=0; ; i++) {
    salsa20_block(block, key, nonce, i);
    for (j = 0; j<16; j++) {
      msgword = fromhost();
      tohost(msgword ^ block[j]);
    }
  }
}
```

addition, we needed to deduce enough of the calling conventions used by the compiler in order to figure out what kinds of values the various registers needed to be initialized to.

We also discovered some unfortunate interactions between existing Kami infrastructure and the taint-analysis function. In particular, the taint-analysis function previously used the `getRs1` and `getRs2` functions provided by Kami in order to figure out the operands of arithmetic operations and similar instructions. If either operand was tainted, the result of the operation would be flagged as tainted. Unfortunately, for immediate instructions with no second operand, `getRs2` still returned an arbitrary value, and if that register happened to be tainted, the result of the immediate operation would be incorrectly marked as tainted. We first attempted to rewrite `getRs2` to return `0` for immediate instructions, as register `0` always holds the value `0` and is never tainted. However, we eventually realized that the ISA treated `JAL` and `JALR` instructions as having a first register operand, as this register is used to calculate the new program counter, even though the value stored to the destination register depends only on the current program counter and not on any registers. Faced with this, we rewrote the taint-analysis function to use a function we wrote ourselves that properly inspects what operands are actually being used, rather than relying on Kami functions.

4.2 Hardware: Unpipelined RISC-V

To prove that the unpipelined RISC-V processor satisfies the processor-only safety condition, assuming that the code on the processor satisfies the machine-code safety condition, we define a relation between Kami traces of the processor and machine-code traces as defined by our abstract semantics. We show that this relation preserves `FROMHOST` value extraction (that is, if a code trace and a Kami trace are

related, their extracted sequences of FROMHOST values are the same); that for every code trace derived from the code semantics, there is a related consistent Kami trace derived from the Kami semantics, and vice versa; and that if two code traces are censorship-equivalent, their related Kami traces are also censorship-equivalent. This allows us to lift the safety property from the machine-code layer to the Kami layer.

Implementation Effort

Proving theorems about a specific processor involved a large amount of wrangling with the Kami semantics. In particular, we needed to write lemmas specialized to the specific modules we were proving that would allow us to perform simpler case-analyses on the possible steps a Kami module could perform under the Kami semantics. For instance, we showed that the only possible step the unpipelined processor could perform was to fire a single rule from its set of rules.

Reasoning about the values of expressions written in Kami's domain-specific language was also difficult, as the function for evaluating Kami expressions could cause extremely slow proofs if it was not manipulated carefully. This could be sidestepped to some extent by using the right sorts of proof tactics, but many of the proofs about the processor safety condition are still annoyingly slow and memory-hungry.

Chapter 5

Conclusion

By building on existing work in Kami and adding a framework for reasoning about timing behavior of code running on compiled Kami circuits, we are able to make powerful new security guarantees about real-world implementations of cryptographic algorithms. This project fills an important gap in the field of formally verified cryptography and hopefully lays groundwork for other proofs about low-level properties of formally verified hardware.

5.1 Related Work

Another project that provides formal guarantees of safety from timing side channels for hardware CPU designs is SecVerilog[1]. SecVerilog is based on the Verilog hardware design language with the addition of security labels; it uses static type analysis to enforce information-flow constraints according to a lattice of security levels. SecVerilog guarantees that no information can flow from a high security level to a low security level, including timing-based information flows. The particular property it enforces is called observational determinism; it states that for any two executions of a hardware module where all low-security inputs are identical, the low-security portions of the

1. Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. "A Hardware Design Language for Timing-Sensitive Information-Flow Security". In: *ASPLOS 2015*. `http://hdl.handle.net/1813/36274`. Istanbul, Turkey, Mar. 2015

execution traces will be identical. These traces include clock-cycle counts, thus guaranteeing that even timing behavior is independent of high-security inputs. (Unlike Bluespec and Kami, Verilog and SecVerilog describe hardware at a lower level where it is possible to reason directly about timing behavior.) Security types of components in SecVerilog can vary dynamically depending on data values, with some constraints for ease of type checking. Existing demonstrations of SecVerilog include a secure MIPS processor and a simplified prototype of the ARM TrustZone hardware security architecture[2].

2. Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. "Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis". In: *ASPLOS 2017*. Xi'an, China, Apr. 2017

While SecVerilog is a powerful tool for designing secure hardware, it cannot be used for verifying preexisting hardware designs, or for implementing hardware architectures that don't involve security levels. It can only be meaningfully used to implement architectures with special instructions for setting the current security level, or architectures that are partitioned into secure and insecure regions. This project constructs proofs of security properties about code running on standard RISC-V processors, which lack security levels of the sort used by SecVerilog. Furthermore, it allows this hardware to be described in a higher-level hardware design language, reducing the burden on hardware developers.

5.2 Further Work

Pipelined Processors

Currently, the only concrete Kami modules that have been proved to satisfy the Kami safety condition are the unpipelined processor with instantaneous memory. We would like to have safety proofs for the more realistic pipelined processors that have been implemented in Kami.

The first step is to define processor-only and memory-only safety conditions for the asynchronous memory interface used by the pipelined processors, and

show that these conditions together imply the Kami safety condition. Much of the proof logic from the synchronous interface can probably be reused for these proofs.

More challenging will be proving the processor-only safety condition for pipelined processors. With the unpipelined processor, it was easy to define a relation where every step of a Kami trace corresponds to a step in a machine-code trace. Once instructions take multiple steps to execute, this becomes more complicated. It may still be possible to define a relation that can be used in the same way as the relation for the unpipelined processor, or an entirely different proof strategy may be needed.

The RTL Layer

For true end-to-end proofs of side-channel safety, we need to extend our reasoning down to the level of a register-transfer language, where actual timing behavior can be analyzed concretely unlike in Kami. Essentially, the goal is to prove a theorem about the Kami-to-RTL compiler, stating that for any Kami module which satisfies the Kami safety condition, the compiler will produce an RTL circuit which satisfies an RTL safety condition analogous to those we have already proved. For any trace produced by the RTL circuit under the RTL semantics, it should be possible to change the `FROMHOST` values received and find a censorship-equivalent trace, for some appropriate definition of censorship. This depends heavily on formalization of RTL semantics and a Kami compiler, which have not yet been accomplished.

Shared Memory

Currently, our proof framework is based on the assumption that our processor running our trusted code has sole access to the entire memory, and no other processor can read or write it. In general, we would like to be able to prove

theorems about multiprocessor systems where some memory is private to our trusted processor, and some memory is shared with other processors. There are several changes that would need to be made to support reasoning about such systems. Firstly and most importantly, we would need to write a new Kami memory module which implements access controls on memory. The processor/memory interface would likely also need to be changed to reflect this new design element. The remaining changes are relatively simple for the most part. The censorship functions for machine code and for the processor-only safety property would need to be changed to only censor values written to private addresses; values written to shared addresses would remain uncensored and thus would be required to be independent of secret data. Furthermore, the machine-code semantics and the memory-consistency property would need to be loosened to allow reads on shared memory addresses to return arbitrary values, as the contents of these addresses could be rewritten by other processors in between our writes.

The most interesting changes would be required in the taint-analysis function for machine code. Currently, the taint-analysis function is able to neatly sidestep the nondeterminism present in the machine-code semantics; the only source of nondeterminism is FROMHOST calls, whose return values are unspecified, but fortunately these unspecified values are required to never be relevant to any decision we make during our simulated execution. No value derived from a FROMHOST value is allowed to affect control flow or to determine what memory addresses we access. Thus, the simulated execution can treat the FROMHOST values as opaque and proceed in a deterministic fashion. If values read from shared memory addresses became an additional source of nondeterminism in the semantics, we might need to modify the taint-analysis function to allow for this. However, it seems inadvisable to allow values from

untrusted sources to affect control flow, so it may be possible to still consider these values as opaque and mandate that our branches, jumps, and memory-access patterns not depend on foreign data.

Other Sources of Secret Data

We model all secret data as being sent into the processor through FROMHOST instructions, which receive data from some unspecified external source. In practice, sensitive data may originate in other places; perhaps there are specific registers or memory addresses that hold secret data at the start of execution of some cryptographic routine. It would be useful to be able to prove safety in this more general case.

At the machine-code level, this is essentially already accomplished. In order to inductively prove the necessary theorems about our taint-analysis function, we proved a connection to a more general safety property which allows for initial states where tainted secret data already exists in registers and/or memory. In this general property, we demand a censor-equivalent trace for all sets of FROMHOST values, tainted register contents, and tainted memory contents. A corresponding Kami-level safety property could presumably be written and proved, but this would likely be more difficult than the current proofs.

References

Bernstein, Daniel J. “The Salsa20 family of stream ciphers”. In: *New stream cipher designs: the eSTREAM finalists*. Ed. by Matthew Robshaw and Olivier Billet. 2007, pp. 84–97. URL: https://cr.yp.to/papers.html#salsafamily.

Carlier, Vincent, Hervé Chabanne, Emmanuelle Dottax, and Hervé Pelletier. *Electromagnetic Side Channels of an FPGA Implementation of AES*. Cryptology ePrint Archive, Report 2004/145. https://eprint.iacr.org/2004/145. 2004.

Chen, Haogang, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. “Using Crash Hoare Logic for Certifying the FSCQ File System”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP ’15. Monterey, California: ACM, 2015, pp. 18–37. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815402. URL: http://doi.acm.org/10.1145/2815400.2815402.

Choi, Joonwon, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. “Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification”. In: *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP’17)*. Sept. 2017.

Dave, N., Arvind, and M. Pellauer. “Scheduling as Rule Composition”. In: *2007 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007)*. May 2007, pp. 51–60. DOI: 10.1109/MEMCOD.2007.371249.

Ferraiuolo, Andrew, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. “Verification of a Practical Hardware

Security Architecture Through Static Information Flow Analysis". In: *ASPLOS 2017*. Xi'an, China, Apr. 2017.

Genkin, Daniel, Adi Shamir, and Eran Tromer. *RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis*. Cryptology ePrint Archive, Report 2013/857. https://eprint.iacr.org/2013/857. 2013.

Gu, Ronghui, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels". In: *In Proc. 2016 USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA, Nov. 2016, pp. 653–669.

Kocher, Paul C. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances In Cryptology - CRYPTO '96, LNCS 1109*. 1996, pp. 104–113.

Kocher, Paul, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. "Introduction to differential power analysis". In: *Journal of Cryptographic Engineering* 1.1 (Apr. 2011), pp. 5–27. ISSN: 2190-8516. DOI: 10.1007/s13389-011-0006-y. URL: https://doi.org/10.1007/s13389-011-0006-y.

Leroy, Xavier. "A Formally Verified Compiler Back-end". In: *J. Autom. Reason.* 43.4 (Dec. 2009), pp. 363–446. ISSN: 0168-7433. DOI: 10.1007/s10817-009-9155-4. URL: http://dx.doi.org/10.1007/s10817-009-9155-4.

Percival, Colin. "Cache missing for fun and profit". In: *In Proc of BSDCan 2005*.

Waterman, Andrew and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. RISC-V Foundation, May 2017. URL: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

Zhang, Danfeng, Yao Wang, G. Edward Suh, and Andrew C. Myers. "A Hardware Design Language for Timing-Sensitive Information-Flow Security". In: *ASPLOS 2015*. http://hdl.handle.net/1813/36274. Istanbul, Turkey, Mar. 2015.