Algorithmic Checking of Security Arguments for Microprocessors

Adam Chlipala
MIT
Cambridge, MA 02139
Email: adamc@mit.edu

***Abstract*—Anticipating all possible attacks on a system is hard work. Malicious actors seem to have an inherent advantage, since they can win by finding single vulnerabilities. In our MIT team within the DARPA SSITH program, we are exploring principled ways to rule out human error as a source of security issues in computer processors. A typical security audit involves prose arguments about attack models and why they are thwarted. We instead write down formal mathematical theorems about real digital hardware designs, and we build their formal proofs that can be checked algorithmically. That is, a program, rather than a potentially distracted human, confirms that the security argument is convincing.**

After giving some background on the general technology, I would like to focus on two concrete uses in the SSITH program. First, we are exploring flexible tagging support, to flow additional security-relevant information through the microarchitectural state of a Linux-capable processor. Unusually, however, we propose to compile custom processor descriptions automatically from descriptions of security policies to be enforced. We aim to do this compilation in a way that gives formal theorems that the generated processors truly enforce the security policies. Second, we are tackling the issues with timing side channels exposed by the Spectre and Meltdown vulnerabilities. Through a synergistic connection with work funded by the National Science Foundation, we are able to prove security theorems for whole hardware-software system stacks. For instance, we can show that a compiled C program with a cryptographic function will run on a specific RISC-V processor, in a way where a secret input flowing into the function provably has no effect on timing of output events flowing out of the processor.

***Keywords*—formal verification; tagged architectures; timing side channels; RISC-V; DARPA; SSITH**

I. Introduction

The DARPA SSITH program is about fundamental tooling approaches to make commodity microprocessors more secure, without requiring superhuman attention to detail by designers. Our SSITH team at MIT (PI Chlipala, co-PIs Arvind and Devadas, and PI Hicks for subcontractor Accelerated Tech, Inc.), in a project called the Hardware Security Compiler, is pursuing one particular "secret weapon" in that direction: *end-to-end mechanized proofs of functional correctness*. That is, we take advantage of computer software that checks completeness of mathematical proofs, to establish that formal security theorems hold of real processor designs. That way, we take the pressure off of human security auditors, to catch mistakes in processor security mechanisms. Instead, we just need to get the theorem statements right, formalizing what security means in some context. This powerful idea can be lifted to a higher level than just concrete designs of processors, and we do so in our project. We show that particular *development tools* produce secure-by-construction designs, whatever source code we apply them to.

Our project pulls together technologies from our team's past work, extends and connects them with formal assurance, and adds one new central tool.

- The **Bluespec hardware-description language** [1] provides a high-level notation for describing IP blocks and their composition to form complete processors, with good support for modular decomposition of designs into libraries, put to good use by our team in the Riscy framework for assembling RISC-V processors [2].
- The **Kami formal-verification library** [3] for the Coq theorem prover lets us prove specifications for individual IP blocks and then compose those results together into whole-theorem results.
- The **Sanctum architecture for secure-enclave execution** [4] lets us protect software components from each other, without trusted operating systems or hypervisors.
- Our new **Hardware Security Compiler** will support a rapid-development workflow for processors where software experts can describe security policies and have them compiled automatically into processor designs guaranteed to enforce the policies accurately. Designs will be generated as Bluespec code, with Kami proofs, calling trap-handler code (for when policy violations are detected) protected inside Sanctum enclaves.

In the next section, we summarize what this style of mechanized proofs is all about and what guarantees it delivers. The final two sections sketch our approaches to compilation of custom tagging schemes and to proved freedom from timing side channels.

II. Mechanized Proofs?

We use the Coq theorem-prover software, though other similar systems could have formed reasonable bases, too. Basic usage involves:

DISTRIBUTION STATEMENT A. Approved for public release: distribution is unlimited. (January 9, 2019).

- Defining *systems under study* as source code in languages with clear semantics. (E.g., we work with Bluespec.)
- Writing down *formal specifications* for those systems, explaining which of their possible behaviors are acceptable, in a language of mathematical source code. (E.g., we write down mappings from IP blocks' input-wire signals to output-wire signals.)
- Developing *formal proofs* in yet another source-code language, laying out arguments for why systems meet their specifications. Manual effort is required to write the proofs, but then they are checked automatically by software.

The crucial properties of the approach include being:

- **Mechanized**, where all arguments are written out as source code and can be checked algorithmically
- **End-to-end**, where we prove each IP block separately but then compose into full-system theorems, such that we only need to worry about getting the full system's specification right, since specification bugs in the IP blocks must be caught in the course of proving the full system
- Proving **functional correctness**, where instead of relatively lax properties like localized assertions, we show very specific correctness properties, like properly implementing the specification of a machine-language instruction set, which tells us which output-wire signals are acceptable given prior input-wire signals

Use of the Bluespec language is crucial to allow separate proof of IP blocks while supporting low-cost composition. Bluespec library modules look something like Java classes, with encapsulated private registers and public methods for accessing them.

III. Compiling Tagging Schemes

Tagged processor architectures attach additional metadata to registers, memory cells, and other microarchitectural state. Different tagging schemes track different metadata to enforce different policies. For instance, to avoid buffer overflows, we might associate pointers with array-length information, trapping if an instruction tries to access a pointer with non-positive length. Configurable tag-tracking units in hardware can provide flexibility at the cost of overhead in power, performance, or area. Conversely, these metrics can be improved by designing processors with specific tagging schemes built in, but then we run into the high costs of developing and verifying new hardware. Our team's approach is to build a Hardware Security Compiler that produces a hardware design automatically, given a high-level description of a tagging policy.

Furthermore, the compiler itself will be proved sound: any hardware design that comes out is guaranteed to implement the tagging policy that went in. There are nontrivial challenges in establishing this property. First, we have the classic problem of showing correctness of a microprocessor with nontrivial optimizations like pipelining, speculation, and caching. Next, we have the challenge of deriving optimized implementations of different tagging schemes automatically. TLB-style caches should probably be introduced for tag information in certain places, and different tagging styles may call for dense (e.g., attach a tag to each memory location) vs. sparse (e.g., use a hash table off to the side) representation. We plan to formalize a menu of such implementation choices and prove that any selection from the menu is sound for any tagging scheme.

IV. Ruling Out Timing Side Channels

Sending the right signals on output wires is a good start, but we also want to make sure that we do not leak secrets via the *timing* of when those signals go out. We have already proved that a very simple, unpipelined processor design avoids leaking secrets through timing. In fact, we were also able to compose that hardware result with a software-level result for the machine code of a popular cryptographic cipher Salsa20, establishing that the full system (hardware and software) avoids information leaks via timing. A crucial challenge here is formulating a separate security property for each main piece: software, processor, and memory system. Any combination of three that meet their respective conditions should be secure.

That preliminary work requires software to meet the *constant time* condition popular in cryptography. Our ongoing work aims to establish timing-side-channel freedom for arbitrary software isolated with Sanctum. We must reason about how, for instance, partitioning of cache lines across protection domains prevents cache timing from leaking secrets.

An appealing property of this approach is that we need not anticipate attack details, trying to guess the next Spectre or Meltdown. Instead, we write down a simple property about relationships between I/O event timing and secret inputs, and in the course of end-to-end proofs that systems meet the property, we will run into any surprising microarchitectural synergies that endanger it.

Acknowledgment

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001118C0018. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).

References

[1] *Bluespec SystemVerilog Reference Guide*, Bluespec, Inc., Waltham, MA, July 2014.

[2] S. Zhang, A. Wright, T. Bourgeat, and Arvind, "Composable building blocks to open up processor design," in *MICRO'18: Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2018.

[3] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: A platform for high-level parametric hardware specification and its modular verification," in *ICFP'17: Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming*, Sep. 2017.

[4] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *CCS'17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.