Expert Opinion

FinalFilter: Asserting Security Properties of a Processor at Runtime

Cynthia Sturton
University of North Carolina at Chapel Hill

Matthew Hicks
Virginia Tech

Samuel T. King
University of California, Davis

Jonathan M. Smith
University of Pennsylvania and DARPA

In an ideal world, it would be possible to build a provably correct and secure processor. However, the complexity of today's processors puts this ideal out of reach. The complete verification of a modern processor remains intractable. Statically verifying even a simple security property—for example, "*hardware privilege escalation never occurs*"—remains beyond the state of the art in formal verification.

Testing can complement formal verification methods, yet testing is incomplete and bugs in the hardware that leave it vulnerable continue to elude test suites. Further, a crafty malicious actor can evade typical testing coverage metrics.

Recent efforts, including that of three of the authors, have explored the use of static analysis on the design files (e.g., hardware description level source code or gate-level netlists) to find suspicious circuitry.[1–3] These techniques rely on heuristics to define patterns that indicate a likely trojan and then search for instances in the design that match the pattern. However, malicious circuitry that does not match the pattern will be missed, as will inadvertent bugs that open vulnerabilities. By the time the weakness is uncovered, the hardware is already in the end user's hands and vulnerable to attack.

In the absence of a full proof of correctness, what is needed is a final filter: a runtime verification technique that works—postdeployment—to detect and respond to security property violations as they occur during execution. In this article, we make the case for final filters using our tool, FinalFilter, as a case study.

Digital Object Identifier 10.1109/MM.2019.2921509
Date of current version 23 July 2019.

FINALFILTER

Prior research, including our own, has shown that assertions hard-coded into the design can be a cheap and effective way to verify the correctness of any single execution run.[4,5] Assertions can cover properties that would be intractable to prove statically for the current state of the art. The downside

Published by the IEEE Computer Society

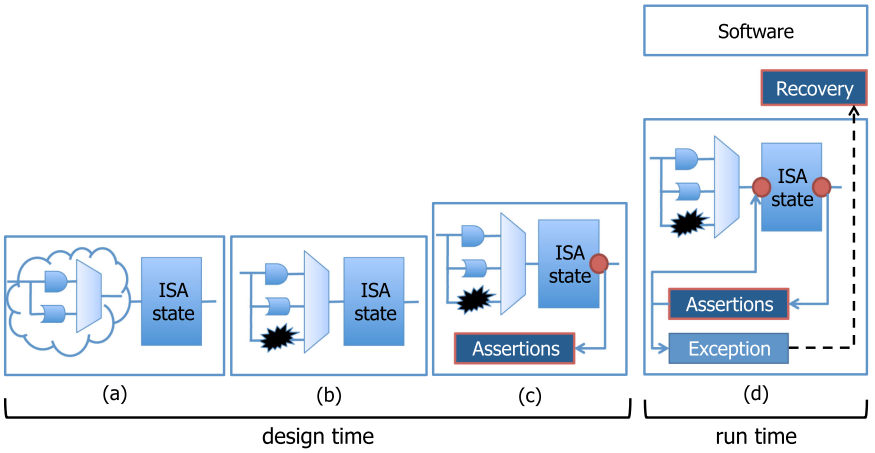

Figure 1. Processor design flow with FinalFilter: (a) Hardware description language implementation of the instruction set specification. (b) Vulnerability is accidentally or maliciously opened in the processor. (c) FinalFilter is added to the design as the last action,[6] with taps directly on the outputs of ISA state storing elements. (d) FinalFilter dynamically verifies the properties encoded by trusted software. FinalFilter triggers existing repair/recovery approaches in the event of an invariant violation. FinalFilter continues to protect the repair/recovery software.

is that, like all execution monitors, this approach cannot prove that the property can never be violated, only that if such a violation occurs the monitor will catch it. As such, a final filter is a verification approach that is complementary to and should be used in conjunction with existing testing and static verification methods.

We extend the basic idea of an assertion-based execution monitor to make it configurable so that the set of properties being monitored can be updated postdeployment to reflect new information about exploitable vulnerabilities in the design. FinalFilter is a reconfigurable, runtime verification system that monitors the state and events of the processor for invalid updates to privileged registers.

The mechanism of a final filter is simple and presents a small attack surface. Yet, making it configurable does add complexity. To minimize FinalFilter's cost to the system's trustworthiness, we formally verify the correctness properties of its component modules and of the composed system. Finally, we show how to verify key properties for individual configurations.

As a formally verified execution monitor, FinalFilter guarantees that any trace violating a given security property will be detected at the point of violation. This is independent of how the violation occurs or what the root cause is.

THREAT MODEL

The trusted computing base for FinalFilter includes our specification and verification process and tools, the fabrication process and tools, and the filter's current configuration.

Lifecycle Assumptions

Referring to Figure 1, we assume we are the last ones to touch the processor design. We rely on orthogonal techniques to ensure that FinalFilter is not tampered with in the supply chain, which includes fabrication of the processor and shipping to the end user.

Architectural Scope

FinalFilter protects privileged instruction set architecture (ISA)-level registers. FinalFilter does not detect side-channel attacks as doing so requires knowledge of more than the current trace of execution. The focus of this paper is the integer core of the processor. Notably, we assume the memory hierarchy is correct.

Attacker Model

The attacker is free to take any action not precluded by our assumptions, either in hardware or in software. This includes an attacker capable of creating and exploiting a hardware defect. An example might be a defect that causes the processor to return from an exception without restoring the privilege level.

DESIGN

FinalFilter enforces properties over privileged ISA state and events necessary for the security of software running on the processor. An example property that we will return to is, "the processor transitions from user mode to supervisor mode if, and only if, there is an interrupt or exception." Any

processor that correctly implements the specification must satisfy this property. Proving this property statically requires a proof across all possible execution traces—currently an intractable task. Yet, as an execution monitor, FinalFilter can verify the property for every trace that is executed. Monitoring is done by a set of hardware-based assertions over architecturally visible states and events.

FinalFilter is designed to be used in conjunction with existing software-level recovery and repair tools. For example, BlueChip,[1] a tool developed by three of the authors, can route execution around vulnerable circuitry. FinalFilter provides precise introspection points and can support a variety of repair and recovery approaches.

Three aspects of the design are worth noting.

1) FinalFilter is reconfigurable after deployment and can protect multiple security-critical properties concurrently.
2) FinalFilter's design is formally specified and its implementation proven correct.
3) Execution overhead is incurred only in the rare case that a processor violates one of the monitored security properties.

The key insight that allowed us to make the monitor both reconfigurable and able to handle multiple invariants concurrently is that many security properties can be implemented as a Boolean combination of more simple assertions, and these simple component assertions are usually in one of only a few forms. Users can specify a number of simple component assertions and combine them into one or more complex assertions that monitor hardware state.

Running Example

We use *security invariants* (or just *invariants*) to describe properties of the ISA that must be true of a secure implementation—that if violated would open an exploitable vulnerability. Invariants are dynamically verified by one or more *assertions* over architecturally visible state.

Consider the following component of the privilege escalation property mentioned before:

$I_0 \doteq$ *A change in processor mode from low privilege to high privilege is caused only by an exception or a reset.*

Invariant I_0 is a statement that the instruction set specification says must be true of the system at all points of execution. It can be written as a concrete assertion in terms of the ISA-level state in the following way:

$$\begin{aligned} A_0 \doteq \text{assert}(&\text{risingEdge}(\texttt{SR}\,[\texttt{SM}]) \rightarrow (\texttt{NPC}[31:12] = 0) \wedge \\ &\text{risingEdge}(\texttt{SR}\,[\texttt{SM}]) \rightarrow (\texttt{NPC}[7:0] = 0) \vee \\ &\text{risingEdge}(\texttt{SR}\,[\texttt{SM}]) \rightarrow (\texttt{reset} = 1)) \end{aligned}$$

where SR [SM]represents the supervisor mode bit of the processor's status register, and an exception is indicated by the next program counter NPCpointing to an exception vector start address. The address will always be of the form 0x00000X00, where the "X" indicates a don't-care value. (This might seem as if it leaves the door open for a processor attack that escalates privilege while executing at an address that matches the form 0x00000X00, but it does not. Pages in that address range have supervisor permissions set which implies that code executing in that address range is already in supervisor mode. If the processor attack attempts to allow user mode execution of supervisor mode pages, FinalFilter includes an invariant to detect such misbehavior.)

We break A_0 into three component assertions.

$$\begin{aligned} A_a &\doteq \text{assert}(\text{risingEdge}(\texttt{SR}\,[\texttt{SM}]) \rightarrow (\texttt{NPC}[31:12] = 0)) \\ A_b &\doteq \text{assert}(\text{risingEdge}(\texttt{SR}\,[\texttt{SM}]) \rightarrow (\texttt{NPC}[7:0] = 0)) \\ A_c &\doteq \text{assert}(\text{risingEdge}(\texttt{SR}\,[\texttt{SM}]) \rightarrow (\texttt{reset} = 1)). \end{aligned}$$

Each of these individual assertions is evaluated at each step of execution, and the results are appropriately combined to form a statement that is equivalent to A_0.

Invariant Monitor

FinalFilter reads in ISA-level state and outputs a signal indicating whether any of the programmed invariants were violated. It works essentially as a programmable finite state machine. Configuration data programs the machine with which invariants to check and ISA-level state acts as the input to the machine. The number of invariants it can monitor concurrently depends on the complexity of the associated component assertions and the number of assertion blocks built into the monitor.

Using our running example, we now describe each module in the configurable monitor, shown in its configured state in Figure 2. In our system, we

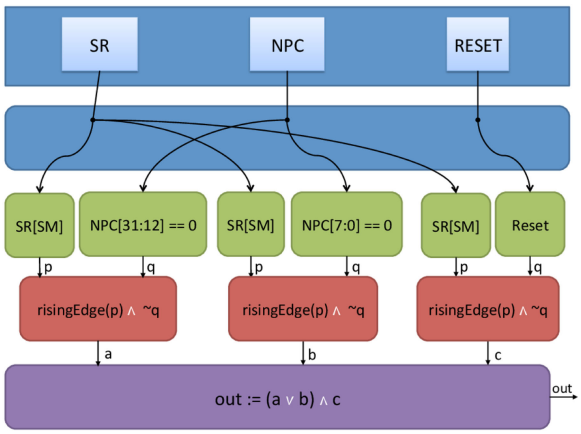

Figure 2. FinalFilter configured with assertion A_0. Starting from the top of the figure, the components are: ISA-level state, Routing block, Logic blocks, Assert blocks, and Merge block. The Routing block sends ISA-level state elements to the Logic blocks; the Logic blocks condense multibit state and constant inputs down to a single bit output that is sent to the Assert block; the Assert block compares the previous value of its inputs to the current value, outputing the result as a one bit value to the Merge block; the Merge block combines the Assertion block results to form a higher level result that indicates if the programmed invariants still hold; this result is tied to the processor's exception generation logic.

refer to A_a, A_b, and A_c as component assertions, and A_0 as simply an assertion. The difference being that A_{number} is the implementation of an invariant, a combination of component assertions, whereas A_{letter} represents a component assertion corresponding to one assertion block in the configurable monitor.

Routing. The Routing block is responsible for feeding the desired ISA-level state to the Logic blocks. The configuration data determines which state element gets routed to which Logic block. To accommodate arbitrary outputs, each Routing block output is 32 bits wide, with zero padding as required. In our running example, `SR[SM]` is output to Logic blocks 0, 2, and 4, `NPC` is output to Logic blocks 1 and 3, and `reset` is output to Logic block 5, as shown in Figure 2.

Logic. Each Logic block implements a comparison operator. Given two inputs A and B, the configuration data can select one comparison operator from the set $\{=, \neq, \leq, <, \geq, >\}$. Additionally, the configuration data can choose to mask off some portion of A or B, or both, or it can substitute a constant value for the value in B. Returning to our running example, Logic block 1 will evaluate $\texttt{NPC\&0xfffff000} = 0$ and output the result. Logic block 3 will evaluate $\texttt{NPC\&0x000000ff} = 0$ and output the result and Logic block 5 will evaluate $\texttt{reset} = 1$ and output the result. Logic blocks 0, 2, and 4 will evaluate $\texttt{SR[SM]} = 1$ and output the result.

Assert. The Assert block implements component assertions of the form $p \rightarrow q$, possibly across several clock cycles (e.g., if p is true then three cycles later, q is true). If it is ever the case that p is true while q is false, the assertion is triggered and the output of the Assert block will be high. In our example, each of A_a, A_b, and A_c are implemented in their own Assert block. The consequent q is always a combinational proposition over ISA state at a single step of execution: it is stateless and is given by the current value sent by the Logic block. However, the antecedent p can be stateful, possibly depending on previous values sent from the Logic block. For example, the individual assertions in our example all have the antecedent $\text{risingEdge}(\texttt{SR[SM]})$. This proposition is true at time t if and only if `SR[SM]`is low at time $t-1$ and high at time t. The Logic block will output a signal that is high whenever `SR[SM]`is high and the Assert block will determine when a rising edge of `SR[SM]`is seen. FinalFilter allows antecedents in one of three forms: $p \in \{\texttt{True}, \neg s_{t-1} \wedge s_t, s_{t-n}\}$. In other words, p can be defined as True, in which case the assertion will trigger whenever q is false, or p can be defined to be the rising edge of some ISA state s, or p can be defined to be the value of ISA state s at time $t-n$, where n is also configurable.

The Assert block uses four of the industry standard Open Verification Library assertions:

- *always*(expression): expression must always be true,
- *edge*(type, trigger, expression): expression must be true when the trigger goes from 0 to 1 (type = positive),
- *next*(trigger, expression, cycles): expression must be true cycles clock ticks after trigger goes from 0 to 1,
- *delta*(signal, min, max): when signal changes value, the difference must be between min and max, inclusive.

Merge. The Merge block takes the outputs from the Assert blocks and combines them as prescribed by the configuration data. It can be viewed as a configurable truth table. The inputs to the truth table are the Assert block outputs—the component assertions A_a, A_b, and A_c in our running example. The function defining how the component assertions combine (i.e., the *out* function) is configurable at run time. The truth table is implemented as a hierarchy of look-up tables. For example, with 16 Assert blocks, rather than a single lookup table with 2^{16} rows, the monitor would have four lookup tables with six inputs (2^6 rows) each. The outputs of the three first-level lookup tables make up the input to a second-level lookup table, the output of which is the output of the Merge block.

We can now complete our running example. Let err_a be the output of the Assert block for A_a, and let err_b and err_c be the output of the Assert blocks for A_b and A_c, respectively. Remembering that the output of each Assert block will be high when the assert triggers, i.e., when the invariant is violated, we combine the results of the component assertions in the following way:

$$\mathrm{err}_0 = (\mathrm{err}_a | \mathrm{err}_b) \& \mathrm{err}_c.$$

As desired, err_0 will be high whenever A_0 is false, i.e., whenever the A_0 assertion is triggered.

Configuration Data. The configuration data are provided by trusted software (e.g., the system BIOS) at initialization (originally, we imagine configuration coming from processor or motherboard manufacturers). It is the mechanism by which FinalFilter is configured, and portions of the configuration data are fed into each block at the appropriate stage.

VERIFICATION

We used the commercial model checking tool Cadence SMV for the verification of the configurable assertion fabric. For each component of FinalFilter shown in Figure 2, we formally specified its behavior and verified that the implementation meets the specification.

In most cases, formally specifying a component's behavior involved little more than extracting the information from the design documents. However, in two cases, the process of formalizing the specification brought out ambiguities in the design, and it was necessary to revisit the design phase of the process. During the course of verification, we found one implementation error: a logical AND was used where an OR was needed.

Ultimately, the monitor's behavior is determined by the configuration data, and it is up to the processor or motherboard manufacturer to provide a correct configuration. A misconfigured fabric could fail to provide the intended protections. We guard against misconfigurations in three ways.

First, we protect against invalid configurations that would result in unpredictable results. Built in to the design of each block is a check that the incoming configuration data are well formed. We verify that if any of the individual components report an invalid configuration, then FinalFilter will not fire any assertion failures. This behavior represents a tradeoff in the design space. On the one hand, an accidentally misconfigured fabric, which will never trigger an assertion, is not protecting the user. On the other hand, never firing in the presence of misconfigured data has the benefit of being a stable behavior—it is what exists today. An alternative is to always fire when the fabric is misconfigured, but this would give an attacker an avenue for launching a denial-of-service attack making FinalFilter a new avenue of attack, something we wish to avoid.

Second, we built a software tool to generate the configuration data from higher level assertion statements. Although only prototypical, we hope that further developing this tool will make generating correct configuration data relatively easy for the user.

Third, we built a validation tool to prove properties about individual configurations. We prove the following sanity checks on the configuration data:

- There are assertions configured.
- None of the assertions are unsatisfiable (e.g., the following does not occur $\{\texttt{True} \rightarrow q \wedge \neg q\}$).
- The configured assertions, as a whole, are satisfiable (e.g., the following does not occur $\{p \rightarrow q;\ p \rightarrow \neg q\}$).

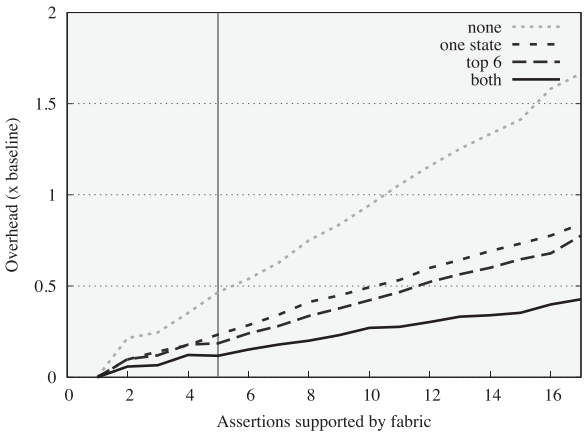

Figure 3. Hardware overhead with respect to the number of assertions supported by the configurable assertion fabric, evaluated at four optimization levels. The range in the number of assertions represents the range in protection required by the processors in our analyzed set from AMD. The vertical line represents the average number of assertions required to protect the processors in our analyzed set. As a reference point, previous work on deployed-bug patching entails hardware overheads of up to 200% and run time overheads of up to 100% in the *common* case.

- Assertions are not trivially violated (e.g., the following does not occur $\{p \rightarrow \neg p\}$).

If any of these checks fail, a misconfiguration error is reported along with information about the offending assertion(s). The user can run this tool before loading the configuration data into FinalFilter. We used the z3 SMT solver as the back end to this tool.

We note that while we formally verify the functional correctness of each module in the filter, we manually audit the connection between modules. That is, we manually check that every module's output signals are appropriately tied to the next module's input signals. There is no logic involved in the composition and our naming convention made the checks straightforward. Our end-to-end verification of the invalid configuration signals, mentioned above, does *not* rely on this manual audit.

EVALUATION

To evaluate the performance and efficacy of FinalFilter, we implement it inside the OR1200 Processor. The OR1200 is an open source, 32-bit RISC processor with a five-stage pipeline, separate data and instruction caches, and MMU support for virtual memory. It is popular as a research prototype and has been used in industry as well[7]; it is representative of what you would see in a mid-range phone today.

We wrote a program that automatically generates the FinalFilter hardware for a given number of Assert blocks to support. Generating the hardware programmatically makes it easy to explore the effect of tuning different parameters, and creates a regular naming and connection pattern that allows us to verify the structural connections of arbitrary filters using an induction type approach.

For a complete system capable of booting Linux, we implemented the processor and filter combination as the heart of a system-on-chip that includes DD2 memory, an Ethernet controller, and a UART controller. We implemented the system-on-chip on the FPGA that comes with the Xilinx XUP-V5 development board. We conservatively clock the system at 50 MHz.

Hardware Area Overhead

Figure 3 shows how the hardware area overhead changes as the number of assertions supported by FinalFilter increases. We built filters with support for as little as 1 assertion to as many as 17 assertions (the number required to protect all AMD processors we analyzed in our previous study on security-critical processor bugs[5]).

The figure contains data at four points in the fabric design space:

1) None. No optimization, this favors expressibility over overhead.
2) One State. This optimization uses Logic blocks with only one state input. Logic blocks were the biggest contributor to the area of the fabric and 83% of our security-critical invariants used only one input to the Logic block. This also reduces the number of required Routing blocks by 50%.
3) Top six. This optimization replaces the Routing blocks with new Routing blocks capable of handling the six most frequently used state elements. We observe that 76% of invariants require the same six ISA-level state items.
4) Both. This includes the two previous optimizations.

USING FINALFILTER

Using FinalFilter requires having a meaningful set of properties to monitor. In prior work, we took a manual approach to develop a set of security critical properties.[5] We studied errata documents to learn what types of exploitable errors can occur and we studied the architecture's specification documents to develop a set of properties necessary—though not sufficient—to protect security critical state of the processor.

Design-time verification alone is insufficient; some exploitable vulnerabilities will make it through. FinalFilter, a last line of defense—one that can be formally verified—protects security critical properties of the processor core.

In subsequent work, one of the authors has developed a semiautomated method for learning new security properties using information gleaned from known exploitable bugs[8]; and demonstrated that properties developed for one RISC processor may be suitable for use, after some translation, on a second RISC processor, even across architectures.[9] However, the development of security-critical properties for use with FinalFilter or any property-based verification method is still in its infancy and more research is needed.

Case Study

We configured FinalFilter with 18 assertions we found to be critical to security in our prior work.[5] We then introduced into the processor 14 vulnerabilities from a mix of previously published hardware attacks and attacks based on exploitable vulnerabilities from several years of AMD processor errata. For each one, we wrote a user-space program that exploits the vulnerability and reports if the attack was successful. FinalFilter is expressive enough to implement all 18 invariants, and the configured filter detects all of the attacks.

PRIOR WORK IN DYNAMIC VERIFICATION

FinalFilter builds on a line of research that uses dynamic verification to catch and patch functional bugs postdeployment. For example, DIVA[10] is a simplified checker core that verifies the computation results of the full-featured core before the processor commits the results to the ISA level. Narayanasamy *et al.*[11] use instruction rewriting routines to avoid triggering a bug that is found postdeployment.

In this article, we have not addressed the problem of measuring coverage. Boulé *et al.* [12] add circuitry to assertions to track and measure coverage. The question of what is a meaningful coverage metric for a set of security properties is an open one, but it is critical: such a measure can give an indication of the number of "unknown unknowns" that remain unprotected.

CONCLUSION

Design-time verification alone is insufficient; some exploitable vulnerabilities will make it through. FinalFilter, a last line of defense—one that can be formally verified—protects security critical properties of the processor core. We believe the idea is broadly applicable and in future work will be exploring the use of a final filter for commercial architectures and for modules outside the processor core.

ACKNOWLEDGMENT

The authors would like to thank the editors for their insightful comments and suggestions, and S. Bellovin for his advice and the phrase "final filter."

REFERENCES

1. M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith, "Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically," in *Proc. IEEE Secur. Privacy*, 2010, pp. 159–172.
2. J. Zhang, F. Yuan, L. Wei, Z. Sun, and Q. Xu, "VeriTrust: Verification for hardware trust," in *Proc. ACM Des. Autom. Conf.*, 2013, pp. 61:1–61:8.
3. A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: Identification of stealthy malicious logic using boolean functional analysis," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2013, pp. 697–708.
4. M. Bilzor, C. Irvine, T. Huffmire, and T. Levin, "Security checkers: Detecting processor malicious inclusions at runtime," in *Proc. IEEE Hardware Oriented Secur. Trust*, 2011, pp. 34–39.

5. M. Hicks, C. Sturton, S. T. King, and J. M. Smith, "SPECS: A lightweight runtime mechanism for protecting software from security-critical processor bugs," in *Proc. ACM Conf. Architectural Support Program. Lang. Oper. Syst.*, 2015, pp. 517–529.
6. A. Waksman and S. Sethumadhavan, "Silencing hardware backdoors," in *Proc. IEEE Symp. Secur. Privacy*, 2011, pp. 49–63.
7. R. Rubenstein, "Open Source MCU core steps in to power third generation chip," Jan. 2014. [Online]. Available: http://www.newelectronics.co.uk/electronics-technology/open-source-mcu- core-steps-in-to-power-third-generation-chip/59110/
8. R. Zhang, N. Stanley, C. Griggs, A. Chi, and C. Sturton, "Identifying security critical properties for the dynamic verification of a processor," in *Proc. ACM Conf. Architectural Support Programming Lang. Operating Syst.*, 2017, pp. 541–554.
9. R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, "End-to-end automated exploit generation for diagnosing processor designs," in *Proc. IEEE/ACM Symp. Microarchit.*, 2018, pp. 815–827.
10. T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *Proc. ACM/IEEE MICRO*, Haifa, Israel, Nov. 1999, pp. 196–207. [Online]. Available: http://www.eecs.umich.edu/taustin/papers/MICRO32-diva.pdf
11. S. Narayanasamy, B. Carneal, and B. Calder, "Patching processor design errors," in *Proc. IEEE Int. Conf. Comput. Des.*, Oct. 2006, pp. 491–498. [Online]. Available: http://cseweb.ucsd.edu/ calder/papers/ICCD-06-HWPatch.pdf
12. M. Boule, J. Chenard, and Z. Zilic, "Adding debug enhancements to assertion checkers for hardware emulation and silicon debug," in *Proc. Int. Conf. Comput. Des.*, 2006, pp. 294–299.

Cynthia Sturton is an assistant professor and Peter Thacher Grauer Fellow at the University of North Carolina at Chapel Hill. She leads the Hardware Security @ UNC research group to investigate the use of static and dynamic analysis techniques to protect against vulnerable hardware designs. Her research is funded by several National Science Foundation awards, the Semiconductor Research Corporation, Intel, a Junior Faculty Development Award from the University of North Carolina, and a Google Faculty Research Award. She was recently awarded the Computer Science Departmental Teaching Award at the University of North Carolina. She has a BSE from Arizona State University and an MS and a PhD from the University of California, Berkeley. Contact her at csturton@cs.unc.edu.

Matthew Hicks is an assistant professor at Virginia Tech, working at the intersection of security, architecture, and embedded systems, with special emphasis on analog-domain hardware security. Contact him at mdhicks2@VT.edu.

Samuel T. King was a professor for eight years at the University Illinois Urbana-Champaign. He then left his tenured position at UIUC to push himself intellectually and professionally in industry. He is currently with the Computer Science Department at the University of California Davis. He is interested in building systems for fighting fraud and rethinking our notion of digital identity. He has a PhD from the University of Michigan, an MS from Stanford University, and a BS from UCLA. Contact him at kingst@ucdavis.edu.

Jonathan M. Smith is currently a program manager in the Information Innovation Office (I2O) at the Defense Advanced Projects Research Agency (DARPA) on leave from the University of Pennsylvania, where he holds the Olga and Alberico Pompa Professorship of Engineering and Applied Science and is a professor of computer and information science. He was previously a Member of Technical Staff at Bell Telephone Laboratories and Bell Communications Research, joining Penn in 1989 after receiving his PhD from Columbia University. He previously served as a Program Manager at DARPA in 2004–2006, and was awarded the Office of the Secretary of Defense Medal for Exceptional Public Service in 2006. He became an IEEE Fellow in 2001. Contact him at jms@cis.upenn.edu.