Position Paper: Sapper: A Language for Provable Hardware Policy Enforcement

Xun Li Vineeth Kashyap Jason K. Oberg* Mohit Tiwari† Vasanth Ram Rajarathinam
Ryan Kastner* Timothy Sherwood Ben Hardekopf Frederic T. Chong

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA
{xun,vineeth,vasanthram,sherwood,benh,chong}@cs.ucsb.edu

*Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA
{jkoberg,kastner}@cs.ucsd.edu

†Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA
{tiwari@eecs.berkeley.edu}

Abstract

We describe Sapper, a language for creating critical hardware components that have provably secure information flow. Most systems that enforce information flow policies place the hardware microarchitecture within the trusted computing base, and also assume that the observable behavior of that microarchitecture is fully and correctly documented. However, the reality is that this behavior is incompletely (and sometimes incorrectly) specified, and that the microarchitecture itself often contains implementation bugs. This fact means that all such systems are vulnerable to attack by exploiting undocumented or buggy hardware features. Sapper addresses this problem by enabling flexible and efficient hardware design that is provably secure with respect to a given information flow policy. Sapper uses a hybrid approach that leverages unique language features and static analysis to determine a set of dynamic checks that are automatically inserted into the hardware design. These checks are provably sufficient to guarantee that the resulting hardware prevents all explicit, implicit, and timing channels even if the hardware is otherwise buggy or poorly documented.

Categories and Subject Descriptors B.5.2 [*Design Aids*]: Hardware description languages

General Terms Security, Languages

Keywords Hardware Description Language, non-interference, information flow

1. Introduction

In this paper we present our ongoing work towards designing hardware components with strong security properties, specifically with respect to information flow. Information flow control mechanisms enforce privacy and integrity security policies by tracking and constraining the propagation of data through a system. This tracking is accomplished by associating *labels* with the data, and propagating these labels appropriately. A wide spectrum of techniques for enforcing information flow policies have been proposed, ranging from language support [26], to operating system support [15, 35], to hardware support [34]. While these techniques are effective in a variety of scenarios, they make the critical assumption that the hardware documentation correctly and fully describes the behavior of the machine with respect to information flow. Unfortunately, even the most widely examined and used processors fail to meet this fundamental assumption.

Modern microprocessors ship with a significant number of bugs, despite being one of the most tested products brought to market today. To create these microprocessor designs, hardware engineers use design tools ranging from synthesis tools that can convert a subset[1] of hardware description languages such as Verilog into low-level netlists,[2] testing tools [9], model checking tools [6], and even verification tools to prove the correctness of *subcomponents* with respect to a reference [14]. However, hardware can have just as many "moving parts" as a fully featured operating system and full verification is a practical impossibility [1]. The size and complexity of these designs mean that real production systems ship with bugs with software level ramifications. While these bugs may not occur in commonplace execution, a motivated attacker will find unique ways to exploit these undocumented behaviors to leak information and circumvent security policies [11, 13].

1.1 Our Approach

The core of our approach is a security-aware hardware design synthesis language called *Sapper*. Sapper uses a hybrid approach that leverages unique language features and static analysis to determine a set of dynamic checks that are automatically inserted into the hardware design. These checks translate runtime *security policy violations* into *safe operations*, and they are provably sufficient to

PLAS'13, June 20, 2013, Seattle, WA, USA.

[1] Unlike software compilers, HDL compilers typically handle only a subset of the language and/or only specific types of constructs that infer certain types of hardware. In this respect they are quite unlike traditional programming languages.

[2] A netlist is a directed graph of circuit elements (e.g. logic gates) and interconnects (i.e. wires) that can be mapped onto a physical substrate (e.g. an FPGA or custom hardware implementation).

guarantee that the resulting hardware prevents all explicit, implicit, and timing channels. Sapper wraps around existing Verilog Hardware Description Language (HDL) code. The Sapper compiler automatically derives and inserts security checks into the system at critical latches; these checks operate in parallel with the logic they analyze. During design testing but before fabrication[3], the inserted checks will detect actions that violate security and translate those actions into pre-specified (by the hardware designer) non-violating actions. In other words, policy violations will show up during the design testing phase as functional bugs. Through careful design, the hardware engineers ensure that the system will operate as intended in the vast majority of cases even with these checks in place (as covered by traditional testing/verification techniques). Once testing is complete the second function of the inserted checks comes into play, as they will remain in the fabricated design. The checks serve as the last line of defense against run-time violations in conditions never encountered during testing and verification. Both undiscovered hardware bugs and rarely occurring combinations of events may provide opportunities to attack an unprotected system; however, hardware designed with Sapper will automatically capture and prevent any runtime violations.

2. Motivation

Sapper enforces a timing-sensitive *noninterference* security policy, where the security principles and relations between principles are defined by the hardware designer. Noninterference means that data tagged with a lower-level security label cannot influence the values of data tagged with a higher-level security label.[4] Timing-sensitive noninterference means that we take into account not only *explicit* (data-dependency related) information flow and *implicit* (control-dependency related) information flow, but also information flow related to *when* events happen. Timing-sensitivity is extremely difficult to enforce at the software level (see, e.g., Kashyap et al [12]), because timing variations generated by hardware components are abstracted away from the software programming model. For example, a memory access instruction can take widely varying amounts of time depending on the cache status, and code with branches can take varying amounts of time depending on branch predictor status. However, timing-sensitivity is important for critical systems: these timing characteristics can be exploited to create covert channels that break security systems with a reasonable level of practicality [10]. One might consider exposing to the software level a low-level model of the microarchitecture that includes timing information, as proposed by previous work [36]. However, the complexity of modern processors prohibits deriving a sound abstraction model for the hardware without digging into every logic gate in the design. Even hardware designers themselves might not be able to figure out the set of hardware components that are responsible for the timing of each instruction. Sound enforcement of information flow security policies demands tools that are tightly integrated into the hardware design process.

Hardware is designed using programming languages known as hardware description languages (HDL). Examples of HDLs include Verilog and VHDL. Intuitively, it might seem that one could apply information flow techniques from software languages directly to HDLs. However, HDLs are different from software languages in many aspects, and information flow analysis techniques for HDLs must be invented from scratch. In particular, there are three major difference between software programming languages and HDLs:

[3] Typically such tests are performed through a combination of hardware simulation and prototyping on reconfigurable hardware.

[4] Non-interference is may be too strong of a property for general purpose systems, but is useful both in the context of crypto systems and safety critical designs, and it matches closely with the existing design goals expressed by both Intel [5] and ARM [3].

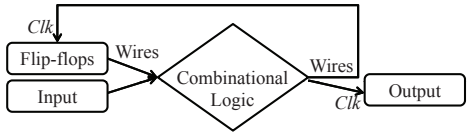


Figure 1. Structure of a synchronous circuit design used by typical modern processors. Flip-flops (e.g. registers) are controlled by a global clock (Clk). Inputs and values of flip-flops are fed into combinational logic, which finishes computation within a clock cycle and writes to output or back to flip-flops on clock edges.

Timing Model. In (most) software programming languages, programs are defined as a sequence of commands that each translate to a series of ISA instructions. The timing of each command in the language depends upon a number of factors, including the compiler implementation and hardware implementation as well as the current state of the hardware. It is almost impossible to determine the precise timing of software programs. On the other hand, hardware designs for modern processors are synchronous circuits based on a global clock. Data coming from inputs or flip-flops (i.e., hardware registers) are wired into a combinational logic circuit for computation, and the results are written to outputs or back to flip-flops as shown in Figure 1. By definition of synchronous circuits, state changes (i.e., changes to outputs and flip-flops) occurr only at clock edges. Under this model, commands written in HDL, no matter how complex, will become part of the combinational logic and the results will not take effect until the next clock edge. All forms of information flow in hardware designs are compressed into a single clock cycle.

It might seem that this timing model makes information flow analysis easier, due to centralized information and strict timing, however it in fact makes hardware designs highly susceptible to taint explosion when typical information flow analysis techniques are applied. Specifically, any input to the combinational logic that is tainted will potentially taint the entire hardware. Thus, it is not hard to see that pure static analysis based on static type systems as used in most software-based information flow mechanisms [26, 33] will unavoidably lead to duplication of all hardware resources in order to be statically verified for noninterference. The economic cost of this duplication is not acceptable, and thus conventional techniques are not adequate for our purpose.

Output Channels. Output channels are interfaces through which attackers are able to observe data. For software programs, output is usually implemented by well-defined library calls. In these programs, information flow policies can be enforced by checking that the security level of users who can observe the output is at least the same as that of the data being output. In hardware designs, there are no predefined "library calls" for I/O operations. Assuming that attackers cannot wire-tap the circuits themselves, output can be observed through a set of ports on the bus. Although enforcing information flow policies on those ports is sound, it is a challenging task to recover the system when violations are captured since the rest of the system would have been largely tainted at the time of output. This setup also makes it difficult to debug the source of violations. Sapper aims at securing the system in a way that is not only recoverable at runtime violations but also useful as a hardware testing tool.

ISA Support. The ultimate goal of enforcing security policies for HDLs is not to make the hardware itself more secure, but to protect systems and applications running on top of the hardware. This extra level of hierarchy makes HDLs different from software

programming languages in nature. In designing secure hardware, not only do we need to ensure that the hardware design enforces noninterference, but the hardware also needs to provide a corresponding ISA interface for systems to interact with the security mechanisms. Designing those ISAs in a formal and secure manner can be a significant challenge.

The philosophy of Sapper is to reinvent information flow analysis techniques for hardware description languages, incorporating all of the distinct characteristics and challenges of HDLs, to enable the creation of efficient, flexible, practical and provably secure computer architectures.

3. Overview of Sapper

We show that modern programming language techniques, when applied in the new domain of hardware description languages (HDLs), can provide static guarantees at design time along with precise control of information flow. Specifically we propose Sapper, a hardware description language that operates at the abstraction of a high-level HDL while enforcing security policies through statically-inserted logic for dynamic tracking and enforcement. A compiler is be responsible for statically analyzing the program in order to generate dynamic tracking and checking logic when translating the program into Verilog, thereby delivering provable guarantees. The generated logic will help eliminate most of the security bugs in the hardware design during testing, and remain in the design to prevent any runtime violations.

A typical Verilog program consists of three parts: signal declarations that define registers and wires (i.e., variables), a synchronous block in which all operations are triggered at the clock edge, and a combinational logic block containing all of the computation that will finish within a clock cycle. The synchronous block is responsible for writing data back to flip-flops at clock edges. The combinational logic blocks contain commands that are similar to those in software programming languages, including assignments, branches and switch/cases. Sapper keeps most of the Verilog syntax and requires minimum changes to the source code. In this section we sketch the details of Sapper. For convenience we assume that the security policy being enforced has two levels, Low and High, such that High information should never affect Low information. Sapper can handle arbitrary finite security lattices in practice.

3.1 Security Tags

Variables (i.e. signals, wires, etc.) in Sapper are associated with security tags that are tracked and checked for security policy violations at runtime. Checking *every* data movement in hardware for violations of noninterference would be extremely expensive, both in terms of additional hardware and performance overhead. We observe that in most hardware designs only certain outputs are exposed and observable by software/programmers and thus require strict enforcement. Many variables, such as internal pipeline registers and wires used to hold intermediate results, are only for temporary storage and are not directly observable. Those non-observable variables only require security tags to be correctly tracked dynamically so that their security level is correctly reflected at runtime. However, if we only enforce security policies for those small set of output ports, it may be extremely difficult for the system to "roll back" from violations or to debug the cause of violations. To this end, we propose Staged Enforcement, in which not only the end output ports are enforced for security policies, but also a set of architectural components that lie on the critical path of data movement are enforced. For example, data movement to the memory can be enforced for noninterference, such that the system can capture violations immediately at the point when data touches the memory. Based on the above observations, Sapper allows designers to declare data variables as one of the following two categories:

- **Enforced Tagged:** Variables having enforced tags will be declared with a default security tag at the beginning, and such security tag will not change until it is *explicitly* modified in the design through provided commands. Information flowing into enforced tagged variables will always be checked for noninterference.
- **Dynamic Tagged:** Variables with dynamic tags are not enforced by security policies, but their tags are dynamically tracked at runtime. Since most of the variables in typical hardware designs are dynamic tagged, variables that are declared without any initial security tags will be dynamic tagged by default.

This dichotomy requires designers to make decisions on what data should be tracked versus enforced, but it is often an easy decision to make since typical architectures only consist of a small portion of components exposed to users or central to data movement. In many architectures, selecting enforced tags for all the bus output ports, the memory and the cache will be sufficient. Note that as long as the I/O ports are enforced, not enforcing policies on some of the other components does not lead to unsoundness, but rather makes the system less precise and thus harder to use. The Sapper compiler is responsible for generating dynamic tracking logic and inserting dynamic checks depending on the tag of the target variable. Below we describe the details of tracking and enforcement.

3.1.1 Tracking

Assignments to dynamic tagged variables will trigger the tracking of security levels: the maximum security level of information that may affect the assigned value shall be propagated to the target variable. Instead of generating tracking logic for every single logic gate as in some previous work [31], Sapper takes advantage of static analysis on the HDL code and inserts tracking logic aggregately at the granularity of expressions and code blocks. Implicit flows (i.e. conditionals) are also derived by the compiler, which inserts logic to ensure sound security label propagation.

Furthermore, unlike previous work that must track information through each bit because it lacks language-level information about the hardware design and thus requires complex but precise tag propagation logic, Sapper tracks information at the register level[5] and uses simple logic to compute security levels (each variable has an n-bit tag independent of its width and the security level of the output is the least upper bound of the security levels of the inputs). In theory, Sapper may be less precise (but still sound) compared to bit-level tracking due to the coarser tracking granularity and relaxed tag propagation. However we observe that the major purpose of using precise bit-level tracking in previous work is to avoid label creep and allow a secure switch from a High to Low context. In the next section we will describe how the "nested states" feature we use in Sapper provides exactly what is needed to satisfy this requirement. In fact, there is nothing that prohibits bit-level tracking in Sapper, but we believe this is not necessary because the state transforms can be expressed in the language itself rather than needing to be inferred from the generated logic. Hence Sapper achieves sufficient precision for security enforcement with significantly less overhead while retaining a high degree of flexibility.

3.1.2 Enforcement

Any assignment to a variable with enforced tags needs to be enforced for noninterference. Specifically, the security level of the target variable can never be lower than the maximum security level of information that may affect the assigned value. The necessary

[5] Note that we do not mean only architectural registers here (like %eax), we mean register-transfer-level register, which is any set of bits used as a group in the hardware description language.

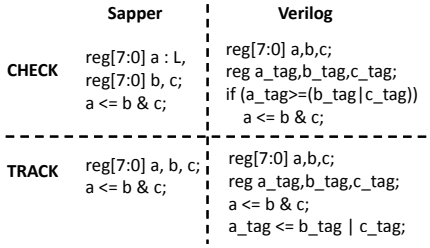


Figure 2. An 8-bit adder written in Sapper along with the generated Verilog code. There are two cases: in the first case register a is enforced tagged hence the assignment needs to be checked for noninterference; in the second case a is dynamic tagged hence only tracking is needed.

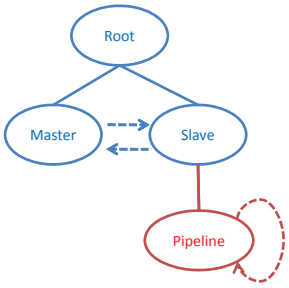
enforcement conditions will be derived by the compiler and the security checks will be automatically inserted into the resulting logic. These assignments will take effect only when they are guaranteed to be secure. Sapper also provides some flexibility for designers to deal with potential violations, which will be described in 3.4. Figure 2 shows the generated Verilog code for an *8-bit-and* design written in Sapper. There are two different cases shown in the figure, one with enforcement (CHECK) while another with tracking (TRACK) only. Note that both the tracking and enforcement logic are automatically generated by the compiler and there is no need for designers to manually specify anything except the initial enforced tags.

3.2 State Machines

Timing in most synchronous hardware designs is strictly aligned to clock edges, and registers are only updated at clock edges. To capture the notion of hardware timing, Sapper explicitly models hardware designs as *state machines*. During a clock cycle the hardware can only be in one of the machine's logical states, and all the logic from that state will be executed within the clock cycle. State transitions take effect at clock edges. Another important motivation behind modeling hardware as state machines is that state machines are a common pattern used by hardware designers, and most hardware designs are already written in or can be easily transformed to state machines.

Since state transitions can be conditional, to catch implicit leaks states must also have security tags and these tags must be correctly propagated or checked during state transitions. In the same manner as variables, states can be declared with enforced or dynamic tags. The security level of states with dynamic tags will be tracked dynamically at runtime, while states with enforced tags will be enforced for noninterference and their security level will not change until it is explicitly modified. An immediate advantage of Sapper is that the same state (if dynamic tagged) can act at different security levels dynamically, and hence the same piece of code can be reused as long as context switches are securely controlled.

Additionally, state transitions carry information by definition. To uphold noninterference, a transition from some state *A* to some state *B* should only occur if A is lower than B. In the case of a state machine diagram that is strongly connected (i.e. every state can reach every other state), the existence of any High state will require all states to be High (label creep). Sapper uses the concept of *nested states* proposed by Li et. al [19] in previous work to solve this problem. States can be organized hierarchically as a tree structure. Within each clock cycle, before executing the logic of any state, its parent state has to be executed first (recursively applied till the root). To give parent states complete control of child states and decide whether/when to fall into them, **fall** commands are used as an indication of transfer from parent state to child state. By having




```
reg[31:0] timer : L;

state Master:L = {
    timer = 100;
    goto Slave;
}

state Slave:L = {
    let state pipeline = {
        // Pipeline logic
        goto pipeline;
    }
    in
        if (timer == 0) begin
            goto Master;
        end else begin
            timer <= timer - 1;
            fall;
        end
}
```

Figure 3. State Machine Diagram example of a secure hardware controller, along with its corresponding implementation in Sapper. Noninterference is achieved by having a trusted timer controlling the behavior of the computation logic.

parent states with Low security levels and child states with High security levels, Low states have the freedom to decide when to terminate High states, without violating security.

Figure 3 shows an example of a state machine diagram for a secure hardware design based on TDMA (Time Division Multiple Access), which is a common design pattern used by secure systems. A trusted timer (Low) is used to control the execution of untrusted components. In particular, the Master state (trusted, labeled with Low, enforced tagged) sets up a timer and transits to the Slave state (also trusted, Low), which falls into its child state (potentially untrusted, dynamic tagged) and executes the computation logic. At the beginning of every cycle, the Slave State is always executed first and the timer is checked. If the timer expires control will transfer back to the Master State. The security level of the child state (i.e. Pipeline State in the diagram) can be either High of Low at runtime depending on the data it is dealing with. No matter what level it is, it will never affect its parent states, thus enforcing noninterference. The corresponding implementation in Sapper is also shown on the side. The timer variable, Master state and Slave state are provided with default types L (Low), indicating they are enforced tagged; while the Pipeline State does not have any default type, indicating it is dynamic tagged. When the code is compiled down to Verilog, tracking and checking logic will be generated based on a formal semantics. Although the runtime security level of the Pipeline State is dynamically changing, the generated checking logic will guarantee that the Master State is always trusted.

3.3 Manipulating Tags

One important advantage of Sapper compared to purely static mechanisms is that security labels can be read, reacted upon and updated at runtime. As we have defined earlier, the security level of enforced tagged registers will not change until they are explicitly modified through the language provided interface. This feature can be used by system kernels to efficiently and securely share memory among different security levels. Although we allow security tags to be modified explicitly, they cannot be modified arbitrarily otherwise information can be leaked through the labels. Sapper provides pre-defined commands to allow modification of the security tags of enforced tagged variables and states. Sapper language rules will ensure that no information can be leaked: a) the security level of any data can only be changed under a context whose level is not higher than the data's (e.g. Low data cannot be hoisted under a High context), thus no information can flow from High to Low by manipulating tags; b) When data is downgraded (e.g., changed from High to Low) the data is automatically *zeroed* instantly to avoid leakage.

The logic for checking, changing the tag and zeroing the data is generated by the Sapper compiler.

3.4 Violation Handling

To give hardware designers full flexibility to decide how to react to runtime security violations, Sapper provides a language interface for specifying the behavior when violation is *about* to happen. The syntax is: [*command*] *otherwise* [*actions*]; which specifies that if there exists any security violation in *command*, *actions* will be executed in replace of *command*. Our compiler will analyze every command that requires enforcement, derive necessary checks (in the form of security tag comparisons) that guarantees noninterference, and insert them into the design. The above code will become the following after compilation: *if* (*derived condition*) [*command*] *else* [*actions*].

Note that *command* will never be speculatively executed, instead, only one of *actions* and *command* will get executed depending on the value of condition. In the case when designers do not provide "otherwise" for commands that require enforcement, our compiler will automatically insert default "otherwise" actions that are guaranteed to be secure (e.g., disabling the operation to make it a noop). These *otherwise* rules can be defined recursively, meaning that the action in the otherwise branch can itself have an otherwise clause. These nested otherwise clauses are terminated by the default, guaranteed safe action; thus all commands in the program are guaranteed to be secure even if designers provide buggy otherwise clauses.

4. Related Work

Denning and Denning were one of the first to show that high level programming language techniques aided by static analysis can be used to enforce information flow policies [8]. This approach was formalized by Volpano [33] and is often implemented as language extensions to existing languages [22, 28]. A more comprehensive study of programming language techniques related to information flow security is found in the survey by Sabelfeld and Myers [26]. Other systems have explored tradeoffs between static and dynamic approaches [29], exploring the space between these two extremes—for example, inserting dynamic checks into the program and then using static analysis to verify that the program will be secure at runtime [4, 21, 25, 27].

While language-level techniques provide strong guarantees inside applications, security enforcement between applications relies on an underlying operating system. There are many approaches tackling this problem from different angles [15–17, 24, 35]. Security mechanisms at the OS level cannot provide full hardware / software system security guarantees in the face of adversaries that take advantage of information leakage in the underlying hardware implementation, such as through caches [23] and branch predictors [2]. Specific secure hardware component designs have been proposed to defend against existing covert channel attacks [34]. More systematic approaches have also been proposed to control timing channels through software/hardware contracts [36], quantitative measurements [7] or fuzzing mechanisms [20]. Towards this end, various approaches have been proposed in previous work towards analyzing and enforcing information flow security in hardware designs, including Gate Level Information Flow Tracking (GLIFT) [31] (and its extensions Execution Lease [30] and Star Logic [32]) and Caisson [18, 19].

GLIFT tracks every single bit of information in the system through each logic gate. Every bit in the system is associated with a shadow bit to represent its security label (either *High* or *Low*), and for every logic gate, a shadow logic is used to calculate the resulting security level based on the security level of inputs as well as the value of inputs. Despite GLIFT's pure dynamic nature, the tracking technique is guaranteed to be complete, i.e. it covers both implicit flows and timing channels, since all forms of information flow become explicit at the gate level. Being a fine-grained dynamic tracking technique, GLIFT can result in a substantial hardware overhead. To reduce this overhead, the authors reworked their method to be used for static analysis in the form of *Star Logic* [32]. It is important to see that Star Logic does not provide assistance or early feedback for hardware designers attempting to create secure hardware; instead it allows for the *after-the-fact static verification of a coordinated processor and kernel design*, which is not the same problem Sapper is solving.

In an attempt to bring a notion of security into hardware design languages, Caisson takes techniques from information flow security at the programming language level and applies them to HDLs, making it possible to verify hardware security policies during design. In Caisson, registers and wires are declared with static security types, and typing rules are used to enforce security policies. To avoid having to treat the design as simply a network of gates (as in GLIFT) and to allow secure switching between High states and Low states, Caisson requires an explicit state-machine-based model for designing hardware. Despite the simplicity of static type checking, it comes with two major problems: a) statically verifying those properties requires that resources be hard partitioned or even duplicated, resulting in large area overheads; and b) there is no way for the system to ever examine, react to, or affect the flow of program metadata (i.e., security labels). Labels are strictly a concept used for analysis, and have no physical manifestation in the final design.

While these past approaches represent a first generation of secure hardware design tools, both the expressiveness of those techniques (the class of hardware systems that could be shown to be secure) and the efficiency of their implementations (the amount of extra logic required to perform checks) can be prohibitively poor.

5. Conclusions and Future Work

As the current technology trends point to increasingly complex hardware platforms and more special-purpose functionality, it is time to reexamine the common assumptions that hardware is both unchangeable and always completely correct with respect to the documentation. While hardware security is a large area to explore, information flow properties are one important aspect of any secure design. This paper is a step towards a new class of tools that inform hardware designers of the information flow ramifications of their design choices and assist them in guarding against unforeseen exploits. We prototype a novel hardware description language that automatically augments a hardware design with appropriate security checks so that it is impossible to violate secrecy or integrity as defined by a policy lattice. The formal semantics and compilation tool-chain is still under development.

There are still many details to be filled in and open questions remaining, such as what kind of hardware and systems can be built using Sapper. We are aiming at using Sapper to design fully functional processors that contain modern architectural components and that are able to run realistic applications.

Acknowledgments

This research was supported by NSF CCF-1117165. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring agencies.

References

[1] OpenSPARC: World's first free 64-bit microprocessors.

http://www.opensparc.net.

[2] O. Accigmez, J. pierre Seifert, and C. K. Koc. Predicting secret keys via branch prediction. In *The Cryptographers' Track at the RSA Conference*, 2007.

[3] T. Alves. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4), 2004.

[4] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, New York, NY, USA, 2009. ACM.

[5] R. Benadjila, O. Billet, S. Gueron, and M. J. Robshaw. The intel aes instructions set and the sha-3 candidates. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '09, pages 162–178, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 2000.

[7] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. Side-channel vulnerability factor: a metric for measuring information leakage. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 106–117, Washington, DC, USA, 2012. IEEE Computer Society.

[8] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[9] A. Gattiker. An overview of integrated circuit testing methods. *Microelectronics Failure Analysis Desk Reference*, page 190, 2011.

[10] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *Security and Privacy*, 2011.

[11] Intel Corporation. AAJ1 Clarification of TRANSLATION LOOKASIDE BUFFERS, Intel® Core™ i7-900 Desktop Processor Extreme Edition Series and Intel® Core™ i7–900 Desktop Processor Series Datasheet. May 2011.

[12] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *IEEE Symposium on Security and Privacy*, 2011.

[13] K. Kaspersky and A. Chang. Remote code execution through Intel CPU bugs. In *Hack In The Box (HITB) 2008 Malaysia Conference*.

[14] C. Kern and M. Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, 1999.

[15] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[16] M. Krohn and E. Tromer. Noninterference for a practical difc-based operating system. In *Security and Privacy*, 2009.

[17] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM.

[18] X. Li, M. Tiwari, B. Hardekopf, T. Sherwood, and F. T. Chong. Secure information flow analysis for hardware design: using the right abstraction for the job. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 8:1–8:7, New York, NY, USA, 2010. ACM.

[19] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf. Caisson: a hardware description language for secure information flow. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 109–120, New York, NY, USA, 2011. ACM.

[20] R. Martin, J. Demme, and S. Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 118–129, Washington, DC, USA, 2012. IEEE Computer Society.

[21] S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 146–160, Washington, DC, USA, 2011.

[22] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. Software release. http://www.cs.cornell.edu/jif, 2001.

[23] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan*, 2005.

[24] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: practical fine-grained decentralized information flow control. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 63–74, New York, NY, USA, 2009. ACM.

[25] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 186–199, Washington, DC, USA, 2010. IEEE Computer Society.

[26] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), Jan. 2003.

[27] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Ershov Memorial Conference*, 2009.

[28] V. Simonet. Flow Caml in a nutshell. In *Proceedings of the first APPSEM-II workshop*, 2003.

[29] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS XI, pages 85–96, New York, NY, USA, 2004. ACM.

[30] M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong, and T. Sherwood. Execution leases: a hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 493–504, New York, NY, USA, 2009. ACM.

[31] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIV, pages 109–120, New York, NY, USA, 2009. ACM.

[32] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 189–200, New York, NY, USA, 2011. ACM.

[33] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4, 1996.

[34] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 83–93, Washington, DC, USA, 2008. IEEE Computer Society.

[35] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[36] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 99–110, New York, NY, USA, 2012. ACM.