A Software Solution for Hardware Vulnerabilities

Komail Dharsee
Department of Computer Science
University of Rochester

Ethan Johnson
Department of Computer Science
University of Rochester

John Criswell
Department of Computer Science
University of Rochester

***Abstract*—Modern processors are becoming increasingly complex with features that improve performance and add new functionality. However, such improvements are a double-edged sword: they improve performance and functionality but also introduce *security-critical* bugs into the processor that attackers can leverage to bypass a system's security policies. Existing solutions require hardware extensions and often lack the flexibility of software. We present the design of a software-only solution to prevent exploitation of these bugs. Our design builds on the Secure Virtual Architecture and employs both static analysis and run-time checks to prevent applications and operating system kernels from triggering security-critical processor bugs. Our evaluation examines security-critical processor bugs identified in AMD processors and analyzes how they can be mitigated by our design.**

I. Introduction

Hardware bugs affect nearly all modern processors; some such bugs introduce security vulnerabilities that attackers can exploit. Exploitation can lead to classic privilege escalation [1] and can violate compiler-enforced policies (such as Control Flow Integrity [2]) by corrupting key hardware state (such as the program counter [3]). Even new processors suffer from hardware bugs. An example is Erratum KBL038 which afflicts Intel's Kaby Lake processors [4]. The bug affects Intel's MPX hardware technology which provides hardware-accelerated memory protection. When triggered, this processor bug causes the processor to incorrectly reinitialize the MPX bounds registers, causing future bounds checks to pass regardless of whether the pointer is in bounds. If triggered, the processor will fail to enforce the security policies configured by software.

Once discovered, hardware bugs are difficult to mitigate in the field. Fixing a hardware bug requires replacing a physical component which is costly and time-consuming compared to software updates which can be performed automatically and cheaply over a network. This cost is especially problematic for mobile devices, large data centers, and cloud computing systems.

We propose a software solution which enables cheap remote software updates to mitigate new hardware bugs as they are discovered. Specifically, we propose that processors be designed, produced, and shipped in conjunction with the Secure Virtual Architecture (SVA) [5], [6]. SVA is a compiler-based virtual machine interposed between the software and processor that abstracts away unnecessary processor details and, with static analysis and compiler instrumentation capabilities, enforces security policies on software. When processor bugs are discovered, processor manufacturers can ship software patches to SVA to mitigate the new processor bugs. Our approach builds security directly into the system by providing software developers with an interface that is sufficiently expressive yet capable of controlling how computation is executed on the processor.

Devising a software-only solution requires solving several challenges:

- **Controlling Arbitrary Native Code**: Current systems allow users to execute arbitrary code on a system. Our solution must provide a way to prevent such arbitrary code from executing a sequence of instructions which may trigger a processor bug.
- **Handling Unbounded Malicious Instruction Sequences**: Some processor bugs can be triggered through an unbounded number of instruction sequences. Furthermore, malicious instruction sequences need not be a consecutive series of instructions; some set of malicious instructions interleaved with benign instructions may trigger a processor bug. We need methods to efficiently represent such infinite sets of instruction sequences and ensure that a piece of code does not execute any instruction sequence in the set.
- **Controlling Privileged and Unprivileged Code**: In addition to user space code, privileged code (such as that running within the operating system kernel) may also trigger processor bugs. Any solution must be able to control both privileged and unprivileged code.
- **Enforcing Higher-Level Security Policies**: Instruction sequences designed to trigger security-critical processor bugs may not necessarily escalate privilege as defined by the processor, e.g. improperly entering kernel mode. Rather, they may disobey expectations of logical control flow and data consistency required for enforcing higher-level security policies, e.g. control flow integrity [2] and software fault isolation [7]. For example, triggering a processor bug might spuriously update the program counter incorrectly [8], making control-flow integrity [2] impossible to enforce. Comprehensive solutions must address this wider set of security requirements.

A key insight of our approach is that it can protect against hardware vulnerabilities by limiting the expressiveness of the instruction set available to software in a way that prevents misuse of hardware features without compromising normal functionality. This is done by leveraging the virtual instruction set from previous work on SVA, which is adequately expressive to encode existing programs (including operating system kernels, such as Linux) [9], [5], while remaining sufficiently restrictive to prevent some errors from being expressed. Remaining errors are then addressed with static analysis and run-time checks [5], [6].

The rest of the paper is organized as follows. Section II discusses SVA and its capabilities in more detail. Section III presents our design for addressing hardware bugs via software. Section IV describes our preliminary study on thwarting hardware bugs with SVA. Section V gives an overview of an expected lower bound on the overhead of our system. Section VI discusses related work, Section VII discusses future work, and Section VIII concludes.

II. Background

Our solution will leverage the Secure Virtual Architecture (SVA) [5]. As Figure 1 shows, SVA is a compiler-based virtual machine residing below the operating system kernel. All software on an SVA system is compiled to a virtual instruction set based on the LLVM Intermediate Representation (LLVM IR) [10]. The original LLVM IR lacks instructions for low-level operations needed by operating system kernels to perform tasks such as MMU configuration and context switching; SVA extends the LLVM IR by introducing a set of low-level instructions, called SVA-OS, which operating system kernels can use to perform these low-level operations. As all code, including operating system kernel and application code, is compiled to SVA's Virtual Instruction Set Architecture (V-ISA) [5], SVA can enforce security policies (such as memory safety [5], [6] and control-flow integrity [11]) on all code by inserting run-time checks into the code while translating it from the virtual instruction set to the native instruction set. Additionally, SVA can employ static analysis to eliminate unnecessary run-time checks.

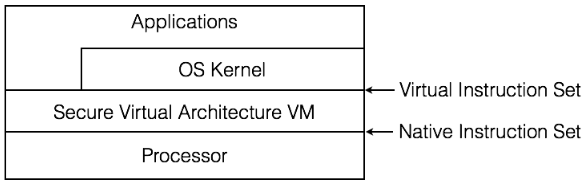

Fig. 1. Secure Virtual Architecture

On an SVA-based system, *all* software is shipped to end users as virtual instruction set code. There are three main benefits in using SVA's V-ISA to encode software that are unavailable to code expressed in most native instruction sets [5]:

- Program memory is separated into code, global variables, stack allocations, and heap memory. This design alleviates the need to reconstruct information on variables from machine code e.g., by using Value Set Analysis [12]. Additionally, by having an explicit code segment protected by code segment integrity [6], [11], SVA alleviates the need to analyze self-modifying code.
- Any function expressed in the V-ISA cannot have any local branches computed at run-time. Explicit local control-flow graphs alleviate the need to construct the local control-flow graph for data-flow analysis.
- The V-ISA's use of an infinite set of virtual registers in Static Single Assignment (SSA) form enables efficient yet powerful data flow analysis. Improvements to analysis include simplifying def-use graphs and avoiding anti/output data dependencies [10]. Some program analysis algorithms, such as constant propagation [13] and flow-sensitive points-to analysis [14], can be performed more efficiently on code in SSA form.

Because SVA translates all user-mode and kernel-mode code from virtual instruction set code to native instruction set code, SVA can enforce security policies on all code executed on the processor [5]. We exploit this capability to defend against hardware bugs. Using a mix of static analysis techniques and instrumentation adding run-time checks to code, we can guarantee that conditions known to trigger hardware bugs cannot arise.

III. Design

There are three methods that the SVA Virtual Machine (SVA VM) can employ to thwart security-critical processor bugs: it can use a code generator that never generates instruction sequences that trigger a processor bug; it can verify that the native code it has generated for a program does not trigger a processor bug; or it can insert run-time checks into that native code to check whether a processor bug is about to be triggered. Figure 2 shows the compilation pipeline.

A. Safe Code Generation

The best solution for thwarting security-critical processor bugs is to ensure that no native code running on the system can trigger such a bug. For some processor bugs, this can be done by modifying the SVA native code generator to ensure that it never generates bug-triggering instruction sequences. Such a solution is unlikely to incur overhead so long as a similarly fast instruction sequence can generate the same result on the processor without error.

For example, there exists a processor bug which is triggered by some invocations of the `FSINCOS` instruction; under some circumstances, execution of this instruction will corrupt the program counter [8]. For processor bugs such as these, the SVA code generator can be modified to emit separate `FSIN` and `FCOS` instructions that do not trigger the bug instead of the `FSINCOS` instruction which does. More generally, the SVA virtual code to native code translator can be modified to avoid generating instruction sequences that trigger processor bugs.

B. Verifying Native Code

In addition to modifying the code generator to proactively avoid generating sequences of instructions which are known to trigger processor bugs, SVA could also statically detect, after native code generation, whether its code generator produced code that can trigger a processor bug. This approach can

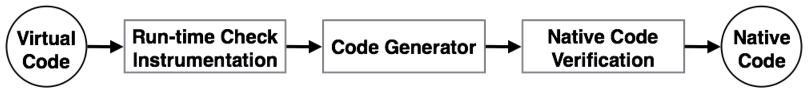

Fig. 2. Secure Code Generation Pipeline

address a much wider variety of processor bugs than those that can be "easily" prevented through simple code generator modifications - for instance, bugs that can be triggered by a large or infinite number of different instruction sequences. It can also verify that modifications to the code generator which avoid generating bug-triggering instruction sequences (as Section III-A describes) were performed correctly. The SVA VM can perform this static analysis prior to running a program; if it observes that the code generator has generated a potentially dangerous sequence of instructions, it can take some protective action (such as refusing to run the program or alerting the system administrator).

We have developed a static analysis procedure which builds a context-free grammar encoding the set of instruction sequences that a program can execute when it runs. Each static instruction in the program is a terminal symbol in the grammar, and each sentence in the grammar's language represents a sequence of dynamic instructions that can arise in some execution of the program. In other words, a sentence can be thought of as a trace of some potential execution of the program, listing the sequence of static instructions (terminal symbols) in the order that they would be executed. The sentence is in the grammar's language if and only if the program's control-flow graph would permit that path through the program to be followed in some execution. Because SVA can enforce control flow integrity (CFI) at run time [5], [11], memory safety errors cannot cause programs to deviate from the control-flow graph used by this analysis; thus, we can be sure that the grammar encompasses all instruction sequences that the program could possibly execute. The grammar accounts for function call/return context in determining whether a trace is valid (e.g., a function cannot return to a different location than where it was called); this invariant can be enforced at run time by using a version of CFI with a shadow stack [2]. The only (potential) source of imprecision in the control-flow graph is due to over-approximation of the potential values of function pointers; since this is also a limitation of CFI, the grammar therefore models exactly the range of behavior that can be exhibited by the program in the presence of memory safety errors.

Using this grammar, we can statically determine whether it is possible for a sequence of dynamic instructions to arise during execution that could potentially trigger a particular hardware bug. To do this, we first characterize all sequences of dynamic instructions that can trigger the bug as a *regular language*. From this, we can construct a finite state automaton (FSA) that checks whether or not a particular sequence of dynamic instructions could trigger the bug. We can then intersect this FSA with a pushdown automaton (PDA) parser for the control-flow grammar that we generated for the program under analysis [15]. The intersection produces a new PDA, the language of which is precisely the set of possible program traces which include bug-triggering sequences. This language will be empty if and only if no execution of the program can potentially trigger the bug. Thus, if the language is empty, we know the program is safe; if not, we conclude that the program is unsafe.

Intersecting the FSA and PDA and checking whether the resultant PDA represents an empty language are both decidable operations that can be performed in linear time [15]. This allows the SVA VM to cheaply and accurately determine at compile time whether a program can potentially trigger a hardware bug.

A potential concern with this method is that the class of regular languages could limit our ability to precisely express bug trigger conditions. It is conceivable that some hardware bugs may be triggered by instruction sequences that cannot be characterized as a regular language. Our preliminary investigations (Section IV-B) did not identify any real-world bugs whose conditions could not be expressed as regular languages; however, our sample size is small and this question bears further investigation. Bugs with non-regular conditions can still be addressed with this approach if the user is willing to tolerate some false positives (harmless programs could be categorized as buggy) by defining a FSA whose language is a regular superset of the actual trigger conditions. We present ideas for more satisfactorily addressing this limitation in Section VII.

C. Run-time Checks

Some processor bugs are not triggered by unusual instruction sequences. Rather, they are triggered by using benign instructions to place the processor into a particular state. For example, some AMD processors suffer from a bug in which the processor generates a debug trap when executing the `syscall` instruction [16]. A strict set of conditions must be met to trigger the bug; one of the conditions requires that the `syscall` instruction be marked as a breakpoint [16].

SVA can insert run-time checks into code during native code generation to prevent a program from triggering such bugs. Native code instructions that could place the processor into a vulnerable state will be instrumented with run-time checks that will ensure that the processor is not placed into an unsafe state. In the case of the aforementioned `syscall` bug, the virtual instruction for setting breakpoints could be enhanced with a run-time check verifying that the address for the breakpoint does not contain a `syscall` instruction. If an attempt to set a breakpoint fails, the virtual instruction can either generate a trap or return an error code indicating that the breakpoint could not be set.

As run-time checks can add overhead, they will only be used as a last resort for processor bugs that cannot be thwarted through other methods.

IV. Experimental Results

A. Bug Survey Statistics

We analyzed the 27 security-critical processor bugs surveyed by Hicks et. al. in their work on SPECS [17]. We studied each bug and determined the feasibility of thwarting the bug using one of our three techniques.

Figure 3 shows the results. Each section in the chart shows the proportion of hardware bugs which we categorized according to the mitigation mechanisms applicable to each bug. We have published details on this survey in a separate technical report [18].

We aimed to categorize each bug into one of three deterrence categories: (1) configuration of *SVA-OS*, (2) analysis with *grammar checks*, or (3) instrumentation with *run-time checks*. Any bugs that could not be completely mitigated, and thus did not fall into any of these categories, are classified as either (4) *SVA Cannot Mitigate*, (5) *Insufficient Documentation*, or (6) *No Hardware Virtualization Support*. Each category is described below:

- *SVA-OS*: Bugs within this category are mitigated by some aspect of system configuration being performed through SVA-OS. SVA-OS abstracts and controls various system configuration operations, such as MMU configuration, interrupt handler configuration, and processor features controlled by Machine Specific Registers (MSRs) [6]. The SVA-OS implementation can be enhanced to perform run-time checks to ensure that configuration does not trigger a processor bug.
- *Grammar Checks*: Bugs that can be mitigated using safe code generation and static verification of native code, as discussed in Sections III-A and III-B, are placed in this category.
- *Run-time Checks*: Any bugs which are deterred through the use of run-time checks that are inserted into the code during native code generation, as discussed in Section III-C, fall into this category. This does not include run-time checks that are manually added to the implementation of the SVA-OS instructions.
- *Insufficient Documentation*: Processor manufacturers do not always provide precise details on security-critical processor bugs. The lack of mitigation techniques for such bugs is not due to limitations in our approach but due to the lack of information available for such processor bugs.
- *SVA Cannot Mitigate*: Some bugs cannot be mitigated by SVA; we placed such bugs into this category.
- *No Hardware Virtualization Support*: SVA does not currently provide support for operating systems and software running on it to take advantage of hardware virtualization extensions (VMX) [19]. Thus, it is immune to processor bugs involving VMX, but only incidentally. VMX-related bugs will need to be mitigated in future versions of SVA that add support for this feature.

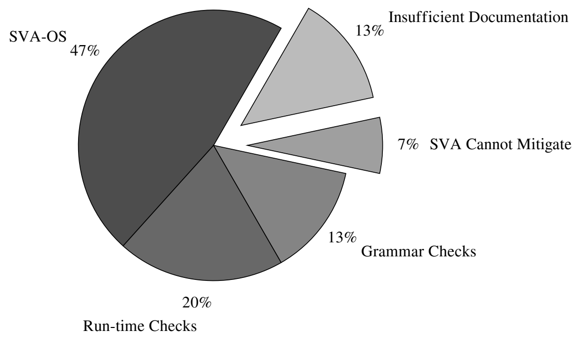

Fig. 3. Processor Bug Analysis

As noted above, we did not analyze bugs involving VMX extensions. These comprise 46% of the 28 bugs studied by Hicks et al. and are not shown in Figure 3. Even so, our study shows that our techniques can thwart 43% of the 28 processor bugs that we studied. Of the non-VMX related bugs (15 total), 47% can be mitigated using SVA-OS, 20% can be mitigated using run-time checks, and 13% can be mitigated using static analysis grammar checks.

Our analysis shows that our design cannot mitigate 27% of the non-VMX-related bugs. However, 13% of those bugs are due to insufficient documentation. It is possible that, with more information, an SVA-based mitigation could be designed.

B. Bug Finite State Automata

Of the 28 bugs considered in our study, we identified two as being suitable for detection with our automata-based static analysis technique. We have created finite-state automata characterizing the conditions triggering both of these bugs, establishing that they can be represented as regular languages. These examples are discussed in more detail below.

1) Example: A Single Bad Instruction: The bug published in Erratum #573 for AMD's Family 11h processors involves a particular instruction, `FSINCOS`, which can sometimes corrupt the program counter when executed [8]. Any instance of this instruction can potentially trigger the bug. This can be represented by a very simple regular language: the language contains precisely all possible sentences (program traces) which contain one or more `FSINCOS` instructions (Figure 4). As noted in Section III-A, this is an example of a bug which can be proactively avoided by modifying the SVA native code generator so that it simply selects a different instruction sequence that performs an equivalent computation. However, a grammar check can provide a second line of defense to confirm that such code generator modifications are indeed in effect.

2) Example: A Long Sequence Triggering a Bug: Erratum #639 for AMD's Family 14h processors describes a bug which can result in the program counter being corrupted when a `CALL RSP` instruction is executed after a long sequence of at

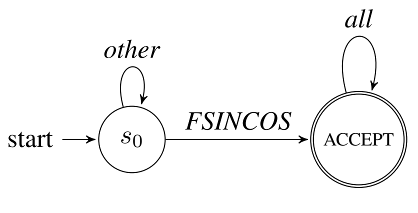

Fig. 4. Finite State Automaton for Erratum #573

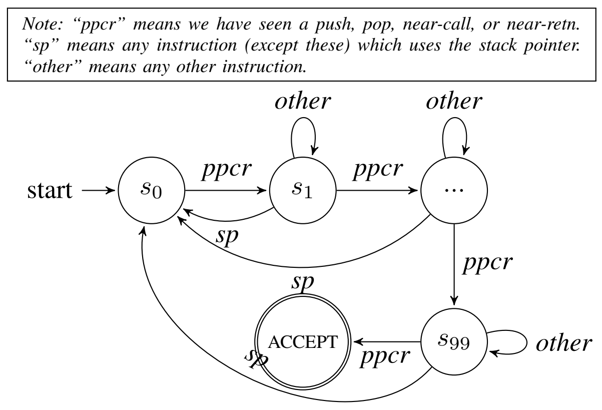
Note: "ppcr" means we have seen a push, pop, near-call, or near-retn. "sp" means any instruction (except these) which uses the stack pointer. "other" means any other instruction.

Fig. 5. Finite State Automaton for Erratum #639

least 100 pushes, pops, near-calls and/or near-returns with no other operations in between that use the stack pointer (register `RSP`) [1]. Unlike the bug we considered in Section IV-B1, these conditions cannot be narrowly addressed with a code generator modification. It is, however, straightforward to precisely represent the conditions which trigger this bug as a regular language: we can design a FSA (Figure 5) that accepts a program trace (i.e., declares it to be vulnerable to the bug) if and only if it sees 100 pushes, pops, near-calls and/or near-returns not separated by some other instruction that uses the stack pointer.

V. Performance Overhead

Previous work on KCoFI [11], a system built using SVA [5], provides a lower bound on the expected overhead that would be introduced by our proposed design for deterring security-critical processor bugs. An implementation of our design would build on the existing KCoFI infrastructure as KCoFI provides the minimal protections needed by our design (control-flow integrity and code segment integrity).

Criswell et al. [11] evaluated KCoFI's performance with the `thttpd` web server, an OpenSSH Secure Shell server, the LMBench microbenchmarks which measure the performance of kernel services [20], and Postmark, a benchmark which simulates mail activity [21].

For the `thttpd` web server, Criswell et al. measured the bandwidth provided by the server by transferring files ranging in size from 1 KB to 2 MB using multiple simultaneous connections; their results showed that the average change in bandwidth was negligible compared to the native FreeBSD kernel [11]. Similar tests on the OpenSSH server using a single network client showed that bandwidth was reduced by 13% on average with a maximum reduction of 27%. Using LMBench [20] to measure the latency of basic kernel operations (such as opening/closing files, creating processes, and signal handler dispatch), Criswell et al. showed that KCoFI decreases kernel-mode performance by a factor of 2 to 3.5. However, this fairly high number was only indicative of kernel-mode execution; the execution of the previously mentioned user-space applications showed much less overhead.

Finally, Postmark was used to further measure file system performance, which showed results similar to the more kernel intensive execution under LMBench [11]. On average, execution took approximately two times longer when compared to the native FreeBSD kernel, consistent with the LMBench results.

VI. Related Work

Several previous approaches have been designed to address processor bugs. However, all such approaches require changes to the processor. Our work, in contrast, requires no processor changes and can be deployed on existing processors.

Man-Lap Li et al. characterized hard errors [22] and used it as ground work for SWAT [23]. SWAT [22], [23] uses low-cost hardware and software monitors to detect hard errors via a thin firmware layer which coordinates detection, diagnosis, and recovery components of their system. SWAT employs a checkpoint and replay system [23] to retry the computation to determine if the error was a soft (intermittent) error or a hard (permanent) error; on a SWAT system, execution must be moved to a fault-free core upon detection of a hard error. Our approach handles design flaws which may affect all cores on a system and is capable of preventing the exploitation of some bugs.

SPECS [17] adds additional hardware that enforces assertions at run-time. When assertions fail, an exception handler for recovery from the caught faulty execution is run. This approach is low-overhead but requires assertions to be deployed with the processor; changes cannot be made after deployment. Our approach alleviates the need for new hardware and can be updated as new hardware bugs are discovered.

Systems such as Phoenix [24] and the work of Constantinides et al. [25] monitor the control signals on a processor for signs of a processor fault; the patterns for which to search can be updated by software. These systems can be updated like software. However, they still require the processor supplier to add specialized hardware. Phoenix's specialized hardware is composed of four units responsible for (1) programming Phoenix, (2) selecting particular logic signals to be monitored, (3) translating the set of selected signals into logic which flags detected defects, and (4) recovery. In contrast, our approach works on existing commodity processors.

The Access-Control Extension (ACE) [26] extends the processor instruction set with new instructions that can query internal processor state; privileged firmware periodically executes and uses these instructions to test the processor for defects. Because ACE scans the processor periodically, it may fail to detect a vulnerability until after damage has been done. Our approach, on the other hand, detects errors before they are exploited.

VII. Future Work

In future work, we will implement defenses for the hardware bugs studied in the SPECS project [17] in SVA and evaluate the performance of both the static analysis and run-time checks. We will also evaluate our work on a larger corpus of processor bugs.

The performance overhead of our system needs to be thoroughly evaluated. Run-time checks are likely to increase execution time. Additionally, replacing instructions during code generation with others that are safe may reduce performance as the new instruction sequences may not yield identical performance.

Our solution makes the assumption that *most* of the functionality of the processor is correct, allowing us to use static analysis and run-time checks. Future research can examine what subset of processor functionality is needed for our security checks and whether strong verification of that subset could guarantee that our checks are correct. If possible, such work could yield very secure processors without greatly increasing verification efforts.

There exist a significant number of processor bugs which are triggered “under a complex set of internal timing conditions” [1]. We surmise that these bugs occur due to errors in processor features such as pipelining, out-of-order execution, register renaming, and speculation. Even if instruction sequences for triggering the bug are known, preventing code that can trigger the bug could prevent benign code from executing. One possible remedy would be to enhance the SVA code generator to insert code to flush the processor pipeline when a program could trigger one of these processor bugs. Such an approach would allow benign code that would otherwise be disallowed from executing to run (albeit with higher execution time).

We will expand our survey of bugs to identify more that are triggered by unusual instruction sequences and thus suitably mitigated with our static verification approach in Section III-B. This will allow us to better understand whether the constraint of needing to express bug conditions as a regular language is a limitation in practice. If so, we will explore ways of dealing with false positives (harmless programs categorized as buggy due to use of a FSA that over-approximates a bug’s conditions) that allow the code to still be run without risking a security breach. For instance, if we can mitigate potentially buggy code by selectively disabling CPU optimizations responsible for the bug in question (as discussed above) or through targeted recompilation of potentially buggy code to avoid the trigger conditions, false positives would only potentially reduce performance, not functionality. Alternatively, we could instrument code statically identified as buggy with more precise run-time checks, again trading performance to maintain full functionality of genuinely harmless programs.

VIII. Conclusions

This paper presents a software solution for processor bugs utilizing the Secure Virtual Architecture (SVA). Since all software on an SVA system is shipped as virtual instruction set code which is translated to native code by the SVA Virtual Machine, SVA can control whether native code triggers a processor bug. We proposed three methods of thwarting security-critical processor bugs (safe code generation, native code verification, and run-time checks) and demonstrated the applicability of our native code verification technique on two real processor bugs.

Acknowledgements

We thank Professor Joel Seiferas of the University of Rochester for inspiring the idea behind the grammar-based verification approach (Section III-B). We also thank the anonymous reviewers for their helpful feedback. This work was supported by National Science Foundation Award Number 1463870.

References

[1] A. M. D. Inc., “Revision guide for AMD family 14h models 00h-0fh processors,” February 2013, http://support.amd.com/TechDocs/47534_14h_Mod_00h-0Fh_Rev_Guide.pdf. [Online]. Available: http://support.amd.com/TechDocs/47534_14h_Mod_00h-0Fh_Rev_Guide.pdf

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” *ACM Transactions on Information Systems Security*, vol. 13, pp. 4:1–4:40, November 2009. [Online]. Available: http://doi.acm.org/10.1145/1609956.1609960

[3] A. M. D. Inc., “Revision guide for AMD family 15h models 00h-0fh processors,” September 2014, http://support.amd.com/TechDocs/48063_15h_Mod_00h-0Fh_Rev_Guide.pdf. [Online]. Available: http://support.amd.com/TechDocs/48063_15h_Mod_00h-0Fh_Rev_Guide.pdf

[4] Intel Corporation, “7th generation intel® processor family specification update,” February 2017. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/7th-gen-core-family-spec-update.pdf

[5] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, “Secure Virtual Architecture: A safe execution environment for commodity operating systems,” in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP’07. Stevenson, WA: ACM, 2007, pp. 351–366. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294295

[6] J. Criswell, N. Geoffray, and V. Adve, “Memory safety for low-level software/hardware interactions,” in *Proceedings of the 18th USENIX Security Symposium*, ser. Security’09, 2009, pp. 83–100. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855774

[7] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, ser. SOSP’93. Asheville, NC: ACM, 1993, pp. 203–216. [Online]. Available: http://doi.acm.org/10.1145/168619.168635

[8] A. M. D. Inc., “Revision guide for AMD family 11h processors,” December 2011, http://support.amd.com/TechDocs/41788_11h_Rev_Gd.pdf. [Online]. Available: http://support.amd.com/TechDocs/41788_11h_Rev_Gd.pdf

[9] J. Criswell, B. Monroe, and V. Adve, “A virtual instruction set interface for operating system kernels,” Boston, MA, USA, June 2006, pp. 26–33.

[10] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO'04. Palo Alto, CA: IEEE Computer Society, 2004, pp. 75–86. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[11] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2014, pp. 292–307. [Online]. Available: https://doi.org/10.1109/SP.2014.26

[12] T. Reps and G. Balakrishnan, "Improved memory-access analysis for x86 executables," in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, ser. CC'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 16–35. [Online]. Available: http://dl.acm.org/citation.cfm?id=1788374.1788377

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, pp. 13(4):451–490, October 1991.

[14] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 289–298. [Online]. Available: http://dl.acm.org/citation.cfm?id=2190025.2190075

[15] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 2nd ed. Addison-Wesley, 2001.

[16] A. M. D. Inc., "Revision guide for AMD family 10h processors," March 2012, http://support.amd.com/TechDocs/41322_10h_Rev_Gd.pdf. [Online]. Available: http://support.amd.com/TechDocs/41322_10h_Rev_Gd.pdf

[17] 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 *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 517–529. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694366

[18] K. Dharsee, E. Johnson, and J. Criswell, "Hardware vulnerability and mitigation survey," Tech. Rep. TR 1000, July 2017. [Online]. Available: http://hdl.handle.net/1802/32871

[19] A. M. D. Inc., "Secure virtual machine architecture reference manual," *AMD Publication*, no. 33047, 2005.

[20] L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, San Diego, CA, 1996, pp. 23–23. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268299.1268322

[21] Postmark, "Email delivery for web apps," July 2013. [Online]. Available: https://postmarkapp.com/

[22] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 265–276.

[23] M.-L. Li, P. Ramachandran, S. K. Sahoo, and S. A. V. S. A. Y. Zhou, "Swat: An error resilient system," *Proceedings of SELSE*, 2008.

[24] J. T. Smruti R. Sarangi, Abhishek Tiwari, "Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware," in *Proceedings of 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-39*, Dec 2006, pp. 26–37.

[25] K. Constantinides, O. Mutlu, and T. Austin, "Online design bug detection: Rtl analysis, flexible mechanisms, and evaluation," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, pp. 282–293.

[26] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "Software-based online detection of hardware defects mechanisms, architectural support, and evaluation," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 97–108.