Model Checking to Find Vulnerabilities in an Instruction Set Architecture

Chris Bradfield
Weebly, Inc.
cbradfield@unc.edu

Cynthia Sturton
University of North Carolina at Chapel Hill
csturton@cs.unc.edu

***Abstract*—Hardware companies conduct extensive testing and verification during the processor design process to reduce the number of errata that persist to the final product. These processes rely on a specification against which to test or verify the design; as a result, they will fail to catch vulnerabilities stemming from errors in the specification itself. In this work we present a model-checking based approach for detecting such vulnerabilities. Our approach is feasible, even for a modern CISC architecture, given the class of properties we are interested in. We demonstrate the value of this approach with a case study of the Intel SYSRET vulnerability.**

I. Introduction

Errata in processors exist. Hardware design and fabrication companies like AMD and Intel regularly publish lists of known errata for their processors (e.g., [8], [7]). Given that modern processor designs require millions of lines of hardware design language (HDL) code [4] it is perhaps no surprise that bugs in the code survive to the final shipped product. This is despite the fact that hardware companies make heavy use of formal verification techniques during the design phase and conduct extensive testing throughout the design and build processes.

Intel defines errata as "design defects or errors [that] may cause ... behavior to deviate from published specifications" [8]. However, this presupposes that the published specifications are themselves correct. It is by no means obvious that this will be true. The specification is large and complex. For example, the Intel 64 and IA-32 Architectures Software Developer Manuals are a three volume set containing over 3400 pages [3] – too large for a person to understand and reason about the specification and all the interactions between modules completely. And as new features are added, but backward compatibility is maintained, complexity of the specification continues to grow. In this work we examine errors in the instruction set architecture (ISA), the specification defining the hardware interface that is presented to software.

Recently, we have seen an example of behavior in an Intel processor leading to a security vulnerability that is exploitable by software [5], [6]. Researchers have shown how the vulnerability could be exploited to allow a guest operating system to escape its virtual machine and gain hypervisor-level privilege on the system [19]. The behavior is not an erratum, according to Intel's definition, because the behavior is consistent with the specification. However, the vulnerability arose because the SYSRET instruction, which is responsible for transitioning the CPU from kernel mode back to user mode, could be interrupted by a general protection fault while the system was in the middle of making the switch and temporarily in an inconsistent state. As a result, user-level (ring 3) code would run with supervisor (ring 0) privilege. We argue that while this may not constitute a processor erratum, it is an example of an error in the ISA. Instructions that may temporarily put the system in an inconsistent and vulnerable state should maintain control of execution until system state has reached a consistent point again.

We present a model checking-based approach to detecting vulnerabilities in the ISA that could leave software running on the CPU vulnerable to attack. Our presentation includes the following.

- We define the class of security property that can both be feasibly verified for a modern CISC ISA and be useful in detecting dangerous security vulnerabilities.
- We provide concrete examples of instructions in the x86-64 ISA for which the defined security property should hold.
- We use the SYSRET vulnerability as a case study and provide a preliminary evaluation of our methodology. We show that our approach finds the vulnerability in the Intel x64 ISA, but not in AMD64, which matches our expected results.

II. Model Checking

In general, model checking does not scale well to such a problem. One naive model checking approach would require creating models for every instruction (over 800 for Intel's x64 [3]) – typically a manual process – and then checking whether a security property of interest holds over any possible sequence of instructions. And, even then, the large state space and the possibility of infinitely long instruction sequences would preclude the possibility of proving a given property holds. The most we could offer is bounded model checking, proving the property is never violated in the first k steps of execution, starting from any initial state. This can be useful as a bug finding tool, but can not provide verification that the property will continue to hold after k steps.

We take an alternative approach and start from an unconstrained initial state which represents all possible states. From this unconstrained initial state we model the execution of a single instruction i and check that the property holds over the resulting states. These states represent an over-approximation

978-1-4673-8826-9/16/$31.00

of the set of states reachable by any sequence of instructions ending with instruction i. If the property is shown to hold for this set, then it will hold for any state reachable by executing i, starting from any reachable state.

This approach has the benefit that it does not require modeling all instructions in order to check the behavior of a single instruction. It also does not require checking a sequence of instructions. However, the over-approximation of the state space will include some unreachable states and the model checker may report a property violation when in fact no reachable state can ever violate the property. In our case study we incurred no false positives, and in ongoing research we are extending the evaluation in order to quantify the risk.

A. Security Properties

We are interested in properties that identify when the CPU is in a dangerous state. Initially, we define the state to be dangerous when the privilege level of the CPU is elevated (e.g., ring 0), but either the stack pointer, base pointer, or instruction pointer are user controlled.

Software may deliberately enter in to such a state. And in fact, it commonly does. We look at the SYSCALL/SYSRET instructions as an example. They are fast system call instructions and, by convention, when invoked the syscall handler must first save the user-level stack pointer (`%RSP`) and set `%RSP` to the kernel stack before beginning to handle the system call. At the end of the call, the handler restores the saved user-level `%RSP` and executes the SYSRET instruction. At the beginning and end of the handler's execution the CPU is in a dangerous state: the CPU mode is privileged, but the stack pointer contains a user-controlled value. The solution to this is for software to mask interrupts and exceptions until it can restore the CPU to a safe state. In this way the software that created the dangerous state maintains control for as long as the state remains vulnerable.

The issue arises when an individual instruction for which exceptions can not forcibly be masked by software operates over potentially dangerous state. In this case, the safe approach is for the ISA to define the instruction such that no exception may occur in the middle of execution. It is violations of this property that we look for, and the SYSRET vulnerability provides one such example.

B. Choosing Instructions to Model

We focus on instructions that update privilege level from low to high and back. These are the instructions most at risk for being interrupted while in a dangerous state. We identify the following list of instructions of interest for the x86-64 ISA:

- SYSCALL/SYSRET
- SYSENTER/SYSEXIT
- INT/IRET
- CALL
- RSM

C. Modeling State

We focus only on properties over the architecturally visible (i.e., visible to software) CPU state. This is key to our approach. There are few hardware-enforced constraints on how software may manipulate the architecturally visible state. An operating system may, with the support of hardware, constrain how user-level code may update CPU state, but privileged code is relatively free to update state as it desires. This property works in our favor as it lowers the risk of false positives in our evaluation. If we find a property violation, there is a good chance it is possible to write code that can run on the CPU and trigger the property violation. In other words, if we find that instruction i, when executed while the CPU is in state s violates one of our security properties, it is likely that software exists that can bring the CPU to state s. While convention may prevent s from being commonly reached – for example, convention dictates that the OS is responsible for saving and restoring user-level state for the SYSCALL and SYSRET instructions – it is preferable to either find and remove the insecure states or state the convention as a requirement in the description of the ISA.

III. Case Study

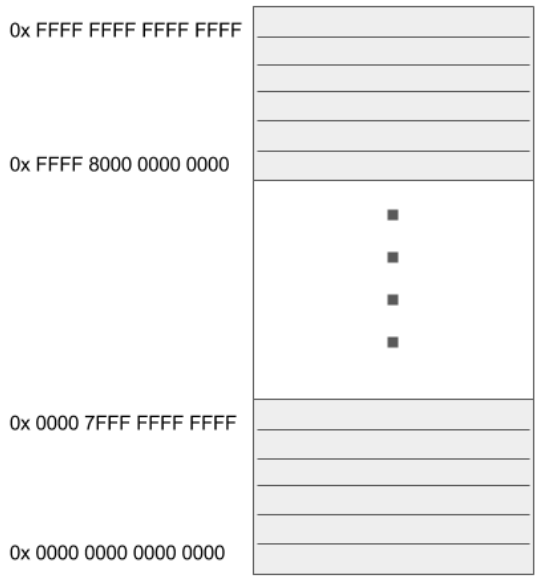
We start with a description of the SYSRET vulnerability that motivates our work. In 2000 AMD defined a 64-bit instruction set architecture (AMD64) that is backwards compatible with the 32-bit x86. Shortly thereafter, Intel developed their version of the ISA (Intel 64).[1] Although Intel 64 is intended to be compatible with AMD64 some subtle differences have been found. One of those differences is in how a general protection fault (#GP) is handled when returning from a SYSCALL instruction. The so called SYSRET vulnerability resulted from this difference.

A. SYSRET Vulnerability

Before describing the vulernability, we first provide some necessary background information. Although AMD64 is defined for a 64-bit architecture, AMD limited the virtual address space to 48-bit addresses. (Doing so limits the depth of the page walk needed to translate addresses – 4 pages deep, rather than 6.) In addition, the processor requires that the high 16 bits (bits 63-48) of a virtual address have the same value as bit 47. An address in this form is said to be *canonical*. The processor will throw a general protection fault (#GP) any time a 64-bit value is used as a virtual address and does not conform to these rules. Enforcing the rule that addresses be in canonical form serves two purposes: it prevents OS developers from using the ignored space as a place to store meta data (e.g., flags or pointer bounds), which would then be broken in future upgrades that do use the space; and it means the usable address space grows up from the bottom and down from the

[1]Because AMD64 is backwards compatible with the x86 family of processors it is often referred to as x86-64. We will use AMD64 and Intel 64 to refer specifically to each company's ISA, and will use x86-64 when we want to refer generically to the 64-bit ISA without focusing on a particular company's version.

Fig. 1. The canonical addresses are shown in gray.



top, and will continue to do so as more address bits are used. Figure 1 illustrates the idea.

The SYSRET instruction returns the processor from privileged-level execution back to user-level execution. When transitioning from user to supervisor and back again, the hardware is responsible for saving and restoring the instruction pointer (%RIP) and changing the privilege level (i.e., the code segment selector register), while the software is responsible for saving and restoring all other registers, including the stack pointer. On SYSCALL, the CPU saves the guest %RIP to the %RCX register and restores it from the %RCX register during the SYSRET instruction. Because the value stored in %RIP will be used as an address the processor will throw a #GP fault if the value is not in canonical form.

The vulnerability arises from a subtle difference in how AMD and Intel processors handle the SYSRET instruction. A processor implementing AMD64 will first change the CPU mode from privileged level down to unprivileged level and then restore the guest %RIP from the %RCX register. A processor implementing Intel 64 will flip the order, restoring the guest %RIP first and then changing the CPU mode. The issue occurs if the %RCX register holds a value that is non-canonical. As soon as the value is copied to %RIP (an address-holding register), the processor will throw a #GP fault if the address is not in canonical form. In the case of the AMD processor the fault happens after the privilege level has been lowered to guest mode. In the Intel processor, however, the CPU mode has not yet been restored at the time of the fault; the CPU is still in privileged mode. And by convention, it is the job of the supervisor to save and restore the user's stack pointer, which means that at the point that the fault occurs the supervisor has already restored the user's stack pointer. The result is that the CPU is in privileged mode, the stack pointer is controlled by user-level code, and execution has been diverted to an exception handler. The result is a vulnerability that can be exploited to give the user privileged control of execution [9].

There is not consensus on whether this counts as a bug in the hardware. The CPU implementation does meet the Intel 64 specification, and therefore can be considered correct. A supervisor that ensures the %RCX register is in canonical form before issuing the SYSRET instruction will never encounter the issue. For example, the system-level fix for the Xen hypervisor is for software to check whether %RCX is in canonical form and if it is not to use the slower IRET (Interrupt RETurn) instruction to return from the hypervisor to the guest. On the other hand, Intel 64 was meant to be compatible with AMD64, and software written for AMD64 should be able to run safely on Intel 64. So perhaps the ISA is buggy. Regardless, we are interested in developing a methodology to detect this vulnerability and others like it. From a security point of view, it is not best practice for a specified, legal general protection fault to put the CPU in an insecure state. In other words, it should not be the software's job to determine when a specified exception will be dangerous and should be avoided at all costs.

B. Model and Property

Our models include the architectural state either read or written in the instructions' pseudocode, which for instructions that change privilege levels includes the current privilege level as given in the code segment register. The models also include a bit for each fault the instruction can possibly throw. We use the bit to indicate whether the particular fault has been thrown at each point in the model's execution. In addition to the state explicitly involved in the instruction, we also include all architectural state that could contribute to a dangerous state, as we define it. In other words, we include in all our models the stack pointer, base pointer, and instruction pointer, whether or not they are used by the instruction. Finally, for each state variable (e.g., %RSP), we include a bit indicating whether the represented register is controlled by user-level or supervisor-level code at each point of execution.

Even a simple model of a single instruction can result in too large of a state space. The problem arises because each state register is 64 bits, meaning 2^{64} possible states for each register. For an instruction that modifies m registers, the number of possible states to explore is 2^{64m}. For even relatively simple instructions, the number can grow quickly. We therefore make a simplification in our model: we abstract the register state space and keep only the information that can affect the control flow through the instruction's pseudocode. For example, %RIP is one of the registers updated by the SYSRET instruction. We abstract its value to capture only whether or not it is in canonical form. If it is not this would cause a #GP fault, which affects control flow.

We constrain the initial state of our model as little as possible. We do constrain the possible exceptions to be in a "not-thrown" state, as the instruction has not yet started executing. All other state we leave unconstrained. In particular, the stack, base, and instruction pointers are free to be either

controlled by user-level code or supervisor-level code. State that affects control flow is fully unconstrained as well.

We formalize the model as a Kripke structure $\mathcal{M} = (S, S_0, R, \mathcal{L})$, where

- S is the set of states – valuations to CPU registers and flags – that the CPU transitions through during the execution of a single instruction;
- $S_0 = \{s \mid \texttt{GP} \notin \mathcal{L}(s)\}$ is the set of initial states defined as the set of states in which a #GP fault has not yet been thrown;
- $\mathcal{R} \subseteq S \times S$ is the transition relation given by the pseudocode of AMD and Intel manuals; and
- $\mathcal{L} : S \to 2^{AP}$ is a labeling function that assigns atomic propositions to each state. The labels in AP define: a) for each CPU register, whether it is controlled by user-level code or supervisor-level code; b) for each register in $\{\%\texttt{RIP}, \%\texttt{RSP}, \%\texttt{RBP}\}$, whether the register is in canonical form; c) whether a general purpose fault has been thrown; and d) the current privilege level (CPL) of the CPU.

The question we are interested in is whether a #GP fault can ever be thrown while the CPU has an elevated privilege level (CPL $= 0$), but at least one of the stack, base, or instruction pointer is user controlled. Formally, we are interested in verifying the invariant $G\phi$ ("globally ϕ"), where ϕ is defined as follows:

$$\begin{aligned}\phi \doteq \forall s \in S_0, \forall t \in S. \quad & R^*(s,t) \rightarrow \\ & t.\texttt{GP} \wedge t.\texttt{CPL} = 0 \rightarrow \neg t.\texttt{RSP-user-controlled} \quad \wedge \\ & \neg t.\texttt{RBP-user-controlled} \quad \wedge \\ & \neg t.\texttt{RIP-user-controlled}.\end{aligned}$$

We use the notation $s.l$ to denote the case that label $l \in AP$ is true in state $s \in S$. In other words,

$$s.l \doteq s \in S \wedge l \in \mathcal{L}(s).$$

We use $R^*(s,t)$ to denote the transitive transition relation between states s and t. In other words, there exist some number of states $s_1, s_2, \ldots, s_n$ such that $R(s,s_1) \wedge R(s_1,s_2) \wedge \ldots \wedge R(s_n,t)$.

C. Implementation

We use DynAlloy [2] as our model checking tool. It is an extension to Alloy, a language and model checker for first-order logic. The DynAlloy extension provides a procedural specification language front end. It is written in Java and provides a graphical interface that makes it easy to explore models and understand reported counter-examples to a given property. The Java implementation means DynAlloy is not as fast as some other model checkers, but for our models, all properties verified in a few seconds.

We use the pseudocode given in the Intel and AMD developers' manuals ([3], [1]) to build the models of both the Intel 64 and AMD64 versions of SYSCALL and SYSRET. The pseudocode is written in a procedural C-style language and it is relatively straightforward to develop a model by following the code.

D. Results

DynAlloy found the dangerous exception in our model of Intel's SYSRET. It noted that starting from an initial state where the stack pointer is controlled by user-level code and the current privilege level is high, the #GP fault will be thrown when the CPU is in a dangerous state if the value in %RCX is not canonical.

For our model of AMD's version of SYSRET DynAlloy completed without finding any violations. Because we initialized the model to the most general state, we have proven that AMD's SYSRET will never exhibit the vulnerability, regardless of the state of the machine or sequence of instructions executed prior to calling SYSRET.

These results support our claim that Intel's ISA exhibits a bug and the reported SYSRET vulnerability is not just the result of buggy operating systems. In the AMD model, no matter what the starting state for the instruction, ϕ is never violated. This means that software may execute SYSRET with a user-controlled stack pointer, a high current privilege level, and a non-canonical address in %RCX and still not exhibit Intel's SYSRET vulnerability.

E. Discussion

Initializing the model to a largely unconstrained state means that if DynAlloy returns without finding any violations of ϕ, we may conclude that indeed ϕ will always hold for the modeled instruction(s). On the other hand, the inverse is not necessarily true. If the model checker does find a violation of ϕ, it may or may not be in a reachable state. This is because the model may include potentially unreachable states in its initial starting state. In our example, we know from prior work (e.g., [19]) that the found violation is indeed a true vulnerability. As we move to modeling instructions for which we don't have a priori knowledge of an exploitable bug, demonstrating a feasible exploit for possible property violations will be a necessary step.

These results suggest that validating the ISA is important, as subtle changes can make a big difference for the security of software running on the CPU. As a next step we plan to extend the technique by defining additional properties that a secure ISA should provide and modeling additional instructions to check for violations. We will focus on instructions that change the current privilege level as well as instructions that save or restore processor state. In doing so our models may become more complex and we will explore moving to models involving valuations to bitvectors and model checking tools to handle the complexity. Finally, we will evaluate the risk of false positives with this method.

IV. Related Work

Existing work in this area has mostly concentrated on manually building an executable ISA for which properties

can be proven or disproven [18], [10], [12], [11]. Godefroid and Taly [13] take a different approach. They derive an executable ISA by looking at the input–output relations for each instruction, using a physical processor as an oracle. These executable ISA models are motivated by a desire to verify software correctness down to the hardware level and have focused mostly on user-level instructions (i.e., not instructions that change privilege levels).

Work in the area of testing CPU emulators and hypervisors has shown that ambiguous or incomplete ISA specifications can lead to two different implementations of the same ISA choosing to update security critical state differently [17], [14], [16], [15]. That work is concerned with identifying differences in implementations, whereas we are interested in discovering whether those differences introduce vulnerabilities.

V. Conclusion

In this paper we have described a class of vulnerabilities that are difficult to detect through testing or verification alone – those vulnerabilities that affect the specification. We have presented an approach to detecting such vulnerabilities in an ISA, and have shown that the model-checking based method can be both feasible and effective. We demonstrated our approach with a case study of the recent Intel SYSRET vulnerability and have shown that the vulnerability does not exist in the AMD64 version of the ISA.

VI. Acknowledgments

We thank the anonymous reviewers for their hard work and expertise; this paper was improved by their insightful comments. This work was supported in part by the National Science Foundation under grant CNS-1464209. Any opinions, findings, conclusions, and recommendations expressed in this paper are solely those of the authors.

References

[1] AMD64 Architecture Programmers Manual Volume 3: General Purpose and System Instructions. *AMD*.

[2] DynAlloy: An efficient extension of Alloy with procedural actions. http://www.dc.uba.ar/inv/grupos/rfm_folder/dynalloy.

[3] Intel 64 and IA-32 Architectures Software Developer Manuals. *Intel*.

[4] Opensparc t2 source code. *Sun*.

[5] Xen Security Advisory 7 (CVE-2012-0217) - PV privilege escalation. *Xen-announce*.

[6] SYSRET 64-bit operating system privilege escalation vulnerability on Intel CPU hardware. *Vulnerability Notes Database*, 2012.

[7] Revision Guide for AMD Family 16h Models 00h-0Fh Processors. *Product Revision*, 2013.

[8] Intel Core i7-600, i5-500, i5-400 and i3-300 Mobile Processor Series. *Specification Update*, 2014.

[9] M. Bonetti. Advanced exploitation of Windows kernel Intel 64-bit mode SYSRET vulnerability (ms12-042). http://www.vupen.com/blog/20120806.Advanced_Exploitation_of_Windows_Kernel_x64_Sysret_EoP_MS12-042CVE-2012-0217.php, 2012. VUPEN Vulnerability Research Team (VRT).

[10] Anthony Fox. Formal Verification of the ARM6 Micro-Architecture. Technical Report UCAM-CL-TR-548, University of Cambridge, Computer Laboratory, November 2002.

[11] Anthony Fox. Formal specification and verification of ARM6. In *Theorem Proving in Higher Order Logics, LNCS*. Springer-Verlag, 2003.

[12] Anthony Fox. Directions in ISA Specification. In *Interactive Theorem Proving*, 2012.

[13] Patrice Godefroid and Ankur Taly. Automated synthesis of symbolic instruction encodings from I/O samples. In *Programming Language Design and Implementation*, volume 47, pages 441–452. ACM, 2012.

[14] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 337–348, New York, NY, USA, 2012. ACM.

[15] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing System Virtual Machines. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 171–182, New York, NY, USA, 2010. ACM.

[16] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing CPU Emulators. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 261–272, New York, NY, USA, 2009. ACM.

[17] Hao Shi, Abdulla Alwabel, and Jelena Mirkovic. Cardinal Pill Testing of System Virtual Machines. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 271–285, San Diego, CA, August 2014. USENIX Association.

[18] Jr. Warrent A. Hunt and Matt Kaufmann. Towards a Formal Model of the x86 ISA. Technical report, University of Texas at Austin, 2012.

[19] Rafal Wojtczuk. A Stitch In Time Saves Nine: A Case Of Multiple OS Vulnerability. *Black Hat*, 2012.