Evaluating Security Specification Mining for a CISC Architecture

Calvin Deutschbein and Cynthia Sturton
University of North Carolina at Chapel Hill
201 S. Columbia St.
Chapel Hill, 27599
{cd, csturton}@cs.unc.edu

***Abstract*—Security specification mining is a relatively new line of research that aims to develop a set of security properties for use during the design validation phase of the hardware life-cycle. Prior work in this field has targeted open-source RISC architectures and relies on access to the register transfer level design, developers' repositories, bugtracker databases, and email archives. We develop Astarte, a tool for security specification mining of closed-source, CISC architectures. As with prior work, we target properties written at the instruction set architecture (ISA) level. We use a full-system fast emulator with a lightweight extension to generate trace data, and we partition the space of security properties on security-critical signals in the architecture to manage complexity. We evaluate the approach for the x86-64 ISA. The Astarte framework produces roughly 1300 properties. Our automated approach produces a categorization that aligns with prior manual efforts. We study two known security flaws in shipped x86/x86-64 processor implementations and show that our set of properties could have revealed the flaws. Our analysis provides insight into those properties that are guaranteed by the ISA, those that are required of the operating system, and those that have become de facto properties by virtue of many operating systems assuming the behavior.**

I. Introduction

Validating the security of a processor starts at the specification and design phases. The current industry practice for security validation is a mostly manual approach. Designers and testers study the specification and design and reason about the necessary and desired security properties of the processor.

Recently, researchers have developed tools to semi-automatically generate properties that capture the security goals of the design [1], [2]. These properties are expressed in SystemVerilog using the industry-standard Open Verification Library (OVL) format [3], making them suitable for use with existing simulation-based verification [4] and formal static analysis [5], [6] methods. An automated approach to security property specification is a first step toward a systematic, comprehensive security validation process.

However, the current security property specification tools were developed for, and are applicable to, only open-source RISC processors. In this paper we develop security specification mining for x86 processor designs. In turning our attention to the x86 instruction set architecture (ISA) we face three challenges. First, the size of the ISA makes even a semi-manual approach prohibitive; Second, x86 is closed-source and prior approaches for mining security properties relied on access to both the source code and the developers' repositories, bugtracker databases, and email forums; Third, compared to today's RISC architectures, x86 offers a richer landscape of security features and privilege modes, increasing the number and complexity of the associated security properties.

We present Astarte, a fully automatic security specification miner for x86. A challenge with mining security critical properties is automatically identifying those properties that are relevant for security, that if violated would leave the processor vulnerable to attack. In general, there is no fixed line separating functional properties from security properties. The environment in which a processor operates and the attacker's motivation and capabilities may move some properties across the security-critical boundary in either direction.

In theory, a design would be validated as correct against a complete specification with well defined and proven security policies. In practice, however, the specification is neither complete, well defined, nor proven secure. It is up to the security validation team to determine whether, and to what degree, a given design may be vulnerable to attack.

Prior work tackled this problem by analyzing existing design bugs and manually sorting them as exploitable

or not exploitable [1]. However, this approach is labor intensive and does not easily scale to x86. Furthermore, perhaps more relevant for our purposes, this approach requires knowledge of and access to the details of known design bugs culled from developers' archives, code repositories, and bugtacker databases, which we do not have for the closed-source x86 designs.

We therefore take a different approach. We focus our attention on mining properties that are relevant to the various control signals that govern security-critical behavior of the processor. These properties are by definition important for the correct and secure behavior of the processor, which in turn is important for the correct implementation of the security primitives that operating systems and software rely one. In this respect, our approach is inspired by prior, manual efforts [7], [8], [9].

We tackle the complexity of the architecture by independently considering the space of properties for each instruction preconditioned on the value of a single security-relevant control signal. In other words, we partition the specification with respect to each control signal. It is perhaps counter-intuitive that this approach works; it would seem necessary to consider all possible combinations of all security-relevant signals for every instruction in order to produce meaningful security properties. Yet, when we compare our found properties to prior manual efforts and to known bugs in shipped x86 products, we find that considering the security-relevant control signals independently produces valuable properties.

We implement Astarte on top of Daikon [10], a popular invariant miner, and we use the QEMU emulator [11] to produce traces of execution. Astarte produces roughly 1300 properties. We evaluate these properties against manually discovered security properties and two historical exploitable bugs in x86. Of the 29 previously identified security properties, Astarte generates 23, and the remaining 6 require processor state unimplemented in QEMU. Astarte generates the properties that could have detected the two exploitable bugs as well.

By generating trace data during the boot of four different operating systems we are able to differentiate between properties that are likely enforced by the processor from those that must be enforced by the operating system. Our analysis also provides insight into properties that are not specified, but that operating systems have come to rely on.

Our main contribution is a security specification miner for closed-source, x86 architectures and its evaluation for the Intel x86 (Ivy Bridge) processor. The novelty in our approach includes:

- a partitioning of the security specification with respect to each security-relevant control signal;
- automatically identifying the control signals of interest;
- differentiating between processor-level properties and operating system-level properties; and
- identifying de facto security-critical properties that operating systems have come to rely on.

II. Properties

The Astarte framework generates properties written over ISA-level state. Properties describe the constraints and behavior of ISA state for a given instruction. For example, the property in Figure 1a states that when the `in` instruction executes, the I/O privilege level (IOPL, as given by bit 13 of the EFLAGS register) must be greater than or equal to the current privilege level (CPL, as given by bit 13 of the Code Segment register).

A property may refer to the state of a signal or register both before and after the instruction executes. For example, the property in Figure 1b refers to the state of the IOPL flag before and after execution of the `addl` instruction. It states that if the I/O privilege level remains unchanged during the `addl` instruction, then the current privilege level must be 3.

$$\texttt{in} \rightarrow \texttt{EFL}[13] \geq \texttt{CS}[13]$$

(a) After the `in` instruction executes IOPL must be greater than CPL

$$\texttt{addl} \wedge \texttt{EFL}[13] = orig(\texttt{EFL}[13]) \rightarrow \texttt{CS}[13] = 3$$

(b) If the `addl` instruction does not modify IOPL then CPL must be 3

Fig. 1: Example properties

III. Astarte Design

A. Overview

Astarte works in three phases: trace generation, property mining, and post-processing. Figure 2 provides an overview of the Astarte workflow. In the first phase we generate traces of execution of the processor. Without access to the source code of the processor design, we can not use a simulator to generate traces of processor execution as prior work has done. Instead we use QEMU, an x86 emulator, to emulate processor execution. QEMU translates blocks of code at a time, and as such produces traces of basic blocks. The miner requires traces of

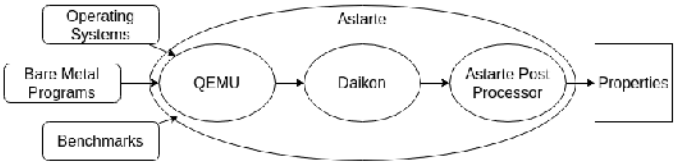

Fig. 2: An overview Astarte, which uses a modified version of the Daikon specification miner.

individual instructions, so Astarte extends the generated traces so that each event in the trace represents a single instruction. This extension is sound with respect to the generated properties. In keeping with prior art, Astarte tracks processor state that is visible to software; the final security properties are written over this software-visible state. We emulate the processor loading and running four different operating systems as well as running software on the bare (emulated) metal.

In the second phase Astarte mines the traces of execution looking for security properties. We build the miner on top of the Daikon invariant generation tool [10]. The closed-source nature of x86 processors precludes our using known, exploitable design bugs to differentiate security-critical properties from functional properties as was done by Zhang et al. [1] when targeting RISC processors. Furthermore, the complexity of the many x86 protection modes and their associated control flags overwhelms the miner. In an initial experiment we let the naive miner run for 7 days, and the result was hundreds of millions of invariants generated, with only a fraction of those invariants representing useful security properties. We address this as follows: Astarte partitions the state space of the processor on each of a small handful of security-critical control signals, and generates invariants within each partition. The approach may seem counter-intuitive: Astarte is treating the control signals as independent of each other, when in fact different combinations of control signals may represent different protection modes within the processor. Yet, we find that it is effective; the result is a manageable set of meaningful security invariants covering the various protection modes.

In the third phase, post-processing, Astarte combines like invariants, integrates results across multiple runs of the miner (e.g., using traces generated from different operating systems), and simplifies expressions. The result is a manageable set of security properties.

B. Trace Generation

To generate traces of execution we use QEMU, a full system, open source machine emulator [12]. Running a processor in emulation allows us visibility into the processor's state. On a QEMU-emulated x86 CPU we can boot an operating system, run user-level applications, and extract log data about the state of the CPU as software executes. The log data forms the basis of the execution traces over which security properties are mined.

QEMU dynamically translates machine instructions from the target architecture (in our case, x86) to the host architecture. To ease portability QEMU translates first to an intermediate language and then to the host instruction set. To improve performance QEMU translates a block of machine instructions at a time, rather than translating line by line.

A QEMU translation block (TB) is akin to a basic block [13]. It is a sequence of instructions with a single entry point—the first instruction in the TB—and a single exit point—the last instruction in the TB. A TB ends at any instruction that modifies the program counter, such as syscall, sysenter, or jmp, or at a page boundary.

The translated TBs can be cached and reused, reducing translation time. However, translating a block of code at a time obscures CPU state at instruction boundaries. In other words, QEMU maintains consistent target CPU state at TB boundaries rather than at instruction boundaries.

From the QEMU execution logs, we can pull *events* corresponding to the execution of a single TB. An event shows the sequence of instructions that make up the TB and the CPU state after the TB executes:

$$\langle \text{instruct}_1, \text{instruct}_2, \ldots, \text{instruct}_n \rangle (r_0, r_1, \ldots, r_m)$$

In the above, $(r_0, \ldots, r_m)$ represents the state of the m ISA-level registers after the TB executes. A *trace* of events gives us the CPU state at every TB boundary. The first event in a QEMU trace is always the single instruction `ljmpw`, which jumps to the code entry point. A trace of TBs might look like this:

$$\begin{aligned} &\langle \texttt{ljmpw} \rangle (r_0^0, r_1^0, \ldots, r_m^0) \\ &\langle \text{instruct}_1, \text{instruct}_2, \ldots, \text{instruct}_n \rangle (r_0^n, r_1^n, \ldots, r_m^n) \\ &\qquad \ldots \\ &\langle \text{instruct}_1, \text{instruct}_2, \ldots, \text{instruct}_j \rangle (r_0^*, r_1^*, \ldots, r_m^*) \end{aligned}$$

The CPU state logged at the end of the first TB gives us the CPU state before the second TB executes. However, for our purposes, we need the CPU state at every instruction boundary, not just every TB boundary. Given the single logged event

$\langle \text{instruct}_1, \text{instruct}_2, \ldots, \text{instruct}_n \rangle (r_0^n, r_1^n, \ldots, r_m^n)$,
we require an extended trace of events:

$$\langle \text{instruct}_1 \rangle (r_0^1, r_1^1, \ldots, r_m^1) \quad (1)$$
$$\langle \text{instruct}_2 \rangle (r_0^2, r_1^2, \ldots, r_m^2) \quad (2)$$
$$\ldots$$
$$\langle \text{instruct}_\text{n} \rangle (r_0^n, r_1^n, \ldots, r_m^n) \quad (3)$$

Producing the extended trace of events would require an emulator that translates code line-by-line. But, the emulator still needs to be fast enough to boot operating systems and run application-level code. Therefore, we take a middle approach: we use QEMU as our emulator and build a lightweight extension to generate partial per-instruction events. For every TB in a trace, the event generator creates a new sequence of events, one event for each instruction in the TB. Each event lists the instruction executed and partial information about the CPU state. Any software-visible register that can be modified by the instruction is marked as invalid, and all other registers retain their value from the previous event. The generated event corresponding to the last instruction in the TB has the full CPU state as given by the original QEMU event. Continuing with the above extended trace of events, and considering the second event at line (2) in the trace, $\forall i, 0 \leq i \leq m$ either $r_i^2 = r_i^1$ or $r_i^2 = \texttt{invalid}$.

The event generator errs on the side of soundness: if it is possible for an instruction to change an aspect of CPU state, the generator assumes it does. We used the Intel Software Developer Manuals [14] as our reference when building the event generator.

C. Property Mining

We use the Daikon invariant detector [10] as the base for the Astarte property mining. We build a custom front-end that reads in the extended traces of events produced in the first phase, and outputs a trace of observations suitable for Daikon.

Daikon was developed for use with software programs: it looks for invariants over state variables for each point in a program. Our front-end treats x86 instructions as program points; Daikon therefore will find invariants over ISA variables for each x86 instruction.

Daikon can handle individual program modules with relatively few program points and few program variables, it is not intended for analysis of entire programs [15]. The amount of ISA state and the number of instructions in x86 is too large for Daikon to handle. The amount of trace data required to achieve coverage of a single instruction, and the size of the state over which to find invariant patterns for a single instruction overwhelm Daikon.

`addl`	$\wedge$	$orig(\texttt{IOPL}) = 0$	$\wedge$	$\texttt{IOPL} = 0$
`addl`	$\wedge$	$orig(\texttt{IOPL}) = 0$	$\wedge$	$\texttt{IOPL} = 1$
`addl`	$\wedge$	$orig(\texttt{IOPL}) = 1$	$\wedge$	$\texttt{IOPL} = 0$
`addl`	$\wedge$	$orig(\texttt{IOPL}) = 1$	$\wedge$	$\texttt{IOPL} = 1$

TABLE I: Four partitions of `IOPL` for instruction `addl`

To mitigate the complexity, for each instruction Astarte partitions the space of properties on individual control signals.

1) Partitioning on Control Signals: For each instruction, Astarte separately considers the space of invariants over ISA state for that instruction, preconditioned on a single control bit. The key insight is that if Astarte chooses the control bits wisely, the partitioning not only mitigates performance and complexity issues with Daikon, it also produces sets of properties that are critical to security, and we can then classify the properties by their precondition. The properties that make up each class provide some insight into the modes and behaviors of the processor governed by the preconditioning control signal.

For each control signal, how Astarte partitions the space of invariants for a single instruction depends on the control signal. For a one-bit signal Astarte creates four partitions, one for each combination of signal values before and after the instruction executes. For example, with the `IOPL` flag and `addl` instruction, Table I shows the four partitions of the space of invariants. Each row of the table represents one of the four possible antecedents of a property. The four antecedents represented in the table completely partition the space. For signals longer than one bit Astarte divides the space of invariants into two partitions for each instruction: $\texttt{instruct} \wedge orig(\texttt{reg}) = \texttt{reg}$ and $\texttt{instruct} \wedge orig(\texttt{reg}) \neq \texttt{reg}$.

The set of properties produced for a particular preconditioning signal tell us something about the behavior governed by that signal. For example, providing $\texttt{CPL} \neq \text{orig}(\texttt{CPL})$ as a precondition will mine properties related to how the current privilege level (CPL) of the processor is elevated and lowered.

2) Identifying Control Signals: The first step is to choose which control signals to use as preconditions. We manually organize the x86 ISA state by category and then let Astarte find the meaningful signals within a category. Here we consider the signals available as variables in QEMU as well as the x86 registers:

- General Purpose Registers: EAX, EBX, ECX, EDX
- Interrupt Pointer: EIP
- Control Registers: EFL, CR0, CR2, CR3, CR4, EFER
- Bitflags: II, A20, SMM, HLT, CPL
- Current Segments: CS, SS, DS
- Special Segments: ES, FS, GS, LDT, TR
- Descriptor Tables: GDT, IDT
- Debug Registers: DR0, DR1, DR2, DR3, DR6, DR7
- Command Control: CCS, CCD, CC0

Of these, we select three categories to focus on: Control Registers, Bitflags, and Current Segments. These registers contain fields that control security critical state, such as privilege levels and location of page tables. We chose these categories based on our knowledge of the x86 ISA. Initially, we had only the Control Registers and QEMU Bitflags, but our initial evaluation led us to add the Current Segments. We expect other categories may also yield interesting properties. Because each control signal is analyzed independently of the others, additional categories of ISA state can be analyzed without incurring a combinational explosion in performance cost. (In Section V-E we discuss the cost.)

During the signal-finding phase Astarte unpacks registers to consider one- and two-bit fields separately. It then looks for and discards any unused fields. It does this by looking for fields that keep a constant value. Astarte collapses all x86 instructions into a single pseudo-instruction and runs the property miner on this modified trace. Any found properties of the form $\texttt{reg} = N$ are an indication that for all instructions `reg` has the constant value N and is therefore unused. Astarte discards these flags from further consideration. At the end of this phase we are left with 24 signals of interest.

D. Postprocessing

The Daikon miner produces tens of millions of properties. In post-processing Astarte removes invalid properties, removes redundant properties, and combines similar properties into a format that is easier to read.

1) Intersection Across Trace Sets: Astarte runs the Daikon miner separately for each set of traces representing separate operating system boots and bare-metal execution. In the first step of post-processing properties from different traces are combined by taking the intersection of all sets with shared elements within a precondition. This ensures that no property that is invalidated by any one trace persists in the property set. It also generalizes properties to the implementation being studied, rather than to just a single trace.

2) Transitive Closure: Frequently, especially in the case of single bit values, many registers will take on the same value and Daikon will return many such equality properties. To make these properties more manageable, Astarte takes the transitive closure of all the equality properties and, instead of lists of pairwise equalities, equality properties are presented as sets of registers that are equal. For example, given the three invariants $\texttt{andb} \rightarrow \mathit{orig}(\texttt{CPL}) = 3$, $\texttt{andb} \rightarrow \texttt{CPL} = \texttt{DS\_DPL}$, and $\texttt{andb} \rightarrow \texttt{CPL} = \mathit{orig}(\texttt{CPL})$, the post-processor would return as a single property $\text{andb} \rightarrow \{\mathit{orig}(\texttt{CPL}), 3, \texttt{CPL}, \texttt{DS\_DPL}\}_=$, where the notation $\{\}_=$ indicates that any two signals in the set are equal ($\forall r, s \in \{\}_=, r = s$).

$$\text{instruct} \wedge \text{precondition} \rightarrow \{\langle \text{var} \rangle, \langle \text{var} \rangle, \ldots\}_=$$

In the next stage, properties that share a common instruction and invariant precondition are combined to form larger properties that more completely express processor behavior with regard to a control signal. These properties are similar to the previous properties with the sole exception of having multiple sets of equal values, registers, or bits.

$$\begin{aligned}&\text{instruct} \wedge \text{precondition} \rightarrow \\ &\quad \{\langle \text{var} \rangle, \langle \text{var} \rangle, \ldots\}_= \\ &\quad \{\langle \text{var} \rangle, \langle \text{var} \rangle, \ldots\}_= \\ &\quad \{\langle \text{var} \rangle, \langle \text{var} \rangle, \ldots\}_= \\ &\quad \ldots\end{aligned}$$

3) OS-Specific Values: In some cases, general purpose registers take on a particular value or set of values for an operating system. These values may differ across operating systems, but there is an underlying pattern that is upheld across operating systems and that is critical to security. For example, values must be word aligned or in a canonical form. To identify these properties the post-processor applies a bit mask to equalities between values and general purpose registers to find which bits change and which do not.

4) Identify Global Properties: As a final step Astarte ensures that all properties are specific to a control signal by comparing against global properties. Recall that Astarte identifies control signals of interest in the first phase. Eleven of the 24 identified signals were found to produce properties specific to those bits. The remaining

13 signals all preconditioned the same global properties. During postprocessing Astarte removes any of these global properties from the sets of properties produced for each of the 11 control signals. These properties are necessarily not specific to a control signal since they have been found to hold globally.

IV. Implementation

The Astarte framework is written in Python. Extensions to the QEMU trace generation, the front-end for Daikon, and the post processor are written in Python. We use QEMU emulating Ivy Bridge 2013 Intel Xeon E3 1200 v2 processor to generate traces of execution. Traces were generated by running disk images within QEMU with debug options to log instructions and processor state (-d in_asm,cpu) with output logged to a file and parsed into traces of execution that could be passed to a miner.

V. Evaluation

We evaluate the Astarte framework on its ability to find security properties of the x86 architecture.

We aim to answer the following research questions:

1) Can Astarte efficiently generate high-quality assertions to prevent known CPU security bugs?
2) How effective are the control signals we use to partition the space of properties in achieving effective security properties?
3) Does Astarte produce a manageable number of properties?

The experiments are performed on a machine with an Intel Core i5-6600k (3.5GHz) processor with 8 GB of RAM.

A. Trace Data

To avoid capturing only properties enforced by, or relevant to, a specific operating system we generate trace data while booting multiple operating systems. We boot two Linux distributions (Ubuntu and Debian), Solaris, seL4, and FreeDOS ODIN.

To achieve high instruction coverage we use Fast PokeEMU [16], a tool for testing consistency between hardware and the QEMU emulator. Fast PokeEMU repeatedly executes an instruction with varying inputs to achieve high path coverage within an instruction with high probability without relying on manual test generation. We execute these instructions on the "bare metal" QEMU emulator.

Over all traces, Astarte modeled 333 distinct instructions while the Intel specification describes 611. Reviewing the specification we find that of the 278 instructions not modeled by Astarte over the trace set, 275 fall into one of five categories: AES and SHA acceleration, mask register operations, packed value operations, and vector operations (see Table II).

Mnemonic	Description	Number
aes	AES acceleration	6
k	mask register operations	13
p	packed value operations	87
sha	SHA acceleration	7
v	vector operations	162

TABLE II: Unmodelled instructions

We analyzed 10.2GB of trace data comprising 4.1 million instruction executions. We expect this trace volume to be sufficient: Amit et al. [17] found that in fewer than 1k iterations of tests of 4096 instructions—a trace volume similar to our own—most known complex race conditions could be found.

B. Control Signals

Of the 24 control signals identified prior to mining (Section III-C2), 11 govern a class of properties preconditioned on that signal. The remaining 13, when used as a precondition, produced only properties common to all preconditions; in other words, they do not govern a particular set of behaviors. Table III shows the 11 control signals along with their common name and a brief description.

C. Effect of Postprocessing

At the end of the property mining phase (Sec. III-C), Daikon produces 13,722,294 properties across all instructions and preconditions. After taking the intersection of properties across distinct trace sets and taking the transitive closure of properties, we are left with 122,122 properties. Identifying the global properties reduces the total to 1,393 properties, a reduction of close to five orders of magnitude from the naive property total. These properties average 6 implied clauses each per precondition. Each class of properties, defined by a single preconditioning control signal, has 127 properties on average. The distribution of the number of properties and average property size by control signal is shown in Table IV.

D. Assessing the Properties with Respect to Security

To evaluate the efficacy of the Astarte framework in producing properties relevant for security we consider two case studies, the 2015 SMM bug from Domas [18] and the SYSRET bug described by Xen [19]. Both

Bit/Reg	Flag	Name	Description
CPL	CPL	Current privelege level	CPL (or CS_DPL) gives the current ring from 0 to 3 while in protected mode
SMM	SMM	System Management Mode	If set, processor is in SMM (ring -2)
EFL[6]	ZF	Zero Flag	Indicates zero result of arithmetic
EFL[9]	IF	Interrupt enable flag	Determines whether to handle maskable hardware interrupts
EFL[11]	OF	Overflow Flag	Indicates overflow result of arithmetic
CR0[0]	PE	Protected Mode Enable	If set, processor is in protected mode
CR0[1]	MP	Monitor co-processor	Controls interaction of WAIT/FWAIT instructions with TS flag in CR0
EFL[4]	AF	Adjust Flag	Indicates arithmetic carry or borrow over four least significant bits
CS	CS	Code Segment	The currently used program code segment (changes only, not values)
SS	SS	Stack Segment	The currently used program stack segment (changes only, not values)
DS	DS	Data Segment	The currently used program data segment (changes only, not values)

TABLE III: Control signals that govern a class of properties preconditioned on that signal

Bit/Reg	Flag	Clauses	Properties	Clauses per Property
CPL	CPL	235	59	4.0
SMM	SMM	335	60	5.6
EFL[6]	ZF	1182	286	4.1
EFL[9]	IF	1102	164	6.7
EFL[11]	OF	390	46	8.5
CR0[0]	PE	1159	173	6.7
CR0[1]	MP	777	68	11.4
EFL[4]	AF	1402	244	5.7
CS	CS	465	55	8.5
SS	SS	432	52	8.3
DS	DS	480	50	9.6
Total		8571	1393	6.2
Globals		4187	246	17.0

TABLE IV: Number of implied clauses in a property by control signal

of these cases received considerable attention from the security and research communities and, thanks to their efforts to reverse engineer the bugs, we have information about the technical details of the bugs beyond the high-level information provided by Intel's errata documents. For each case study we examine whether the properties generated by Astarte could have caught these bugs. We also compare the properties generated by Astarte to the set of x86 security properties manually written by Brown [9]. Tables V and VI show the results.

1) SMM: At Black Hat 2015, Dumas [18] disclosed the Memory Sinkhole escalation vulnerability in SMM. The vulnerability allows an OS-level attacker to enter System Management Mode and execute arbitrary code. The attack relies on using the `call` instruction with a particular parameter while in SMM. The security properties discovered by Astarte would disallow this exploit. The properties prohibit the execution of the `call` instruction while in SMM.

No.	Property	Found	Ctrl Signal	Astarte Property
1	CALL → SMM=0	✓	SMM	G5
2	SYSRET → canonical(ECX)	✓	CPL	5, 7

TABLE V: Finding case study properties with Astarte

2) SYSRET: This vulnerability, as described by Xen [19] arises from the way in which Intel processors implement error handling in their version of AMD's SYSRET instruction. If an operating system is written according to AMD's specification, but run on Intel hardware, an attacker can exploit the vulnerability to write to arbitrary addresses in the operating system's memory.

The crux of the vulnerability has to do with when the Intel processor checks that, when returning to user mode, the address being loaded into the `RIP` register from the `RCX` register is in canonical form. Astarte generates properties that require `RCX` to always be in canonical form when the current privilege level is elevated, which would prevent the vulnerability. It is interesting to note that Astarte only finds this property over traces produced by operating systems, an indication that this desired behavior is not enforced by the hardware and must be enforced by an operating system, as is indeed the case.

3) Manually Developed Properties: Brown studied the Intel specification and crafted 29 properties they found to be critical to security [9]. The Astarte properties cover 23 of the 29 manually written properties. The remaining 6 properties required exercising processor state unimplemented in QEMU.

E. Performance

Generating the trace data took approximately 8 hours, processing the traces to make them suitable for Daikon took 57 minutes, identifying control bits on which to

No.	Property	Found	Ctrl Signal	Astarte Property
1	IN/OUT/INS/OUTS → IOPL ≥ CPL	✓	CS[13]	G65, G68-71, G104-107, G243-244
2	!(JMP/CALL/RET/SYS*) → CS=orig(CS)	✓	CS	298, 300, 302, 304, 306, 308-309, 311-313, 315, 317-318, 320-322, 324, 326, 328, 330-332, 335-337, 339, 341-343, 345, 347-349, 351
3	POPF/IRET & !CPL=0 → EFL_13=orig(EFL_13)	✓	EFL[13]	G72, G111-G112, G206
4	STI/CLI & CPL > EFL_13 → EFL_9=orig(EFL_9)	✓	EFL[9]	G53, G141
5	IRET & EFL_9=orig(EFL_9) & CPL > EFL_13 → EFL_9=orig(EFL_9)	✓	EFL[9]	1044-1045
6	IRET & CPL≠ 0 → EFL_13=orig(EFL_13)	✓	CS[13]	G72, G206
7	SYSEXIT → CPL=0	✓	CS[13]	3, 7
8	SYS* → CPL ≤ DPL	✓	CS[13]	37, 39
9	SYS* → CS_DPL ≤ CPL	✓	CS[13]	37, 39
10	JMP/CALL(FAR) & CS≠orig(CS)→ DPL = CPL	✓	CS	15, 16, G18
11	JMP/CALL(FAR) & CS≠orig(CS)→ DPL ≤ CPL	✓	CS	15, 16, G18
12	CALL(FAR) & CS≠orig(CS)→ CPL ≤ DPL	✓	CS	15, 16
13	JMP(FAR) & CS≠orig(CS)→ DPL ≤ CPL	✓	CS	G18
14	JMP/CALL(FAR) & CS≠orig(CS)→ CS_DPL ≤ orig(CPL)	✓	CS	15, 16, G18
15	RET & CS≠orig(CS) → CS_DPL≥ CPL	✓	CS	G35, G72, G90, G120, G206, G224
16	SS≠orig(SS) → SS_DPL=CPL	✓	SS	G245
17	DS≠orig(DS) → DS_DPL≥CPL	✓	DS	G245
18	CS[11]=1 CS[12]=0	✓	CS	G245
19	SS[9]=1 & SS[12]=0 & SS[11]=0	✓	SS	G245
20	DS[9]=DS[11]=DS[12]=1	✓	DS	G245
21	IRET & EFL_13 → CS_DPL ≤ SS_DPL	✓	CS[13]	G35, G72, G90, G120, G206, G224
22	SYSENTER & CR0_0=1→CS=val,EIP=val,SS=val,SP=val	✓	CS[13]	37, 39
23	SYSEXIT & CR0_0=1→CS=val,EIP=old_EIP,SS=val,SP=val	✓	CS[13]	37, 39
24-25	Properties over unimplemented MSRs	-	unknown	
26-29	Properites requiring unimplemented VMX instructions	-	unknown	

TABLE VI: Finding manually crafted properties with Astarte

partition the property space took 44 minutes. Mining along all preconditions took approximately 16 hours with each control bit costing roughly 44 minutes. Overall, the Astarte framework completed the property generation in 29 hours.

We completed the full mining process for 24 control signals as preconditions. Excluding unused control signals from consideration provided a speedup of of 5.82x. Comparing all control signals pairwise rather than treating them independently would have yielded 1936 preconditions over the the 24 bits used, or 262144 comparisons over all possible bits, giving speedups of 44.00x and 5957.82x respectively. (When computing these speedups, we do assume that control bits all take roughly the same amount of time to mine. We found all mined control bits completed within tens of seconds of each other; we believe it reasonable to assume this timing trend would apply to other untested preconditions.)

F. Operating System-Enforced Properties

When mining over traces from different operating systems, we note that some properties are found over all operating systems and some over only a subset of operating systems or only on bare metal traces with no operating system. In Figure 3 we show how many operating systems are found to enforce each property. Figure 4 shows for each property enforced by 1, 2, or 3 operating systems, which operating system it is enforced by.

We found that properties were predominantly enforced either by a single operating system or by all operating systems. We interpret properties enforced by a single operating system to likely fall into two main possibilities: either the properties are well-founded properties that, when enforced, make the operating system more secure

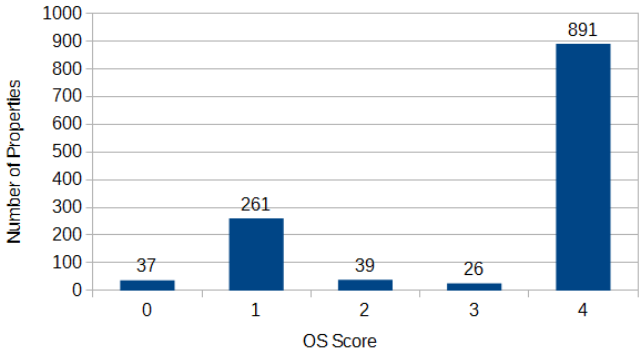

Fig. 3: Distribution of properties by how many OSes are found to enforce them shows clustering around one or every OS.

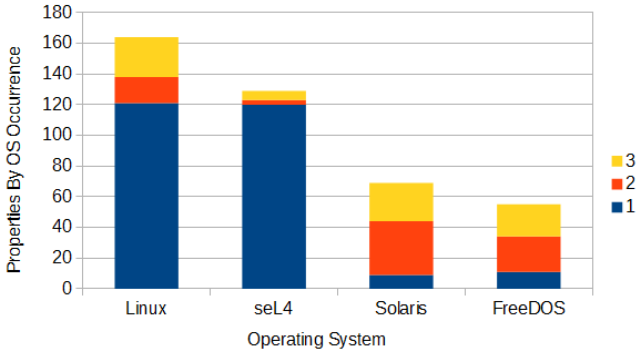

Fig. 4: Within the properties implemented by a subset of the OSes, Linux and seL4 enforce many unique properties and Solaris and FreeDOS enforce properties with each other or with Linux.

in some way, or that they are false positives and found only within a single operating system for this reason. In manual inspections of properties, we found that many of the properties unique to Linux and seL4 were related to ensuring the safety of the specific implementations of system calls used by the operating system. Unsurprisingly more properties were enforced on Linux and seL4 which have the highest usage levels and most rigorous theoretical assurances respectively. The remainder of unique properties governed specific instruction usage from specific processor states only exercised by that operating system that may or may not be associated with security.

We interpret properties enforced by all operating systems to be necessary implementation features as changing any one of them would would likely cause compatibility issues across many operating systems. We extend this understanding to properties implemented by all but one operating system, especially as the operating system most frequently missing was seL4. As seL4 by design has provably correct behavior it cannot rely on undocumented or incidental features. Without the burden of provable correctness and security enforcement, other operating systems may make reasonable assumptions of processor behavior. These assumptions may eventually become part of the processor specification if many operating systems come to rely on them, making it difficult for hardware designers to modify the expected, though undocumented, behavior.

The few properties enforced by just two operating systems usually govern behavior of a very specific type of system call that is enforced by precisely two operating systems. A few properties govern specific instruction usage enforced by precisely two operating systems. Similarly, these may be best practice or false positive properties.

There were also a few properties found to be enforced on bare metal traces but not operating system traces. We regard these as either false positives or these are vestigial properties that persist in hardware but OSes no longer need to rely upon.

G. Properties in the Specification

To provide a sense for how difficult the properties generated by Astarte would be to find manually, we developed a scoring function for properties that considered each bit or register within a property against how many times that bit or register is referenced in the Intel Software Developers Manuals [14] to give a sense of how many pieces of discrete information must be considered to generated a property. Figure 6 shows the cumulative distribution function of this specification score for properties.

Properties typcially would require reviewing approximately 7000 mentions (median 6874, mean 7088) with a minimum of 413, a maximum of 19669, and about 8.9 million in total. The distribution is nearly uniform with slight clustering at the minimum and slightly longer tails on the maximum. Figure 5 shows how many discrete mentions of each bit or register occur in the ISA specification.

VI. Related Work

Generating Security Critical Properties for Hardware Designs: The first security properties developed for hardware designs were manually crafted [8], [7], [9]. SCIFinder semi-automatically generates security critical properties for a RISC processor design [1]. Recent

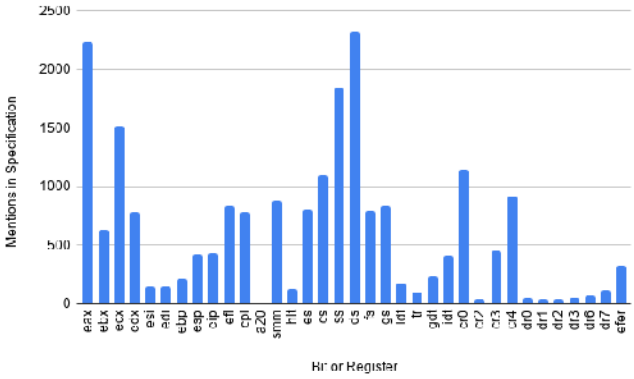

Fig. 5: Different bits or registers may be mentioned from 1s to 1000s of times in the specification.

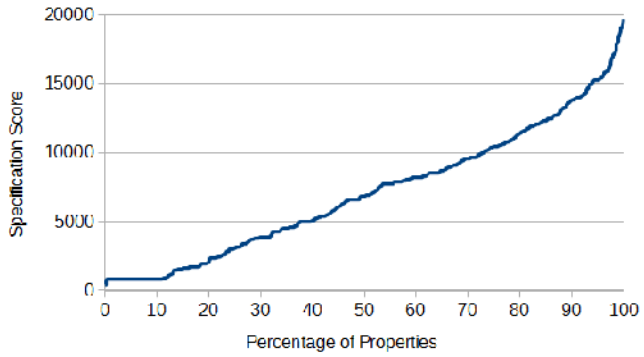

Fig. 6: Different properties may requiring considering from 100s to 10000s of mentions in the specification.

hackathons have revealed the types of properties needed to find exploitable bugs in the design of a RISC-based system-on-chip [20]. Our tool builds on this prior work to generate properties for closed-source CISC (x86) processors.

Extracting Functional Properties from Hardware Designs: General properties that do not focus on security have been found by looking for instances of known patterns, such as one-hot encoded signals [21], [22]. An alternative approach is to use data mining to infer properties from traces of execution of the design [23], [24], [25]. More recent work has focused on mining temporal properties from execution traces [26], [27], [28], [29]. A combination of static and dynamic analysis can extract properties expressed over words [30].

Mining Specifications for Software: The seminal work in specification mining comes from the software domain [31] in which execution traces are examined to infer temporal specifications in the form of regular expressions. Subsequent work used both static and dynamic traces to filter out less useful candidate specifications [32]. More recent work has tackled the challenges posed by having imperfect execution traces [33], and by the complexity of the search space [34], [35], [36]. Daikon, which produces invariants rather than temporal properties, learns properties that express desired semantics of a program [10].

Generating Security Properties of Software: In contrast to the hardware domain, in the software domain a number of papers have developed security specific specification mining tools. These tools use human specified rules [37], observe instances of deviant behavior [38], [39], [10], or identify instances of known bugs [40].

VII. Conclusion

In this paper we proposed Astarte, a framework for mining security critical properties from a closed source, CISC architecture. Astarte produces manageable numbers of properties that capture the behavior of the processor under security-relevant control signals. Astarte addresses issues of complexity and closed-source designs not seen in RISC-based security specification miners. We evaluated the efficacy of the properties generated and the performance of the framework.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1816637 and by the Semiconductor Research Corporation (SRC). Any opinions, findings, conclusions, and recommendations expressed in this paper are solely those of the authors.

References

[1] R. Zhang, N. Stanley, C. Griggs, A. Chi, and C. Sturton, "Identifying security critical properties for the dynamic verification of a processor," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2017, pp. 541–554. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037734

[2] C. Deutschbein and C. Sturton, "Mining security critical linear temporal logic specifications for processors," in *Proceedings of the International Workshop on Microprocessor and SoC Test, Security, and Verification (MTV)*. IEEE, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8746060

[3] H. Foster, K. Larsen, and M. Turpin, "Introduction to the New Accellera Open Verification Library," 2006.

[4] L.-T. Wang, Y.-W. Chang, and K.-T. Cheng, *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann, 2009.

[5] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verfication tool," in *Comuter Aided Verification (CAV)*. Lecture Notes in Computer Science, 2010.

[6] D. Brand, "Verification of large synthesized designs," in *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. IEEE/ACM, 1993.

[7] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, "Security checkers: Detecting processor malicious inclusions at runtime," in *International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, June 2011, pp. 34–39. [Online]. Available: https://calhoun.nps.edu/handle/10945/35004

[8] 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 (ASPLOS)*. ACM, 2015, pp. 517–529. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694366

[9] M. Brown, "Cross-validation processor specifications," University of North Carolina at Chapel Hill, Master's Thesis, 2017.

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35–45, Dec. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2007.01.015

[11] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX Association, 2005. [Online]. Available: https://dl.acm.org/citation.cfm?id=1247401

[12] QEMU. [Online]. Available: https://www.qemu.org/

[13] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

[14] "Intel 64 and IA-32 architectures software developer manuals," *Intel*. [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[15] Daikon. [Online]. Available: https://plse.cs.washington.edu/daikon/

[16] Q. Yan and S. McCamant, "Fast PokeEMU: Scaling generated instruction tests using aggregation and state chaining," in *Proceedings of the 14th International Conference on Virtual Execution Environments (VEE)*. ACM, 2018, pp. 71–83. [Online]. Available: http://doi.acm.org/10.1145/3186411.3186417

[17] N. Amit, D. Tsafrir, A. Schuster, A. Ayoub, and E. Shlomo, "Virtual CPU validation," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. ACM, 2015, pp. 311–327. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815420

[18] C. Domas, "The memory sinkhole: An architectural privelege escalation vulnerability," *Black Hat USA*, 2015. [Online]. Available: https://www.blackhat.com/docs/us-15/materials/us-15-Domas-The-Memory-Sinkhole-Unleashing-An-x86-Design-Flaw-Allowing-Universal-Privilege-Escalation.pdf

[19] "The Intel SYSRET privilege escalation," *Xen Project*. [Online]. Available: https://xenproject.org/2012/06/13/the-intel-sysret-privilege-escalation/

[20] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "Hardfails: Insights into software-exploitable hardware bugs," in *28th USENIX Security Symposium*. USENIX Association, 2019, pp. 213–230. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky

[21] S. Hangal, S. Narayanan, N. Chandra, and S. Chakravorty, "IODINE: A tool to automatically infer dynamic invariants for hardware designs," in *Proceedings of 42nd Design Automation Conference (DAC)*. IEEE, 2005.

[22] E. El Mandouh and A. G. Wassal, "Automatic generation of hardware design properties from simulation traces," in *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2012, pp. 2317–2320.

[23] P.-H. Chang and L. C. Wang, "Automatic assertion extraction via sequential data mining of simulation traces," in *Proceedings of the 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2010, pp. 607–612.

[24] S. Hertz, D. Sheridan, and S. Vasudevan, "Mining hardware assertions with guidance from static analysis," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 6, pp. 952–965, 2013.

[25] L. Liu and S. Vasudevan, "Automatic generation of system level assertions from transaction level models," *Journal of Electronic Testing*, vol. 29, no. 5, pp. 669–684, Oct. 2013. [Online]. Available: http://dx.doi.org/10.1007/s10836-013-5403-y

[26] W. Li, A. Forin, and S. A. Seshia, "Scalable specification mining for verification and diagnosis," in *Proceedings of the 47th Design Automation Conference (DAC)*. ACM, 2010, pp. 755–760. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837466

[27] A. Danese, N. D. Riva, and G. Pravadelli, "A-TEAM: Automatic template-based assertion miner," in *Proceedings of the 54th Design Automation Conference (DAC)*. ACM/EDAC/IEEE, June 2017, pp. 1–6.

[28] A. Danese, T. Ghasempouri, and G. Pravadelli, "Automatic extraction of assertions from execution traces of behavioural models," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2015, pp. 67–72.

[29] A. Danese, G. Pravadelli, and I. Zandonà, "Automatic generation of power state machines through dynamic mining of temporal assertions," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 606–611.

[30] L. Liu, C. Lin, and S. Vasudevan, "Word level feature discovery to enhance quality of assertion mining," in *International Conference on Computer-Aided Design (ICCAD)*. IEEE/ACM, Nov 2012, pp. 210–217.

[31] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *Proceedings of the 29th Symposium on Principles of Programming Languages (POPL)*. ACM, 2002, pp. 4–16. [Online]. Available: http://doi.acm.org/10.1145/503272.503275

[32] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer-Verlag, 2005, pp. 461–476. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31980-1_30

[33] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal API rules from imperfect traces," in *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. ACM, 2006, pp. 282–291. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134325

[34] M. Gabel and Z. Su, "Javert: Fully automatic mining of general temporal properties from dynamic traces," in *Proceedings of the 16th International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2008, pp. 339–349. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453150

[35] G. Reger, H. Barringer, and D. Rydeheard, "A pattern-based approach to parametric specification mining," in *28th International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 2013, pp. 658–663.

[36] M. Gabel and Z. Su, "Symbolic mining of temporal specifications," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 51–60. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368096

[37] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "AutoISES: Automatically inferring security specifications and detecting violations," in *Proceedings of the 17th USENIX Security Symposium*. USENIX Association, 2008, pp. 379–394. [Online]. Available: http://dl.acm.org/citation.cfm?id=1496711.1496737

[38] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP)*. ACM, 2009, pp. 87–102. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629585

[39] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking semantic correctness: The case of finding file system bugs," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. ACM, 2015, pp. 361–377. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815422

[40] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning," in *Proceedings of the 5th USENIX Conference on Offensive Technologies (WOOT)*. USENIX Association, 2011, pp. 13–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028052.2028065