Validation with Code Introspection of a Virtual Platform for Sandboxing and Security Analysis

Yves Lhuillier, Gilles Mouchard, and Franck Vedrine

CEA LIST, Software Reliability and Security Laboratory,
Palaiseau, France.
firstname.lastname@cea.fr

Abstract

Validating the safety and security of software computing systems often involves testing code in simulators of these systems, called virtual platforms. Because security breaches often come from implementation details, such simulators must reach a high level of accuracy. However, validating an instruction set simulator is a heavy development task involving large test campaigns. In this paper, we propose a novel technique to automatically generate and evaluate simulator tests. Using C++ polymorphism, we developed a code introspection software library that enables automatic test generation. By leveraging this automated approach, we were able to develop a self-testing simulator, providing a superior level of validation with minimal development overhead.

Keywords: virtualization, simulation, cybersecurity, validation, software validation, instruction sets, processors, code introspection, polymorphism

1 Introduction

Software safety and security engineers face more and more complex issues when dealing with validation of software computing systems. Due to constantly growing complexity of electronic systems and software components, the ability to identify safety issues, security breaches and even malicious code has been challenged considerably lately.

Analyzing code and being able to observe the final behavior of the real system is the role of *virtual platforms*. Virtual platforms not only allow to do software simulation of hardware components; they also offer the ability to instrument the simulation in order to perform a wide range of analyses. However, virtual

This work has been supported by the European Union's Horizon 2020 RIA programme under grant agreement No 780075; CHARIOT - Cognitive Heterogeneous Architecture for Industrial IoT.

platforms, like every models, face accuracy issues that become crucial when dealing with security. Safety issues are often related to functional, algorithmic and system-level issues and can be dealt with coarse-grained models. On the contrary, security issues require much more precision regarding code execution, as security breaches are often hidden in implementation details. Furthermore, most malware now try to detect sandboxes with so-called *anti-tampering methods* [12, 9], typically based on differences between emulators and real hardware.

The UNISIM-VP framework [1] was originally developed with hardware and software validation in mind. With the fast-paced growth of safety critical systems and cyber-threats, it is nowadays increasingly used for reliability and security concerns. In this context, a UNISIM-VP ARMv7 platform is currently used as front-end decoder for the static binary analyzer BINSEC [4]. The resulting ARMv7 binary code analyzer is currently being used for formal proof and certification of a safety-critical real-time OS. It also has been used for reverse compilation of inline assembly [11] to enable use of the Frama-C [7] static C code analyzer. In all these purposes and more, validating the functional accuracy and minimizing the difference between these virtual platforms and the real hardware is a critical concern for UNISIM-VP developers.

Most of UNISIM-VP CPU simulators (e.g. PowerPCs and ARMv7) have already undergone massive automated test campaigns, comparing each single instruction execution against real hardware execution with random inputs on billions of samples. These test campaigns revealed large amount of simulator bugs that were completely transparent to legitimate software, either because compilers never emit these instructions or because input data combinations are impossible in a sound execution environment. Nevertheless, a malicious software could use these differences to apply anti-tampering strategies [2].

Though the test campaigns on UNISIM-VP were successful, they suffered from one issue preventing them from being generalized: single instruction test patterns were to be written by hand. Producing new tests for either new instruction sets (e.g. ARMv8) or extended instruction sets (e.g. x86-64) was a heavy burden on developers.

This observation led us to develop techniques for automatic instruction test generation and verification based on processor simulators code introspection. Leveraging the C++ source code of these simulators, and using various techniques ranging from polymorphism to template meta-programming, a special instance of the simulator was developed to self-verify its implementation. This paper presents how this self-verification is implemented with the following organization: first, the principles of code introspection is presented in Section 2. Test selection, generation and verification are discussed in Section 3. Then the experimental setup and design choices specific to the x86-64 instruction set are discussed in Section 4. Finally, results and discussions come in Section 5.

2 Self-verifying simulator

2.1 Principles and Benefits

As mentioned in previous section, validating instruction set simulators accuracy is a critical but very costly task for developers. To speed up the process a novel automatic test generation method is proposed. From a high-level perspective, simulators perform symbolic executions of instructions to extract, for each of them, a behavioral description, as shown in the left part of Figure 1. Instruction input accesses and output results are then processed to filter relevant instructions to test. In a second step (see right part of Figure 1), each selected instrution is packaged in a unit test executed using random combinations of inputs. Executions from both the simulator and the underlying hardware are compared to check for discrepancies. This assumes of course that the self-verifying simulator runs on the same architecture than the one it models (x86-64 in this study).

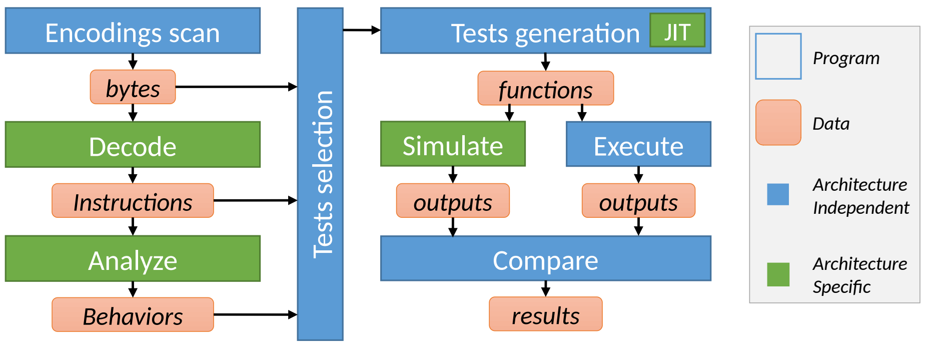


Figure 1: Architecture of the self-verifying simulator.

The test extraction using code introspection is the most original part of the self-verifying simulator since the other steps have been used for decades in processor simulator verification. Analyzing simulator code to generate tests for virtual machines (VM) or real hardware has already been implemented successfully by Fonseca et al. [6]. The authors instrumented the Bochs simulator [10], to extract behavior and generate tests. We advocate that our approach, though similar, goes one step further by encapsulating the instrumentation in a C++ code introspection framework that can easily fit various simulators of different brands (e.g. UNISIM-VP, Bochs) and targeting different instruction set architectures (Intel, PowerPC and ARM). In the self-verifying simulator, architecture-specific procedures (decode, analyze, and simulate in Figure 1) are simple template instanciations of the C++ processor simulator under test. In addition, the integration of all steps from code introspection to online test verification in a single program provides simulator developers with a great tool to improve the simulator accuracy (through quick test-and-fix cycles).

The C++ code overloading that allows instrumentation is designed to require minimal modifications to existing instruction set simulators (ISS). The instrumentation activation is performed at compilation time thus, with no performance penalty, if deactivated. Moreover, once the overloading method is understood, development of new simulators may benefit from this instrumentation without any additional development efforts. In our case, we initially patched the code of existing ARM, PowerPC and Intel simulators, whereas an ARMv8 simulator was developed from scratch with this instrumentation-ready method.

In this paper, we focus exclusively on the x86-64 instruction set for experimental results. Nevertheless, the general operation of this C++ code overloading technique is described below.

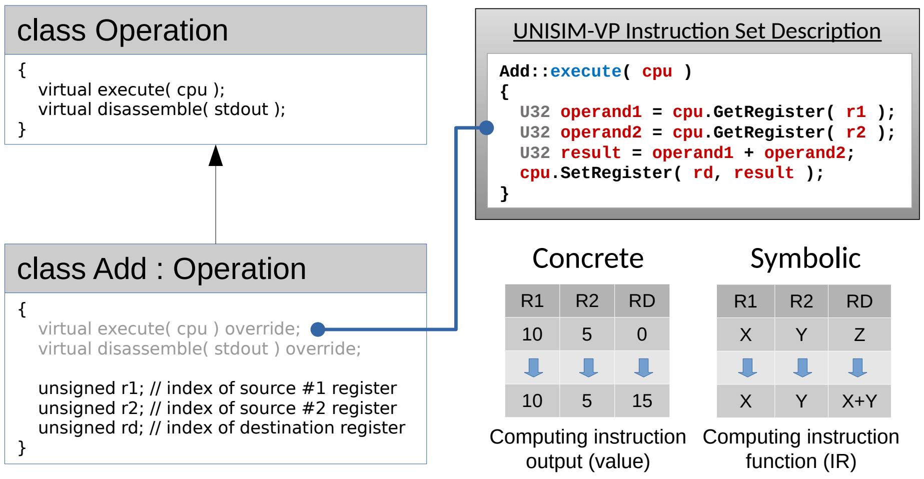
2.2 The UNISIM-VP framework and its Instruction Set Simulators.

UNISIM-VP [1] is a collection of tools and libraries designed to help development of precise executable models of various hardware computing components ranging from electronic embedded systems to general-purpose computing systems. In addition to hardware component models, SystemC-related tools, debugging and profiling tools, UNISIM-VP provides CPU Instruction Set Simulators (ISS). These simulators are developed in C++ and some of them leverage some source code generation using a specialized ISS compiler (GenISSLib), part of the UNISIM-VP framework.

Basically a UNISIM-VP ISS is implemented using a conventional C++ scheme where instructions are represented using objects where data members encode the instructions' variable opcodes (register operand references, constant immediate values) and methods encode the specific operation of the instruction (e.g. execution and disassembly). All instruction classes inherit from a generic instruction class (called *Operation*) that provides a virtual interface to the instruction specific methods. The virtual interface provides means to execute or textually disassemble the instruction (see Figure 2).

2.3 Instruction sets behavioral extraction

The UNISIM-VP framework allows to add any number of methods to each instruction. Thus, it would have been possible to add a method to compute the behavioral description of the instruction in the form of a well-chosen intermediate representation (IR). Nevertheless, separating the behavioral description from the behavioral implementation (its `execute` method) would be error-prone as developers would have to develop and maintain redundant codes in two different methods. Since knowing with accuracy the behavior of the instruction is a strong requirement of our testing methodology, we decided to extract the behavioral description of the instruction directly from the existing execute methods.


UNISIM-VP Instruction Set Description
Concrete
Symbolic
Computing instruction output (value)
Computing instruction function (IR)

```
class Operation
{
   virtual execute( cpu );
   virtual disassemble( stdout );
}
```

```
class Add : Operation
{
   virtual execute( cpu ) override;
   virtual disassemble( stdout ) override;

   unsigned r1; // index of source #1 register
   unsigned r2; // index of source #2 register
   unsigned rd; // index of destination register
}
```

```
Add::execute( cpu )
{
  U32 operand1 = cpu.GetRegister( r1 );
  U32 operand2 = cpu.GetRegister( r2 );
  U32 result = operand1 + operand2;
  cpu.SetRegister( rd, result );
}
```

R1	R2	RD
10	5	0
⬇	⬇	⬇
10	5	15

R1	R2	RD
X	Y	Z
⬇	⬇	⬇
X	Y	X+Y

Figure 2: Principles of UNISIM-VP introspection. Left is the working class hierarchy of ISS instructions. Tables on the right depict the behavioral capture of instructions through symbolic execution.

2.3.1 Self-extraction of arithmetic and logic expressions

In order to extract code from the compiled original execution code, we used C++ polymorphism capabilities to divert the execute method from its main purpose. Replacing concrete types (carrying computation values) with abstract types (carrying variables and equations), we allowed the simulator to perform symbolic execution of instructions. This symbolic execution incrementally extracts code expressions that are gathered at instruction output to reconstruct the instruction's genuine code. The right part of Figure 2 shows how things actually happen; overloaded arithmetic and logic operators process abstract data by producing the expression tree corresponding to their outputs. Final outputs of instructions are extracted from machine state (register and memory) to determine the expressions associated with the instruction operation.

2.3.2 Handling the control flow

Arithmetic and logic computations are easily tracked by abstract type variables. Nevertheless, control flow instructions expecting a boolean condition such as if-then-else statement can not be overloaded to handle abstract type variables. Thus abstract boolean variables have to be "concretized" which practically goes together with the fact that the code is only taking one if-then-else branch at a time. Thus, we have programmed a multiple-execution scheme that keeps track of condition concretization sequences in order to cover all possible execution paths. Using this mechanism effectively allows to reconstruct the original code as a binary decision diagram with arithmetic and logic equation at its nodes.

3 Test campaign

3.1 Unit test selection

Based on the extracted description of instruction behaviors, the unit test generator scans all possible instructions and classifies them to gather a collection of instruction tests that are both feasible and evenly representative. A feasible instruction test is a test that can be run natively in the test program without corrupting the program state (illegal instructions or dangerous memory accesses). This property is ensured by static analyses of the behavioral description. In this work, we deliberately do not address system instructions. Though the side effect of these instructions may be safely controlled and sandboxed in a simulator (e.g. system traps do not translate into real host machine traps), a native execution may come with a series of issues. One way of overcoming these issues is with native sandboxing by catching every possible hardware and software exceptions. In [5], it is shown that with sufficient care, one can execute any x86-64 instruction with perfect sandboxing and rollback of its side-effects.

Additionally, we need to select instruction instances that are evenly distributed according to their behaviors and computations. That means that, on one hand, each instruction selected for test should exhibit a behavior sufficiently distinct from other tested instructions. On another hand, all sufficiently distinct behaviors should be represented. These properties are obtained by sorting behavioral equations of instructions according to a well-chosen order relation. This relation compare recursively each node of the control flow graph (CFG). For each CFG node, local expressions and conditional expressions are compared, in that order according to an expression comparator. The expression comparator recursively compares the trees of operations. Finally, if no difference appears up to the source nodes, these source nodes are compared in a way to qualitatively favor good coverage of tests. Source nodes are either registers or numeric constants. Their comparisons are detailed below.

Intel integer registers are less symmetric and exchangeable than is the case in a pure `RISC` architecture. For example, `RCX` and `RAX` may seem relatively similar, but in practice, many instructions uses `RCX` as a "counter register" and cannot use any other register for that matter. Registers `RSP` or `RBP` are closely related to stack management, without any compiler convention being involved, since some instructions implicitly use them as stack or frame pointers (e.g. `PUSH` or `POP`). Nevertheless, since there are now 16 integer registers in x86-64 architectures, a lot of them are absolutely symmetric and exchangeable. Our comparison relation takes that symmetry into account by having exchangeable source registers renamed to only give credit to their order of appearance in instruction behavioral equations.

Numeric constants are also a source of combinatorial explosion in instruction test generation. Testing all combinations of immediate values in x86-64 instructions would not be feasible since most instructions with immediate values accept 32-bits forms of these immediate values. Yet, different immediate values can lead to radically different behaviors. In particular, some remarkable values (e.g. zero, binary string maximal values) may trigger very specific parts of the code. Our current code introspection mechanism does not capture these conditions and thus cannot isolate proper tests. So far, the best solution we have is to cover evenly immediate values according to the number of zeroes and ones in their binary representation. Thus, the relation order for numeric constant tells that a constant is greater than another is if it has more ones in its binary representation. For 32-bits integers, this generates 33 constant classes. After a sufficient number of draws, each of these classes have a representative. This forces the selection of remarkable values (such as 0) despite having a low initial selection probabilty (e.g. 2^{-32} for 0).

The scanning of the instruction set is done using random draw in the encoding space of the instruction set. In the past, we dealt essentially with fixed size instruction sets or variable length instruction sets with reasonable upper bound on length. In the case of the x86-64 instruction set, the upper bound on instruction length is 15 bytes. Without caution, the encoding space exploration would span over more than 10^{36} possibilities. By taking into account the already decoded instructions and proper mapping of not scanned regions, it is possible to reach an evenly distributed exploration of the encoding space.

3.2 Unit test generation and execution

Once the instructions to test are selected and their behavioral description extracted, the test generator runs to package the instructions under test in small sequences of instructions suitable for native and simulated executions. This packaging involves writing input registers with random inputs coming from an external buffer. Additionally, some register values need to be fixed in order to restrain memory references to safe memory locations. The packaging also involves writing back modified values to a predetermined memory region for later processing.

Finally, the generated tests are executed in a round robin fashion with random inputs. Random generation is performed using reproducible strategies so that found bugs can be easily pinned. The execution is done natively by casting the generated code to a function pointer and directly calling it from the testing program. A simulator is embedded in the program to run the same code. At each instruction, native execution and simulation are compared according to their produced output.

4 Experimental Setup

4.1 Reference hardware

In our approach, the test generator and the simulator, both written in C++, are seamlessly integrated in the same program. Compiling the program for the architecture targeted by the tests (in our case x86-64) allows to go one step further in integration: allowing the tests generation, the tests simulation, the tests native execution and the tests result comparison to happen in the same program. This integration allows straightforward verification of billions of randomly generated tests in seconds.

The verification program was executed on a simple laptop sufficiently recent to provide all instruction set features needed to run generated tests. Experiments run on an Intel Core^{TM} i7-4810MQ (Haswell) CPU. The available features are summarized in Table 1.

Category	Support and comments	Sim.	Nat.	Test
Legacy Floating Point	x87 not tested	Yes	Yes	No
MultiMedia Ext.	MMX no supported	No	Yes	No
Streaming SIMD Ext. Advanced Vector Ext.	SSE SSE2 SSE3 SSE4.1-2 AVX AVX2 (approx. FP)	Yes	Yes	Yes
Optional instructions	cmov popcnt lzcnt tzcnt	Yes	Yes	Yes

Table 1: Instruction set features of Real hardware, Simulator and Test sequences

4.2 Unit test filtering

As mentioned in Section 3.1, the whole instruction encoding space is scanned to select instructions to be tested. Nevertheless, not all scanned encodings translate in valid testing instructions. First, encoded instructions may really be undefined. Though increasingly rare as the instruction set is continuously enriched with new instructions, there are still holes in the x86-64 instruction set. In [5], Christopher Domas shows techniques to probe existence of instructions in the whole encoding space. Surprisingly, he finds instructions that seem valid (following conventional patterns of other legitimate instructions), but are not (yet) documented in Intel specifications. In this work, we only select instructions that our simulator recognizes. This choice may legitimately appears as a strong limitation, but our simulator already recognizes all instructions decoded by the latest *GNU binutils* (relying on the widely spread *libbfd* library). Second, selected instructions may be already over-represented in previously selected tests (e.g. no need, and no time to test all 2^{32} variants of a move-immediate-to-register instruction). Finally, some instructions may not safely be executed in a standard userland environment (e.g. system instructions).

Table 2 provides the main test rejection categories encountered during a 10^5 samples instruction scan. In the end, only 11% of the scanned instructions are kept. More than 20% of all scanned instructions are left aside because they are considered redundant. Note that this figure grows significantly with the amount of scanned instructions, because the distinct behaviors coverage grows with scanned instructions. More details about instruction tests coverage are discussed in Section 5.

A significant portion of rejected instructions (the remaining 69% of all scanned instructions) is due to limitations of our current tester. Some instructions would need additional work to be handled correctly during their native execution. Currently, we do not handle branch instructions that represent 11% of all scanned instructions. A random branch instruction can transfer control to arbitrary memory locations, which is relatively unfortunate for our self-tester. Using virtual memory, memory protection and well-designed exception handlers ([5]), we could overcome the limitation and actually observe the native branch behavior, but that is not yet implemented.

Instructions	Rejection reason
29416	Reserved register access
20786	Redundant tests
14454	Undefined instructions
10376	Branches instructions
4078	Malformed addresses
3257	Not implemented
2951	Legacy x87 floating-point
1640	System (privileged) instructions
1562	RIP relative addressing
342	FS/GS segmented addressing
30	Environment dependent instruction (counters, caches)

Table 2: Instruction scans and test selection

Additionally, our current testing capabilities prevent us from handling a significant amount of memory instructions (6% of scanned instructions) because their address computation is not yet compatible with our address generator: RIP-relative, segmented and absolute addressing. In addition, legacy x87 floating-point requires special non-IEEE floating-point arithmetic, which is currently not supported in the simulator.

Finally, there are two special categories of rejected instructions, which are hard to test but crucial for cybersecurity: system and environment dependent instructions. They represent a small fraction of scanned instructions (less than 2%), but they are a major vector of cyber-attacks. On the one hand, the behavior (or misbehavior) of system instructions is often abused in malware codes. On the other hand, environment-dependent instructions are often the cornerstone of timing attacks or anti-tampering techniques. For the former system instructions, testing is hard because their outputs and side effects are often difficult to confine and capture in a simple test application. For the later environment dependent instructions, testing is also hard because their inputs (hardware counters, timers, caches state...) are naturally difficult to control in a native execution environment. However, in sandboxed analysis environments, these instructions will often be handled differently from the other instructions with special techniques ranging from taint analysis to symbolic execution. We thus argue that the testing of these instructions requires a dedicated effort, which goes beyond the scope of this current work.

5 Results

Our current simulator has been used to validate and analyze several Intel 64-bits applications for more than two years. The simulator was also used in an earlier form dedicated to analyze 32-bits applications (as an IA-32 instruction set simulator) for 10 additional years. During the last year no bug were found, despite using the simulator on more than 5 real-life software codes with trace comparison with native execution; a trace was extracted from simulator execution and that trace was compared to the native execution using single-stepping instructions in a debugger (here GDB, the classic GNU debugger).

The application of our approach eventually lead to a lot of results regarding bugs in our code but also our very understanding of the x86-64 instruction set. Two bugs were due to development errors with roots in straightforward misunderstanding of the specification. Nevertheless, some other bugs were really due to specification misunderstanding on some ambiguous points.

5.1 Instruction tests coverage

Before actually running tests, the self-testing simulator performs a phase of unit tests selection. As mentioned in Section 3.1, the target instruction set is scanned to discover relevant instructions to test. The scanning algorithm is designed to perform a random, but evenly distributed, instruction set space exploration. In order to get an idea of our selected test coverage, we measured the evolution of selected tests during the scanning phase.

Figure 3 shows the result of coverage measure for two algorithms. The left part of the figure represent selected tests for our baseline selection algorithm, as explained in Section 3.1. For this algorithm, we scanned $1M$ instructions, which resulted in approximately $45K$ selected instructions. When we exceed $500K$ scanned instructions, the scanner begins to slow down significantly, mainly due to the associative container holding records for already scanned instructions. Because of the slowdown, we failed to obtain a satisfactory asymptote. Thus we introduced a simplified selection algorithm to see whether we could obtain better coverage figures. The second selection algorithm (corresponding to the right part of Figure 3) differs from the baseline algorithm in its handling of constants. In this second version, constants are considered all equivalent, whereas in the former algorithm only constants with an equal number of ones where considered equivalent. This heavy cut in the instruction set exploration space allowed us to get closer to a satisfactory coverage. Nevertheless, the slowdown limitation of our current algorithm would require an optimization effort.

The final database of $45K$ instruction tests revealed bugs of various forms. Many bugs were operand swaps occurring in rare instructions (input and outputs). Only one computation-centric bug was found; the three-operand form of the `imul` instruction was incorrectly computed in our simulator (this three-operand form is rarely emitted by conventional compilers). Surprisingly enough no address-computation related errors were found. This may be due to 1/ the strongly factorized addressing modes (ModRM) of x86 machines that makes

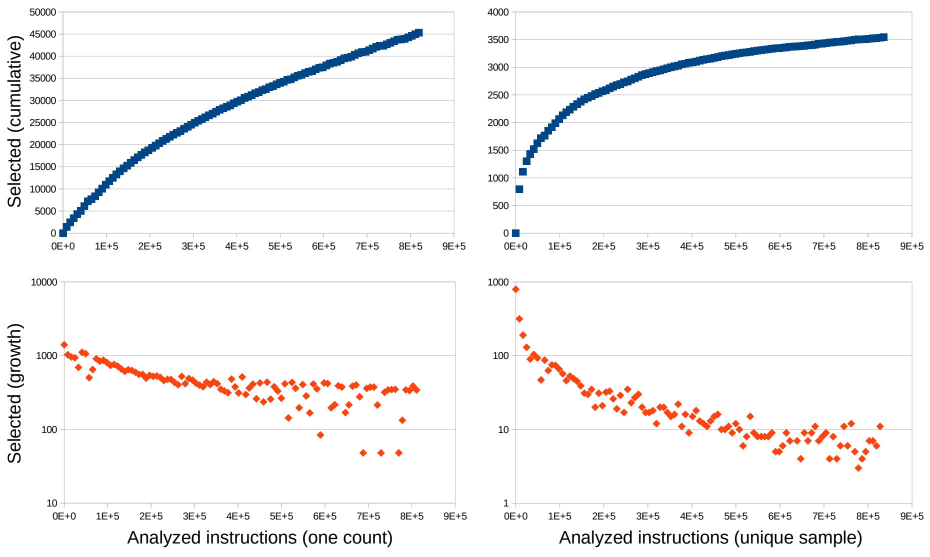


Figure 3: Evolution of selected tests over analyzed instructions for two different selection algorithms.

various forms easily covered by conventional code and 2/ the significant portion of the addressing modes that we cannot yet handle (absolute, RIP-relative, no-base register...). Finally, we found many undefined instructions that were mistakenly recognized by our simulator (illegal prefixes, shadow encodings...).

5.2 QEMU detection using x86-64 segment prefixes

Intel processor architectures provide an addressing mode for memory operations called the segmented addressing mode. This mode allows the computed memory address of a memory operation to be relocated in a given memory region (segment). Though this mechanism was initially designed to extend accessible address space on 16-bits machine, it has evolved among processor generations. It is now slowly being deprecated, especially since AMD's 64-bits transition. Though two segments have been preserved, the four first data segments have been drastically simplified:

> "In 64-bit mode: CS, DS, ES, SS are treated as if each segment base is 0, regardless of the value of the associated segment descriptor base. This creates a flat address space for code, data, and stack. FS and GS are exceptions. Both segment registers may be used as additional base registers in linear address calculations (in the addressing of local data and certain operating system data structures)." ([3])

The 64-bit specification regarding segmented addressing mode seems to imply that segments still exist but that their parameters (base address and size) are hardwired to a flat mapping. The manual later specifies segment prefixes in a section called "2.1.1 Instruction Prefixes". Instruction prefixes are cumulative modifiers (1-byte long) located before the instruction in the instruction flow. These instruction prefixes modify the behavior of the immediately following instruction. Among these instruction prefixes, the segment prefixes allow modifying the default memory reference segment of each instruction. AMD, which has originally created the x86-64 instruction set, specifies (in Revision 3.2.2, 2017) that each segment override legacy prefix (explicitly CS, DS, ES, SS, FS and GS) forces use of the corresponding segment for memory operand. Moreover, to the best of our knowledge, this Intel manual section has not seen any relevant modification to segment prefix description since the beginning of the 64-bit transition. One can naturally think that CS, DS, ES, SS prefixes are still active but that the segment they are referring to are now forced to flat mapping. The reality is in fact different. Our self-testing simulator has generated an instruction test that revealed a discrepancy between native execution and some well-known x86-64 tools.

Let us consider the instruction `#1` of Table 3, which basically loads the content of the memory located at address `$rdi` into register `$rax`. If we prepend the prefix FS to the instruction (see `#2` of Table 3), the instruction now loads unambiguously from the address `$rdi` offset by the base address of the FS segment. Now if we insert an ES prefix in between the FS prefix and the instruction itself (see `#3` of Table 3), things get weird. The Intel and AMD specifications seem to imply that the ES prefix will override the preceding FS prefix and that the `mov` instruction will act in flat memory mapping (because the ES segment is hardwired to flat mapping in 64-bit mode). Popular debuggers and emulators such as GDB (version 8.2), QEMU (version 4.1.0) and most probably other x86-64 tools are in fact doing this interpretation. Our simulator was also making this assumption, which turned out to be wrong; after native execution inspection, we concluded that CS, DS, ES, SS prefixes are now completely ignored. Thus when an ES follows a FS prefix, the FS prefix remains active for the following instruction, meaning the access will be FS-mapped and not flat-mapped.

	prefixes	semantic interpretation
`#1`		`mov ($rdi), $rax`
`#2`	FS	`mov ($rdi), $rax`
`#3`	FS ES	`mov es:($rdi), $rax` (QEMU, GDB) `mov fs:($rdi), $rax` (Intel)

Table 3: Semantic interpretations of an x86-64 load instruction with a variable combination of segment prefixes. Semantic interpretations are described using assembly code in the AT&T syntax.

To sum up, all tools relying on the *GNU binutils* to perform x86-64 instruction decode and disassembly (e.g. GDB, the GNU debugger) produce wrong outputs. QEMU (v4.1.0) also computes wrong addresses when this kind of instruction is issued. Interestingly enough, the Valgrind tool [8] regards this instruction as illegal, which is quite desirable for a code sanitizer. The Bochs simulator [10], known for its accuracy, processes these instructions correctly. It is worth mentioning the source code comes with a comment that explicitly states "ignore segment override prefix" (for the CS, DS, ES, SS prefixes). Finally, the Intel and AMD architectures agree on the treatment of these prefixes (ignoring them), whereas their specifications clearly state the opposite (even explicitly in the case of the AMD specification).

This bug is now fixed in our simulator and a bugfix request is currently submitted to the QEMU tool maintainers. We also developed a small application that is able to use this discrepancy by comparing the results of loads with different prefixes combination and easily differentiating between a native execution and QEMU execution (sandbox detection).

6 Conclusion

In this paper, we have shown a novel technique to produce self-testing simulators. This technique allowed quick setup of billions of relevant tests with a high coverage and stress of the developed simulator. The self-verifying simulator has successfully revealed critical bugs, one of those being also present in QEMU (v4.1.0).

The use of C++ code introspection allows automatic high test coverage with minimal code writing and no redundancy. Because it relies mainly on data polymorphism and operator overloading, this technique is portable and can be applied to other architecture simulators. We have already performed similar instrumentations on PowerPC and ARM simulators, but focused on x86-64 in this paper because the comparison with the real hardware can be performed online, speeding up the test-and-fix cycle.

Finally, using C++ code introspection we can also easily repurpose our simulators for various applications involving dataflow or taint analysis, and also just-in-time compilation and instrumentation. Thus, we strongly believe that this approach to simulator development significantly improves code quality and reuse.

References

[1] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. A. Penry, O. Temam, and N. Vachharajani. Unisim: An open simulation environment and library for complex architecture design and collaborative development. *IEEE Computer Architecture Letters*, 6(2):45–48, Feb 2007.

[2] Alexei Bulazel and Bülent Yener. A survey on automated dynamic malware analysis evasion and counter-evasion: PC, Mobile, and Web. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, ROOTS, pages 2:1–2:21, New York, NY, USA, 2017. ACM.

[3] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, August 2019.

[4] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. L. Potet, and J. Y. Marion. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, volume 1, pages 653–656, March 2016.

[5] Christopher Domas. Breaking the x86 ISA. In *Black Hat 2017*, July 2017.

[6] Pedro Fonseca, Xi Wang, and Arvind Krishnamurthy. MultiNyx: A multi-level abstraction framework for systematic analysis of hypervisors. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 23:1–23:12, New York, NY, USA, 2018. ACM.

[7] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015.

[8] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[9] M. Polino, A. Continella, S. Mariani, S. D'Alessio, L. Fontana, F. Gritti, and S. Zanero. Measuring and defeating anti-instrumentation-equipped malware. In *Proceedings of the 14th Intl Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 73–96, 2017.

[10] The Bochs Project. bochs: The open source ia-32 emulation project. http://bochs.sourceforge.net/, 2017.

[11] F. Recoules, S. Bardin, R. Bonichon, L. Mounier, and M-L. Potet. Get rid of inline assembly through verification-oriented lifting. In *Proceedings of the 34th International Conference on Automated Software Engineering*, ASE 2019, 2019.

[12] X. Wang, S. Zhu, D. Zhou, and Y. Yang. Droid-AntiRM: Taming control flow anti-analysis to support automated dynamic analysis of android malware. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, pages 350–361, New York, NY, USA, 2017. ACM.