December 2016

Selective Dynamic Analysis of Virtualized Whole-System Guest Environments

Andrew William Henderson
Syracuse University

ABSTRACT

Dynamic binary analysis is a prevalent and indispensable technique in program analysis. While several dynamic binary analysis tools and frameworks have been proposed, all suffer from one or more of: prohibitive performance degradation, a semantic gap between the analysis code and the execution under analysis, architecture/OS specificity, being user-mode only, and lacking flexibility and extendability.

This dissertation describes the design of the Dynamic Executable Code Analysis Framework (DECAF), a virtual machine-based, multi-target, whole-system dynamic binary analysis framework. In short, DECAF seeks to address the shortcomings of existing whole-system dynamic analysis tools and extend the state of the art by utilizing a combination of novel techniques to provide rich analysis functionality without crippling amounts of execution overhead. DECAF extends the mature QEMU whole-system emulator, a type-2 hypervisor capable of emulating every instruction that executes within a complete guest system environment.

DECAF provides a novel, hardware event-based method of just-in-time virtual machine introspection (VMI) to address the semantic gap problem. It also implements a novel instruction-level taint tracking engine at bitwise level of granularity, ensuring that taint propagation is sound and highly precise throughout the guest environment. A formal analysis of the taint propagation rules is provided to verify that most instructions introduce neither false positives nor false negatives. DECAF's design also provides a plugin architecture with a simple-to-use, event-driven programming interface that makes it both flexible and extendable for a variety of analysis tasks.

The implementation of DECAF consists of 9550 lines of C++ code and 10270 lines of C code. Its performance is evaluated using CPU2006 SPEC benchmarks, which show an average overhead of 605% for system wide tainting and 12% for VMI. Three platform-neutral DECAF plugins - Instruction Tracer, Keylogger Detector, and API Tracer - are

described and evaluated in this dissertation to demonstrate the ease of use and effectiveness of DECAF in writing cross-platform and system-wide analysis tools.

This dissertation also presents the Virtual Device Fuzzer (VDF), a scalable fuzz testing framework for discovering bugs within the virtual devices implemented as part of QEMU. Such bugs could be used by malicious software executing within a guest under analysis by DECAF, so the discovery, reproduction, and diagnosis of such bugs helps to protect DECAF against attack while improving QEMU and any analysis platforms built upon QEMU. VDF uses selective instrumentation to perform targeted fuzz testing, which explores only the branches of execution belonging to virtual devices under analysis. By leveraging record and replay of memory-mapped I/O activity, VDF quickly cycles virtual devices through an arbitrarily large number of states without requiring a guest OS to be booted or present. Once a test case is discovered that triggers a bug, VDF reduces the test case to the minimum number of reads/writes required to trigger the bug and generates source code suitable for reproducing the bug during debugging and analysis.

VDF is evaluated by fuzz testing eighteen QEMU virtual devices, generating 1014 crash or hang test cases that reveal bugs in six of the tested devices. Over 80% of the crashes and hangs were discovered within the first day of testing. VDF covered an average of 62.32% of virtual device branches during testing, and the average test case was minimized to a reproduction test case only 18.57% of its original size.

SELECTIVE DYNAMIC ANALYSIS OF VIRTUALIZED WHOLE-SYSTEM GUEST ENVIRONMENTS

by

Andrew W. Henderson

B.S., Embry-Riddle Aeronautical University, 1999

M.B.A., Jacksonville University, 2004

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical and Computer Engineering.

Syracuse University

December 2016

To my wife, Cheryl.

ACKNOWLEDGMENTS

The work presented in this thesis could not have been created without the encouragement and guidance provided by many others. I would like to acknowledge those who have helped me throughout this effort.

First and foremost, I am proud to acknowledge my wife Cheryl, daughter Olivia, and father Bill for all of the support and encouragement that they have given me during my doctoral studies. I cannot begin to express how excited I am to spend more time with them after five years of late nights in the lab away from them all. Without their help to make the doctoral process more tenable, I doubt I would have made it through. For this, I will forever be thankful. I look forward to all of the long walks and trips to the park that we will have in our future.

My advisor, Dr. Heng Yin, who has been my mentor in both research and technical pursuits, has shown me the importance of striking a balance between research work and life. His expertise has taught me the importance of communication, attention to detail, and the art of finding "good" research problems to think about. He has also taught me more than I could ever imagine about the fundamentals of research and technical presentation. All of these aspects will serve me very well in the years to come, and I am very grateful that he decided to take a chance on working with the random graduate student that simply walked into his office one day and asked to learn more about his research work.

While my advisor has been my primary source of guidance during my time at Syracuse University, he is one of many faculty members that have given me an opportunity to gain both breadth and depth of knowledge during my doctoral studies. Dr. Wenliang Du has provided me with a variety of opportunities to not only explore mobile platform security, but also to lecture in his classes, collaborate with the students in his lab, and discuss research and industry trends and opportunities. Dr. Ehat Ercanli's coursework and guidance introduced me to the world of single board computers, from which has grown so many opportunities for me that I cannot even begin to count them. Dr. Roger Chen has provided valuable mentorship, advice, and insight into teaching students and navigating the world of higher education.

The help and friendship of my labmates and fellow graduate students (both current students and those long since graduated), has made the long doctoral process much more enjoyable. In particular, the guidance of Lok, Aravind, and Mu and the company of Qian, Xunchao, Rundong, Amit, Mahmuda, and Yousra have helped see me through qualifier exams, submission deadlines, and years of cold and snowy Syracuse weather.

Intelligent Automation, which has funded my research for the past three years, has been very supportive of my doctoral studies. Dr. Jason Li and Dr. Julia Deng have both helped to mentor me through the process of creating, writing, and presenting funding proposals and technical reports. Their guidance has also provided me with training that will serve me well as a principle investigator on my own research projects in the future.

TABLE OF CONTENTS

Page

ABSTRACT i

LIST OF TABLES xi

LIST OF FIGURES xii

1 Introduction 1

1.1 Dynamic analysis design goals 2

1.2 Hardening DECAF against malicious guest activity 5

1.3 Overview of dissertation 7

1.4 Previous publications 8

2 Background 9

2.1 Process-level dynamic analysis 9

2.2 System-level dynamic analysis 11

2.3 Using fuzzing for dynamic analysis 13

3 DECAF 16

3.1 Key challenges 17

3.1.1 DECAF components 21

3.1.2 Example DECAF Plugin 25

3.2 Selective Code Instrumentation 27

3.3 Just-in-Time VMI 32

3.3.1 Goals and Challenges 32

3.3.2 Solution 33

3.4 Precise Lossless Dynamic Taint Analysis 37

3.4.1 Taint Propagation in CPU Registers 38

3.4.2 Taint Propagation in Memory and IO Devices 41

3.4.3 Asynchronous Tainting 42

Page

3.5 Formal Model and Definitions . . . 44
3.5.1 Taint Propagation Rules in Practice . . . 46
3.5.2 Verifying Taint Propagation Rules . . . 48
3.5.3 Constructing Tainting Rules . . . 51
3.6 Evaluation . . . 61
3.6.1 SPEC CPU2006 Benchmarks . . . 62
3.6.2 Per-Trace Verification of DECAF's Tainting . . . 65
3.6.3 API Tracer . . . 69
3.6.4 Keylogger Detector . . . 71
3.6.5 Instruction Tracer . . . 74
3.7 Limitations of DECAF . . . 77
4 Virtual Device Fuzz Testing . . . 80
4.1 VDF Overview . . . 82
4.2 Background . . . 84
4.2.1 Understanding guest access of virtual devices . . . 87
4.2.2 Understanding memory mapped I/O . . . 88
4.3 Fuzzing virtual devices . . . 91
4.3.1 Fuzzing workflow . . . 93
4.3.2 Virtual device record and replay . . . 96
4.3.3 Selective branch instrumentation . . . 104
4.3.4 Creation of minimal test cases . . . 107
4.4 Evaluation . . . 109
4.4.1 Virtual device coverage and bug discovery . . . 110
4.4.2 Classification of all discovered virtual device bugs . . . 113
4.5 Limitations of VDF . . . 121
4.6 Related Work . . . 122
5 Summary . . . 126
A Rule construction and verification: A 2-bit `and` example . . . 128
B VDF Sample Fuzzing Results: SDHCI Virtual Device . . . 133

Page

LIST OF REFERENCES 137

VITA 145

LIST OF TABLES

Table Page

2.1 The scope and purpose of existing dynamic analysis tools. 14

3.1 DECAF supported x86 instructions.
The tainting rules for all these instructions are sound, and most are also precise. The imprecise ones are marked with "*". 40

3.2 Flow Type Results for x86 Instructions Flow Types: (U)p, (D)own, (I)n-place, (A)ll-around, (S)pecial, (N)ot-Supported, (S)pecial, (E)ax is tainted in `cmpxchg`, * - Zeroing Idiom, **Boldface** - Generated Policy is more precise 55

3.3 Precise Rules and Verification Results: Length of operands verified (in bits).
✓Verified for all lengths. * Shift amount is untainted. ^{z} Non-zero operand for `bsf, bsr`. 59

3.4 New Precise Bit-level Taint Rules: `rcr` and `bsr` are similar to `rcl` and `bsf` respectively, and so omitted. The `bsf` rule is shown for a 16-bit value which must be non-zero, and the rule for `rcl` is precise only when the rotate amount is untainted. `x1`, `x2`, and `cf` (carry flag) are the operands while `t1`, `t2`, and `tcf` are the respective shadow taints. 60

3.5 Execution Overhead for DECAF and DECAF with VMI on different architecture/OSs without tainting. 63

3.6 Code breakdown of DECAF, VMI, and various plugins. The code introduced by DECAF in addition to the code of QEMU, which by itself has over 500K LOC. 64

3.7 Trojan.Win32.KeyLogger Trace. 72

3.8 Comparing DECAF with TEMU on tainted shell commands.
"n / m" indicates that "n" bytes are tainted, and "m" tainted EIPs are observed. 73

4.1 QEMU virtual devices seed data sources. 97

4.2 QEMU virtual devices tested with VDF. 110

A.1 Query Results for 2-bit `and` 130

LIST OF FIGURES

Figure		Page
3.1	The overview of DECAF.	23
3.2	A sample plugin for tracking tainted keystrokes.	24
3.3	DECAF inserts instruction execution callbacks into the original TCG code stream (a) to create an instrumented opcode stream (b) to trigger helper function calls to plugin callback functions.	29
3.4	The VMI flowchart	34
3.5	Register liveness tests determine which TCG instructions in the TB (a) should be instrumented for taint propagation, and instrumentation is inserted as needed (b). TCG's optimization logic eliminates unnecessary opcodes, resulting in an optimized, instrumented TB (c).	39
3.6	All events(a) are logged into a staging buffer(b). Logging logic(c) decides which events should be recorded and places them into a circular buffer(d) that is asynchronously written to disk(e).	43
3.7	Example SMT queries checking for information flow (equation 3.1) from the low bit `a` of an input to the 4th bit `b` of an operation output. The first query, for `not`, is unsatisfiable, indicating no flow. The second query, for `add`, is satisfiable, for instance by `x1 = 0` and `x2 = 0xf`: there is flow.	56
3.8	Information flow of `dst` in `or` instruction	56
3.9	Information flow of bits 7, 20 and 31 of `dst` in `sbb` instruction	57
3.10	Pseudocode for `cmpxchg` (flags are omitted)	58
3.11	CINT2006 benchmarks that measure overhead for VMI (a) and inline taint propagation (b).	64
3.12	Per-Trace Verification Overview	66
3.13	Trace entry for `and` bug	68
3.14	Evaluation of API Tracer plugin.	70
3.15	A simple buffer overflow example.	75
3.16	Buffer overflow detection on ARM.	75

Figure Page

3.17 Buffer overflow detection on x86. 76

4.1 Device access request originating from inside of a QEMU/KVM guest. Note that the highest level of privilege in the guest (ring 0') is still lower than that of the QEMU process on the host (ring 3). 87

4.2 The x86 address space layout for port- and memory-mapped I/O. 90

4.3 VDF's process for performing fuzz testing of QEMU virtual devices. 94

4.4 The record format of VDF for an MMIO read/write event. 98

4.5 Simplified control flow graph of the `ide_ioport_write()` function within the QEMU IDE core. 100

4.6 Sample branch coverage data for the `voice_set_active()` function within the AC97 virtual device. 106

4.7 Process for minimizing test cases. 107

4.8 Average percentage of branches covered during fuzz testing. 111

4.9 Average percentage of total bugs discovered during fuzz testing. 112

4.10 The backtrace of the deadlock in the worker thread pool shutdown, which occurs in the TPM backend (entries #2 and #3 in the backtrace). 119

A.1 SMT2 for 2-bit `and` 129

A.2 SMT2 for verifying the 2-bit `and` rule 131

A.3 Sat model for simple 2-bit `and` rule 132

B.1 Function call depth and test cases triggering crashes and hangs during the fuzzing of the SDHCI virtual device in QEMU source file `hw/sd/sdhci.c`. 135

B.2 Discovered, explored, and pending paths during the fuzzing of the SDHCI virtual device in QEMU source file `hw/sd/sdhci.c`. 136

1. INTRODUCTION

Dynamic analysis is the observation and modification of a guest system as it executes for the purpose of understanding the runtime behavior of that system. It has demonstrated its strength in many research problems, such as malware analysis, protocol reverse engineering, vulnerability signature generation, software testing, profiling, and performance optimization. While static analysis, the examination of code or binaries without requiring the execution of that code, can determine many aspects of individual binaries (control flow graphs, path predicates, use of uninitialized memory, "dead code" that can never be reached, etc.), it is unable to determine behaviors that are only observable at runtime. Examples of such runtime behaviors are interactions among concurrent threads, dynamically modified/created code, time-sensitive logic, and complex multi-process interactions via IPC.

Compared to process-level binary instrumentation and analysis, whole-system dynamic binary analysis has unique advantages. First, it provides a complete view of the guest system, including the OS kernel and all running applications, which enables the analysis of kernel activity and the interactions among multiple user-space processes. Second, the code instrumentation and analysis are performed from entirely outside of the context of the guest system under analysis (typically by executing the guest within a virtual machine (VM)). In contrast, process-level instrumentation tools share the same memory space as the instrumented program execution. Leveraging virtualization

techniques, whole-system dynamic binary analysis provides better transparency and stronger isolation than that of process-level instrumentation tools. This is especially important within the context of analyzing malicious code that attempts to detect, evade, and/or tamper with the analysis environment. To discover vulnerabilities within the infrastructure of modern software frameworks, be they embedded, virtualized, or desktop, we must be able to capture and analyze the complete execution of some functionality of interest within an arbitrarily complex guest environment.

1.1 Dynamic analysis design goals

A generic, whole-system dynamic binary analysis platform that can instrument any portion of the guest's execution environment is highly desirable, but challenging to design and create. Unless system-wide dynamic analysis is performed at a reasonable speed, it is useless. Observation of time-sensitive runtime events, such as network communications or GUI interactions, is one of the primary reasons to use dynamic analysis over static analysis methods. Time-sensitive events must be performed in a timely fashion within an instrumented guest to be useful and representative of their non-instrumented execution.

The two primary limitations of dynamic analysis are that guest code must be executed to be observed and that overhead is imposed by the instrumentation necessary to observe, and optionally record, the behavior of the guest. The analysis of every executed instruction within the guest is infeasible. The performance and storage overhead of such a task is too great, and the analysis of such a large dataset is not possible within a

reasonable timeframe. In addition, any instrumentation added to the system to observe code execution may limit or interfere with the functionality of the code under observation.

Some subset of the guest's entire execution, such as the behavior of a particular user-space process or kernel module, is typically the desired subject of an analysis. However, the interaction of this subset with the remainder of the guest environment context must be considered. Because of this, it is infeasible to extract only the guest code of interest and observe its execution in isolation. **Therefore, this dissertation makes the thesis statement that it is possible to unobtrusively dynamically analyze a subset of the guest system's execution while that subset executes within the context of the guest.** In particular, the following questions must all be satisfactorily answered to prove this thesis statement to be true:

1. How and when is context information about the guest environment gathered? Specifically, virtual machine instrospection (VMI) [58] must be implemented in a way that effectively gathers all required guest context information at the proper time to accurately reconstruct the semantics of the guest environment. Existing VMI approaches place an agent within the guest to gather guest context information [82], or continually poll the guest environment [66] (which incurs additional overhead to guest execution). Is there a better way to accomplish this semantic reconstruction without adding unreasonable instrumentation overhead?

2. How can you specify which subset of code within the guest to analyze? Specifically, how is it determined what user or kernel space addresses belong to code of interest? How are these pages of virtual memory mapped to physical memory locations?

How are only those code sections selectively instrumented, rather than instrumenting all code within the guest?

3. How is selective instrumentation of the guest performed without modifying guest execution? Specifically, the overhead for guest execution speed overhead must be low enough that guest behavior is unchanged, and no additional instrumentation (such as a VMI agent) must execute within the context of the guest. The order of instructions executed within the guest must not be perturbed, and any instrumentation added to the guest to collect information must not have side effects that impact guest execution.

4. How can these principles of selective dynamic analysis be coupled with existing analysis tools to form a complex, focused analysis effort? How can heavyweight analysis tools be selective applied to accomplish analysis efforts that were previously considered infeasible?

This dissertation addresses these four questions by presenting a novel new whole-system dynamic analysis platform capable of selectively applying heavy-weight instrumentation to any subset of code executing within the guest environment. This platform is the Dynamic Executable Code Analysis Framework, or DECAF [62, 63].

Although much research has been performed to make use of whole-system dynamic binary analysis to solve various security problems [40, 41, 46, 77, 90, 91], little attention has been paid to the analysis framework itself. Such tools are often tailored to solve specific problems in an ad-hoc manner. Many times, analysts must still develop new analysis tools from scratch to meet their own specific needs. DECAF is built upon the

QEMU whole-system emulator [29], a popular type-2 hypervisor. It aims to address these issues to "Make It Work, Make It Right, Make It Fast". This means that DECAF must not only provide the same set of capabilities as existing analysis systems such as TEMU [82], but it must also follow proper principles in its design. DECAF offers analysis results of better quality, and with a higher correctness guarantee, than TEMU while still conducting analyses more efficiently.

1.2 Hardening DECAF against malicious guest activity

The primary intended purpose of DECAF is the transparent observtion and analysis of the behaviors of malicious software (malware). DECAF is an open-source project [10], and since its first release in January 2013, it has received over 5000 downloads and has been utilized in a number of malware and security analysis studies [25, 37, 54, 85]. It is reasonable to assume that malware authors familiar with DECAF will attempt to attack or evade analysis by attacking and exploiting vulnerabilities in QEMU. *Under no circumstances should activity originating from within the guest be able to attack and compromise QEMU (and by extension, DECAF), so effectively identifying vulnerabilities in QEMU is a difficult, but valuable, problem to consider.*

QEMU uses a virtualized device model: the hardware devices provided to the guest environment are implemented in software within QEMU. Whether QEMU completely emulates the guest CPU or uses another hypervisor, such as KVM [11] or Xen [27], to execute guest CPU instructions, the hardware devices made available to the guest environment will still be QEMU's virtualized devices. Such virtual devices appear as real

hardware devices to the guest environment, and can be interacted with in the same manner. Each virtual device emulates the corresponding interfaces (memory-mapped I/O (MMIO), interrupts, and DMA) of its analogous physical device. Virtual devices may completely emulate the internal state of a piece of hardware, provide a pass-through to a physical device on the host system, or provide some combination of the two.

Because these virtual devices are part of the QEMU binary, they execute at a higher level of privilege than any code executing within the guest environment. They are not directly part of the guest environment, per se, but they are QEMU subsystems that the guest environment directly interacts with. Because of this, a malicious or misbehaving guest may attempt to use these virtual devices in an unpredictable manner. QEMU's virtual devices are a common source of security vulnerabilities [4, 5, 6, 7], are written by a number of different authors, and the most complex virtual devices are implemented using thousands of lines of code. Therefore, it is desirable to discover an effective and efficient method to test these devices in a scalable and automated fashion without requiring expert knowledge of each virtual device's state machine and other internal details.

To ameoliorate the threat of malicious guests attacking DECAF via virtual device bugs, this dissertation also presents Virtual Device Fuzzer (VDF), a novel new fuzz testing [73] framework that provides targeted fuzz testing of QEMU's virtual devices. VDF selectively explores interesting branches within complex programs, namely the portions of the QEMU codebase that implements specific virtual devices. While QEMU does provide a mechanism for testing virtual devices [19], this mechanism is intended for regression testing, rather than the discovery of unknown bugs. By providing such focused testing capable of discovering new bugs within QEMU, VDF aims to better protect not

only DECAF against virtual device attacks, but also QEMU in general and any other QEMU-based analysis platforms [40, 50, 82].

Providing proper seed test cases to the fuzzer is important for effective exploring the branches of a program [38, 79], as a good starting seed will focus the fuzzer's efforts in areas of interest within the program. Therefore, VDF utilizes *record and replay* of virtual device activity to create fuzzing seed test cases that are guaranteed to reach states of interest and initialize each virtual device to a known good state from which to start testing. It then mutates this seed data to generate and replay fuzzed inputs that exercise additional branches of interest within the virtual device.

1.3 Overview of dissertation

This dissertation describes the theory and design of DECAF, as well as three of its analysis plugins, and evaluates their ability to provide a whole-system binary analysis solution that provides answers to the four questions laid out by the thesis statement. It also describes the theory and design of VDF and evaluates its ability to test virtual devices, discover any vulnerabilities within the virtual device code, and produce minimized test cases suitable for the reproduction of discovered issues.

The dissertation is presented in the following manner. Chapter 1 is the introduction of the thesis and an overview of the material presented within the dissertation. Chapter 2 is a survey of background material that presents the current state of the art of dynamic analysis at both the whole-system and process levels. Chapter 3 presents the DECAF system, explains the novel contributions of its design, and evaluates both its benchmarked

performance and capability to perform common analysis tasks. Chapter 4 presents the VDF system, provides additional background material on QEMU's virtual devices, evaluates VDF by fuzz testing a variety of virtual devices, and analyzes the nature of each discovered virtual device issue. Chapter 5 provides a summary of all findings and conclusions. Finally, Appendix A provides a rule construction and verification example of the dataflow rules used within DECAF's system-wide data flow tracking implementation. Appendix B provides a sample set of coverage and result graphs for the fuzz testing of a virtual device using VDF.

1.4 Previous publications

The research material presented within this dissertation is derived from three publications. A portion of the DECAF material presented within Chapter 3 was first published as a peer-reviewed conference paper in the Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA '14)[63]. The remainder of the DECAF work presented in Chapter 3 and Appendix A has been peer-reviewed and accepted for publication in a future issue of the IEEE Transactions in Software Engineering[62] journal. The VDF material presented in Chapter 4 is currently under submission as a peer-reviewed conference paper for the 2017 Network and Distributed System Security Symposium (NDSS '17).

2. BACKGROUND

This chapter presents a survey of existing dynamic analysis tools and techniques. Understang the capabilities and limitations of these prior works provides an understanding of how the state of the art in whole-system dynamic analysis is advanced by the design and features of the DECAF and VDF systems.

2.1 Process-level dynamic analysis

There are many analysis platforms for process-level binary instrumentation, as the dynamic analysis of user-space processes has been a long-studied technique. Several instrumentation solutions perform data flow analyses (known as "dynamic taint analysis" or "tainting") within the scope of a single process or binary. Such solutions are generally much faster than their counterparts implemented for whole-system analysis because process-level instrumentation is limited in scope to only the instructions executed by a single process, rather than all instructions executed across an entire system.

The Pin [70] API is a flexible C/C++ interface used to create process-level instrumentation tools (known as Pintools). Examples of such Pintools are libdft [69] and Dytan [43]. Pintools do not have the benefit of a plugin development API that works at a semantic level higher than that of individual instructions. Dytan is designed as a platform for prototyping different tainting policies. libdft offers a less flexible, but faster, solution

for tracking explicit data flows. It has the same limitations of other Pintools and only supports instrumenting x86 binaries.

DynamoRIO [32] is a runtime code manipulation system that translates process execution on-the-fly to add, remove, and execute instrumentation. Like Pin, it also supports an instrumentation development API to support instrumentation that is triggered during key events such as the execution of individual instructions, loading of libraries, execution of specific function calls, and triggering of system calls. Example tools created using DynamoRIO trace library function calls, count executed instructions and basic blocks [24], track code coverage during execution, and assist in debugging memory. It supports instrumenting 32/64-bit x86 binaries and 32-bit ARM binaries. Similar in functionality to DynamoRIO is Strata [80], which is another runtime code manipulation system. Strata targets Sparc, MIPS, and x86, but provides a coarser level of instrumentation (system call level) than DynamoRIO.

Many efforts have been made to reduce the runtime overhead of process-level dynamic taint analysis. LIFT [78] assumes that taint propagation is not needed for most code execution, so it optimizes performance by taking the fast paths (without taint instrumentation) most of time. It also exploits extra registers in x86 64-bit architectures to shadow taints in x86 32-bit applications. This is a form of selective instrumentation, though the code is actually duplicated into instrumented and non-instrumented forms and the particular version run for any path through the code is selected dynamically at runtime. Minemu [31] leverages the x86 SSE registers to provide lightweight taint tracking for 32-bit x86 applications. Jee et al [65] build upon libdft to create a system that performs a static analysis on a process to selectively instrument the process for dynamic analysis per

the rules of a Taint Flow Algebra. All of these tainting implementations only track taint status, and apply imprecise and sometimes unsound tainting rules, to achieve high efficiency.

Unlike approaches that sacrifice precision and correctness for performance, Memcheck [81] focuses on applying precise and correct tainting rules to troubleshoot memory errors within a process. It uses bitwise tainting to accurately track which bits of memory within the process's memory space have been initialized. Memcheck is able to detect double freeing of memory, usage of uninitialized variables, overlapping source/destination blocks when copying memory, and memory leaks. It favors correctness over efficiency, and does so without relying upon architecture-specific features (e.g., SSE) to improve runtime performance. The dramatic overhead of Memcheck (an average slowdown of 2650%) makes it unsuitable for analyzing software that performs time-dependent tasks. It supports a number of 32/64-bit architectures, include x86, ARM, MIPS, and PPC.

2.2 System-level dynamic analysis

Whole-system instrumentation platforms leverage binary emulation and VMI, and they have long suffered from poor performance. Typically, the guest environment is executed under some form of virtual machine manager (VMM), such as QEMU [29], VMWare [17], or KVM [11], and the guest is unaware that its execution is being virtualized or emulated. The VMM is augmented to perform some form of instrumentation of the guest environment during the guest's execution.

Early whole-system analysis platforms, such as TaintBochs [41], favored accuracy over performance. Ether [48] attempts to elude and analyze VM-aware malware by leveraging Intel VT hardware virtualization extensions. By triggering a debug exception after every instruction, Ether is able to stealthily analyze the state of the system at the cost of heavy execution overhead. However, performing practical, accurate analyses of interactive systems makes the reduction of such high overhead an important focus. ReVirt [51] uses an instrumented UMLinux VMM for the record and replay of compromised guest systems. This allows for more heavyweight analyses based upon repeated replays of previously recorded guest sessions. Aftersight [42] attempts to record information from the guest environment and then analyze it on a different system, offloading the analysis overhead to a different machine.

More recent whole-system instrumentation platforms have been built upon QEMU [29]. Argos [77] performs whole-system taint tracking within honeypot systems for the purpose of generating signatures for network-based attacks. Argos extends the earlier process-level taint tracking system TaintCheck [75]. TEMU [16], part of the BitBlaze binary analysis suite [82], serves as the base for a variety of security analysis tools that perform whole-system analysis, such as HookFinder [91], Panorama [90], and Renovo [67]. TEMU is also not capable of emulating newer OSes such as Windows 7 and 8, and it is only capable of instrumenting x86 platforms. Its design, while feature-rich, creates execution overheads that may be far too heavyweight for simpler analyses that do not require all of TEMU's features.

S2E [40] uses QEMU to perform inline symbolic execution on subsets of guest execution. Guest instructions are transformed into a Low-Level Virtual Machine

(LLVM [22]) intermediary representation, and when execution of the guest environment reaches a branch within code of interest, S2E forks the current QEMU process to explore both branches using LLVM-based symbolic execution. While powerful, this process is quite slow and memory intensive. PANDA [50] leverages the LLVM work performed by S2E to create an analysis platform using record and replay. Tasks of interest are executed within the guest platform and recorded in a log, and then the recorded activity is replayed through a PANDA analysis plugin. This allows for increasingly heavyweight analyses to be performed on the same recorded activity without placing heavyweight runtime performance penalties on the guest at recording time.

The DECAF tool is designed to assist in performing such heavyweight analyses by using lightweight plugins to capture detailed system information and instruction traces that provide enough detail to allow other tools to perform heavyweight analyses offline, if necessary. DroidScope [88] is a dynamic analysis platform for the security analysis of the Android OS. The core idea of DroidScope is to seamlessly reconstruct both Dalvik VM-level and OS-level semantic views and to provide a unified interface for Android malware analysis. DroidScope is an extension to DECAF for Android-specific analyses. Table 2.1 summarizes the scope and purpose of existing dynamic analysis tools, including DECAF.

2.3 Using fuzzing for dynamic analysis

Fuzzing [73] can be leveraged for both system- and process-level dynamic analysis. Because dynamic analysis is only useful if the behavior to be observed is triggered during

Table 2.1: The scope and purpose of existing dynamic analysis tools.

TOOL	PURPOSE	PLUGIN SUPPORT	FINEST TAINT GRANULARITY	ANALYSIS SCOPE	X86	ARM	BASED UPON	LINUX GUEST	WIN32 GUEST	ANDROID GUEST
Aftersight [42]	Record/Replay		N/A	System	✔		VMWare	✔	✔	
Argos [77]	Tainting		Byte	System	✔		TaintCheck	✔	✔	
DECAF [62, 63]	General	✔	Bit	System	✔	✔	QEMU	✔	✔	✔
DroidScope [88]	General	✔	Bit	System	✔	✔	DECAF			✔
DynamoRIO [32]	General	✔	Bit	Process	✔	✔	N/A	✔	✔	✔
DYTAN [43]	Tainting	✔	Byte	Process	✔		Pin	✔	✔	
Ether [48]	General		N/A	System	✔		Xen		✔	
HookFinder [91]	Analysis Plugin		N/A	System	✔		TEMU	✔	✔	
KLEE [36]	Symbolic Exec		N/A	Process	✔		LLVM	✔		
libdft [69]	Tainting	✔	Byte	Process	✔		Pin	✔		
LIFT [78]	Tainting		Byte	Process	✔		StarDBT		✔	
Memcheck [81]	Tainting		Bit	Process	✔	✔	Valgrind	✔	✔	✔
Minemu [31]	Tainting		Byte	Process	✔		N/A	✔		
PANDA [50]	Record/Replay	✔	Byte	System	✔	✔	QEMU	✔	✔	✔
Panorama [90]	Analysis Plugin		N/A	System	✔		TEMU	✔	✔	
Pin [70]	General	✔	Bit	Process	✔		N/A	✔	✔	
Renovo [67]	Analysis Plugin		N/A	System	✔		TEMU	✔	✔	
ReVirt [51]	Record/Replay	✔	N/A	System	✔		UMLinux	✔		
S2E [40]	Symbolic Exec	✔	N/A	System	✔	✔	QEMU	✔	✔	
Strata [80]	General	✔	N/A	Process	✔		N/A	✔	✔	
TaintBochs [41]	Tainting		Byte	System	✔		Bochs	✔	✔	
TaintCheck [75]	Tainting		Byte	Process	✔		Valgrind	✔	✔	
TaintDroid [53]	Tainting		Byte	System		✔	Android OS			✔
TEMU [16]	General	✔	Byte	System	✔		QEMU	✔	✔	

analysis, it is necessary to automate the discovery of inputs that trigger interesting behaviors. Once interesting inputs are discovered, they can then later be replayed while the guest is executing under dynamic analysis. Work in the area of fuzzing has focused on discovering interesting input "seed" data (KLEE [36], AEG [26], COVERSET [79]) and fuzzing with symbolic execution (SAGE [59], Driller [83], TaintScope [84], Mayhem [38], Bitfuzz [35]). EmuFuzzer [71] fuzz tested various x86 emulators (QEMU [29], Valgrind [81], Pin [70], and Bochs [41]) for emulation correctness, showing that fuzz testing not only aids in performing dynamic analysis, but can be used to improve the analysis tools themselves.

The VDF fuzzing framework presented within this dissertation seeks to use record and replay (similar to that seen in tools like PANDA [50]), to test QEMU virtual devices. This provides a solution to the difficult problem of determining seed input that will trigger branches of interest within a complex program.

3. DECAF

DECAF is built on top of QEMU [29], the whole-system emulator and dynamic translator. By extending QEMU, DECAF inherits a mature and feature-rich platform to use as a starting point when implementing its instrumentation and analysis functionality. Because all aspects of the guest environment (e.g. CPU, RAM, hardware devices) are emulated in software, DECAF has many opportunities to monitor the runtime behavior of the guest system.

QEMU's whole-system emulator functionality acts as a type-2 hypervisor for executing guest virtual machines (VMs). It makes use of dynamic binary translation techniques to emulate multiple target guest architectures, so the architecture of the guest environment can differ from that of the host machine. Virtual guest hardware devices, such as network interfaces and IDE/SCSI controllers, are implemented in software and pass data through to the devices physically present on the host system as needed.

QEMU decouples the specific details of the guest CPU from that of the host using its Tiny Code Generator (TCG). TCG translates the instructions of the guest environment into an intermediary representation (IR) of architecture-neutral set of RISC-like instructions. These instructions include common ALU operations (e.g. `add`, `sub`, `xor`), memory load/store, and control flow transfer. This IR is then dynamically translated into the native instructions of the host system and executed. This effectively decouples the

CPU architecture and instruction set of the emulated guest environment from that of the host platform.

DECAF modifies QEMU's TCG to selectively insert instrumentation into the IR at the point of guest-to-IR translation. At the point of IR-to-host translation, the instrumentation becomes embedded within the host instruction stream without disturbing the semantic meaning of the guest's execution. This enables DECAF to support the analysis of a wide variety of different guest architectures while requiring only a minimal amount of architecture-specific code, and without requiring ad-hoc modifications to numerous subsystems. This process is detailed in Section 3.2.

3.1 Key challenges

The following key challenges must be overcome when building a whole-system dynamic binary analysis platform:

1. How to reconstruct a fresh OS-level semantic view from completely outside of the guest system? As we run a virtual machine inside a whole-system binary analysis framework and perform various analysis tasks from outside, we must reconstruct the OS-level semantic view of the guest VM from outside, known as Virtual Machine Introspection (VMI). Several efforts (such as VMWatcher [66], Virtuoso [49], and VMST [56]) have been made to bridge this *semantic gap* and reconstruct the OS-level semantic view. However, the question of "when to reconstruct" has not been addressed. In a running system, the OS-level semantic views constantly change (e.g., a process starts or terminates, a code module is loaded or unloaded). For dynamic analysis, we must be

aware of these new events "just-in-time" at the moment they occur. The TEMU [82] analysis platform circumvented this problem by inserting a kernel module into the guest OS within the VM. This kernel module hooks several system events, retrieves OS-level information, and passes it to the hypervisor through a spare port. This circumvention clearly violates the external monitoring principle for VMI, and it can be easily subverted by the malicious code inside the VM.

DECAF proposes a new, novel solution to reconstructing a fresh OS-level semantic view by only monitoring hardware-level events. Such an approach has not, to our knowledge, been proposed before. It provides notification of OS-level events without requiring the expensive polling of guest kernel data structures or the violation of the external monitoring principle.

2. How to provide an event-based programming paradigm that is both correct and efficient? Most of the existing analysis platforms provide instrumentation interfaces only, through which a plugin can specify which instructions to instrument and what instrumentation code should be run. While this instrumentation approach is simple and flexible, it places a burden on the plugin developers to decide exactly how to instrument guest program execution. Such an approach is acceptable for user-level instrumentation, but it becomes difficult within a whole-system setting. Properly instrumenting whole-system execution requires the analyst to be familiar with the low-level system details of the guest system, such as exceptions, interrupts, page faults, context switches, etc.

Therefore, DECAF must provide an event-based interface, through which an analyst can register for events in various selected contexts (e.g., a process, the kernel space, or a kernel module). DECAF automatically determines what instrumentation code to selectively insert and where, and it ensures that the inserted instrumentation code is correct and efficient. TEMU provides a similar high-level interface, but achieves it in a naive way: it inserts instrumentation code uniformly in *all* translated code blocks and decides at execution time whether to deliver the events to the plugin. This guarantees the correctness of event processing, but incurs unnecessarily high runtime overhead. DECAF selectively inserts instrumentation into only the code blocks where it is needed, dramatically lowering overhead and improving performance.

3. How to implement precise, sound, and lossless tainting? Dynamic taint analysis (tainting) is a powerful dynamic binary analysis technique. Many taint system implementations exist [31, 43, 75, 78, 82]. Among these implementations, two important factors are often overlooked. First, most of these implementations are not *precise* enough (resulting in overtainting), and some of them are not even *sound* (resulting in undertainting). This means that these taint analysis systems would unnecessarily mark many memory locations as tainted and/or fail to taint certain memory locations and CPU registers that should be tainted. When dealing with security problems, an unsound implementation may miss real attacks, while an imprecise implementation may raise too many false alarms.

Second, we often need to track tainted data originating from multiple taint sources by applying multiple labels. Many taint analysis implementations do not distinguish among

multiple taint labels. For the ones that do, they do not provide a lossless guarantee. Each tainted byte or word is associated with up to a small number of taint labels, due to space constraints on shadow memory. When a memory location or CPU register is tainted from more taint sources than those that can be kept in the shadow memory, the remaining are *lost*!

To achieve high precision, DECAF maintains taint information for every bit of registers and memory locations, and it applies precise tainting rules for most instructions at the QEMU TCG IR level. This thesis examines the information-flow patterns in integer operations experimentally, survey previous systems, and in several cases designs new propagation rules when no previous rule was sound and precise. The soundness and precision of these best rules are verified for each operation using two decision procedures (automatic theorem provers), and also using a new technique called *per-trace verification*. An analysis of these rules, using definitions based upon bit-level non-interference, is provided in Section 3.5.2.

To support any number of taint labels without the information loss seen in other systems, DECAF separates tracking of taint status from tracking taint labels. Taint status is tracked efficiently and inline during execution, while taint labels are tracked in an asynchronous manner via plugin-based logging. Taken together with its sound and precise information-flow rules, DECAF offers a novel, sound implementation of whole-system tainting without prohibitive amounts of runtime overhead.

4. How to provide strong support for cross-platform analysis? Ideally, the same analysis code (with minimum platform-specific code) works for different guest CPU

architectures (e.g, x86 and ARM) and different guest operating systems (e.g., Windows and Linux). This requires the analysis framework to hide guest architecture- and OS-specific details from the analysis plugins. Further, to make the analysis framework maintainable and easily extensible to new architectures and OSes, the platform-specific code within the framework must be minimized. Some instrumentation tools, like Pin [70], can run under both Linux and Windows, but, until now, no analysis tool provides support for both multiple architectures and multiple OSes. DECAF provides support for multiple platforms by implementing core instrumentation and analysis tasks at the TCG IR level, independent of the CPU architecture of the VM. DECAF's plugin API is engineered to hide many architecture and OS specific details.

3.1.1 DECAF components

Figure 3.1 provides an overview of DECAF. Inside the virtual machine, programs of interest are run and various analyses are conducted externally via analysis plugins. DECAF has the following key components:

Just-In-Time VMI. DECAF's VMI component reconstructs a fresh, OS-level view of guest execution within the virtual machine, including each of the guest's processes, threads, code modules, and symbols, to support binary analysis. Further, to support multiple architectures and operating systems, DECAF follows a platform-neutral design principle. The workflow for extracting OS-level semantic information is common across multiple architectures and OSes. The only platform-specific handling lies in what guest

kernel data structures are examined and which fields to extract information from. Further details about the VMI implementation is provided in Section 3.3.

Precise, lossless dynamic taint analysis. DECAF ensures precise tainting by maintaining bit-level taint precision for CPU registers and memory, and inlining precise tainting rules within translated code blocks. Thus, the taint status of every CPU register and memory location is processed and updated synchronously during the code execution of the virtual machine. The propagation of taint labels is done by recording to a taint propagation log via a plugin. Later, this log can be analyzed to determine label propagation. This label analysis is done in an asynchronous manner for two reasons: 1) it is impractical and expensive to maintain an unlimited number of labels for each tainted bit in the shadow memory; and 2) for most taint analysis problems, it is not necessary to know which taint labels are associated with all tainted bits in real time. The majority of tainting analyses are only interested in when a key data sink (e.g., the x86 `EIP` register or a sensitive memory buffer) becomes tainted. Once taint reaches such a taint sink, the taint propagation log can be reviewed and the taint labels present in the sink retrieved. By implementing such a tainting logic mainly at QEMU's largely architecture-independent IR level, it becomes much simpler to extend tainting support to a new CPU architecture. Section 3.4 provides more details about DECAF's taint analysis implementation.

Event-driven programming interface. Compared to many existing analysis frameworks [70, 74] that provide only an instrumentation interface, DECAF provides an event-driven programming interface. This means that DECAF's design of "instrument in the translation phase and then analyze in the execution phase" is invisible to the analysis plugins. Plugins only need to register for specific events and implement the corresponding

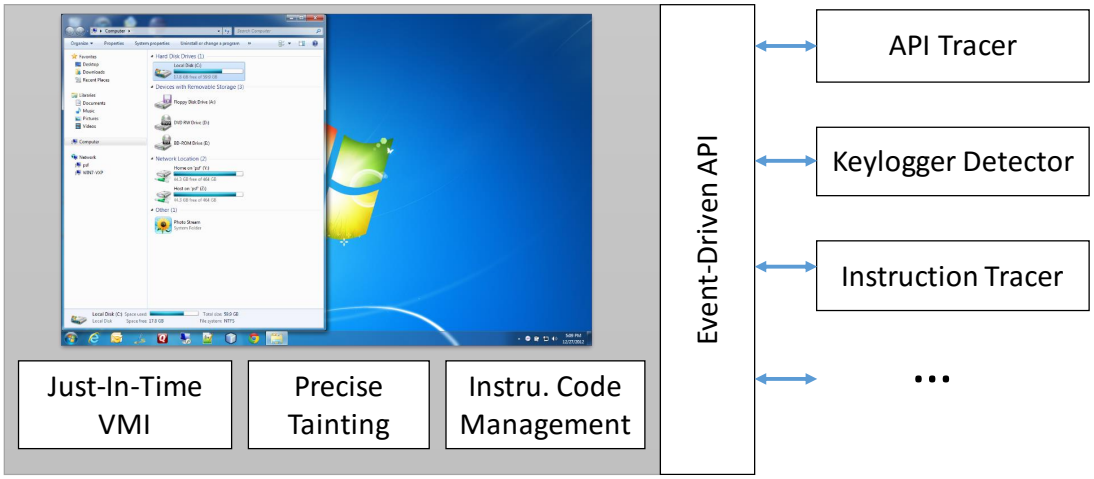


Fig. 3.1.: The overview of DECAF.

event handling functions. The details of how the code is instrumented are handled by the framework, not by the plugins. Such details include how to generate the instrumentation code for inserting these event handlers into the translated code stream and how to maintain instrumentation code consistency when new event handlers are registered and old ones are removed.

Dynamic instrumentation management. To reduce runtime overhead to the guest environment, the instrumentation code is inserted into the translated IR code only where necessary. For example, when a DECAF plugin registers a function hook for a function's entry point, the instrumentation code for this hook is only placed once (at the function entry point). When the plugin unregisters this function hook, the instrumentation code will also be removed from the translated code accordingly. To ease the development of plugins, the management of dynamic code instrumentation is completely taken care of in the framework, and thus invisible to the plugins.

```
/* Define some globals for our plugin logic. */
static plugin_interface_t my_interface;
static DECAF_Handle handle_keystroke_cb;
static DECAF_Handle handle_read_taint_mem_cb;
static int taint_key_enabled = 0;

/* Define the callback triggered when tainted memory
   is read. */
static void my_read_taint_mem_cb(DECAF_Callback_Params *param) {
  char name[128];
  tmodinfo_t tm;
  /* Find the code module accessing tainted memory. */
  if ( VMI_locate_module_c( DECAF_getPC(cpu_single_env),
    DECAF_getPGD(cpu_single_env), name, &tm ) == 0 )
      /* Virtual address and module of the access. */
      DECAF_printf("INSN 0x%08x, Module '%s' Read Key\n",
        DECAF_getPC(cpu_single_env), tm.name);
}

/* Define the callback triggered when a keystroke is
   entered into the guest via a QEMU monitor command. */
static void my_sendkey_cb(DECAF_Callback_Params *params) {
  *params->ks.taint_mark = taint_key_enabled;
  taint_key_enabled = 0;
  DECAF_printf("taint key %d \n", params->ks.keycode);
}

/* Define the function called when the plugin-specific
   "taint_sendkey" QEMU monitor command is used. */
static void do_taint_sendkey(Monitor *mon, const QDict *qdict) {
  if (qdict_haskey(qdict, "key")) {
    /* Enable tainting for the next keystroke */
    taint_key_enabled = 1;
    /* Send the tainted keystroke into the guest */
    do_send_key(qdict_get_str(qdict, "key"));
  }
}

/* Define the "taint_sendkey" QEMU monitor command. */
static mon_cmd_t my_term_cmds[] = {
  {
    .name      = "taint_sendkey",
    .args_type = "key:s",
    .mhandler.cmd = do_taint_sendkey,
    .params = "taint_sendkey key",
    .help = "Send a tainted keypress to the guest"
  },
  {NULL, NULL, },
};

/* Define a cleanup function for plugin unload. */
static void my_cleanup(void) { /* Perform cleanup here. */ }

/* This is executed upon loading this plugin. */
plugin_interface_t *init_plugin(void) {
  /* Register plugin-specific QEMU monitor commands. */
  my_interface.mon_cmds = my_term_cmds;
  /* Register cleanup function called at plugin unload. */
  my_interface.plugin_cleanup = my_cleanup;
  /* Register for DECAF callback events. */
  handle_read_taint_mem_cb = DECAF_register_callback(
    DECAF_READ_TAINTMEM_CB, my_read_taint_mem_cb, NULL);
  handle_keystroke_cb = DECAF_register_callback(
    DECAF_KEYSTROKE_CB, my_sendkey_cb, NULL);
  /* Done! Return this new plugin interface to DECAF. */
  return &my_interface;
}
```

Fig. 3.2.: A sample plugin for tracking tainted keystrokes.

3.1.2 Example DECAF Plugin

Figure 3.2 presents the source code for an example DECAF plugin that detects keylogger malware within the guest system. This plugin tracks the propagation of tainted keystrokes throughout the entire guest environment, and it is both guest architecture and OS independent. The same plugin code works for x86 and ARM, Windows and Linux. Whenever possible, DECAF provides generic functions to abstract away any architecture-dependent details of the guest. For example, `DECAF_getPC` will return the program counter (e.g., `EIP` in x86 and `R15` in ARM), and `DECAF_getPGD` will return the page table directory (e.g., `CR3` in x86 and `CP15` in ARM).

DECAF plugins work by registering callback functions that are executed when events of interest occur within the guest. The sample plugin defines two functions, `my_read_taint_mem_cb` and `my_sendkey_cb`, that ware registered as callback functions. `my_read_taint_mem_cb` is called whenever tainted guest memory is read (the `DECAF_READ_TAINTMEM_CB` event). `my_sendkey_cb` is called whenever a tainted keystroke is entered into the system (the `DECAF_KEYSTROKE_CB` event).

Because it is often necessary for an analyst to interact with a plugin during guest execution, DECAF leverages the QEMU command monitor. The monitor is a shell that accepts commands for controlling and querying the runtime behavior of QEMU, such as starting/stopping guest execution, saving the state of the VM, and profiling QEMU's resource usage. The example plugin code specifies a plugin-specific monitor command, `taint_sendkey`, in the `my_term_cmds[]` array. When this command is entered into the QEMU monitor, the plugin's `do_taint_sendkey` function is called and a tainted

keypress is entered into the guest VM. The `taint_sendkey` command is only available while the plugin is loaded. Upon unloading the plugin, any plugin-specific commands are removed from the monitor.

Every plugin must have an `init_plugin` function. This function is called to initialize the plugin and return a pointer to a `plugin_interface_t` structure, which specifies any plugin-specific monitor commands and a cleanup function (`my_cleanup` in the sample plugin) to be called when the plugin is unloaded. The `init_plugin` function typically registers callback functions for any guest events of interest, but registering and unregistering callbacks can be performed at any point after the plugin has been loaded.

When the analyst loads this sample plugin and then enters the `taint_sendkey` command into the monitor, the registered callback `my_send_keystroke` is called and the corresponding keystroke is tainted. Thereafter, the tainted keystroke will propagate from the keyboard device, through the OS kernel, and to the destination user-level program. Since DECAF performs whole-system dynamic taint analysis, the analyst is able to observe this entire taint propagation flow. Whenever an instruction reads a tainted memory location, the DECAF calls the registered `my_read_tainted_mem` callback, which checks the code module in which this instruction is located. Any relevant information about this taint event is then logged for offline analysis.

3.2 Selective Code Instrumentation

To meet the requirements of efficiency and cross-platform for code instrumentation, DECAF selectively inserts instrumentation code at QEMU's intermediate representation (IR) level.

Dynamic binary translation in QEMU. To support multiple architectures, QEMU makes use of a compiler backend, called Tiny Code Generator (TCG), as its dynamic binary translation engine. QEMU translates each basic block of guest instructions into an architecture-independent TCG IR instructions within a TCG translation block (TB). The TCG compiler then translates each TB into a piece of native code to be executed on the host. Figure 3.3(a) provides an example of how two x86 instructions are translated into these TCG instructions.

TCG instructions include common ALU operations (e.g. `add`, `sub`, `xor`), memory load/store, and control flow transfer. The parameters for each TCG instruction can be temporary variables (registers that exist only within the scope of the current TB), global variables, and constants. For more complex, guest-specific instructions (e.g. floating point operations), a `call` TCG instruction exists for making calls to high-level language helper functions that implement the complex functionality. In this manner, TCG cleanly decouples specific details of the guest's architecture and instruction set from that of the host.

Placement of code execution events. DECAF's code instrumentation integrates coherently into the TCG-based dynamic binary translation process. Events like "block begin/end" (for reaching the beginning/end of a TB) and "instruction begin/end" (for

reaching the IR that begin/end a guest instruction) are used for tracing guest execution. When callbacks for these events are registered by a plugin, DECAF inserts the proper helper function calls into the necessary TBs by pausing the guest's execution, flushing the necessary TBs, retranslating those TBs to include calls to the helper functions (via an inserted `call` IR), and then resuming the guest's execution. Because callbacks are triggered inline with the guest's execution, they are guaranteed to be synchronized to the occurance of events of interest.

Figure 3.3(b) shows the insertation of the two helper functions `DECAF_invoke_insn_begin_callback` and `DECAF_invoke_insn_end_callback` at the beginning and end of each guest instruction, respectively. For many analyses, the analyst is only interested in the execution of a small subset of the guest system, such as the instructions belonging to a single kernel module or user-level process. Plugins can specify ranges of memory addresses, or even a single address, of interest when registering for callbacks. Callback helper functions are only placed into the necessary TBs, and only at the proper locations within each TB, to capture these events as they occur. This greatly reduces the runtime overhead of DECAF.

An important design decision of DECAF is its callback dispatch mechanism. For each kind of event (e.g., "block begin"), only a single helper function (e.g., `DECAF_invoke_block_begin_callback`) is inserted at each desired program location. Within the helper function, DECAF iterates through all registered callbacks for that event and decides which callbacks to trigger. There are two important reasons for this: avoiding multiple callbacks at the same location and efficiently removing stale instrumentation code.

```
// Start of translation block
// Original instruction: orl  %ebx, %eax
mov_i32  tmp11, ebx
mov_i32  tmp12, eax
or_i32   tmp13, tmp12, tmp11
// Original instruction: addl $0x01, %eax
movi_i32 tmp14, $0x01
add_i32  tmp15, tmp14, tmp13
mov_i32  eax, tmp15
// End of translation block
goto_tb  $0x0
```

(a)

```
// Start of translation block
// Insert DECAF_BLOCK_BEGIN callback
movi_i32  tmp21, $<CURRENT_ADDRESS>
movi_i32  tmp22, $DECAF_invoke_block_begin_callback
call      tmp22, $0x0, $0, env, tmp21
// Original instruction: orl  %ebx, %eax
// Insert DECAF_INSN_BEGIN callback
movi_i32  tmp23, $DECAF_invoke_insn_begin_callback
call      tmp23, $0x0, $0, env
mov_i32   tmp11, ebx
mov_i32   tmp12, eax
or_i32    tmp13, tmp12, tmp11
// Insert DECAF_INSN_END callback
movi_i32  tmp24, $DECAF_invoke_insn_end_callback
call      tmp24, $0x0, $0, env
// Original instruction: addl $0x01, %eax
// Insert DECAF_INSN_BEGIN callback
movi_i32  tmp25, $DECAF_invoke_insn_begin_callback
call      tmp25, $0x0, $0, env
movi_i32  tmp14, $0x01
add_i32   tmp15, tmp14, tmp13
mov_i32   eax, tmp15
// Insert DECAF_INSN_END callback
movi_i32  tmp26, $DECAF_invoke_insn_end_callback
call      tmp26, $0x0, $0, env
// End of translation block
// Insert DECAF_BLOCK_END callback
movi_i32  tmp27, $DECAF_invoke_block_end_callback
call      tmp27, $0x0, $0, env
goto_tb   $0x0
```

(b)

Fig. 3.3.: DECAF inserts instruction execution callbacks into the original TCG code stream (a) to create an instrumented opcode stream (b) to trigger helper function calls to plugin callback functions.

DECAF and its plugins may register multiple callbacks on the same event. A dispatch mechanism like this avoids inlining repeated helper function `call` IRs into the TBs, which would negatively impact guest performance. More importantly, in whole-system analysis, callback functions inserted into the code stream are executed within the context of the entire guest system. For example, instrumentation code inserted into the TB containing code for a shared library is executed in all guest processes with that library loaded. So, DECAF's dispatch mechanism must decide at execution time if the current execution context is the correct one for each registered callback.

DECAF also provides a mechanism to efficiently remove any stale instrumentation code. Plugins may frequently register and unregister callbacks at runtime. A common example of such activity is function hooking. A plugin may need to examine the return value and output parameters when an API call returns. To do so, the plugin registers a hook on the entrypoint of that call. When that hook is invoked, the plugin retrieves the return address of the API call and then registers a second hook on its return address. When the second hook is invoked, the plugin inspects the return value and any output parameters. After that, the plugin can remove the second hook for efficiency.

Using the dispatch mechanism described above, it is no longer necessary to immediately remove the second hook, which would require flushing the corresponding code cache and forcing a retranslation of the TB (which hurts runtime guest performance). If no callbacks are associated with an inserted helper function, then no callbacks will be dispatched, which is expected. This little extra function call overhead is several magnitudes smaller than frequent code cache flushing and retranslation. Therefore, DECAF postpones the actual code cache flush to a much later time to improve efficiency.

MMU, IO, and higher-level events. Events like "memory read/write" and "tainted memory read/write" are related to the Software Memory Management Unit (in short, SoftMMU) in QEMU. QEMU must translate each guest virtual address into a guest physical address, and then translate that into a host virtual address. Therefore, the instrumentation for MMU-related events is straightforward: the helper functions are directly inserted into the SoftMMU code. Of course, a dispatch mechanism is still needed to properly deliver the callbacks to the plugin. Some higher-level events are derived from these low-level memory events. For example, VMI events (such as process creation and deletion) are derived from the "TLB execute miss" event.

QEMU emulates a set of common IO devices, such as hard disks, keyboards, and network cards. DECAF instruments the IO events related to these devices by inserting helper functions inside each virtual device's implementation. Such helper functions monitor events related to these IO devices, allowing plugins to taint network input and keystrokes and track tainted data that is swapped out of main memory to secondary storage and vice-versa. A more in-depth discussion of QEMU virtual devices is provided in Section 4.

Dynamic tainting control. A unique feature of DECAF is that it can dynamically enable or disable tainting during analysis. This is a particularly important feature for a whole-system analysis framework. Due to the considerable runtime overhead of tainting, tainting should only be enabled when needed for an analysis. When a user or plugin requests to switch tainting on or off, DECAF flushes the entire translation code cache and reinstruments the new code blocks under the new settings. Details of the implementation of tainting instrumentation at the TCG-instruction level are explained in Section 3.4.

3.3 Just-in-Time VMI

As a binary analysis platform, DECAF must reconstruct the following OS-level semantics of the guest to facilitate custom analysis tasks "out of the box": (1) **Processes.** DECAF must know what processes are running within the guest VM. As many analysis tasks only focus on one or very few user-level processes, this process information is essential to limit the amount of added instrumentation. (2) **Threads.** Many programs are multi-threaded. Knowing which threads are running within a given process is also important for many analysis tasks. (3) **Code modules.** Within a process's memory space, a main executable and several shared libraries are loaded. Binary analysis often needs to know which code module an instruction comes from. Thus, this code module information is also required. (4) **Exported symbols.** Shared libraries export a list of functions to enable other code modules to dynamically link with each other and call exported functions by name. Retrieving exported symbols greatly helps in understanding a program's behavior at the API level, as APIs are exported symbols.

3.3.1 Goals and Challenges

Three primary design goals guide the design and implementation of DECAF's just-in-time VMI. First, a fresh view of the guest OS must always be available to the analyst. For many analysis tasks, the analyst must be immediately notified when a new process is created or a new code module is loaded so a program's complete execution can be observed from beginning to end. No existing VMI techniques are able to provide such a strong timing guarantee.

Second, the VMI technique must be as platform-independent as possible, as the same techniques should work for different CPU architectures and different OSes with minimal platform-specific handling. While one could simply hook specific system calls (e.g., `fork` and `exec`) or kernel functions to meet the first design goal, this approach is very OS-specific and often changes across different OS versions. Doing so would fail to meet the second design goal of platform-independence.

Third, as VMI is a basic functionality required by almost every analysis plugin, the performance overhead for DECAF's VMI technique must be minimal. A key challenge is to meet both this performance requirement and the strong timing guarantee of the first goal simultaneously. DECAF must monitor certain system events more frequently, which may incur high runtime overhead, to continually maintain a fresh view of the guest OS.

3.3.2 Solution

DECAF relies upon the following three observations that commonly hold true across modern platforms to achieve its goals for just-in-time VMI. First, each process must have its own memory space, and each CPU architecture must have a register to indicate the current base address of the memory space of that process (e.g., `CR3` in x86 and `CP15` in ARM). DECAF uses this register to uniquely identify each new process. Second, a Translation Look-aside Buffer (TLB) will have an "execute" cache miss whenever a new code page is loaded and executed. Third, upon context switch, the old mappings in the TLB will be flushed. Therefore, whenever a new process is created or a new module is

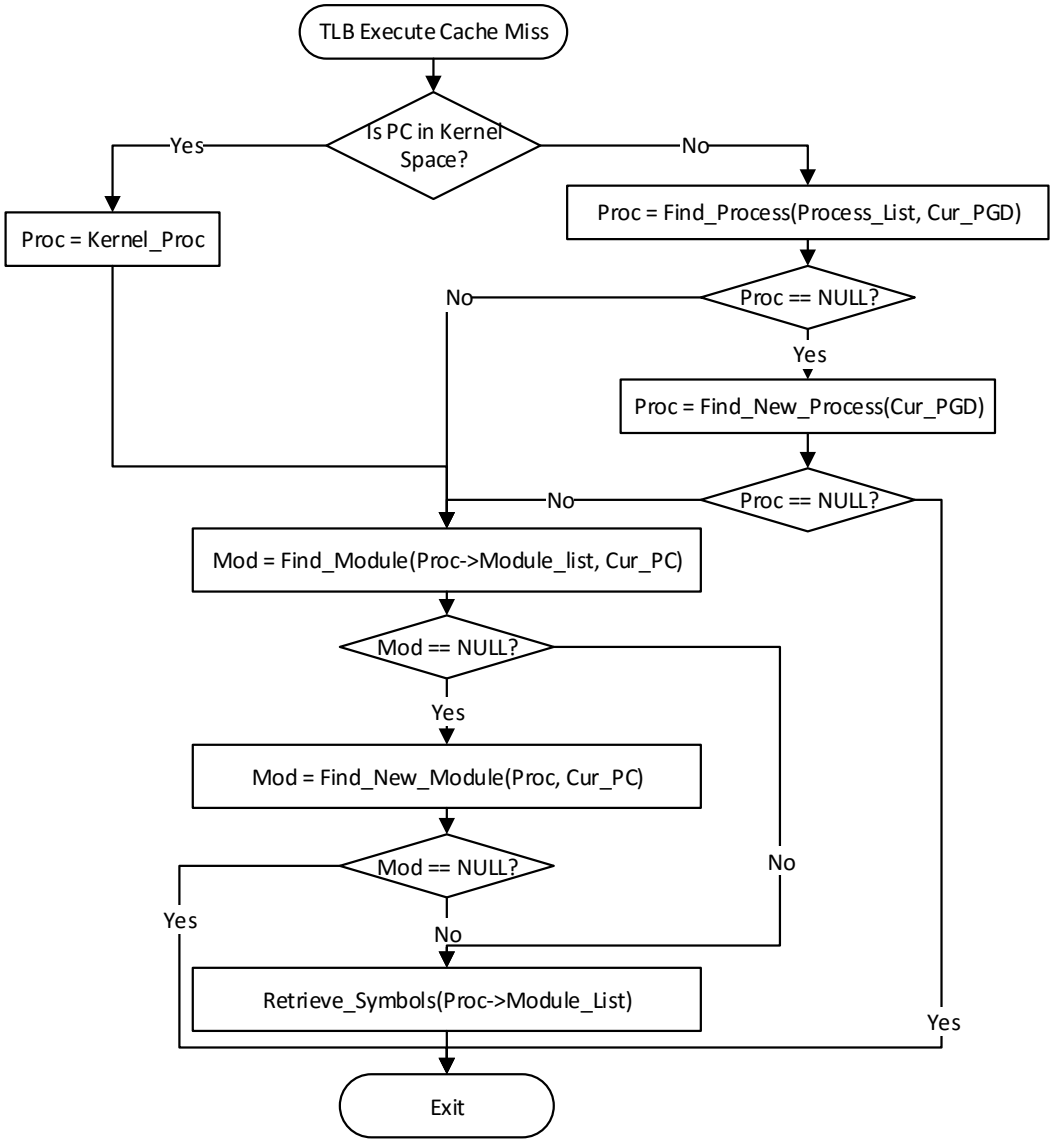


Fig. 3.4.: The VMI flowchart

loaded, DECAF's VMI captures the exact moment it occurs via a TLB Execute cache miss hardware event.

The usage of TLB Execute cache misses for VMI is a novel contribution of the DECAF system. Process-level VMI approaches do not have visibility of such hardware events, but they generally have no need to observe them because the semantics of the process under analysis are already well-known. Whole-system VMI approaches must either continually poll key kernel data structures for changes or violate the external monitoring principle by placing notification code within the guest kernel (using a custom kernel driver or module). Monitoring cache misses allows DECAF to eliminate the overhead of polling key data structures while not violating the external monitoring principle. This results in lower VMI overhead when executing guest environments.

Figure 3.4 illustrates the VMI workflow. Whenever DECAF observes a TLB Execute cache miss, it first checks whether the current program counter is in the kernel space. If not, it determines if the current process is newly created by searching for the current PGD in DECAF's list of current guest processes[1]. If it cannot find the PGD, the process must be new. So, DECAF traverses the kernel data structures (i.e., active process list) of the guest to retrieve information about the newly created process. Thus, DECAF only traverses kernel data structures (which can be a costly operation) when there is a new process.

After DECAF locates the correct process (either it already exists or is newly created), it checks if a new code module has been loaded. Again, DECAF uses a hash table to quickly determine whether the current program counter falls into any code modules that

[1]DECAF uses a hash table to store its list of existing guest processes, so checking for the presence of a particular guest process in the hash table takes constant time.

have been loaded into the current process memory space. If not, DECAF has found a new code module and will traverse the module list in the guest kernel to retrieve information (such as module name, base address, and size) about the new module.

Once DECAF locates the current code module, it starts retrieving the exported symbols of the code modules directly from memory. DECAF must parse the headers (PE for Windows, and ELF for Linux) of each code module to extract symbols. Note that it may not be able to completely retrieve symbols for a newly loaded module the first time DECAF sees it, as related pages of the module may not yet be loaded into guest RAM. Therefore, on future TLB Execute misses, DECAF rechecks the code module to see if additional symbols are now available for retrieval.

This symbol extraction process is fairly heavyweight because it requires many memory reads from the guest to parse executable headers and copy the symbols. However, DECAF only needs to do it once for each code module across all guest processes. Since most code modules are shared libraries (.so files in Linux and .dll files in Windows), this overhead is amortized across the creation of multiple processes.

Unfortunately, TLB cache misses cannot inform DECAF of the exact moment when a process has terminated or a module has been unloaded. To find such events, DECAF must periodically traverse the kernel data structures to find deleted process objects and unloaded code modules. In general, these events are not so timing critical for binary analysis purposes, unlike process creation and module loading events. So, periodically checking (e.g., every 1 or 5 seconds) is acceptable. If an analyst must know the precise time when such termination events happen, the plugins must implement their own mechanism to do so, such as hooking specific functions in the guest execution.

This VMI workflow avoids inserting OS-specific hooks into the VM to obtain a fresh view of the guest OS, and it also avoids frequent memory reads in the VM. The only platform-specific knowledge for this VMI workflow is what kernel data structures to examine and how to interpret the related fields in those structures. The definition of these data structures are publically available. Compared to hooking into system calls and kernel functions, this approach is more stable. Changes on kernel data structures are less frequent than code. It is also fairly straightforward to extract the data structure information from the public symbols of guest OSes.

3.4 Precise Lossless Dynamic Taint Analysis

The primary limitation of all dynamic taint analysis implementations is the runtime performance penalty imposed upon the guest system under analysis. This penalty becomes even greater when multiple taint sources are tracked separately using unique taint labels. Tracking the propagation of multiple taint labels requires either a single heavyweight taint propagation operation that accommodates all tracked labels or multiple lightweight taint propagation operations (one for each tracked label). Neither of these approaches scale when using a large number of taint labels, imposing a limit on the number of taint labels in use simultaneously.

DECAF ameliorates this limitation by performing precise, lightweight taint *status* propagation inline with guest execution while an asynchronous, heavyweight taint propagation of multiple taint *labels* is performed in parallel to the guest execution. DECAF implements its lightweight taint propagation mostly at the TCG instruction level,

so it is easily extended to support multiple CPU architectures. To achieve bit-level precision, DECAF propagates tainted bits through CPU registers, memory, and IO devices.

3.4.1 Taint Propagation in CPU Registers

DECAF creates TCG global variables to shadow the TCG global variables that represent general-purpose and flag CPU registers. Each shadow variable is the same size as the variable that it shadows, and each bit of the shadow variable represents the taint associated with the analogous bit in the variable. For example, the global variable `eax` for an x86 guest is shadowed by `taint_eax`, `ebx` is shadowed by `taint_ebx`, etc. When `eax` contains tainted data, `taint_eax` contains a bitmask that marks which bits of `eax` are tainted. These shadow variables emulate a set of dedicated taint-tracking registers in the guest CPU. DECAF also creates a shadow temporary variable on-the-fly to shadow each temporary variable present inside each TB. For the x86 target, DECAF creates shadow variables for the `cc_src`, `cc_dst` global variables so that taint propagates to CC flags naturally.

Currently, DECAF does not create a shadow memory for the FPU stack and the MMX stack, and it does not have special tainting rules for instructions that operate on these stacks. This is a design decision common in security applications, and this thesis leaves it as a future work to investigate sound and precise tainting rules for the floating point and MMX/SSE instructions.

Once TCG translates guest instructions into a TB containing TCG instructions, DECAF performs a translation pass on the TB to insert additional TCG instructions which

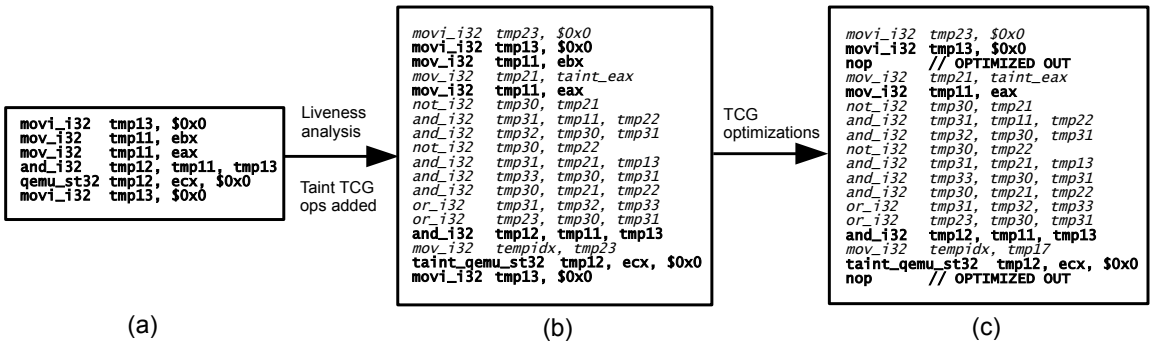


Fig. 3.5.: Register liveness tests determine which TCG instructions in the TB (a) should be instrumented for taint propagation, and instrumentation is inserted as needed (b). TCG's optimization logic eliminates unnecessary opcodes, resulting in an optimized, instrumented TB (c).

implement taint propagation rules that shadow each of the original TCG instructions. For example, Figure 3.5b shows that the instruction "`mov_i32 tmp11, eax`" is shadowed by "`mov_i32 tmp21, taint_eax`". Some tainting rules are far more complex in order to be precise. For example, the `add` operation in Figure 3.5 requires nine extra TCG instructions to precisely propagate the taint bits from two source operands to the destination. DECAF's tainting rules have been formally verified to be sound (guarantee of no under-tainting at instruction level), and most of them have also been verified to be precise (guarantee of no over-tainting). The details are presented in Section 3.5.

Figure 3.5 illustrates this instrumentation pass. TCG translates a basic block of guest instructions into a TB of TCG instructions (a). DECAF performs its instrumentation pass on this TB by first performing a variable liveness analysis on the TCG code to determine if any TCG instruction is unnecessary or redundant. A TCG instruction that fails this analysis will be removed by TCG's optimization later, so there is no need to instrument it. Each opcode to be instrumented is compared against DECAF's list of tainting rules to determine which TCG instructions must be inserted to instrument it. The instrumentation

Table 3.1: DECAF supported x86 instructions.
The tainting rules for all these instructions are sound, and most are also precise. The imprecise ones are marked with "*".

AAA* AAD* AAM* AAS* ADC ADD AND ARPL BOUND BSF BSR BSWAP BTC BTR BTS BT CALLF CALL CBW CDQ CLC CLD CLI CLTS CMC CMOVB/CMOVC/CMOVNAE CMOVBE/CMOVNA CMOVL/CMOVNGE CMOVLE/CMOVNG CMOVNB/CMOVAE/CMOVNC CMOVNBE/CMOVA CMOVNL/CMOVGE CMOVNLE/CMOVG CMOVNO CMOVNP/CMOVPO CMOVNS CMOVNZ/CMOVNE CMOVO CMOVP/CMOVPE CMOVS CMOVZ/CMOVE CMPS CMPXCHG8B CMPXCHG CMP CPUID CWDE CWD DAA* DAS* DEC DIV ENTER FWAIT/WAIT HINT_NOP HLT IDIV* IMUL* INC INS INT0 INT1/ICEBP INT INVD INVLPG IN IRET JB/JNAE/JC JBE/JNA JCXZ/JECXZ JL/JNGE JLE/JNG JMPF JMP JNB/JAE/JNC JNBE/JA JNL/JGE JNLE/JG JNO JNP/JPO JNS JNZ/JNE JO JP/JPE JS JZ/JE LAHF LAR LDS LEA* LEAVE LES LFS LGDT LGS LIDT LLDT LMSW LOCK LODS LOOPNZ/LOOPNE LOOPZ/LOOPE LOOP LSL LSS LTR MOVBE MOVSX MOVS MOVZX MOV MUL* NEG NOP NOT OR OUTS OUT POPAD POPA POPFD POPF POP PUSHAD PUSHA PUSHFD PUSHF PUSH RCL RCR RDMSR RDPMC RDTSCP RDTSC REPNZ/REPNE REPZ/REPE REP RETF RETN ROL ROR RSM SAHF SAL/SHL SAR SBB SCAS SETB/SETNAE/SETC SETBE/SETNA SETL/SETNGE SETLE/SETNG SETNB/SETAE/SETNC SETNBE/SETA SETNL/SETGE SETNLE/SETG SETNO SETNP/SETPO SETNS SETNZ/SETNE SETO SETP/SETPE SETS SETZ/SETE SGDT SHL/SAL SHLD SHRD SHR SIDT SLDT SMSW STC STD STI STOS STR SUB SUB SYSENTER SYSEXIT TEST VERR VERW WBINVD WRMSR XADD XCHG XLAT/XLATB XOR

TCG instructions are inserted prior to the original TCG instruction because some tainting rules (e.g. `and`, `or`) depend upon the values held in both the variables and shadow variables when determining taint propagation. Values held in the variables may change if the same variable is used as both the source and destination of the TCG instruction. Once this pass is complete, the TB now contains both the original and instrumentation code (b). The TCG engine performs an optimization pass on the instrumented TB and generates the final, optimized TB (c), which is then translated into the native instructions of the host and executed.

By implementing tainting rules at the IR level and with some special helper functions, DECAF is able to provide full tainting support for all integer-based x86 instructions and a few floating point and SSE instructions with simple semantics (totaling 369 opcodes - operand types and widths ignored). A complete list of mnemonics with respect the soundness and precision guarantees can be found in Table 3.1.

3.4.2 Taint Propagation in Memory and IO Devices

The guest's physical RAM is shadowed bit-for-bit by a three-level shadow page table. While other instrumentation platforms perform byte-level precision tainting of RAM[82, 88, 90] by representing each byte of taint as a single bit, that approach requires bit masking and shifting operations to represent a 32-bit register in a 4-bit space. DECAF's bit-level precision of shadow memory ensures that taint precision is not lost as taint propagates throughout the guest.

At DECAF start-up, this shadow page table is empty and a pool of available pages is allocated. When tainted data is copied into guest RAM, its taint is placed into the appropriate entry in the page table. If no shadow page exists yet for that location, one is taken from the available page pool and added to the page table. Periodically, a garbage collector traverses the page table and deallocates shadow pages that no longer contain any taint. If the available page pool becomes exhausted, another set of pages will be allocated for the pool. This approach ensures that taint information for a large amount of guest physical RAM can be tracked and retrieved with a minimum of processing overhead and without require a large amount of host memory to be allocated up-front and potentially never used.

An implementation challenge is to re-factor the existing TCG instructions that access guest memory (`qemu_ld/st`) to also access shadow memory at the same time. This is necessary to ensure that taint propagation occurs at the same time that memory accesses occur. The inlined SoftMMU code already uses most of the host's x86 registers for TLB lookup and parameter passing, meaning that the stack must also be used for passing taint

information. This causes performance degradation and potential side effects if unexpected register spillage occurs when taint information is fetched from the stack. To counter this problem, additional shadow global variables are used specifically for copying taint information to and from the shadow page table.

Taint propagation in DECAF's virtualized devices (NE2000 NIC, IDE hard disk, PS/2 keyboard) is similar to taint propagation in memory. Each instrumented virtual device has a device-specific shadow memory, and a specific global variable passes taint data back and forth between device and RAM when programmable I/O or DMA operations occur.

3.4.3 Asynchronous Tainting

DECAF's lightweight taint propagation occurs inline with guest execution so that DECAF can halt execution at the exact moment that taint reaches a specific taint sink (i.e., instruction pointer, system call, virtual device). Asynchronous heavyweight taint propagation relies upon DECAF's Instruction Tracer plugin to efficiently log the taint propagation history. While the plugin is designed to log TCG instructions to record instruction traces, DECAF's flexible plugin interface enables Instruction Tracer to also record memory accesses, CPU states, and taint events. The plugin quickly logs enough information about the taint propagation for the log to be processed asynchronously offline by any custom analysis tool that executes as a separate process. Such tools can consume the taint log information as it is generated (running simultaneously with DECAF) or after DECAF's taint log has completed, performing a much more heavyweight taint analysis on the trace (i.e. reconstructing taint labels and propagation via backward slicing[23]). The

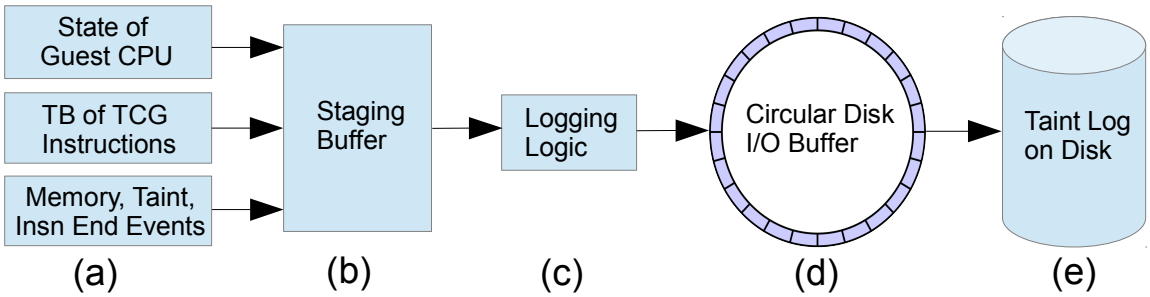


Fig. 3.6.: All events(a) are logged into a staging buffer(b). Logging logic(c) decides which events should be recorded and places them into a circular buffer(d) that is asynchronously written to disk(e).

combination of synchronous lightweight and asynchronous heavyweight taint tracking guarantees that taint detection is both timely and more scalable than the inline tracking of multiple taint labels.

Figure 3.6 shows the steps of the logging process. As each TB begins execution, the plugin writes an identifier for the TB and the current taint state of the CPU registers (a) to a staging buffer (b). If the TB has not been logged previously, or the TB has been flushed and retranslated since it was last logged, all TCG instructions and their arguments held in the TB are written to the staging buffer. Only the original, non-instrumented TCG instructions are written. Any memory and shadow memory accesses (both access size and both the virtual and TLB-resolved physical addresses) are written, as are the introduction of any new taint labels. As each group of TCG instructions implementing a single guest instruction complete execution, an “instruction end” event is recorded in buffer. This is necessary because TB execution can cease early due to jumps, branches, and exceptions. There must be a record of which instructions within the TB were executed so that execution can be reconstructed. When the execution of the next TB begins, the staging buffer is examined (c). If any global shadow variable contains taint, shadow memory is

accessed, or a shadow memory location is marked with a taint label, the buffer is written to a circular buffer (d) that asynchronously writes log data to disk (e). Otherwise, the staging buffer is discarded.

3.5 Formal Model and Definitions

This section begins with an overview of the data-centric noninterference model [87] used by this thesis to analyze instruction level taint trackers. Observations on the model and how it relates to taint tracker implementations in practice are provided. This discussion helps to motivate some of DECAF's design decisions. Finally, this section concludes with the definitions used for formal verification of taint propagation rules and taint analysis implementations.

The original formulation of noninterference by Goguen and Meseguer [60] was applied to a multi-level secure operating system and used a state-machine model. A more modern formulation divides the state of an arbitrary system into two parts, named "high" and "low." Consider two possible starting states of the computation that are the same in their low portions ("low-equivalent"), though the high portions may be different. If the computation satisfies noninterference, then the output states of the computation on those two inputs will also be low-equivalent. Intuitively, this definition captures a lack of information flow from high to low.

When the noninterference principle is applied to dynamic taint analysis, the tainted values correspond to high. Noninterference is a soundness property for tainting, saying intuitively that *tainted values before the computation never affect untainted values after*,

or equivalently that *any value affected by a tainted value is itself tainted*. Precise tainting is also desirable: subject to the constraint of noninterference, the amount of data tainted should be as small as possible.

The usual definition of noninterference considers the entire tainted (high) state of a system, but for reasoning about noninterference it suffices to consider the effect of changing an arbitrarily small part of the state. Stated informally, if a large change has an effect, then among the smaller changes that make it up, at least one must also have an effect. Taking advantage of this property, the analysis can be narrowed to consider the effects of the smallest possible change: changing a single bit from 0 to 1 or vice-versa.

For the purposes of this thesis, the state of the computation system (e.g., a CPU) is modeled as a vector of bits. The symbols $\wedge$, $\vee$, $\oplus$, and an overbar are used to represent the Boolean operations of AND, OR, XOR, and NOT either on single bits or bitvectors, equivalent to the `&`, `|`, `^`, and `~` operators in C. S is the set of possible states, equal to all the bitvectors of a particular fixed size. Bit positions are identified with bitvectors that have just that one bit set, and use the notation $v|_b$ for extracting a single bit b from a bitvector v.

Definition 3.5.1 *Let a and b be bits in the state of a system. A computation has an* information flow from a to b *if there are two input states s_0 and s_1 that are identical except that s_0 has $a = 0$ and s_1 has $a = 1$, and in the corresponding output states s'_0 and s'_1, the values of b are different (one 0 and the other 1, in either order). In other words, if*

the computation is a function f on state vectors, and a is a bitvector with only a single position set to `1`*, there is a state vector $s \in S$ such that:*

$$f(s \vee a)|_b \neq f(s \wedge \overline{a})|_b \tag{3.1}$$

From the untainted (low) perspective on a computation, tainted bits are ones whose values are unknown. Thus we can use a shorthand notation analogous to three-valued logic with three kinds of digits: `0` to represent a bit with value zero which is untainted, `1` for a bit with value one which is untainted, and `X` for any tainted bit. Thus `1X0` represents a number whose second bit is tainted; in effect, the value from the high perspective might be either 4 (binary `100`) or 6 (binary `110`).

3.5.1 Taint Propagation Rules in Practice

There are three important observations about this data-centric noninterference model. First, the model is defined using *information flows between bits*. Thus, it directly describes systems in which taint is labeled per bit. Not all implementations take this approach, but the model extends naturally to coarser-grained taint. For example, there is information flow from byte x to byte y as long as there is information flow from any bit of x to any bit of y. Results from a coarse-grained analysis are inherently limited in their precision, but for any granularity, DECAF can try to achieve the most precise results expressible at that

granularity. DECAF seeks to explore this at the maximum possible precision (bit-level tainting).

Second, the precision of taint results also depends upon the granularity of the *computation* analyzed. The reason for this is that the taint status of bits does not include information about how some bits might be correlated with others. For instance, suppose that a single tainted bit `X` (representing either 0 or 1) is multiplied by an untainted value 3 (binary `11`). The result must be either 0 (`00`) or 3 (`11`); thus, both the low bits should be tainted and represented as `XX`. If the source of where the untainted value came from is known, then it is known that the first and second bit positions must have the same value. But, this information is missing in the tainted-bit representation, which could equally well describe a 1 (`01`) or 2 (`10`). This inherent imprecision of the representation leads, in turn, to imprecision in later results. For instance if the tainted bit value `XX` is multiplied by 3 again (i.e., ? $\times$ 3 $\times$ 3), the result is `XXXX`, since there is information flow to each of the four bits of the result. On the other hand, if instead of multiplying it by 3 twice (as two separate operations), we had started with the tainted bit `X` and multiplied it by 9 in one operation, the result becomes the more precise `X00X`.

This is a general phenomenon: expressing a larger computation in terms of smaller ones and applying sound taint analysis to each operation separately will always give sound final results. However, applying precise taint analysis to each operation separately will often not give as precise of a result as analyzing the entire computation at once.

At the binary level, there are two common choices for taint analysis: either perform the analysis and update the taint labels after each instruction, or translate each instruction into a sequence of simpler IR operations and analyze the taint effects of each IR operation

separately. Though this IR-level approach has other advantages, it can come at a cost to precision for the reason described in the previous paragraph. As an instruction-level example, consider an instruction (such as the `BIC` instruction on ARM) which computes the bitwise-AND of one register and the bitwise negation of another: $z = x \wedge \overline{y}$. If the two inputs are the same register, this has the effect of clearing the output register, so if this instance of this instruction is analyzed as a unit, the output should be completely untainted. On the other hand, an IR-level taint analysis that treated the AND and NOT as separate operations would be unable to tell that one operand of the AND was the negation of the other, so the result would still be tainted. The formal verification described in this thesis can reveal these kinds of imprecision.

A final remark is that, as specified so far, the model does not place any further restrictions on the choice of the input state s; the specific selection comes from the context in which we are verifying a taint analysis. To analyze the taint propagation in a particular situation, a concrete value can be specified for s. For instance, a program state encountered during testing can be used. On the other hand, in constructing rules for taint propagation, such rules should work correctly in *all* situations. So, taint rules should soundly and precisely capture information flow for any choice of s. In short, s is a free variable when constructing rules and s is concretized when verifying rules.

3.5.2 Verifying Taint Propagation Rules

Taint propagation rules have usually been defined based on domain expertise and then reasoned about manually, or simply left unverified due to the difficulties of manual

verification. For example, Memcheck [81] has many special case rules, but according to its project suggestions webpage, formal verification of the rules is still needed [12]. The concepts for formal verification of tainting rules are introduced in this section.

The most obvious representation for bit-level taint, used by Memcheck, is to maintain taint bits parallel to data bits with the same structure: for instance, the taint information for a 32-bit data word is represented by another 32-bit word, with the first bit of the taint word reflecting the taint status of the first bit of the data word, etc. DECAF's implementation of shadow memory also uses a bit-for-bit mapping, adopting the convention that a taint bit value of `1` indicates that the corresponding data bit is tainted, while `0` indicates untainted. Memcheck uses the opposite convention in its implementation (for what are referred to as validity or "V" bits), but because of the duality of Boolean algebra, the choice makes little difference.

The suffix $\_t$ denotes shadow variables that hold taint; for instance $S\_t = S$ is the set of all possible taint states. The taint propagation rule for a given operation is a function that takes as inputs the data state before the operation and the taint state before the operation, and yields the taint state after the operation: $\mathrm{rule}_{op} : S \times S\_t \rightarrow S\_t$. This thesis uses the definition of a sound and precise rule as one where the taint bit for an output position b should be set if (soundness) and only if (precision) there is an input bit position a for which there is information flow from a to b and a is tainted.

An equivalent perspective on the soundness of a rule, analogous to noninterference, is that for each bit position b that is untainted after an operation, it should be the case that for any choice of values for the tainted input bits, the value of that untainted output bit is constant. If this condition fails, and there is an output bit that is affected by the tainted

input but is not itself tainted, the rule suffers from a false negative error. As a formula, let y_t be the output taint after applying the rule for the operation *f* to the input data state x and the input taint x_t. This is formally expressed as the following definition.

Definition 3.5.2 *A rule* $y\_t = \mathrm{rule}_f(x, x\_t)$ *applied to an operation* $y = f(x)$ *has a* false negative *error if:*

$$\exists b, x_1, x_2 : (y\_t|_b = 0) \wedge ((x_1 \wedge \overline{x\_t}) = (x_2 \wedge \overline{x\_t})) \wedge$$
$$(f(x_1)|_b \neq f(x_2)|_b)$$

for some bit position b*. Equivalently, all of the untainted output positions can be compared at once:*

$$\exists x_1, x_2 : ((x_1 \wedge \overline{x\_t}) = (x_2 \wedge \overline{x\_t})) \wedge$$
$$((f(x_1) \wedge \overline{y\_t}) \neq (f(x_2) \wedge \overline{y\_t})) \tag{3.2}$$

Conversely, a rule has a false positive error if there is a bit position which is tainted, but does not in fact depend on the tainted input:

Definition 3.5.3 *A rule* $y\_t = \mathrm{rule}_f(x, x\_t)$ *applied to an operation* $y = f(x)$ *has a* false positive *error if:*

$$\exists b : (y\_t|_b = 1) \wedge \forall x_1, x_2 : \tag{3.3}$$
$$((x_1 \wedge \overline{x\_t}) = (x_2 \wedge \overline{x\_t})) \Rightarrow (f(x_1)|_b = f(x_2)|_b)$$

Observe that the input state variables x and x_t are free in Equations 3.2 and 3.3. When checking the taint propagation in a trace, they are instantiated with values taken from an execution. When checking the correctness of a rule in the abstract, we quantify over all possible values for x and x_t: a rule is sound if there is no value of x and x_t for which Equation 3.2 holds, and precise if there is no value of x and x_t for which Equation 3.3 holds.

3.5.3 Constructing Tainting Rules

In the previous section, a formal model was presented for taint analysis based on noninterference, and defined soundness and precision based on information flow. In security applications, unsoundness can lead to missed attacks (a result considered worse than false alarms). So, a set of rules must first be constructed to be guaranteed sound, and then refined to maximize precision.

This section focuses on the key concepts in constructing precise tainting rules. The reader can refer to an in-depth technical report [89] on the material for a more detailed

treatment of topic, including examples of how these were actually applied for DECAF. Constructing tainting rules is separated into three key steps. First, sound tainting rules are constructed by identifying all bit-wise information flows in operations. Second, SMT solvers are used to verify that the rules are indeed sound. Third, the sound rules are improved upon to create precise rules; these precise rules are formally verified as well. Examples are drawn from the x86 instruction set, but the techniques and most of the specific rules are applicable to other architectures, since the same basic data and ALU operations (such as addition and bit shifts) are provided by all CPUs.

Constructing Sound Rules

Recall that a rule is sound if every information flow from a tainted input bit to an output bit is noted by making the output bit tainted. Thus, to construct a sound rule, all possible information flows within an instruction are first identified and then these flows are summarized with a rule. Since definition 3.5.1 is a satisfiability problem, satisfiability-modulo-theories (SMT) solvers are used to identify the information flows. To do this, the behavior of each instruction of interest is modeled using the bitvector operations of SMT solvers. Then, queries are submitted to SMT solvers to identify all information flows. For DECAF, all of the instructions are modeled in SMT-LIB Version 2 [28] ("SMT2" for short) to maintain compability with a wide range of solvers.

Stage 1: Behavioral Definitions: There are two general ways to define the behavior of instructions: manual and automatic. Godefroid and Taly [59] presented algorithms to automatically generate the behavioral specifications of common x86 instructions. The key

intuition behind their approach is that many instructions follow specific behavioral "templates" (e.g., an *addition* template will cover the `add`, `sub`, `inc`, and `dec` instructions). Thus, their algorithms use a small number of manually defined templates to automatically specify the behaviors of a large number of instructions. While an automated approach is available, the effort for DECAF's uses the manual approach. This is because previous experiments making extensive use of the x86 instruction set and BAP [33] facilitated a manual definition of the behaviors much quicker than reimplementing Godefroid and Taly's algorithms. Additionally, templates for special instructions such as `cmpxchg` were not readily available, requiring the manual definition of those instructions anyway.

Since the correctness of the behavioral definitions is paramount, both BAP and the Intel developer's manuals [64] were used to help define the models. Please note that if any errors exist in the behavioral definitions at this point, they will be revealed when Per-Trace Verification (described in Section 3.6.2) fails as well. To manually define the behavior of the x86 instructions, the instruction set was first divided into four categories: *data transfer*, *control transfer*, *arithmetic and logic*, and *special*. Data and control transfer instructions have simple semantics with obvious bitwise information flow relationships and do not warrant further analysis. The arithmetic and logic category includes instructions that are likely to be supported in any general-purpose architecture. The remainder of the instructions fall into the *special* category. The `cmpxchg` is a prime example of a special instruction since it has an unusual information-flow pattern.

For all of the instructions of interest, assembly code was written to exercise different aspects of their behavior. This assembly was then linked them into an executable and

lifted the executable into BAP's internal IR (BIL). This resulted in a collection of BIL that summarizes the instruction. A single SMT2 behavioral representation was then extrapolated from the instruction's BIL instances and cross-checked against the processor documentation.

In total, over 150 different arithmetic and logic instructions were analyzed. After some initial tests, the precise mnemonics and operand choices (e.g., `add r/m8, r8` vs. `add r8, r/m8` vs. `add r16/32, r/m16/32`), were found to not affect the information flow patterns. Thus, the focus of the analysis was on only generic 32-bit register instruction formats (e.g., `add dst, src`). The 26-instruction test set that was used is outlined in Table 3.2. Please note that, similar to Godefroid and Taly's intuition on templates, while the analysis focused on these 26 instructions, the design for DECAF (using IR-level tainting) enables the use of the rules for these 26 instructions to support most of the x86 instructions.

Stage 2: From Information Flow to Sound Rules: The goal of this stage is to take the SMT2 files from stage 1 and identify all possible information flows. For each file, all possible pairs of input and output bits are iterated through and Z3 [47] is queried for the satisfiability of the condition in Definition 3.5.1. A "sat" result means that there is information flow and an "unsat" result means there is none. Figure 3.7 shows two (simplified) example queries. The resulting statistics for all the instructions are summarized in the first five columns of Table 3.2. The instructions are presented in the first column; the input operands, both implicit and explicit, in the second; output operands, both implicit and explicit, in the third; the total number of input-bit to output-bit combinations in column four; and the time it took for Z3 to process the queries is shown

Table 3.2: Flow Type Results for x86 Instructions
Flow Types: (U)p, (D)own, (I)n-place, (A)ll-around, (S)pecial, (N)ot-Supported, (S)pecial, (E)ax is tainted in `cmpxchg`,
* - Zeroing Idiom, **Boldface** - Generated Policy is more precise

Instruction	Inputs	Outputs	# Cases	Runtime	Flow Type	DroidScope[88]	libdft[69]	Minemu[31]	TEMU[?]	Memcheck[?]	Codename:suthesis
adc *dst, src*	dst,src,cf	dsr,src,zf,of,sf,af,cf,pf	4550	1m19s	U	**A**	I	**A**	S	U	S
add *dst, src*	dst,src	dst,src,zf,of,sf,af,cf,pf	4480	1m13s	U	**A**	I	**A**	**A**	S	S
and *dst, src*	dst,src	dst,src,zf,sf,pf	4288	1m05s	I	**A**	I	**A**	I	S	S
dec *dst*	dst	dst,zf,of,sf,af,pf	1184	20s	U	**A**	I	**A**	**A**	U	S
div *rm*	edx,eax,rm	edx,eax,rm	9216	95m48s	D	**A**	I	N	**A**	**A**	D
idiv *rm*	edx,eax,rm	edx,eax,rm	9216	307m04	A	A	I	N	A	A	A
imul1 *rm*	eax,rm	edx,eax,rm,of,cf	6272	289m51s	U	**A**	I	N	**A**	U	U
imul2 *dst, rm*	dst,rm	dst,rm,of,cf	4224	52m37s	U	**A**	I	N	**A**	U	U
imul3 *dst, rm, imm*	rm,imm	dst,rm,imm,of,cf	6272	53m56s	U	**A**	I	N	**A**	U	U
inc *dst*	dst	dst,zf,of,sf,af,pf	1184	19s	U	**A**	I	**A**	**A**	U	S
mul *rm*	eax,rm	edx,eax,rm,of,cf	6272	16m02s	U	**A**	I	N	**A**	U	U
not *dst*	dst	dst	1024	15s	I	**A**	I	**A**	I	I	I
or *dst, src*	dst,src	dst,src,zf,sf,pf	4288	1m05s	I	**A**	I	**A**	I	S	S
rcl *dst, imm8*	dst,imm8,cf	dst,imm8,of,cf	1722	42s	A	A	N	A	A	A	S
rcr *dst, imm8*	dst,imm8,cf	dst,imm8,of,cf	1722	42s	A	A	N	A	A	A	S
rol *dst, imm8*	dst,imm8	dst,imm8,of,cf	1680	41s	A	A	N	A	A	S	S
ror *dst, imm8*	dst,imm8	dst,imm8,of,cf	1680	41s	A	A	N	A	A	S	S
sal *dst, imm8*	dst,imm8	dst,imm8,zf,of,sf,af,cf,pf	1840	35s	U	**A**	N	**A**	S	S	S
sar *dst, imm8*	dst,imm8	dst,imm8,zf,of,sf,af,cf,pf	1840	34s	D	**A**	N	**A**	S	S	S
sbb *dst, src*	dst,src,cf	dst,src,zf,of,sf,af,cf,pf	4550	1m21s	U	**A**	I*	**A***	**A**	**A**	S
shr *dst, imm8*	dst,imm8	dst,imm8,zf,of,sf,af,cf,pf	1840	35s	D	**A**	N	**A**	S	S	S
sub *dst, src*	dst,src	dst,src,zf,of,sf,af,cf,pf	4480	1m17s	U	**A**	I*	**A***	**A***	S	S
xor *dst, src*	dst,src	dsr,src,zf,sf,pf	4288	1m05s	I	**A**	I*	**A***	**A***	I	I
bsf *dst, src*	src	dst,src,zf	2080	31s	A	N	I	N	A	A	S
bsr *dst, src*	src	dst,src,zf	2080	31s	S	N	I	N	**A**	**A**	S
cmpxchg *rm, r*	eax,rm,r	eax,rm,r,zf,of,sf,af,cf,pf	9792	2m39s	S	N	**E**	N	**E**	**E**	S
TOTAL			102064	13h52m48s							

in column five. A new instance of Z3 is used for each test case and thus the timing results include process creation overhead.

As expected, logical operations return results extremely quickly whereas signed multiply and divide takes the most time. Overall, it took less than 14 hours on an Intel Core-i7 860 to automatically identify all information flow relationships for the arithmetic and logical instructions.

The Rules: Once all of the possible information flows were revealed, the flows are then summarized into simple rule types. The sixth column of Table 3.2 indicates the

```
(declare-fun x1 () (_ BitVec 32))
(declare-fun x2 () (_ BitVec 32))
(declare-fun a () (_ BitVec 32))
(assert (= a #x00000001))
(declare-fun b () (_ BitVec 32))
(assert (= b #x00000010))

;; For NOT:
(assert
 (not
  (= (bvand b (bvnot (bvor x1 a)))
     (bvand b (bvnot (bvand x1 (bvnot a)))))))

;; For ADD:
(assert
 (not
  (= (bvand b (bvadd (bvor x1 a) x2))
     (bvand b (bvadd (bvand x1 (bvnot b))
                     x2)))))
```

Fig. 3.7.: Example SMT queries checking for information flow (equation 3.1) from the low bit `a` of an input to the 4th bit `b` of an operation output. The first query, for `not`, is unsatisfiable, indicating no flow. The second query, for `add`, is satisfiable, for instance by `x1 = 0` and `x2 = 0xf`: there is flow.

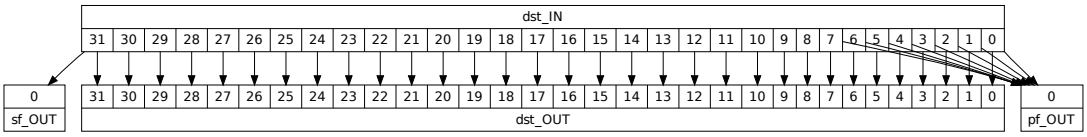


Fig. 3.8.: Information flow of `dst` in `or` instruction

general flow type for each instruction. There are four distinct information flow patterns between the source and destination operands. A fifth type, *special*, is reserved for more complex cases. The four basic flow types serve as four different sound rules that will later be refined for precision.

The four rules are: *1. In-place:* Information can only flow from bit i of the source to bit i of the destination (as shown in Figure 3.8). *2. Up:* Information can only flow from bit

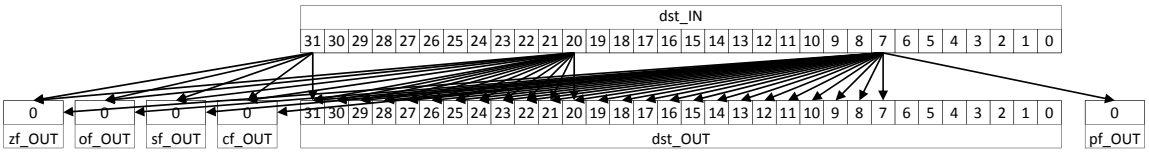


Fig. 3.9.: Information flow of bits 7, 20 and 31 of `dst` in `sbb` instruction

i of the source to bits j of the destination where $j \geq i$. Figure 3.9 depicts this behavior, showing the combination of the information flow graphs for bits 7, 20, and 31. It is evident from the figure that information only flows from bit 7 of the source operand to bit 7 and higher of the destination. The same applies to bits 20 and 31, where bit 31 of the source only flows to bit 31 of the destination. *3. Down:* Dual to up, information can only flow from bit i of the source to bits j of the destination where $j \leq i$. *4. All-around:* Information can flow from bit i of the source to any bit of the destination.

There are times when a single instruction requires multiple tainting rules. Table 3.2 is not an exhaustive list. The divide instructions are good examples of this. In the divide operation, `edx:eax` is divided by `rm`, the quotient placed into `eax` and remainder into `edx`. Intuitively, division is similar to shift right and thus the flow type for `edx:eax` to `eax` should be *down*. On the other hand, the flow type for `edx:eax` to `edx` is *all-around* since nothing definitive can be said about the relationship between the divisor and the remainder without concrete value analysis.

Special Instructions: Implicit information flows (those due to control dependencies) are a known source of imprecision in taint analysis; they can even occur within a single instruction, making tainting rule definition difficult. The `bsf`, `bsr`, and `cmpxchg` instructions exhibit such behavior and are thus considered special.

```
1. cmpxchg (rm32, r32) {
2.   if (eax == rm32 ) then
3.     rm32 = r32;
4.   else eax = rm32;
5. }
```

Fig. 3.10.: Pseudocode for `cmpxchg` (flags are omitted)

The `cmpxchg` (Figure 3.10) x86 instruction illustrates such a potential pitfall. Applying Definition 3.5.1 to the instruction shows that there is no information flow from `eax` to `eax` because the output value of `eax` is fully dependent on the input value of `rm32`. On the other hand, if information flow was analyzed line-by-line using the technique proposed by Ferrante et al. [55] (both Dytan [43] and DTA++ [68] use this technique), `eax` will be tainted if `eax` was tainted before the instruction. This is because `eax` was unchanged in the equals branch (line 3) and thus retains its taint. The case for simple control flow dependencies is even worse. Since `eax` is used in the comparison on line 2 and also as an l-value on line 4, it will remain tainted in the not-equals branch. The false positive arises from the fact that the above mentioned techniques analyzed the information flows line-by-line - this is what IR level tainting does -, thus knowledge of the logic in the other branch is not taken into consideration. Overall, striking a balance between handling all possible special instructions and only handling a smaller subset of instructions that can be used to emulate the rest is a design decision. DECAF uses a compromise approach that leaves some special cases unhandled, but uses per-trace verification to minimize errors.

Table 3.3: Precise Rules and Verification Results: Length of operands verified (in bits).
✓Verified for all lengths. * Shift amount is untainted. ^{z} Non-zero operand for `bsf, bsr`.

		Precise				Precise	
Operation	**Sound**	**Z3**	**MONA**	**Operation**	**Sound**	**Z3**	**MONA**
add	256	16	✓	adc	256	16	✓
and	256	256	✓				
cmpEq	256	256					
or	256	2	✓				
rol	256	16*		rcl	256	16*	
ror	256	16*		rcr	256	16*	
sal/shl	256	16*					
sar	256	16*					
shr	256	16*					
sub	256	4	✓	sbb	256	4	✓
				bsf	32		16^z
				bsr	32		16^z

Constructing Precise Rules

The previous section focused on the construction of sound rules. Four basic rules were arrived at that are sound by construction. However, special cases such as `cmpxchg` motivate the need for formal verification of tainting rules. Tainting rule verification is accomplished in two steps: the operation and the tainting rules under test are formally specified, and then solvers are used to determine whether Equations 3.2 and 3.3 are satisfiable. The formal specification step is straightforward using the models from Stage 1 and will not be discussed further.

When verifying the sound rules from the previous section, it was found that, while all of the rules were sound (as expected), many of them were not precise. In order to construct precise rules, Memcheck's rules were examined, since it has many specially-defined rules. Many of Memcheck's rules were found to be precise. In total, six new precise rules were added for `adc`, `sbb`, `rcr`, `rcl`, `bsf`, and `bsr`, summarized in Table 3.4. SMT2 code for a 2-bit `and` verification example is provided in Appendix A to show a more detailed example of this verification process.

Table 3.4: New Precise Bit-level Taint Rules: `rcr` and `bsr` are similar to `rcl` and `bsf` respectively, and so omitted. The `bsf` rule is shown for a 16-bit value which must be non-zero, and the rule for `rcl` is precise only when the rotate amount is untainted. `x1`, `x2`, and `cf` (carry flag) are the operands while `t1`, `t2`, and `tcf` are the respective shadow taints.

Operation	Rule (C-like pseudocode)

adc	`x1_min = x1 & ~t1; x2_min = x2 & ~t2; cf_min = cf & ~tcf;`
	`x1_max = x1 | t1; x2_max = x2 | t2; cf_max = cf | tcf;`
	`t1 | t2 | ((x1_min + x2_min + cf_min) ^ (x1_max + x2_max + cf_max))`

sbb	`t1 | t2 | ((x1_min - (x2_min + cf_min)) ^ (x1_max - (x2_max + cf_max)))`
rcl	`pcast(v) { v == 0 ? 0 : -1 /* all ones */ }` `pcast(t2) | rcl(t1, x2, tcf)`

bsf	`xc = x1_max & ~((x1_min << 1) | -(x1_min << 1));`
	`((xc & 0x5555) && (xc & 0xaaaa) ? 1 : 0) |`
	`((xc & 0x3333) && (xc & 0xcccc) ? 2 : 0) |`
	`((xc & 0x0f0f) && (xc & 0xf0f0) ? 4 : 0) |`
	`((xc & 0x00ff) && (xc & 0xff00) ? 8 : 0);`

The verification results of all specially defined rules are summarized in Table 3.3. Memcheck rules are placed on the left and DECAF rules on the right side of the Memcheck rules if the rules are similar. There are four columns: the operation, Z3 result for soundness, the Z3 result for precision and finally, if the Z3 result was inconclusive (i.e., Z3 did not return a result after 24 hours of processing), the MONA [52] result of whether the rule is precise, and the corresponding rule that was verified. Note that MONA was chosen as a complementary decision procedure to Z3 since it deals gracefully with alternating quantifiers, which Z3 does not. On the other hand, MONA is less expressive, making it difficult to use MONA for all cases.

As the results show, all of the special rules defined in Memcheck are sound for operands up to 256 bits[2] Additionally, the special rules for `and` and `cmpEq` are also precise up to 256 bits. In most cases, Z3 times out for operands beyond 16 bits in length. The size of the state space to explore is the most likely culprit for these time outs, since

[2]We chose 256 bits as the maximum length to test, since we are unaware of any architectures with operands greater than 256 bits.

smaller bit lengths returned quickly. MONA was able to verify precision of the `add`, `adc`, `or`, `sub`, and `sbb` rules.

All of the shift rules were shown to be imprecise. This is because the shift amount can be tainted, which causes all bits of the output to be marked as tainted. Subsequently, when the shift amount was asserted to be not tainted and the rules were re-verified, they were shown to be precise for up to 16 bit operands using Z3.

3.6 Evaluation

The performance overhead of DECAF has been evaluated under benchmarked guest performance under different feature configurations (such as VMI or tainting functionality enabled), and the results are presented in Section 3.6.1. The correctness of DECAF's tainting implemention was verified using *per-trace verification* in Section 3.6.2. DECAF's analysis capabilities are evaluated using three plugins: API Tracer (Section 3.6.3), Keylogger Detector (Section 3.6.4), and Instruction Tracer (Section 3.6.5). The source code for these plugins are available for download from DECAF's project page[10].

The hardware used for all evaluations is a 32-core 2.0GHz Intel Xeon ES-2650 CPU server with 128 GB of RAM. The server uses Ubuntu 12.04 Linux (3.2.0 kernel) as its OS. DECAF was executed on this server using an ARM Debian 6.0 Linux (2.6.32 kernel) VM image and three x86 guest VM images: Windows 7, Windows XP SP3 and Ubuntu 12.04 Linux (3.2.0 kernel). 4 GB of RAM was allocated to each of the x86 VMs, and 128 MB of RAM was allocated to the ARM VM. The priority of DECAF was `nice`'d to -20 to

minimize the performance impact of other processes executing on the benchmark hardware.

3.6.1 SPEC CPU2006 Benchmarks

DECAF's instrumentation impact on guest performance was measured using the CINT2006 integer component of the SPEC CPU2006 benchmark suite.[3] The CINT2006 tests were chosen because the tainting instrumentation is applied to the TCG instructions, which all implement RISC-like integer operations. Floating point operations are implemented as a set of guest architecture-specific helper functions. Performance of ARM VMs under DECAF cannot be measured using the benchmark suite due to the memory requirements of the tests. The majority of the tests exceed the RAM allocated to the VM[4] and will measure the performance of the memory paging to disk, rather than the instrumented operations of interest. While a direct comparison of TEMU and DECAF performance using these benchmarks would be informative, this is infeasible because TEMU is too slow to correctly execute the tests. When executing the benchmark suite under TEMU, the first benchmark test of the suite (400.perlbench) was allowed to run for over a day before its execution was terminated after failing to complete even a single iteration of the benchmark test.

Baseline DECAF, without any instrumentation enabled, experiences an average of 15.20% overhead over the execution performance of a similarly-configured QEMU. DECAF updates `EIP` (x86) and `R15` (ARM) after every guest instruction to ensure

[3] 462.libquantum was omitted from the test suite due to Visual Studio's Visual C++ not supporting some C++ features used by the test.
[4] The "versatilepb" platform QEMU uses to emulate ARM VMs has a 256MB RAM limitation.

accurate analysis, while QEMU updates these registers at the end of each TB. DECAF must also maintain its plugin infrastructure by continually watching for the registration of new plugin callbacks.

The VMI overhead measurements in Figure 3.11 show the difference in performance between running DECAF in a baseline configuration with all features disabled and a configuration with only VMI enabled. Average overhead is 12.07% for Windows 7 and 14.48% for Linux. The negative overhead result for the Linux 400.perlbench test can be attributed to the short execution time of the test and the general variability in execution times within an emulated VM environment. The result of 429.mcf has considerably higher VMI overhead than the other tests with 54.36% for Windows 7 and 55.23% for Linux. This test incurs almost twice as many TLB misses as the next closest test (471.omnetpp). VMI callbacks are triggered when TLB misses occur, explaining the larger amount of observed overhead.

Table 3.5: Execution Overhead for DECAF and DECAF with VMI on different architecture/OSs without tainting.

Setup	**XUbuntu**	**WinXP SP 3**	**Debian Squeeze (ARM)**
DECAF with VMI	3m 25.9s	1m 4.36s	2m 50.16s
QEMU 1.0.1	2m 45.85s	0m 52.79s	2m 36.52s
DECAF + VMI Overhead %	24.14	21.91	8.72

Furthermore, Table 3.5 presents guest boot time overhead under DECAF, and Table 3.6 presents the source code distribution between architecture dependent and independent components in QEMU to add DECAF functionality. DECAF and VMI

Table 3.6: Code breakdown of DECAF, VMI, and various plugins. The code introduced by DECAF in addition to the code of QEMU, which by itself has over 500K LOC.

	OS/Arch independent (LOC)	**OS Specific (LOC)**	**Total (LOC)**
DECAF	18470	1350	19820
Insn Tracer	3770	90	3860
API Tracer	840	880	1720
Key Logger	120	0	120

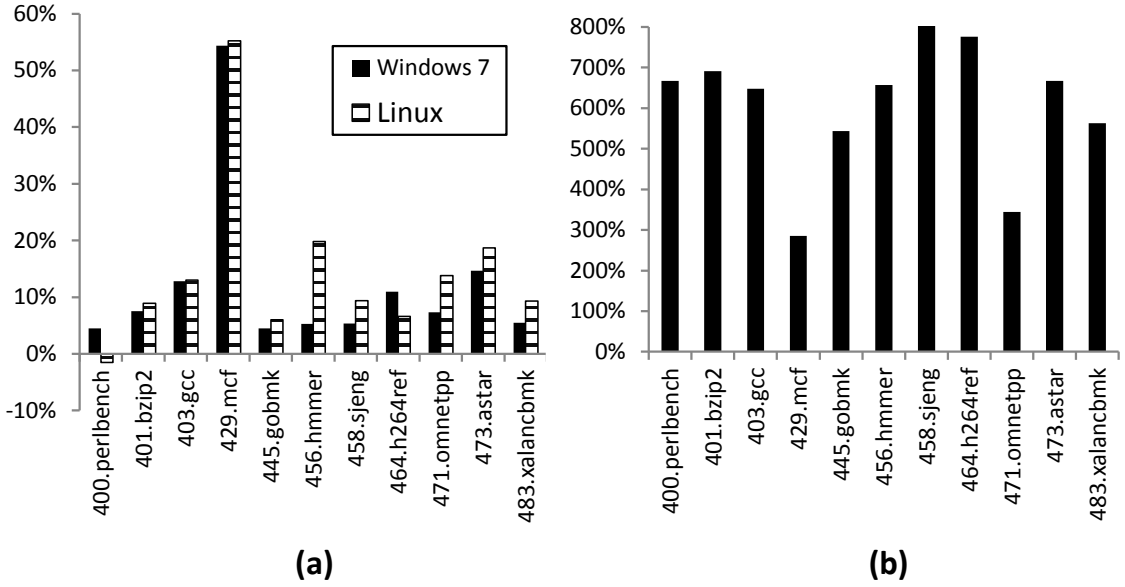


Fig. 3.11.: CINT2006 benchmarks that measure overhead for VMI (a) and inline taint propagation (b).

impose a combined overhead under 25% on x86 and 8.72% on ARM. Also, from Table 3.6 we can see that most of the plugin code is architecture independent. API Tracer includes OS-specific code to interpret some OS-specific data structures, but, the core part of API Tracer contains no OS-specific code.

The inline taint propagation measurements in Figure 3.11 show the difference between running DECAF in a baseline configuration with all features disabled and with inline tainting enabled for the Windows 7 VM. The average overhead is 605.07%, ranging from

285.32% (429.mcf) to 815.77% (458.sjeng). Taint propagation overhead is directly related to the number of TCG instructions being executed, so it is highest for CPU-bound tests. Because DECAF's inline tainting executes multiple taint propagation TCG instructions for each TCG instruction that executes, an average slowdown of six-times is justified.

The internal QEMU profiler (the "`info jit`" QEMU monitor command) was used to obtain translation block (TB) statistics. For the QEMU baseline, the average TB contains 45.3 IR instructions with the largest TB having 464 instructions. An average of 29.3 temporary registers were used by the TBs, with a maximum of 68 temporary registers used. On the other hand, DECAF TBs have an average of 86.7 IR instructions with the largest TB containing 520 instructions. On average, 74.0 registers were used with a maximum of 358 temporary registers.

3.6.2 Per-Trace Verification of DECAF's Tainting

Per-trace verification was used to verify the correctness of a taint analysis system's implementation. A high level overview of the process is depicted in Figure 3.12.

In per-trace verification, the taint analysis system under test (e.g., DECAF) executes a program and generates a tainted execution trace. The trace is a log of all instructions executed, along with additional metadata. Each log entry contains the instruction executed, the input operand values, the output operand values, and the corresponding taint label assignments. (A sample log entry is shown in Figure 3.13)

For each entry in the instruction trace, an *oracle* is used to determine whether the resulting taint matches the noninterference model. The oracle consists of three main

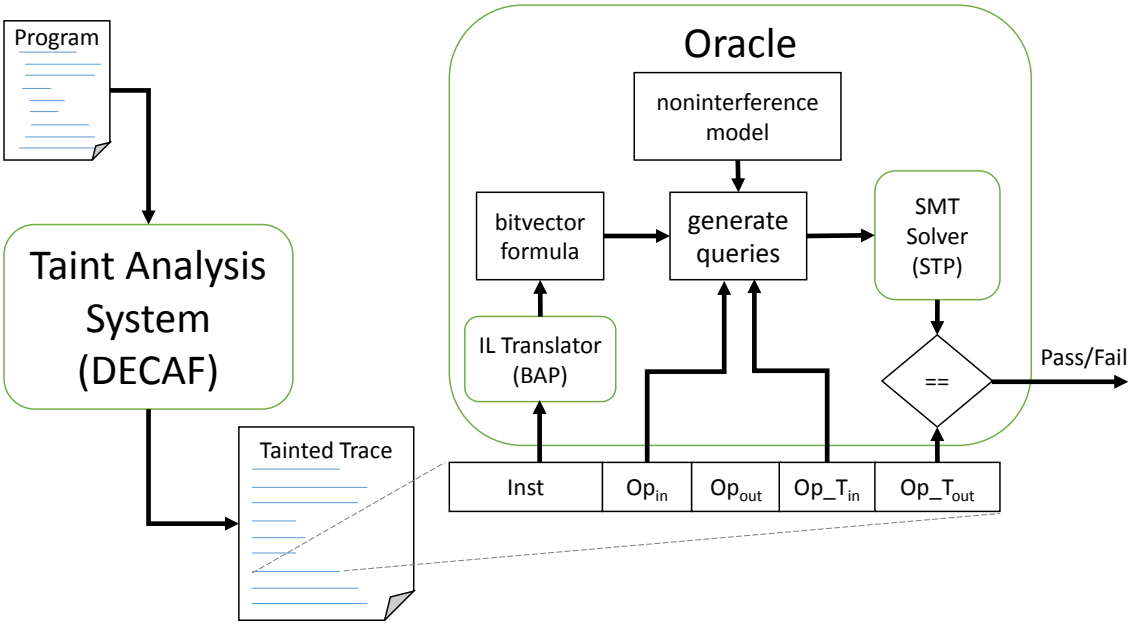


Fig. 3.12.: Per-Trace Verification Overview

components. An *IL translator* is used to translate the operation (`and` in the example) into a bitvector formula. A *query generator* then takes the translated formula, the concrete values from the trace entry, and the input taint assignments and generates a query to determine the correct output taint labels. This query is subsequently sent to an SMT solver and the results compared to the output taint as recorded in the trace entry. If they agree, then the implementation is correct for this particular operation and machine state. If they disagree, either the rule is imprecise or there is an implementation bug.

Per-trace verification has a number of advantages. First, the traces can be generated and verified independently and thus processed in parallel. Second, the problem of verifying traces one instruction instance at a time is more tractable: using concrete values reduces the state space to explore. Third, the oracle can also be used as a taint analysis system itself. For example, a taint analysis system might use sound but imprecise tainting

rules to improve runtime performance and then use the oracle to reprocess the trace offline and remove any false positives.

The major limitation of per-trace verification is coverage. Per-trace verification will not be complete unless the traces used to verify the system cover all possible system states (i.e., all possible combinations of operations, operand values and taint values). To maximize coverage, a collection of over 600,000 test programs from the PokeEMU [72] project was used for the evaluation. These test programs were automatically generated by exploring all of the different instruction decode and execution paths of the Bochs x86 emulator [3]. They provide full path coverage of more than 800 protected-mode x86 instructions, and so the per-trace verification results for DECAF inherit this same extensive coverage.

In order to verify DECAF, an instruction tracer plugin was implemented to generate the tainted trace. The oracle was implemented using BAP as the IL translator and STP [57] as the SMT solver (Z3 works as well). Specifically the bitvector formula and queries are expressed in the BAP IL, allowing the use BAP's existing interface to STP (or Z3).

The correctness of DECAF's rule implementations was verified using the 600,000+ PokeEMU test cases. Each test case was executed using DECAF, and all instructions executed were logged into a tainted trace, one per test case. Due to the sheer number of test cases, there was not exhaustive testing that attempted to try all possible taint assignments to the program state. Instead, random taint values were assigned to the program state at the beginning of execution and propagated through the program.

Each trace was then passed through the oracle to determine whether there were any differences in the output taints. If the verification failed, the offending instruction was

```
/* ebx = eax & ebx */
Inst: and %eax, %ebx
Inputs:  eax = 0x84be2329, ebx = 0xaed66ce1
Outputs: ebx = 0x84962021
Input Taints:  eax_t = 0x7369C667, ebx_t = 0xec4aff51
Output Taints: ebx_t = 0xe44ae761
/* Expected Output Taints: ebx_t = 0xe64ae761 */
```

Fig. 3.13.: Trace entry for `and` bug

manually reviewed in an attempt to track down the source of the failure. If a bug was found, it was patched in DECAF and then the offending test case was re-run to ensure that the bug was fixed. In total, it took over 16 days to complete the verification task by running 80 verification instances in parallel. Each trace took approximately 3 minutes to complete. This does not include the extra time needed to address the few bugs that were discovered.

This method of verification uncovered two incorrectly implemented tainting rules in DECAF (`and` and `add`). Both errors were due to the same implementation mistake. A text version of the offending trace entry is shown in Figure 3.13. The figure shows the concrete values of the operands, as well as the input and output taints. According to DECAF, the output taint was 0xe44ae761, which failed verification because the expected taint was 0xe64ae761. Notice that bit 25 is 0, but should actually be 1.

As it turns out, this error was due to the way the extra TCG IR to propagate taint was inserted. In the code for adding the propagation IRs for `and`, the propagation IRs were incorrectly placed *after* the original `and` operation. As a result, instead of using the concrete value of 0xaed66ce1 for `ebx` to calculate the taint, the *result* of `ebx` (0x84962021) was used. In fact, this bug was pervasive throughout DECAF's implementation, and it was not understood it until it was discovered that the `add`

implementation had the same problem. In general, this bug only surfaces if the destination operand is also a source operand, and the value written to the destination happens to affect the final taint calculation, meaning it depends on both the concrete values as well as the taint assignments. This discovery led to the insertion of all IRs that implement taint propagation instrumentation for an IR immediately *prior* to the IR that they instrument.

3.6.3 API Tracer

The API Tracer plugin leverages the VMI and function hooking features of DECAF to capture API-level traces of the user- and kernel-mode execution of a program.

At its core, API Tracer is a minimal and stand-alone cross-platform component, comprised of 340 lines of C code, that retrieves function-level execution traces of programs on any platform/OS supported by DECAF. Furthermore, it contains a custom configuration parser, comprised of 500 lines of C code, and a Windows-specific extension component, comprised of 880 lines of C code, to decipher the higher-level OS-specific semantics. For example, in Windows the `kernel32.dll::CreateProcess()` API call contains newly created process information and the creation flag parameters required to extend analysis into child processes. The OS-specific component interprets such information and acts accordingly.

Unlike static analysis tools that are unable to analyze dynamically generated code, and user-space dynamic analysis tools (such as Pin [70]) that are unable to analyze activity in the kernel, API Tracer keep track of any kernel modules loaded by a user program and traces such modules automatically. It also monitors the memory allocation and

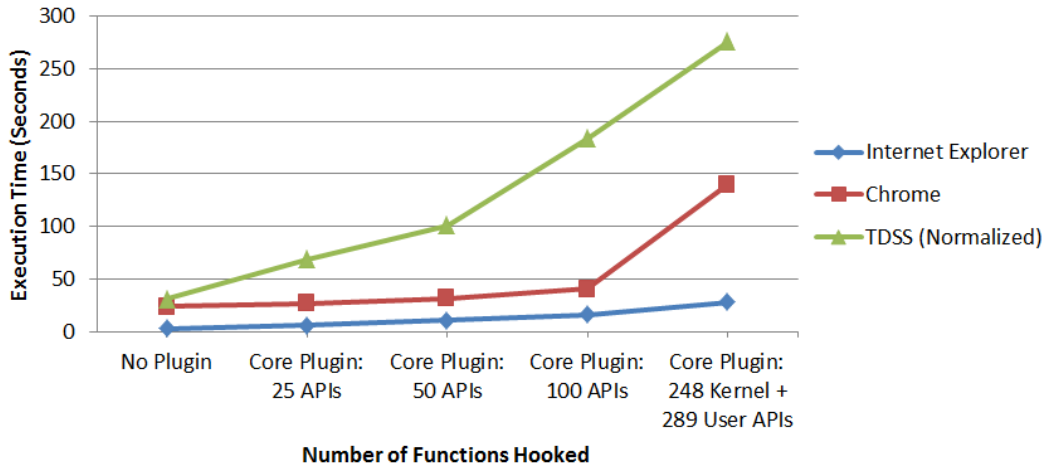


Fig. 3.14.: Evaluation of API Tracer plugin.

deallocation of a program to identify and trace any unpacked/dynamically generated code, thereby providing rich cross-platform and system-wide analysis capabilities.

Figure 3.14 shows the overhead introduced by API Tracer[5] on the execution of a Windows XP SP3 guest as the plugin scales with the number of functions in the plugin's configuration file [6]. DECAF selectively instruments only the TCG TBs that correspond to the hooked functions, thereby significantly improving performance. An un-optimized implementation would instrument *all* TBs and filter the ones that correspond to hooks - similar to what TEMU [82] does. As a comparison, Internet Explorer loads the webpage in 22.6 seconds and 217.79 seconds with selective optimization on and off, respectively.

For the sake of evaluation, two popular web browser clients for Windows (IE and Chrome) and a notorious bot (TDSS [61], which inserts a kernel module to hide itself in the kernel) were evaluated using the plugin. API Tracer is not only able to trace the inserted kernel module, but is also able to extract the unpacked code in memory for further

[5]TDSS values are normalized because it stalls for 360 seconds to evade analysis. Loading of the web page `http://www.gnu.org` was used as a reference to measure execution times of IE and Chrome.
[6]Configuration file consists of all functions that must be captured, along with their parameter list/types, return types, and calling conventions.

analysis. The Chrome browser uses a *multiple-processes architecture* and keeps tabs, extensions, web apps, and plug-in processes independent from each other and spawns new processes when required. API Tracer is able to automatically and successfully trace the parent Chrome process and any spawned child processes.

3.6.4 Keylogger Detector

The Keylogger Detector plugin is an extended version of the sample plugin shown in Figure 3.2. Leveraging the VMI, tainting, and event-driven programing features of DECAF, this plugin is capable of identifying keyloggers and analyzing their stealthy behaviors. The core of Keylogger Detector is cross-platform and OS-independent, comprised of only 120 lines of C code.

By sending tainted keystrokes into the guest system and observing whether any untrusted code modules access the tainted data, keylogging behavior can be detected. This is similar to the functionality provided by Panorama [90]. The sample plugin can introduce tainted keystrokes into the guest system and identify which modules read the tainted keystroke by registering `DECAF_READ_TAINTMEM_CB` and `DECAF_KEYSTROKE_CB` callback events. To capture the detailed stealthy behaviors, Keylogger Detector implements a shadow call stack by registering a `DECAF_BLOCK_END` callback. Whenever the callback is triggered, the current instruction is checked. If it is a `call` instruction, function information is retrieved using VMI and the current program counter is pushed onto the shadow call stack. If it is a `ret` instruction and pairs with the entry on the top of the shadow call stack, it is popped from the stack. When the

DECAF_READ_TAINTMEM_CB callback is invoked, information about which process, module, and function read the tainted keystroke data from the shadow call stack is retrieved.

Two experiments were run to evaluate the Keylogger Detector. First, a set of malware samples, known to have key-logging functionality, was collected. This sample set has 117 malware samples in total, spanning 29 malware families. They were tested on a Windows XP SP3 guest by sending keystrokes to the notepad.exe application and observing whether any tainted keystrokes were accessed by the tested sample. Keylogger Detector successfully detected the keylogging behaviors in all of these samples. Table 3.7 is the trace of Trojan.Win32KeyLogger. It shows which module of the process read the tainted keystroke using which function. The trace shows that the tainted keystroke entered the system and was fetched by the untrusted code of MPK.exe, which clearly depicts a keylogging activity. Furthermore, the trace shows which functions were used to steal keystrokes. Such information is very valuable when performing malware analysis.

Table 3.7: Trojan.Win32.KeyLogger Trace.

PROCESS	MODULE	FUNCTION
<KERNEL>	i8042prt.sys	hall.dll:READ_PORT_UCHAR
<KERNEL>	win32k.sys	ntoskrnl.exe:PsGetProcessWin32Process
<KERNEL>	win32k.sys	hal.dll:HalEndSystemInterrupt
...	...	...
notepad.exe	Mpk.dll	ntoskrnl.exe:ProbeForWrite
notepad.exe	Mpk.dll	user32.dll:SendMessageA
...	...	...
MPK.exe	user32.dll	ntoskrnl.exe:ProbeForWrite
...	...	...
MPK.exe	MPK.exe	kernel32.dll:InterlockedIncrement
...	...	...
MPK.exe	MPK.exe	hal.dll:HalEndSystemInterrupt
MPK.exe	MPK.exe	ntdll.dll:wcscpy
MPK.exe	user32.dll	ntoskrnl.exe:ProbeForWrite
...	...	...

Second, an analogous Keylogger Detector plugin was created for the TEMU tool and tested some tainted shell commands in both Windows XP Service Pack 3 and Linux 2.6.20 guests. Tainted keystrokes were sent as commands to the shell and each of the tainted commands was observed as it was processed in the operating system. For each command, after it finishes execution, the number of tainted bytes in main memory and the occurrences of the `EIP` register becoming tainted were recorded. Note that, by design, the number of bytes tainted should be more correlated with the length of the commands than the actual commands used.

Table 3.8: Comparing DECAF with TEMU on tainted shell commands.
"n / m" indicates that "n" bytes are tainted, and "m" tainted EIPs are observed.

Windows		
Command	**DECAF**	**TEMU**
`dir`	207 / 0	639 / 0
`cd`	146 / 0	616 / 0
`cipher c:`	929 / 0	3617 / 0
`echo hello`	660 / 0	3808 / 0
`find "jone" a.txt`	967 / 0	5684 / 0
`findstr /s /i jone ./*`	945 / 0	1333 / 0
Linux		
Command	**DECAF**	**TEMU**
`ls`	350 / 3	34923 / 0
`cd`	306 / 3	301 / 0
`cat ./readme`	545 / 31	26619 / 0
`echo hello`	744 / 9	704 / 0
`ln -s a.txt nbench`	1122 / 35	24707 / 0
`mkdir test`	551 / 9	23766 / 0

The results for both Windows and Linux are listed in Table 3.8. The results for Windows show that the number of tainted bytes in DECAF is much smaller than the number in TEMU, demonstrating the benefit of DECAF's tainting implementation being more precise (less overtainting). No instances of a tainted `EIP` register were observed in either system. The Linux results are somewhat different. Although the number of tainted

bytes marked by DECAF was generally much smaller than that of TEMU, DECAF reported tainted `EIP` registers for all of the commands, whereas TEMU reported none.

These results look contradictory to the claim that DECAF should be more precise, so the taint propagation logs generated by DECAF and TEMU were manually examined. Not every instance of a tainted `EIP` register (a total of 93) was examined, but it was confirmed that every examined sample was indeed correct. A common case is that a tainted character (from the tainted keystroke entered) was being used as an index into a function pointer table to call a function. The same instruction sequences were discovered in the trace generated by TEMU. This means that TEMU has an under-tainting problem, even though its tainting rules are generally sound.

3.6.5 Instruction Tracer

The Instruction Tracer plugin records a TCG IR instruction-level trace with concrete and taint values for a specific guest user-space process or kernel code region. Similar to the other two plugins, Instruction Tracer is largely platform-neutral, capable of collecting execution traces for programs in x86 and ARM, Linux and Windows. Moreover, it is also easier to perform formal verification on the TCG trace, due to its RISC-like instruction semantics, than on the original code of the guest. For example, it has been demonstrated as feasible to convert the TCG trace into LLVM IR and then perform symbolic execution on the trace [39]. Instruction Tracer is implemented in 3860 lines of C code, though this includes the code for both the plugin and the parser for the log file that the plugin generates.

To demonstrate the practical effectiveness of this plugin, Instruction Tracer was used to detect a buffer overflow at runtime. The sample code in Figure 3.15 was compiled and executed inside of x86 and ARM Linux guest VMs running under DECAF with Instruction Tracer loaded.

```
1. int func1(char *input) {
2.   char buffer[4];
3.   strcpy(buffer, input);
4. }

5. void main(void) {
6.   char buffer[16];
7.   scanf("%s", buffer);
8.   func1(buffer);
9. }
```

Fig. 3.15.: A simple buffer overflow example.

The code contains a simple buffer overflow vulnerability. If more than three characters are entered by the user, `buffer` in `func1()` will overflow and begin corrupting data stored on the stack. To capture the corruption, characters are entered into the program via tainted keypresses until the return address is modified by the overflow. Under the ARM environment, Instruction Tracer identified the buffer overflow when `R15` (PC) became tainted after entering five characters. `R14` (Link Register) was also monitored for taint, but it never became tainted during the test. Figure 3.16 shows the log output at the point where `R15` first becomes tainted. Tainted character data is fetched from stack memory, masked to ensure that the value is properly aligned, and then stored in `R15`.

```
qemu_ld32 tmp61[00000000],tmp50[00000000],$0x0
 --> TAINT HAS BEEN READ FROM MEMORY:
     Address: 0x07837e5c (4 bytes)
     Taint: [ffffffff]
movi_i32  tmp62[00000000],$0xfffffffe
and_i32   pc[00000000],tmp61[00000000],tmp62[ffffffff]
 --> TAINT NOW PRESENT IN PROGRAM COUNTER (R15)
```

Fig. 3.16.: Buffer overflow detection on ARM.

Under the x86 environment, the TCG global variable for the `EIP` register can't be directly passed to an opcode as an argument. `EIP` is modified by writing to host memory via the `st_i32` opcode. Watching for tainted writes to `EIP`'s offset (`0x4C`) in the `CPUState` data structure identifies that the buffer overflow occurs. Figure 3.17 shows the log output at the point where `EIP` first becomes tainted. Tainted character data is fetched from memory located at the address in `ESP`, the stack size is reduced by four bytes, and the tainted data is then placed into `EIP`'s offset in the `CPUState` data structure.

```
mov_i32   tmp2[00000000],esp[00000000]
qemu_ld32 tmp0[00000000],tmp2[00000000],$0x0
 --> TAINT HAS BEEN READ FROM MEMORY:
     Address: 0x0bfffff30 (4 bytes)
     Taint: [ffffffff]
movi_i32  tmp15[00000000],$0x4
add_i32   tmp4[00000000],esp[00000000],tmp15[00000000]
mov_i32   esp[00000000],tmp4[00000000]
st_i32    tmp0[ffffffff],env,$0x4c
 --> TAINT NOW PRESENT IN EIP
```

Fig. 3.17.: Buffer overflow detection on x86.

Instruction Tracer's performance was also compared against that of the TEMU's Tracecap plugin. Tracecap generates a trace of the guest's instructions as they execute to facilitate analyses similar to that of the buffer overflow analysis performed with Instruction Tracer. DECAF and TEMU were used to emulate the same Windows XP VM and trace the execution of an instance of the DOS `sort` application. For both plugins, tainting was disabled. A text file 5.4 MB in size[7] was selected to be sorted, and both plugins were configured to log their execution traces of the application directly to `/dev/null`. The `sort` execution completed in 39m 57.33s with Tracecap under

[7]The text file selected for this test was "The Complete Works of William Shakespear", which was downloaded from Project Gutenberg (http://www.gutenberg.org).

TEMU, but in only 2m 5.23s with Instruction Tracer under DECAF (almost 20 times faster). The same sort with a stock QEMU completed in 5.89s.

3.7 Limitations of DECAF

While the novel design for DECAF is suitable for a wide range of analysis tasks, no single analysis platform will meet the requirements for every analysis task. Therefore, the key limitations of the DECAF platform are enumerated here to both better understand the reasoning behind those limitations and document them as tasks to be explored in future work.

1. DECAF is based upon QEMU v1.0.1. This was the most recent version of QEMU at the time when DECAF's initial development began in 2012. A number of improvements, such as additional features and bug fixes, have been added to the QEMU codebase since then. DECAF does not contain these additional improvements as it has been maintained as an independent fork of the QEMU codebase. To receive the benefit of these additional items, the DECAF-specific code must be ported to a newer QEMU codebase.

2. DECAF is currently only usable with 32-bit guest environments. This is an implementation limitation, rather than an insurmountable technical limitation, as support for 64-bit guests was never completed. The SoftMMU TLB lookup paths have only been instrumented for taint propagation along the 32-bit cases (Section 3.2). Implementation of the 64-bit cases is partially completed. Likewise, the insertion of additional taint-propagation IRs within each TB is complete for

32-bit IRs (also Section 3.2), but support for 64-bit guests is partially completed. Thus, adding 64-bit support is largely a matter of updating the implementation of the SoftMMU TLB lookup code, shadow memory data structures, and 64-bit taint-propagation IRs that currently exist in the DECAF codebase.

3. DECAF's VMI reconstructs the semantics of the guest environment (as described in Section 3.3.2) and provides an interface for plugins to easily utilize this information during analysis. This abstracts away many of the guest-specific details to simplify analysis tasks. However, DECAF's VMI is designed to examine the known internal data structures of either a Linux or Windows guest kernel. There is currently no VMI support for other OSes, such as FreeBSD, QNX, or SunOS. Android inherits its VMI support due to its use of the Linux kernel. Adding VMI support for additional OSes is largely an implementation limitation, though an OS whose design deviates from that of the monolithic kernels of the supported OSes may require significant effort to support.

4. DECAF's taint propagation is designed to shadow the activity of the IR of the guest's native instructions. However, when complex guest instructions cannot be easily translated into TCG IR, high-level helper functions are called via a TCG `call` IR (Section 3.2). DECAF has taint propagation rules for many of these special cases (for example, the x86 `cmpxchg` instruction). But, floating point, MMX, and SSE instructions do not have taint propagation rules (Section 3.4.1). By using a sequence of these instructions, malicious guest software could potentially remove taint from data and cause undertainting. Both the discovery of sound and

precise tainting rules and their implementation for these instructions is left as a future work.

4. VIRTUAL DEVICE FUZZ TESTING

As cloud computing becomes more prevalent, the usage of virtualized guest systems for rapid and scalable deployment of computing resources is increasing. Major cloud service providers, such as Amazon Web Services (AWS), Microsoft Azure, and IBM SoftLayer, continue to grow as demand for cloud computing resources increases. Amazon, the current market leader in cloud computing, reported that AWS's net sales exceeded *7.88 billion USD* in 2015 [2], which demonstrates a strong market need for virtualization technology.

This popularity has led to an increased interest in mitigating attacks that target hypervisors from within the virtualized guest environments that they host. Unfortunately, hypervisors are complex pieces of software that are difficult to test under every possible set of guest runtime conditions. *Virtual hardware devices* used by guests, which are emulated in software (rather than directly map to physical devices on the host system), are particularly complex and a source of numerous bugs [4, 5, 6, 7]. This has leading to the ongoing discovery of vulnerabilities that exploit these virtual devices to attack or spy on the host environment.

Because virtual devices are so closely associated with the hypervisor, if not integrated directly into it, they execute at a higher level of privilege than any code executing within the guest environment. They are not part of the guest environment, per se, but they are privileged subsystems that the guest environment directly interacts with. Because of this, a malicious or misbehaving guest may attempt to use these virtual devices in an

unpredictable manner. Under no circumstances should activity originating from within the guest be able to attack and compromise the hypervisor, so effectively identifying vulnerabilities in these virtual devices is a difficult, but valuable, problem to consider. However, these virtual devices are written by a number of different authors, and the most complex virtual devices are implemented using thousands of lines of code. Therefore, it is desirable to discover an effective and efficient method to test these devices in a scalable and automated fashion without requiring expert knowledge of each virtual device's state machine and other internal details.

Such issues have led to a strong interest in effectively testing the code that implements these virtual devices [19, 44] to discover bugs or other behaviors that may lead to vulnerabilities. However, this is a non-trivial task as virtual devices are often tightly coupled to the hypervisor codebase and may need to pass through a number of device initialization states (i.e. BIOS and guest kernel initialization of the device) before representing the device's state within a running guest system.

The technique of fuzzing [73] has long been used to explore the execution states of programs in an unexpected manner to discover bugs. This form of testing, when applied to untrusted inputs provided to the program, is able to discover vulnerabilities such as buffer overflows and infinite loops. While simple in concept, fuzzing is capable of executing large numbers of dynamically-generated test cases in an automated fashion, making it a powerful way to explore and test programs [59, 83, 84].

The benefits of fuzzing are very appealing when attempting to test virtual devices, but it has its limitations. Fuzzing attempts to visit *all* states of a program to discover bugs, but the number of branches belonging to a particular virtual device form a very small fraction

of the total branches within the entire hypervisor codebase. Even worse, some states of interest may only be reachable after first visiting an arbitrary pattern of uninteresting states, so random program inputs have no guarantee of actually reaching the interesting states of a virtual device. Therefore, it is important to generate fuzzing test cases that not only trigger bugs, but that are able to reach the specific program states where such bugs may exist.

4.1 VDF Overview

This dissertation chapter presents Virtual Device Fuzzer (*VDF*), a novel fuzz testing framework that provides targeted fuzz testing of interesting subsystems within complex programs: the portions of a hypervisor's codebase that implement virtual devices. VDF enables the testing of virtual devices within the context of the hypervisor as the hypervisor executes. It utilizes *record and replay* of virtual device memory-mapped I/O (MMIO) activity to create fuzz testing seed inputs that are guaranteed to reach states of interest and initialize each virtual device to a known good state from which to begin each test. Providing proper seed test cases to the fuzzer is important for effective exploring the branches of a program [38, 79], as a good starting seed will focus the fuzzer's efforts in areas of interest within the program. VDF mutates these recorded seed inputs to generate and then replay fuzzed MMIO activity to exercise additional branches of interest within virtual devices.

As a proof of concept, VDF is evaluated by using it to test eighteen virtual devices implemented within QEMU. QEMU, and tools such as DECAF that extend QEMU,

provide a wide variety of virtual devices, such as network, audio, block, and character devices. Such virtual devices may completely emulate the internal state of a piece of hardware, provide a pass-through to a physical device on the host system, or provide some combination of the two. Whether QEMU completely emulates the guest CPU or uses another hypervisor, such as KVM [11] or Xen [27], to execute guest CPU instructions, hardware devices made available to the guest are software-based virtualized devices.

This chapter thus presents the following material:

1. The design of VDF, a record and replay fuzz testing framework for virtual devices, is presented. VDF uses selective instrumentation to perform fuzz testing of each virtualized device, by providing fuzzed MMIO activity to the virtual device, to target only the branches of execution which belong to the virtual device under analysis. This testing is performed within the context of a running hypervisor, but without the need for a guest environment to be booted, or even present. This allows for large numbers of tests to be executed quickly.

2. The VDF solution is motivated by using it to test eighteen QEMU virtual devices, executing over 2.28 billion test cases in several parallel VDF instances within a cloud environment. This testing discovered a total of 348 crashes and 666 hangs within six of the tested virtual devices. Bug reports for these crashes and hangs have been reported to the QEMU maintainers and its security team where applicable.

3. A method is described that reduces each discovered crash/hang test case to a minimal test case that is still capable of reproducing the same bug. Using this method, the average test case is reduced to only 18.57% of its original size, greatly

simplifying the analysis of discovered bugs and discovering duplicate test cases that reproduce the same bug. This method also automatically generates source code suitable for reproducing the activity of each test case to aid in the analysis of each discovered bug.

4. The discovered virtual device bugs are analyzed and organized into four categories: excess host resource usage, invalid data transfers, debugging asserts, and multithreaded race conditions.

4.2 Background

The functionality of complex programs is implemented as a collection of individual *subsystems*. Each such subsystem implements some portion of the functionality of the program and maintains the current state of that portion as it relates to the entire program. For example, substates include values held in variables that describe attributes of that subsystem, encapsulating these values within the scope of the subsystem.

Typically, an interesting subsystem provides an API for accessing that subsystem's functionality. This API could be as simple as a single function that acts as a gateway to the subsystem, though more complex APIs can expose dozens of functions that access the features the subsystem provides. Programs with well-defined subsystems can be unit tested by testing each subsystem via its entry point (i.e. API functions that request that the subsystem perform some action). If a subsystem's behavior is based solely upon the interface function calls that exercise the subsystem, then calling those functions with different arguments will exercise different sections of the subsystem. If a subsystem's

functionality cannot be triggered via these function calls, then that functionality will never be exercised and that code within the subsystem is unreachable.

Within QEMU, virtual devices register callback functions with QEMU's virtual memory management unit (MMU). These callback functions expose virtual device functionality to the guest environment, and they are called when specific memory addresses within the guest memory space are read or written. QEMU uses this mechanism to implement memory-mapped I/O (MMIO), mimicking the MMIO mechanism of physical hardware.

The following attack model describes how malicious guest activity might attempt to attack these virtual devices:

1. The virtual device is correctly instantiated by the hypervisor. The details of the virtual device's hardware have been correctly specified in the hypervisor's configuration (if necessary) and the hypervisor has created or claimed the device and made it available to the guest environment.

2. The virtual device is correctly initialized via the guest's BIOS and OS kernel and brought to a stable state during the guest boot process. The virtual device has been assigned resources via the PCI host controller (if necessary), and any needed kernel device drivers have been loaded and initialized.

3. After the guest has booted, control of the system is given to the attacker. The attacker then acquires privileged access within the guest and attempts to attack the virtual devices via memory reads and writes to the MMIO address(es) belonging to these virtual devices.

Unfortunately, it is non-trivial to perform large-scale testing of virtual devices in a manner analogous to the attacks described by this model. The MMIO read/write activity must originate from within the guest environment, requiring the guest to completely boot and initialize prior to performing a test from within the guest[1]. Because any read or write to a virtual device control register may change the internal state of the device, the device must be returned to a known good "just initialized" state prior to the start of each test. While utilizing virtual machine (VM) state snapshots to save and restore the state of the guest would ameliorate a great deal of this overhead, the time required to continually restore the state of the guest to a known good state makes this approach inefficient for large-scale testing.

While the code that implements each virtual device within QEMU is fairly well isolated from the remainder of the QEMU codebase, it may still have strong dependencies on the current state of the running QEMU hypervisor (e.g., guest memory layout for DMA, timers, and contextual state information). Virtual devices must be executed within the context of a running QEMU, making their extraction and isolated testing infeasible. There has been some prior work that explores stubbing out the functions of virtual devices and extracting them for symbolic analysis outside of QEMU [44], but this focuses on only a small number of virtual network devices as a proof of concept. When considering a virtual device from a symbolic execution standpoint, the flow of execution for the virtual device becomes disjointed because the code of the virtual device only represents a portion of the overall execution of QEMU. The flow of execution will appear to "jump" from

[1]QEMU does provide a *qtest* framework to perform arbitrary guest memory read/write activity without a guest present. We discuss qtest, and its limitations, in Section 4.3.

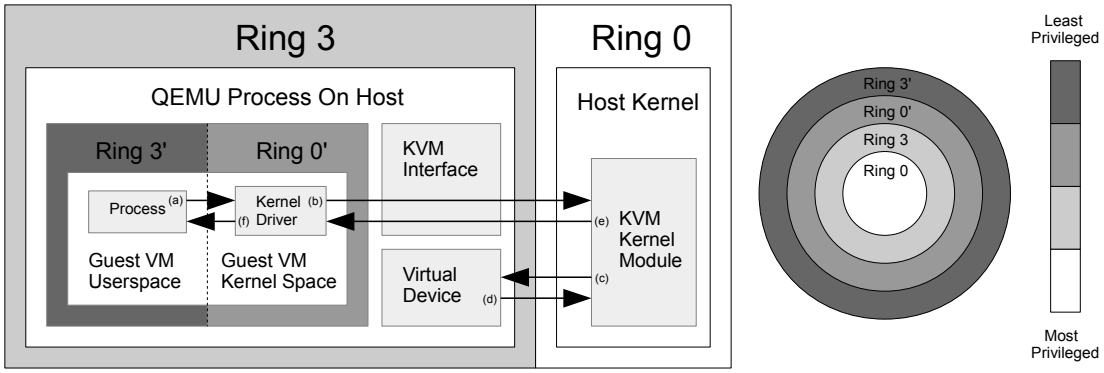


Fig. 4.1.: Device access request originating from inside of a QEMU/KVM guest. Note that the highest level of privilege in the guest (ring 0') is still lower than that of the QEMU process on the host (ring 3).

callback to callback, even though there is no direct correlation between the calling of one callback function and another.

4.2.1 Understanding guest access of virtual devices

The flow of activity for virtual device access from within a QEMU-hosted guest is shown in Figure 4.1. This figure shows a KVM-accelerated QEMU hypervisor configuration. The guest environment executes within QEMU, and the virtual devices are provided to the guest by QEMU. CPU instruction execution and memory accesses, however, are serviced by the KVM hypervisor running within the host system's Linux kernel. A request is made from a guest process (a) and the guest kernel accesses the device on the process's behalf (b). This request is passed through QEMU's KVM interface to the KVM kernel module in the host's kernel. KVM then forwards the request to a QEMU virtual device (c). The virtual device responds (d) and the result is provided to the

guest kernel (e). Finally, the guest process receives a response to its device request from the guest kernel (f).

Unlike the standard 0-3 ring-based protection scheme used by x86 platforms, virtualized systems contain two sets of rings: rings 0 through 3 on the host, and rings 0' through 3' on the guest. The rings within the guest are analogous to their counterparts on the host with one exception: the highest priority guest ring (ring 0') is at a lower priority than the lowest priority ring on the host (ring 3).

An exploit seeks to gain any privileges possible beyond those legitimately granted so as to access data or resources that it would otherwise be able to use. While a guest environment may be compromised by malicious software, it can still be safely contained within a virtualized environment so as not to harm the host. However, if that software were to compromise the hypervisor and gain ring 3 privileges on the host, it would effectively "break out" of the virtualized environment and gain the opportunity to attack the host system.

4.2.2 Understanding memory mapped I/O

Both physical and virtual peripherals must provide a method for software to interface with them. Devices have one or more registers that control the behavior of the device. By reading data from and writing data to these control registers, the hardware is instructed to perform tasks and provide information about the current state of the device. Each device's control registers are organized into one or more *register banks*. Each register bank is mapped to a contiguous range of guest physical memory locations that begin at a particular

base address. Thus, the physical memory addresses mapped to a particular device are specified by the address of each base address and the size of the register bank located at that base address. To simplify interaction with these control registers, the registers are accessed via normal memory bus activity. From a software point of view, hardware control registers are accessed via reads and writes to specific physical memory addresses.

The x86 family of processors is unique because it also provides port I/O-specific memory (all memory addresses below `0x10000`) that cannot be accessed via standard memory reads and writes [45]. Instead, the x86 instruction set provides two special I/O-specific instructions, `IN` and `OUT` [64], to perform 1, 2, or 4 byte accesses to port I/O memory. Other common architectures, such as Alpha, ARM, MIPS, and SPARC, do not have this port I/O memory region and treat all control register accesses as regular memory-mapped I/O. For simplicity of discussion, port-mapped I/O (PMIO) is refered to as memory-mapped I/O throughout this chapter.

Figure 4.2 shows where MMIO devices are mapped to in guest physical memory on x86-based systems. PCI-based PMIO mappings occur in the addresses ranging from `0xC000` through `0xFFFF`, with ISA-based devices mapped into the sub-`0xBFFF` PMIO range. PCI devices may also expose control registers or banks of device RAM or ROM in the PCI "hole" memory range `0xE0000000` through `0xFFFFFFFF`.

Each ISA device claims some address range within the ISA port I/O space to map its control registers. While some ISA devices have historically always been mapped to the same specific addresses (for example, `0x3F8` for the COM1 serial port), other ISA devices can be configured to use one or more of a small set of selectable base addresses to avoid conflicts with other devices.

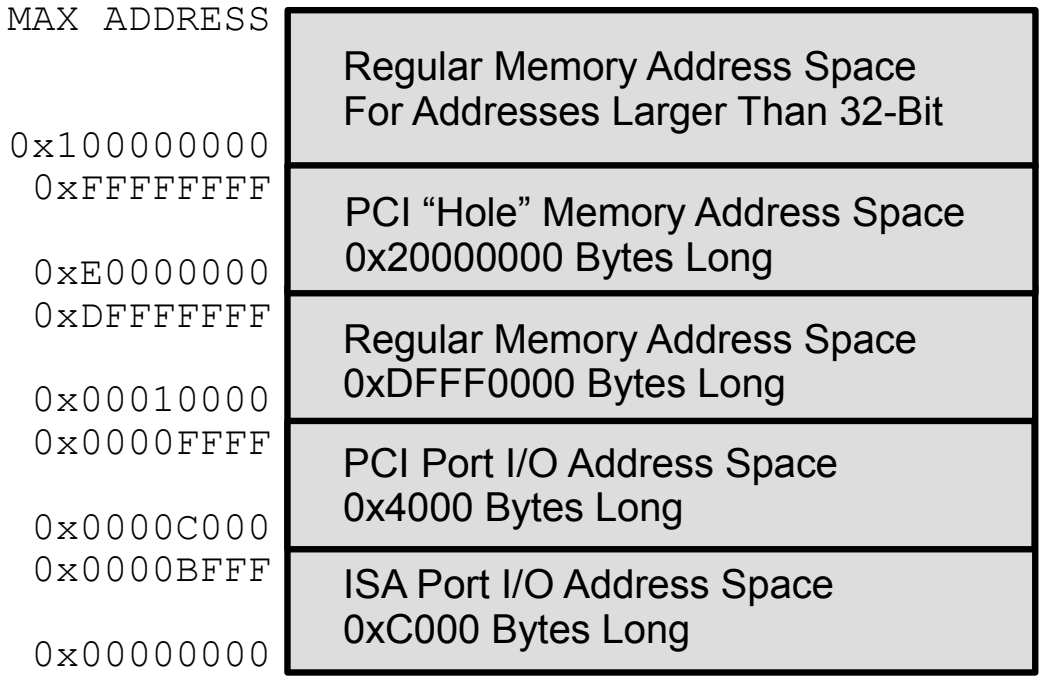


Fig. 4.2.: The x86 address space layout for port- and memory-mapped I/O.

PCI devices are far more flexible in the selection of their address mapping. At boot, the BIOS queries the PCI bus to enumerate all PCI devices connected to the bus. The number and sizes of the control register banks needed by each PCI device are reported to the BIOS. The BIOS then determines a memory-mapping for each register bank that satisfies the MMIO needs of all PCI devices without any overlap. Finally, the BIOS instructs the PCI bus to map specific base addresses to each device's register banks by configuring the PCI base address registers (BARs) of each device. This process, commonly performed by a "plug-and-play" BIOS, greatly simplifies and automates PCI device MMIO configuration at boot.

However, PCI makes the task of virtual device testing more difficult. By default, the BARs for each device contain invalid addresses. Until the BARs are initialized by the

BIOS, PCI devices are unusable. The PCI host controller provides two 32-bit registers in the ISA MMIO/PMIO address space for the task of configuring each PCI device BAR: `CONFIG_ADDRESS` at `0xCF8` and `CONFIG_DATA` at `0xCFC` [13]. Until the proper sequence of reads and writes are made to these two registers, PCI devices remain unconfigured and inaccessible to the guest environment. Therefore, ensuring that the configuration of a virtual PCI-based device is correct involves not only correctly initializing the state of the virtual device itself, but also the state of the PCI bus on which the virtual device resides.

4.3 Fuzzing virtual devices

Fuzzing mutates seed input to generate new test case inputs that exercise new paths of execution within a program. Simple fuzzers naively mutate seed inputs without any knowledge of the program under test, effectively treating the program being tested as a "black box". However, more sophisticated fuzzers, such as AFL [92], can insert compile-time instrumentation into the program under test. This instrumentation, placed at every branch and label within the instrumented program, tracks which branches have been taken at runtime when specific inputs are supplied. This method of "white box" fuzzing requires the program under analysis to be compiled from source, but it is much more effective at exploring new branches within a program.

If AFL generates a test case that covers one or more new branches within a program, that test case becomes a new seed input. As AFL continues to generate new seeds, more and more states of the program are exercised. Unfortunately, such an approach has its

limitations. All branches are considered to be of equal priority during exploration, so uninteresting states are explored as readily as interesting states are. This leads to a large number of wasted testing cycles as uninteresting states are unnecessarily explored. VDF leverages AFL's powerful white box fuzzing functionality to perform state exploration of virtual devices, but it is only interested in exploring branches belonging to the code that implements virtual devices. Therefore, the AFL fuzzer used within VDF has been modified to only instrument the portions of the hypervisor source code that belong to the virtual device currently being tested. This effectively makes AFL ignore the remainder of the hypervisor codebase when selectively mutating seed inputs.

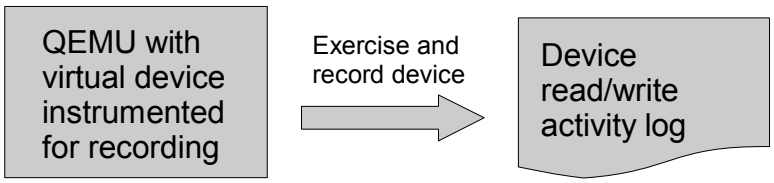
AFL maintains a "fuzz bitmap", with each byte within the bitmap representing a count of the number of times a particular branch within the fuzzed program has been taken. Programs may contain an arbitrarily large number of branches, and only a subset of branches may be exercised during fuzz testing, so AFL does not perform a one-to-one mapping between a particular branch and a byte within the bitmap. Instead, AFL's embedded instrumentation places a random two-byte constant identifier into each branch to identify that branch. Whenever execution during fuzzing reaches an instrumented branch, AFL performs an `XOR` of the new branch's identifier and the last branch identifier seen prior to arriving at the new branch. This captures both the current branch and the unique path taken to reach it (such as when the same function is called from multiple locations in the code). AFL then applies a hashing function to the `XOR`'d value to determine which entry in the bitmap represents that particular branch combination. Whenever a particular branch combination is exercised, the appropriate byte is incremented within the bitmap.

VDF's modified AFL uses a much simpler block coverage mechanism that provides a one-to-one mapping between a particular instrumented branch and a single entry in the bitmap. Because VDF selectively instruments *only* the branches within a virtual device, the bitmap fuzz contains more than enough entries to accommodate all such instrumented branches. VDF's modifications do away with the `XOR`ing of branch identifiers and AFL's hash function. Instead, branch identifiers are assigned linearly. This allows for a simpler mapping between identifiers and particular locations in the instrumented code, which simplifies determining the ground truth of whether a particular branch has been reached during testing. It also eliminates the possibility that randomly-generated branch identifiers are duplicated.

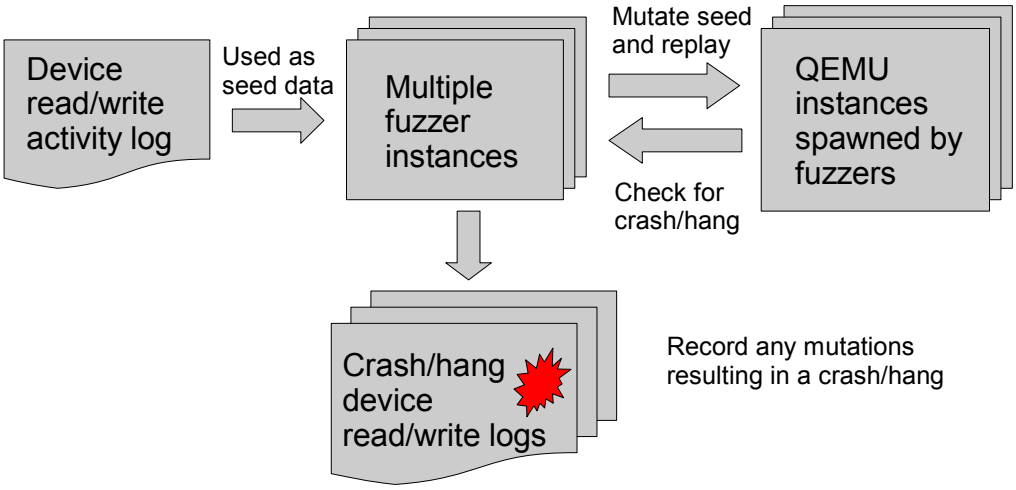
One large benefit of using AFL within VDF is the increase in test case execution speed provided by AFL's *fork server*. By default, AFL executes the program to be fuzzed and then makes a `fork` call in thta program once its `main` function is reached. This `fork` call is inserted into the program during AFL's compile-time instrumentation. Because the forking at `main` occurs after all shared libraries are loaded and all static resources are allocated, future test cases executed by child processes created at this fork point leverage copy-on-write of memory pages to eliminate much of the program's start-up time.

4.3.1 Fuzzing workflow

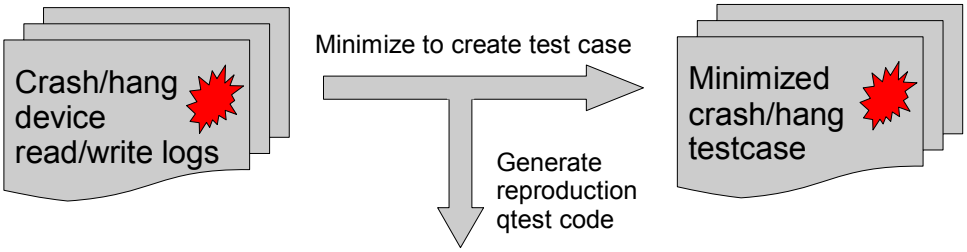
Figure 4.3 shows the three-step flow used by VDF when testing a virtual device. In the first step, virtual device activity is recorded while the device is being exercised. This log of activity includes any initialization of PCI BARs for the virtual device via the PCI host



Step 1: Record read/write activity of a virtual device by using our instrumented QEMU and generate a device activity seed test case.



Step 2: Execute multiple fuzzer instances in parallel to repeatedly mutate the seed, launch QEMU instances to replay the mutated seed, and discover any crashes or hangs.




```
#if 1 /* START: Reproduce case for qtest */
  qpci_io_writew(dev, dev_base[0]+0x4, 0x00007214);
  qpci_io_writew(dev, dev_base[0]+0x6, 0x00000001);
  qpci_io_writew(dev, dev_base[0]+0xE, 0x0000333A);
  qpci_io_writeb(dev, dev_base[0]+0x0, 0x00001780);
  qpci_io_writel(dev, dev_base[0]+0x1, 0x00000000);
#endif /* END: Reproduce case for qtest */
```

Step 3: Minimize crash/hang tests to simplify analysis and generate qtest code for future reproduction of each discovered crash or hang.

Fig. 4.3.: VDF's process for performing fuzz testing of QEMU virtual devices.

controller (if needed), initialization of any internal device registers, and any MMIO activity that exercises the virtual device. This log is then saved to disk, and it becomes the seed input for the fuzzer. This collection of seed input is described further in Section 4.3.2.

In the second step, the collected virtual device read/write activity is then provided as seed data to AFL. Multiple AFL instances can be launched, with one required master instance and one or more optional slave instances. The primary difference between master and slave instances is that the master will use a series of mutation strategies (bit/byte swapping, setting bytes to specific values like `0x00` and `0xFF`, etc.) to explore the program under test. Slave instances only perform random bit flips throughout the seed data. The mutated test cases simulate guest misbehavior that could be due to a badly written device driver within the guest's kernel, the actions of a malicious program executing within the guest OS, or even a combination of the two.

Once the seed input has been mutated into a new test case, a new QEMU instance is spawned via AFL's fork server. VDF replays the test case in the new QEMU instance and observes whether the mutated data has caused QEMU to crash or hang. It is important to note that VDF does not blindly replay events, but rather performs strict filtering on the mutated seed input during replay. The filter discards malformed events, events describing a read/write outside the range of the current register bank, events referencing an invalid register bank, etc. This prevents mutated data from potentially exercising memory locations unrelated to the virtual device under test. If a test case causes a crash or hang, the test case is saved to disk and logged.

Finally, in the third step, each of the collected crash and hang test cases is reduced to a minimal test case capable of reproducing the bug. Both a minimized test case and source

code to reproduce the bug are generated. The minimization of test cases is described further in Section 4.3.4.

4.3.2 Virtual device record and replay

Fuzzing virtual devices is non-trivial because virtual devices are *stateful*. It may be necessary to traverse an arbitrarily large number of states within both the virtual device and the remainder of the hypervisor prior to reaching a desired state within the virtual device. Because each virtual device must be initialized to a known good start state prior to each test, VDF uses *record and replay* of previous virtual device MMIO activity to both prepare the device for test and perform the test itself.

First, VDF records any guest reads or writes made to the virtual device's control registers when the device is initialized during guest OS boot[2]. This captures the setup performed by the BIOS (such as PCI BAR configuration), device driver initialization in the kernel, and any guest userspace process interaction with the device's kernel driver. Table 4.1 shows the different sources of initialization activity used by VDF when recording device activity during our testing.

Second, the recorded startup activity is partitioned into two pieces: an *init* set and a *seed* set. The init set contains any seed input required to initialize the device for testing, such as PCI BAR setup, and the activity in this set will never be mutated by the fuzzer. VDF plays back the init set at the start of each test to return the device to a known, repeatable state. The seed set contains the seed input that will be mutated by the fuzzer. It

[2]VDF can also capture this initialization activity if the device is exercised via a QEMU qtest test case, if only a minimal amount of recorded activity is desired. However, most seed input used in this dissertation was simply recorded during the boot of the guest OS.

Table 4.1: QEMU virtual devices seed data sources.

Device Class	Device	Seed Data Source
Audio	AC97	Linux guest boot with ALSA [1] `speaker-test`
	CS4231a	
	ES1370	
	Intel-HDA	
	SoundBlaster 16	
Block	Floppy	qtest test case
Char	Parallel	Linux guest boot with directed console output
	Serial	
IDE	IDE Core	qtest test case
Network	EEPro100 (i82550)	Linux guest boot with `ping` of IP address
	E1000 (82544GC)	
	NE2000 (PCI)	
	PCNET (PCI)	
	RTL8139	qtest test case
SD Card	SD HCI	Linux guest boot with mounted SDHCI volume
TPM	TPM	Linux guest boot with TrouSerS test suite [20]
Watchdog	IB700	qtest test case
	I6300ESB	Linux guest boot

can be any sequence of reads and writes that exercises the device, and it usually originates from some guest user space activity that exercises the device (playing an audio file, pinging an IP address, etc.).

Even with no guest OS booted or present, a replay of these two sets returns the virtual device to the same state that it was in immediately after the register activity was originally recorded. While the data in the sets could include timestamps to ensure that the replay occurs at the correct time intervals, VDF does not do this. Instead, VDF takes the simpler approach of advancing the virtual clock of the guest environment one microsecond for

each read or write performed. The difficulty with including timestamps within the seed input is that the value of the timestamp is too easily mutated into very long virtual delays between events. While it is true that some virtual device branches may only be reachable once an longer, abitrary virtual time interval has passed (such as interrupts that are raised when a device has completed performing some physical event), performing a fixed increment of virtual time on each read and write is a reasonable approach to the issue.

Event record format

Fig. 4.4.: The record format of VDF for an MMIO read/write event.

The format of the VDF read/write event record is shown in Figure 4.4. This format captures all data needed to replay an MMIO event and represents this information in a compact format requiring only 3-8 bytes per event. The compactness of each record is an important factor because using a smaller record size decreases the number of bits that can be potentially mutated by the fuzzer.

The header in Figure 4.4a is a single byte that captures whether the event is a read or write event, the size of the event (1, 2, or 4 bytes), and which virtual device register bank the event takes place in. The base offset field in Figure 4.4b is the offset from the base address of the register bank specified in the header. The size of this field will vary from

device to device, as some devices have small register bank ranges (requiring only one byte to represent an offset into the register bank) and other devices map much larger register banks and device RAM address ranges (requiring two or three bytes to specify an offset). The data field in Figure 4.4c is the data written to a memory location when the header field specifies a write operation. Some devices, such as the floppy disk controller and the serial port, only accept single byte writes. Most devices accept writes of 1, 2, or 4 bytes, requiring a 4 byte field for those devices to represent the data. For read operations, the data field of the record is ignored.

While VDF's record and replay of MMIO activity captures the interaction of the guest environment with virtual devices, some devices may make use of interrupts and DMA. However, such hardware events are not necessarily required to recreate the majority of the behavior of most devices during fuzz testing. Interrupts are typically *produced* by a virtual device, rather than *consumed*, when some hardware event has completed. Interrupts alert the guest environment that some hardware event has completed. Another read or write event would then be initiated by the guest in reaction to an interrupt, but since VDF records all read/write activity to the virtual device, the guest's response to the interrupt is captured without explicitly capturing the interrupt itself.

DMA events perform copies of data between guest and device RAM. DMA copies typically occur when buffers of data must be copied and the CPU isn't needed to copy this data byte-by-byte. These buffers contain data to be processed by the virtual device, such as pixel data to be displayed on a video framebuffer. Thus, when only copying data to be processed, it is not necessary to actually place data in the correct location within guest RAM and then copy it into the virtual device. It is enough to say that the data has been

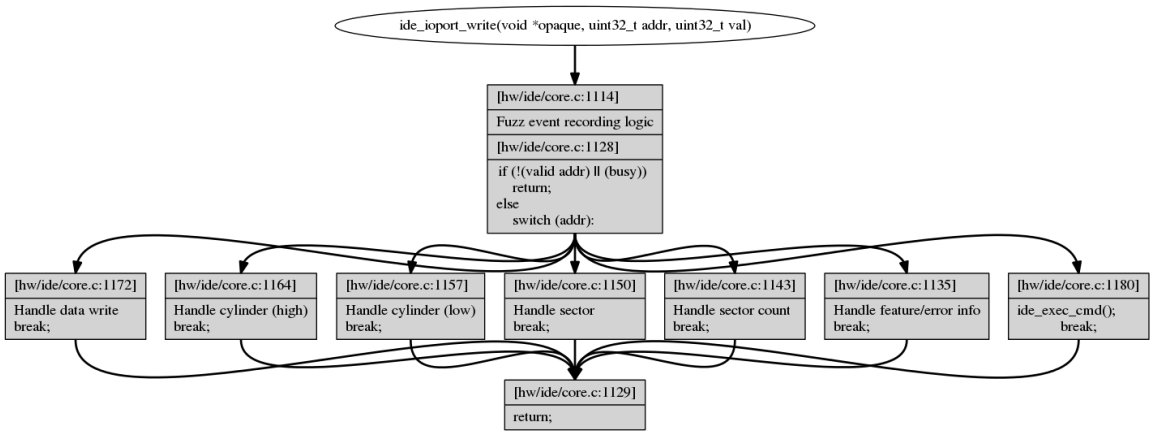


Fig. 4.5.: Simplified control flow graph of the `ide_ioport_write()` function within the QEMU IDE core.

copied and then move onto the next event. While the size of data and alignment of the data may have some impact on the behavior of the virtual device, such details are beyond the scope of VDF as described in this dissertation.

Recording virtual device activity

Consider the simplified control flow graph (CFG) shown in Figure 4.5. This is a CFG for the function `ide_ioport_write()`, which is implemented within QEMU's IDE virtual device. This function is registered with QEMU's MMU when an IDE bus is instantiated, and the function is executed whenever the guest performs a write to memory addresses `0x170` (primary IDE bus) or `0x1F0` (secondary IDE bus). The guest interacts with IDE-based devices, such as hard disks and CD-ROM drives, by writing to the control registers mapped to these memory locations.

This CFG has a structure seen in most QEMU virtual device callback functions. The function receives a state structure of the virtual device (passed as an opaque `void *`), the

memory address written to by the guest, and the data written to that memory address. Once the address is filtered and the current state of the device is examined, a `switch` statement uses the address to dispatch the data to the proper subfunction of the virtual device. This allows a single function to provide an interface that controls a large number of the device's features. While the simplest virtual devices register only a single set of MMIO callback functions to handle read/write activity to a single MMIO address, more complex devices register multiple sets of callbacks to represent several sets of MMIO control registers provided by the device.

The structure of the callback function's CFG explains the primary reason that VDF's targeted fuzzing is so effective for exploring virtual devices. A minor mutation in the address provided to the callback function is all that is needed to reach different pieces of the device's functionality. Mutated test cases that exercise each of the various cases of the `switch` statement will reach the majority of the branches of interest within the device.

Almost every interaction between the guest environment and virtual devices occurs via this MMIO interface, so it is an ideal location to record the virtual device's activity. Rather than attempt to capture the usage of the each device by reconstructing the semantics of the guest's kernel and memory space, VDF captures device activity at the point where the hardware interface is provided to software. In fact, there is no immediate need to understand the semantic details of the guest environment as virtual devices execute at a level above that of even the guest's BIOS or kernel. By placing recording logic in these callback functions, VDF is able to instrument each virtual device by adding only 3 to 5 LOC of recording logic to the beginning of each MMIO callback function.

Playback of virtual device activity

Once VDF has recorded a stream of read/write events for a virtual device, it must have a mechanism to replay these events within the context of a running QEMU. Because QEMU traverses a large number of branches before all virtual devices are instantiated and testing can proceed, it isn't possible to provide the event data to QEMU via the command line. The events must originate from within the guest environment in the form of read/write activity to memory locations registered for MMIO. Therefore, QEMU must first be initialized before performing the replay of MMIO events.

QEMU provides *qtest*, which is a lightweight framework for testing virtual devices. qtest is a QEMU *accelerator*, or type of execution engine. Common accelerators for QEMU are *TCG* (for the usage of QEMU TCG IR) and *KVM* (for using the host kernel's KVM for hardware accelerated execution of guest CPU instructions). The qtest framework works by using a test driver process to spawn a separate QEMU process which uses the qtest accelerator. The qtest accelerator within QEMU communicates with the test driver process via IPC. The test driver remotely controls QEMU's qtest accelerator to perform guest memory read/write instructions to virtual devices exposed via MMIO. Once the test is complete, the test driver terminates the QEMU process.

While the qtest accelerator and test driver programs are convenient, they are inadequate for the type of testing that VDF performs for two reasons. First, the throughput and timing of the test is slowed because of QEMU start-up and the serialization, deserialization, and transfer time of the IPC protocol. Commands are sent between the test driver and QEMU as plaintext messages, requiring time to parse each string. While this is

not a concern for the virtual clock of QEMU, wall clock-related issues (such as thread race conditions in a virtual device backend) are less likely to be exposed with the slower pace of virtual device activity.

Second, qtest does not provide control over QEMU beyond spawning the new QEMU instance and sending control messages back and forth. It is unable to determine exactly where a hung QEMU process has become stuck. A hung QEMU also hangs the qtest test driver process, as the test driver will continue to wait for input from the non-responsive QEMU. If QEMU crashes, qtest will respond with the feedback that the test failed. Reproducing the test which triggers the crash may repeat the crash, but the analyst still has to attach a debugger to the spawned QEMU instance prior to the crash to gain insight into exactly why the crash is occurring.

VDF's seeks to automate the discovery of any combination of virtual device MMIO activity that triggers a hang or crash in either the virtual device or some portion of the hypervisor. qtest excels at running known-good, hard-coded tests on QEMU virtual devices for repeatable regression testing. But, it becomes less useful when searching for unknown vulnerabilities. Such a search requires generating new test cases that cover as many execution paths as possible through a virtual device, as quickly as possible.

To address these shortcomings, VDF contains a new *fuzzer* QEMU accelerator, based upon qtest. This new accelerator adds approximately 850 LOC to the QEMU codebase. It combines the functionality of the qtest test driver process and the qtest accelerator within QEMU, eliminating the need for a separate test driver process and the IPC between QEMU and the test driver. More importantly, it allows VDF to directly replay read and

write activity that exercises virtual devices as if the event came directly from within a complete guest environment.

4.3.3 Selective branch instrumentation

Fuzz testing must explore as many branches of interest as possible within a program to perform effective testing. Therefore, determining the *coverage* of those branches during testing is a metric for measuring the thoroughness of our approach. While the code within any branch may host a particular bug, execution of the branch must be performed to trigger the bug. Thus, reaching more branches of interest increases the chances that a bug will be discovered. However, if the fuzzer attempts to explore *every* branch it discovers, it can potentially waste millions of test cycles trying to test branches unrelated to the virtual device of interest.

To address this issue, VDF leverages the instrumentation capabilities of AFL to selectively place this instrumentation in only the branches of interest (those belonging to a virtual device). By default, the compiler toolchain supplied with AFL instruments programs built using it. This instrumentation places a randomly-generated ID and some trampoline logic at every branch within the instrumented program. When the program is executed under AFL for testing, reaching a branch results in the ID of the branch being loaded into a register and the trampoline code being called. The trampoline code notes the ID and marks the branch as having been visiting in the fuzz bitmap. This scheme is designed to encourage the exploration of all branches within a program, as each branch represents a new area for AFL to focus on to expand coverage during test.

VDF modifies AFL to selectively instrument only code of interest within the target program. The modifications provide a one-to-one mapping between branches of interest and locations within AFL's fuzz bitmap, so measuring coverage becomes as simple as comparing the current state of the fuzz bitmap with the bitmap locations known to represent branches of interest within the virtual device. Uninstrumented branches are ignored by the fuzzer as they are seen as (very long) basic blocks of instructions that occur between instrumented branches. Aside from the instrumented branches within the virtual device, a stub `main()` function is also instrumented. This stub `main()`'s purpose is to trigger AFL's deferred forking and then call the program's true `main()` function.

Prior to the start of each testing session, VDF dumps and examines all function and label symbols found in the instrumented hypervisor. If a symbol is found that maps to an instrumented branch belonging to the current virtual device under test, the name, address, and AFL branch identifier (embedded in the symbol name) of the symbol are stored and mapped to the symbol's location in the fuzz bitmap. At any point during testing, the AFL fuzz bitmap can be dumped using VDF to provide ground truth of exactly which branches have been covered thus far.

Figure 4.6 shows an example of the coverage information report that VDF provides. This example shows both the original source code for a function in the AC97 audio virtual device (top) and the generated branch coverage report for that function (bottom). The report provides two pieces of important information. The first is the ground truth of which branches are instrumented, including their address within the binary, the symbol associated with the branch (inserted by the modified AFL), and the original source file line

```
static void voice_set_active (AC97LinkState *s, int bm_index, int on)
{
    switch (bm_index) {
    case PI_INDEX:
        AUD_set_active_in (s->voice_pi, on);
        break;
    case PO_INDEX:
        AUD_set_active_out (s->voice_po, on);
        break;
    case MC_INDEX:
        AUD_set_active_in (s->voice_mc, on);
        break;
    default:
        AUD_log ("ac97",
            "invalid bm_index(%d) in voice_set_active",
            bm_index);
        break;
    }
}
```

```
ID: COVERED: ADDRESS: SYMBOL:                          LINE:
--- -------- -------- -------                          -----
00c COVER    002e92e0 voice_set_active                 296
00d COVER    002e9324 REF_LABEL__tmp_ccBGk9PX_s__27_39 296
00e COVER    002e9368 REF_LABEL__tmp_ccBGk9PX_s__28_40 296
00f UNCOVER  002e93a4 REF_LABEL__tmp_ccBGk9PX_s__29_41 296
```

Fig. 4.6.: Sample branch coverage data for the `voice_set_active()` function within the AC97 virtual device.

number where the branch's code is located. The second is whether a particular branch has been visited yet during testing.

The four branches listed in the report are associated with the four cases in the switch statement of the `voice_set_active()` function, which is located on line 296 in the source file. By examining the coverage report, we can see that the first three cases have been reached during the testing performed thus far. The default fall-through case has not yet been triggered by any tests. An analyst familiar with the internals of the AC97 virtual device could review this report and then devise a new seed input that contains the necessary register activity to trigger the final case in the switch statement. Thus, such reports are useful for not only an understanding of *which* branches have been reached, but

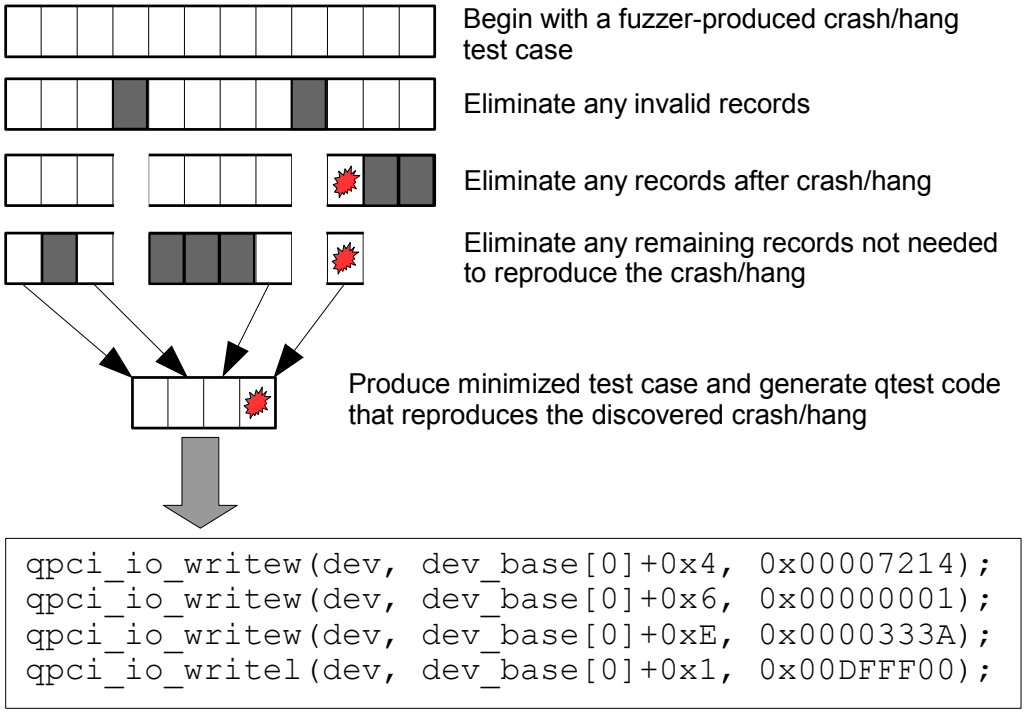


Fig. 4.7.: Process for minimizing test cases.

they also providing insight into *how* the unexplored virtual device functionality might be reached.

4.3.4 Creation of minimal test cases

Once VDF detects either a crash or a hang in a virtual device, the test case that produced the issue is saved for later examination. This test case may contain a large amount of test data that is not needed to reproduce the discovered issue, so it is desirable to reduce this test case to the absolute minimum number of records needed to still trigger the bug. Such a minimal test case simplifies the job of the analyst when using the test case to debug the cause of the discovered issue.

VDF performs a three-step test case post-processing, seen in Figure 4.7, to produce a minimal test case from any test case shown to reproduce an issue. First, the test case file is read into memory and any valid test records in the test case are placed into an ordered dataset in the order in which they appear within the test case. Because the fuzzer lacks semantic understanding of the fields within these records, it produces many records via mutation that contain invalid garbage data. Such invalid records may contain an invalid header field, describe a base offset to a register outside of the register bank for the device, or simply be a truncated record at the end of the test case. After this filtering step, the dataset contains only valid test records.

Second, VDF eliminates all records in the dataset that are located after the point in the test case where the issue is triggered. To do this, it generates a new test case using all but the last record of the dataset and then attempts to trigger the issue using this truncated test case. If the issue is still triggered when using the new test case, the last record is then removed from the dataset and another new truncated test case is generated in the same fashion. This process is repeated until a truncated test case is created that no longer triggers the issue, indicating that all dataset records located after the issue being triggered are now removed.

Third, VDF eliminates any remaining records in the dataset that are not necessary to trigger the issue. Beginning with the first record in the dataset, VDF iterates through each dataset record, generating a new test case using all but the current record. It then attempts to trigger the issue using this generated test case. If the issue is still triggered, the current record is not needed to trigger the issue and is removed from the dataset. Once each

dataset record has been visited and the unnecessary records removed, the dataset is written out to disk as the final, minimized test case.

While simple, VDF's test case minimization is very effective. The 1014 crash and hang test cases produced by the fuzzer during testing have an average size of 2563.5 bytes each. After reducing these test cases to a minimal state, the average test case size becomes only 476 bytes, a mere 18.57% of the original test case size. On average, each minimal test case is able to trigger an issue by performing approximately 13 read/write operations. This average is misleadingly high due to some outliers, however, as over 92.3% of the minimized test cases perform fewer than six MMIO read/write operations.

4.4 Evaluation

The configuration used for all evaluations is a cloud-based 8-core 2.0GHz Intel Xeon ES-2650 CPU instance with 8 GB of RAM. Each such instance uses a minimal server installation of Ubuntu 14.04 Linux as its OS. Eight such cloud instances were utilized in parallel. Each device was fuzzed within a single cloud instance, with one master fuzzer process and five slave fuzzer processes performing the testing. A similar configuration was used for test case minimization: each cloud instance ran six minimizer processes in parallel to reduce each discovered crash/hang test case.

A set of eighteen virtual devices, shown in Table 4.2, were selected for the evaluation of VDF. These virtual devices utilize a wide variety of hardware features, such as timers, interrupts, DMA, and MMIO. Each of these devices provides one or more MMIO interfaces to their internal registers, which VDF's fuzzing accelerator interacts with. All

Table 4.2: QEMU virtual devices tested with VDF.

Device Class	Device	Branches of Interest	Coverage Via Initial Seeds	Coverage Via Fuzz Testing	Crashes Found	Hangs Found	Total Tests Per Fuzzer Instance (Millions)	Cumulative Test Duration
Audio	AC97	164	43.9%	53.0%	87	0	24.0	59d 18h
	CS4231a	109	5.5%	56.0%	0	0	29.3	65d 12h
	ES1370	165	50.9%	72.7%	0	0	30.8	69d 18h
	Intel-HDA	273	43.6%	58.6%	238	0	23.1	59d 12h
	SoundBlaster 16	311	26.7%	81.0%	0	0	26.7	58d 13h
Block	Floppy	370	44.9%	70.5%	0	0	21.0	57d 15h
Char	Parallel	91	30.8%	42.9%	0	0	14.6	25d 12h
	Serial	213	2.3%	44.6%	0	0	33.0	62d 12h
IDE	IDE Core	524	13.9%	27.5%	0	0	24.9	65d 6h
Network	EEPro100 (i82550)	240	15.8%	75.4%	0	0	25.7	62d 12h
	E1000 (82544GC)	332	13.9%	81.6%	0	384	23.9	61d
	NE2000 (PCI)	145	39.3%	71.7%	0	0	25.2	58d 13h
	PCNET (PCI)	487	11.5%	36.1%	0	0	25.0	58d 13h
	RTL8139	349	12.9%	63.0%	0	6	24.2	58d 12h
SD Card	SD HCI	486	18.3%	90.5%	14	265	24.0	62d
TPM	TPM	238	26.1%	67.3%	9	11	2.1	36d 12h
Watchdog	IB700	16	87.5%	100.0%	0	0	0.3	8h
	I6300ESB	76	43.4%	68.4%	0	0	2.1	26h

devices were evaluated using QEMU v2.5.0, with the exception of the TPM device. The TPM was evaluated using QEMU v2.2.50 with an applied patchset that provides a libtpms emulation [30] of the TPM hardware device [34]. Fewer than 1000 LOC were added to each of these two QEMU codebases to implement both the fuzzer accelerator and any recording instrumentation necessary within each tested virtual device.

4.4.1 Virtual device coverage and bug discovery

Four metrics were collected during testing to measure both the speed and magnitude of VDF's coverage. These metrics are 1) the number of branches covered by the initial seed test case; 2) the total number of branches in the virtual device; 3) the current total number of branches covered (updated at one minute intervals); and 4) the percentage of total bugs discovered during each cumulative day of testing. Taken together, these metrics describe not only the total amount of coverage provided by VDF, but also the speed at which coverage improves via fuzzing and how quickly it discovers crash/hang test cases.

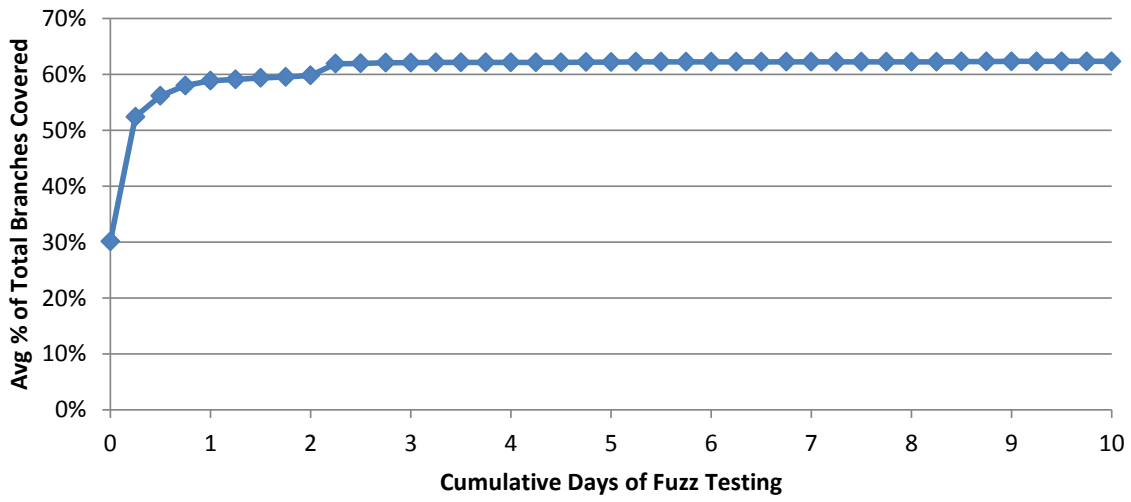


Fig. 4.8.: Average percentage of branches covered during fuzz testing.

Figure 4.8 shows the average percentage of covered branches over cumulative testing time. Of the eighteen tested virtual devices, 30.15% of the total branches were covered by the initial seed test cases. After nine cumulative days of testing (36 hours of parallel testing with one master and five slave fuzzing instances), 62.32% of the total branches were covered. The largest increase in average coverage was seen during the first six cumulative hours of testing, where coverage increased from the initial 30.15% to 52.84%. After 2.25 days of cumulative testing, average coverage slows considerably and only 0.43% more of the total branches are discovered during the next 6.75 cumulative days of testing. While eleven of the eighteen tested devices stopped discovering new branches after only one day of cumulative testing, six of the seven remaining devices continued to discover additional branches until 6.5 cumulative days had elapsed. Only one virtual device (serial) discovered additional branches after nine cumulative days of testing.

Table 4.2 presents some insightful statistics about coverage. The smallest improvement in the percentage of coverage was seen in the AC97 virtual device (9.1%

increase), and the largest improvement in coverage was seen in the SDHCI virtual device (72.2% increase). The smallest percentage of coverage for any virtual device with discovered crashes/hangs was 53.0% (AC97), but eight other virtual devices had a greater level of coverage than 53.0% with no discovered crashes/hangs.

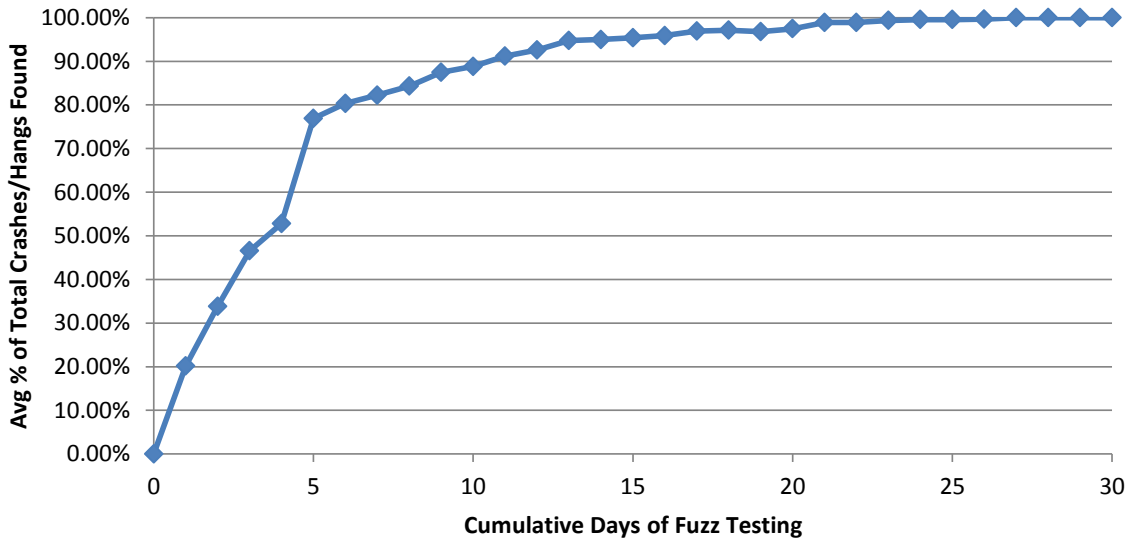


Fig. 4.9.: Average percentage of total bugs discovered during fuzz testing.

Figure 4.9 shows the average percentage of discovered hangs/crashes over cumulative testing time. As shown in Table 4.2, a total of 1014 crashes and hangs were discovered in six virtual devices. These 1014 test cases were all discovered within 27 days of cumulative testing for each device, with no additional test cases being discovered after that point. Approximately 50% of all test cases were discovered after four days of cumulative testing, with approximately 80% of all test cases discovered after five days of cumulative testing.

One interesting insight is that even though the number of branches covered is very close to its maximum after approximately 2.5 cumulative days of testing, only approximately 25% of all crash/hang test cases were discovered at that point in time. This

shows that it is not necessarily an *increase in branch coverage* that leads to the discovery of bugs, but rather the *repeated fuzz testing of those discovered branches*.

4.4.2 Classification of all discovered virtual device bugs

While it is straightforward to count the number of discovered crash/hang test cases generated by the fuzzer, it is non-trivial to map these test cases to their underlying cause without a full understanding of the virtual device under test. This understanding involves knowing which reads/writes commands perform which register commands within the virtual device, the state machine of the virtual device, and any additional requirements (external files mounted as secondary storage, pass-through requirements, etc.) of the virtual device.

The proposed test case minimization method greatly simplifies this process, as many unique bugs identified by VDF minimize to the same set of read/write operations. The ordering of these operations may differ, but the final read/write that triggers the bug remains the same. Thus, it becomes far simpler to more accurately assess the number of unique bugs discovered.

The discovered virtual device bugs fell into one of four categories: Excess resource usage (AC97), invalid data transfers (E1000, RTL8139, SDHCI), debugging asserts (Intel-HDA), and thread race conditions (TPM).

Excess host resource usage

Some set of resources belonging to the host system must be allocated to QEMU to represent the resources allocated to the guest environment. Such resources include RAM to represent the physical RAM present on the guest, CPU cores and cycles to perform CPU and virtual device emulation, and disk space to hold the guest's secondary storage.

The crash discovered while testing the AC97 audio virtual device caused QEMU to allocate approximately 500MB of additional host memory when the control register for `AC97_MIC_ADC_Rate` is set to an invalid, non-zero value. Additional resources may be allocated by QEMU at runtime to meet the data needs of virtual devices, which presents a potential opportunity for a malicious guest to trick QEMU into allocating large amounts of unnecessary resources at runtime. An important observation on this type of resource bug is that it will easily remain hidden unless the resource usage of the QEMU process is strictly monitored and enforced. For example, using the Linux `ulimit` command to place a limit on the virtual memory allocated to QEMU will discover this bug when the specified memory limit is exceeded. VDF enforces such a limitation during its testing, using AFL to limit the amount of virtual memory allocated to each QEMU instance. Once this virtual memory is exceeded, memory allocations within QEMU fail, leading to a `SIGTRAP` signal being raised and a crash test case saved for later analysis.

While a single hypervisor allocating excessive resources for a single guest instance is typically not a concern, the potential impact of this issue increases greatly when considering a scenario with large numbers of instances deployed within a cloud environment. Discovering and correcting such bugs can have a measurable impact on the

resource usage of hosts implementing cloud environments. This bug has been reported to the QEMU maintainers.

Invalid data transfers

Many virtual devices emulate hardware that transfers blocks of data. Such transfers are used to move data to and from secondary storage and guest physical memory via DMA. However, invalid data transfers can cause virtual devices to hang in an infinite loop. This type of bug can be difficult to deal with in production systems as the QEMU process is still alive and running while the guest's virtual clock is in a "paused" state. If queried, the QEMU process will appear to still be running without issue and will respond to signals from the host OS. The guest will remain frozen and cause a denial of service for any processes running inside of the guest.

VDF discovered test cases that triggered invalid data transfer bugs in the E1000 and RTL8139 virtual network devices and the SDHCI virtual block device. In each of these cases, a transfer was initiated with either a block size of zero or an invalid transfer size, leaving each device in a loop that either never terminates or executes over an arbitrarily long period of time.

For the E1000 virtual device, the guest sets the device's `E1000_TDH` and `E1000_TDT` registers (TX descriptor head and tail, respectively) with offsets into guest memory that designate the current position into a buffer containing transfer operation descriptors. The guest then initiates a transfer using the `E1000_TCTL` register (TX control). However, if the values placed into the `E1000_TDH/TDL` registers are too large, then the transfer logic

enters an infinite loop. VDF discovered this by mutating writes into the `E1000_TDH/TDL` registers. A review of reported CVEs has shown that this issue was already discovered in January 2016, a CVE [8] was reserved, and a patch [9] was included into mainline QEMU to address it.

For the RTL8139 virtual device, the guest resets the device via the `ChipCmd` (chip control) register. Then, the `TxAddr0` (transfer address), `CpCmd` ("C+" mode command), and `TxPoll` (check transfer descriptors) registers are set to initiate a DMA transfer in the RTL8139's "C+" mode. However, if an invalid address is supplied to the `TxAddr0` register, QEMU becomes trapped in an endless loop of DMA page lookup operations. An interesting observation on this particular bug is that six test cases were generated by VDF that demonstrate this same bug once the test cases were minimized. However, the seed RTL8139 test case used in this evaluation testing was recorded from a qtest test case (`tests/rtl8139-test.c`) that only tests the raising of an interrupt after a QEMU timer has expired. This demonstrates that VDF is capable of discovering interesting bugs that are completely unrelated to the register activity recorded in the seed input. This was an undiscovered bug, which was reported to the QEMU security team for their assessment due to its potential as a denial of service exploit.

For the SDHCI virtual device, the guest sets the device's `SDHC_CMDREG` register bit for "data is present" and sets the block size to transfer to zero in the `SDHC_BLKSIZE` register. The `switch` case for `SDHC_BLKSIZE` in the `sdhci_write()` MMIO callback function in `hw/sd/sdhci.c` performs a check to determine whether the block size exceeds the maximum allowable block size, but it does not perform a check for a block size of zero. Once the transfer begins, the device becomes stuck in a loop, and the

guest environment becomes unresponsive. Luckily, fixes for this issue were integrated into mainline QEMU as part of an overall effort to correct SD card support for the Raspberry Pi platform [18] in December 2015.

While most test cases reproducing invalid transfer bugs minimize down to a test case containing only four read/write operations, some test cases contained six or seven operations, instead. However, all of the test cases still resulted in triggering the same bugs. After some examination, the reason for this was determined to be the fragmentation of MMIO reads/writes on non-word aligned memory accesses. For example, a single two-byte write made to a word-aligned memory address (say, `0x100`) will appear as a single write operation to a virtual device. The same two-byte write, when made to the non-aligned address `0x101`, will be fragmented into two one-byte writes at consecutive addresses (`0x101` and `0x102`). Therefore, it is possible to shift the address of a read/write to a non-aligned address in a test case and exercise a very different set of registers. This results in test cases that appear quite different after minimization, but which still trigger the same underlying bugs.

Debugging asserts

The Intel-HDA audio device demonstrates a limitation in using the `assert` function to test for invalid conditions within virtual devices. All of the VDF evaluation testing used a debug build of QEMU to assist in the assessment of each discovered hang or crash within the virtual devices. The `intel_hda_reg_write()` function in `hw/audio/intel-hda.c` uses an `assert` call to trigger a `SIGABRT` when a write is

made to an address offset of 0 from the MMIO register base address. The mutated seed data did indeed attempt to make 1-, 2-, and 4-byte writes to an address offset of 0, resulting in VDF's discovery of the issue. Thus, a guest could make a single, one-byte write to the Intel-HDA control register bank and crash a debug QEMU!

While using an `assert` is a commonly-used debugging technique in mature software codebases, `asserts` are used to catch a particular case that should "never happen". If that impossible case actually *can* happen as a result of untrusted input, proper error-handling logic should be added to the code to address it. If the virtual device code is built with `NDEBUG` defined (rendering the GNU libc `asserts` within the code into no-ops), then the invalid input would continue past the `assert` check and into the remainder of the virtual device. This item was reported as a bug to the QEMU development team.

Thread race conditions

The virtual TPM in mainline QEMU is a pass-through device to the host's hardware TPM device. It is possible to implement a TPM emulated in software using libtpms [30] and then have QEMU pass TPM activity through to the emulated hardware. QEMU interacts with the separate process implementing the TPM via RPC. However, it is also possible to integrate libtpms directly into QEMU by applying a patchset provided by IBM [34]. This allows each QEMU instance to "own" its own TPM instance and directly control the start-up and shutdown of the TPM. This patchset was selected for testing because it was never accepted into mainline QEMU and any bugs discovered would not have an immediate security impact on deployed production systems. Instead, it serves as

an example of how proposed virtual device patches could be tested using VDF as part of the patch vetting process.

The crashes/hangs discovered in the TPM demonstrate a race condition between the shutdown of the main QEMU process and the worker threads in a thread pool that are executing TPM commands in the TPM virtual device's libtpms backend. It is possible for the thread pool to hang while waiting on a mutex when shutting down the thread pool. This hang is more noticeable on a single CPU, single core host system, though it could potentially still occur on a multicore system within an extremely small time window. A backtrace of the stack, as captured under GDB while using one of the crash datasets, is shown in Figure 4.10.

```
Program received signal SIGINT, Interrupt.
pthread_cond_wait@@GLIBC_2.3.2 () at
  ../nptl/sysdeps/unix/sysv/linux/x86_64/pthread_cond_wait.S:185
185  ../nptl/sysdeps/unix/sysv/linux/x86_64/pthread_cond_wait.S:
  No such file or directory.
(gdb) bt
#0  pthread_cond_wait@@GLIBC_2.3.2 () at
  ../nptl/sysdeps/unix/sysv/linux/x86_64/pthread_cond_wait.S:185
#1  0x00007ffff75e5bf7 in g_cond_wait () from
  /lib/x86_64-linux-gnu/libglib-2.0.so.0
#2  0x00007ffff75c9f60 in g_thread_pool_free () from
  /lib/x86_64-linux-gnu/libglib-2.0.so.0
#3  0x0000555555b5b68b in tpm_backend_thread_end
  (tbt=0x555556eadef8) at backends/tpm.c:163
#4  0x0000555555fb8788 in tpm_ltpms_terminate_tpm_thread
  (tb=0x555556eade80) at hw/tpm/tpm_libtpms.c:708
#5  tpm_ltpms_destroy (tb=0x555556eade80) at
  hw/tpm/tpm_libtpms.c:821
#6  0x000055555560ae01 in main (argc=<optimized out>,
  argv=0x7fffffffdd38,envp=<optimized out>) at vl.c:4503
```

Fig. 4.10.: The backtrace of the deadlock in the worker thread pool shutdown, which occurs in the TPM backend (entries #2 and #3 in the backtrace).

This issue is the result of either a premature exit of QEMU, such as an `exit()` triggered by a bad command line argument or a failed VM migration attempt, or an extremely short QEMU session that triggers the `main_loop_wait()` function a

minimal number of times before QEMU terminates. In both of these cases, the thread pool shutdown will occur before the tasks allocated to the thread pool have all been completed. Without an adequately long call to `sleep()` or `usleep()` prior to the thread pool shutdown to force a context switch and allow the thread pool worker threads to complete, the thread pool will hang on shutdown. Because the shutdown of the TPM backend is registered to be called at `exit()` via an `atexit()` call, any premature `exit()` prior to the necessary `sleep()` or `usleep()` call will trigger this issue. QEMU registers signal handlers that are never unregistered, so attempts to kill the hung process via a `SIGTERM` signal are unsuccessful. The hung QEMU instance must be killed via a `SIGKILL` signal.

Note that this thread pool is part of the TPM backend design in QEMU. It is not part of the libtpms library that implements the actual TPM emulator. Most likely this design decision was made to avoid any noticeable slowdown in QEMU's execution by making the TPM virtual device run in an asynchronous manner to avoid any performance impact caused by performing expensive operations in the software TPM. Other TPM pass-through options, such as the Character in User Space (CUSE) device interface to a stand-alone TPM emulator using libtpms, should not experience this particular issue. In fact, the patchset that VDF was tested against has been made obsolete by a newer patchset [14] that uses a stand-alone TPM emulator, so reporting the discovered threading issue was not necessary.

This issue highlights a limitation of working with virtualized systems: while virtual time within the guest can be manipulated, there is still a finite amount of wall clock time required to execute the hypervisor on the host and emulate the hardware provided to the guest. It also demonstrates a limitation of VDF's test case minimization process. VDF's

minimization operates under the assumption that a test case will consistently trigger a hang or crash, so repeated executions of the test case during the iterative removal of unnecessary records is reasonable. Unfortunately, race conditions, such as those seen in the TPM backend, are not consistently triggerable. During reproduction of the issue by using crash/hang test cases collected by VDF, each test case had to be executed an average of 5.7 times under GDB before the hang in the thread pool shutdown was observed to occur.

4.5 Limitations of VDF

While VDF is a novel approach to the fuzz testing of virtual devices, it has a number of limitations that must be considered both when testing additional virtual devices and attempting to adapt the techniques of VDF to testing other types of software. A summary of those limitations, as mentioned throughout this chapter, are listed below:

1. VDF will only fuzz test a subsystem that has been instrumented for record and replay. This is a simple matter for most virtual devices, as MMIO provides a convenient entry point to the subsystem. However, if there are multiple methods of triggering subsystem functionality (i.e. multiple API functions exposed that each trigger a different piece of functionality), the record and replay mechanism must be extended to capture each function's arguments in a format suitable for fuzz testing.

2. VDF does not take hardware features such as timers, interrupts, or DMA into account when fuzzing virtual devices. While much of a virtual device's functionality can be exercised via MMIO, some devices may have specific

functionality that is only triggered when these hardware features are used. This limits the maximum branch coverage that can be acheved using VDF. However, VDF does perform fuzzing that mimics the MMIO attack model proposed in Section 4.2. The expansion of the attack model to incorporate these additional hardware features is left as a future work.

3. VDF does not incorporate the timing of MMIO events during its record and replay. As discussed in Section 4.3.2, including a timestamp within the event record exposes that timestamp to the possibility of mutation during fuzzing, making the addition of timing information difficult. This limitation does not limit VDF's ability to test, though it may limit the ability of VDF to reach a virtual device state that is time-dependent.

4. VDF reduces discovered crash/hang test cases to a minimal test case still capable of reproducing the issue. However, the issue must be consistently triggered by the test case during the test case reduction process described in Section 4.3.4. For issues that are not consistently reproducable (such as thread race conditions), additional event records may be present in the final test case, resulting in non-minimal test cases.

4.6 Related Work

Fuzzing has been a well-explored research topic for a number of years. The original fuzzing paper [73] used random program inputs as seed data for testing Unix utilities. Later studies on the selection of proper fuzzing seeds [38, 79] and the use of fuzzing to discover software vulnerabilities [26] have both been used to improve the coverage and

discovery of bugs in programs undergoing fuzz testing. By relying on the record and replay of virtual device activity, VDF provides proper seed input that is known to execute branches of interest within the virtual device under test.

A number of tools utilize record and replay to analyze programs and systems. ReVirt [51] records system events to replay the activity of compromised guest systems to better analyze the nature of the attack. Aftersight [42] records selected system events and then offloads those events to another system for replay and analysis. Its primary contribution of decoupled analysis demonstrates the ability for record and replay to facilitate repeated heavyweight analysis that does not occur at the moment that the event under analysis originally occurred. PANDA [50], a much more recent work in this area, is a dynamic analysis tool that uses a modified QEMU to record non-deterministic events that occur system-wide within a guest system. These events are then replayed through increasingly heavier-weight analysis plugins to reverse engineer the purpose and behavior of arbitrary portions of the guest.

Symbolic execution of complex programs is also a common technique to calculate the path predicates and conditionals needed to exercise branches of interest. KLEE [36] performs symbolic execution at the process level. Selective Symbolic Execution (S2E) [40] executes a complete guest environment under QEMU leverages the previous KLEE work to perform symbolic execution at the whole-system level. The approach proposed by Cong et al [44] attempts to extract the code for five network virtual devices from QEMU, stub out key QEMU datatypes, and then perform symbolic execution on the resulting code. VDF is capable of performing its testing and analysis of a much larger set of virtual devices, within the context of QEMU, without requiring the effort of extracting

and stubbing the virtual device code. However, the techniques laid out in [44] could benefit VDF by generating new seed test cases designed to augment VDF's ability to reach new branches of interest.

Driller [83] uses both white box fuzzing and symbolic execution to discover vulnerabilities within programs. Unlike VDF, which is interested in focused fuzzing to explore branches of interest, Driller seeks to explore all branches within a program. It focuses on switching back and forth between symbolic execution and fuzzing when fuzzing gets "stuck" and can no longer discover data values that explore new branches. VDF focuses on executing large numbers of fuzzing test cases without using expensive instruction tracing and path conditional calculations to create new seeds.

Forced execution tools, such as X-Force [76], attempt to explore new branches of execution by changing runtime data to force specific branches to be taken within a program. Program context that has not yet been created or initialized, such as pointer references and the contents of memory buffers, are dynamically created during execution with just enough data to allow execution to continue. However, forced execution seeks to explore new branches of a program while avoiding crashing the program. VDF explores branches of interest, while executing those branches within a complete program context, while actively seeking input that can crash or hang the program.

The discovery of vulnerable code is a difficult and ongoing process, and there is interest in research work orthogonal to VDF that seeks to protect the host system and harden hypervisors. DeHype [86] reduces the privileged attack surface of KVM by deprivileging 93.2% of the KVM hypervisor code from kernel space to user space on the host. The Qubes OS project [15] compartimentalizes software into a variety of VMs,

allowing the isolation of trusted activities from trusted ones within the OS. Qubes relies upon the bare-metal Xen hypervisor, which is much harder to exploit than a hypervisor executing under the host OS (like QEMU or KVM).

5. SUMMARY

Dynamic analysis is the observation and modification of a running system for the purpose of understanding the runtime behavior of the system. It has demonstrated its strength in many research problems, such as malware analysis, protocol reverse engineering, vulnerability signature generation, software testing, profiling, and performance optimization. Compared to process-level binary instrumentation and analysis, whole-system dynamic binary analysis has unique advantages. First, it provides a full system view, including the OS kernel and all running applications, allowing the analysis of kernel activity and the interactions among multiple user-space processes. Second, the code instrumentation and analysis are performed from entirely outside of the context of the guest system under analysis (typically executing within a virtual machine (VM)).

Building a generic, whole-system dynamic binary analysis platform that can instrument any portion of the guest's execution is desirable, but challenging. Unless system-wide dynamic analysis is performed at a reasonable speed, it is useless. Observation of time-sensitive runtime events, such as network communications or GUI interactions, is one of the primary reasons to use dynamic analysis over static analysis methods. Time-sensitive events must be performed in a timely fashion within an instrumented guest to be useful and representative of their non-instrumented execution.

This dissertation states the thesis that it is possible to unobtrusively and dynamically analyze a subset of whole-system execution as that subset executes within the context of a

virtualized guest environment. To that end, this dissertation presents the design and evaluation of two new and novel tools for the dynamic analysis of software: DECAF and VDF. The primary intended purpose of DECAF, presented in Chapter 3, is the transparent observation and analysis of the behaviors of malicious software (malware) via whole-system dynamic analysis. VDF, presented in Chapter 4, is a fuzz testing framework designed to test virtual devices, such as those seen in DECAF and QEMU. This is done by exercising the MMIO interfaces of those devices as they interact with the guest environment and discovering vulnerabilities that are susceptible to attack by malicious software executing within the guest.

DECAF's performance and functionality evaluation successfully demonstrates that the tool is capable of performing system-wide data flow analysis that is both sound and precise. The evaluation also proved that DECAF is capable of utilizing a novel, hardware-based VMI solution to aid in detecting and analyzing key loggers, buffer overflows, rootkits, and other behaviors commonly exhibited by malware. VDF's evaluation successfully demonstrates that it is capable of discovering and aiding in the analysis of vulnerabilities seen in virtual devices. This not only helps to protect QEMU from attack from a malicious guest environment, but it also does the same for any analysis framework (such as DECAF) that extends QEMU.

In conclusion, both DECAF and VDF perform selective dynamic analysis using a collection of novel techniques. These tools improve upon the current state of the art, providing empirical results applicable to real-world problems. Future work, as mentioned throughout the dissertation, will be required to address the limitations of each tool and extend their functionalities to handle additional hardware features and architectures.

APPENDICES

A. RULE CONSTRUCTION AND VERIFICATION: A 2-BIT `AND` EXAMPLE

The `and` instruction is a good candidate for illustrating the different stages of rule construction and verification. The simple in-place flow type plus the straight forward taint propagation rule means the analysis for a 2-bit `and` is equivalent to the analysis for 32-bit `and`, as well as other bit lengths.

The SMT2 declarations for revealing the flow-type of the 2-bit `and` is listed in Figure A.1. Lines 1 to 4 declare a new sort named STATE that is an alias for a bitvector of length 5 (2 for *dst*, 2 for *src* and 1 for *zf*[1]), as well as functions to extract the corresponding bits of the state. Helper functions are defined on lines 6, 8, and 10. The update function which returns the new state value given an input state is defined on line 12. Similar declarations of helper functions and final update functions are defined for all of the instructions that are listed in Table 3.2.

Recall that Definition 3.5.1 tests whether there is a system state where changing a single bit of the input will result in a change in the output. All possible system states are tested by declaring the input state components as free variables on lines 15-19.

Once all preparatory declarations have been made, all possible input-to-output bit-wise combinations are iterated through and apply Definition 3.5.1. Since the 2-bit `and` instruction has 4 bits of input and 5 bits of output, there are a total of 20 possible

[1]We removed *sf* and *pf* for brevity.

```
 1. (define-sort STATE () (_ BitVec 5))
 2. (define-fun dst ((S STATE)) (_ BitVec 2) ((_ extract 4 3) S) )
 3. (define-fun src ((S STATE)) (_ BitVec 2) ((_ extract 2 1) S) )
 4. (define-fun zf ((S STATE)) (_ BitVec 1)  ((_ extract 0 0) S) )
 5.
 6. (define-fun f_bool2bv ((b bool)) (_ BitVec 1) (ite b #b1 #b0) )
 7.
 8. (define-fun f_and ((S STATE)) (_ BitVec 2) (bvand (dst S) (src S)) )
 9.
10. (define-fun f_zf ((S STATE)) (_ BitVec 1) (f_bool2bv (= (f_and S) #b00)) )
11.
12. (define-fun x86_and ((S STATE)) (STATE) (concat (f_and S) (src S) (f_zf S)) )
13.
14. ;---- FREE VARIABLE DECLARATIONS ----
15. (declare-const DST_1 (_ BitVec 1)) ; dst[1:1]
16. (declare-const DST_0 (_ BitVec 1)) ; dst[0:0]
17. (declare-const SRC_1 (_ BitVec 1)) ; src[1:1]
18. (declare-const SRC_0 (_ BitVec 1)) ; src[0:0]
19. (declare-const ZF_0 (_ BitVec 1)) ; zf
20. ;----      END DECLARATIONS      ----
21.
22. ; 1: dst [1:1] -> dst [1:1]
23. (push)
24. (assert (exists ( (i (_ BitVec 1)) (j (_ BitVec 1)) )
25.   (not (= ( (_ extract 1 1) (dst (x86_and (concat i DST_0 SRC_1 SRC_0 ZF_0  ))))
26.           ( (_ extract 1 1) (dst (x86_and (concat j DST_0 SRC_1 SRC_0 ZF_0  ))))
27.       )) ) )
28. (check-sat)
29. (pop)
30. ;sat
31. ; 2: dst [1:1] -> dst [0:0]
32. (push)
33. (assert (exists ( (i (_ BitVec 1)) (j (_ BitVec 1)) )
34.   (not (= ( (_ extract 0 0) (dst (x86_and (concat i DST_0 SRC_1 SRC_0 ZF_0  ))))
35.           ( (_ extract 0 0) (dst (x86_and (concat j DST_0 SRC_1 SRC_0 ZF_0  ))))
36.       )) ) )
37. (check-sat)
38. (pop)
39. ;unsat
40. ; 8: dst [0:0] -> src [1:1]
41. (push)
42. (assert (exists ( (i (_ BitVec 1)) (j (_ BitVec 1)) )
43.   (not (= ( (_ extract 1 1) (src (x86_and (concat DST_1 i SRC_1 SRC_0 ZF_0  ))))
44.           ( (_ extract 1 1) (src (x86_and (concat DST_1 j SRC_1 SRC_0 ZF_0  ))))
45.       )) ) )
46. (check-sat)
47. (pop)
48. ;unsat
49. ;... TRUNCATED ...
```

Fig. A.1.: SMT2 for 2-bit `and`

combinations. Three of these combinations are shown on lines 23-29, 32-38, and 41-47. In the first two test cases, there is a query whether there exists two assignments to bit 1 of *dst* (the most significant bit) such that bits 1 and 0 of the resulting *dst* are different. The first query returns *sat* (line 30) while the second returns *unsat* (line 39), meaning that there is information flow from the highest bit of *dst* to itself, but not down to the lowest bit. The

third test case illustrates how the query is changed to determine whether there is information flow from bit 0 of *dst* (it is now replaced with *i* and *j*) to bit 1 of *src* (the bit being *extract*'ed).

The complete sat/unsat results are summarized in Table A.1. The in-place flow type for *dst* is evident from the first two columns as information only flows from a bit from either *src* or *dst* to the same bit in the *dst*. The third and fourth columns show that there is only information flow from *src* to itself because it is unchanged by the instruction. The final column indicates that the status of the zero flag changes based on changes from either operand as expected.

Table A.1: Query Results for 2-bit `and`

	dst[1:1]	**dst[0:0]**	**src[1:1]**	**src[0:0]**	**zf**
dst[1:1]	sat	unsat	unsat	unsat	sat
dst[0:0]	unsat	sat	unsat	unsat	sat
src[1:1]	sat	unsat	sat	unsat	sat
src[0:0]	unsat	sat	unsat	sat	sat

Once the flow-type is understood, a taint propagation rule is defined and a check performed on its soundness and precision as listed in Figure A.2. Since the flow-type of `and` is *in-place*, the most basic sound rule would be "a resulting bit is tainted if either of the corresponding input bits are tainted"; however, this is known to be imprecise since it does not take short-circuiting into account. The `and` of any bit with 0 will always be 0. Therefore, an untainted 0 bit `and`ed with any other bit (even if it is tainted) will always result in 0. In accordance with Definition 3.5.1, the resulting bit should be untainted. This special behavior is embedded into the taint propagation rule defined on lines 3-7.

```
 1. (define-fun f_and ( (x (_ BitVec 2)) (y (_ BitVec 2)) ) (_ BitVec 2) (bvand x y))
 2.
 3. (define-fun f_rule ( (x (_ BitVec 2)) (y (_ BitVec 2)) (x_t (_ BitVec 2)) (y_t (_ BitVec 2)) ) (_ BitVec 2)
 4.   (bvand (bvor x_t y_t)  ; either x or y is tainted
 5.          (bvand (bvor x x_t) ; unless x is 0 and not tainted
 6.                 (bvor y y_t) ; or unless y is 0 and not tainted
 7.          ) ) )
 8.
 9. ; A rule is precise if a bit of the result is NOT tainted implies all possible
10. ; re-assignments of tainted bits will not change the value of that bit
11. (define-fun isSound ( (x (_ BitVec 2)) (y (_ BitVec 2)) (x_t (_ BitVec 2)) (y_t (_ BitVec 2))
12.                       (result_t (_ BitVec 2)) ) (Bool)
13.   (and (implies (= #b0 ((_ extract 1 1) result_t) )
14.          (forall ( (i (_ BitVec 2)) (j (_ BitVec 2)) (k (_ BitVec 2)) (l (_ BitVec 2)) )
15.            (=
16.              ( (_ extract 1 1) (f_and (bvor (bvand i x_t) (bvand (bvnot x_t) x))
17.                                       (bvor (bvand k y_t) (bvand (bvnot y_t) y)) ) )
18.              ( (_ extract 1 1) (f_and (bvor (bvand j x_t) (bvand (bvnot x_t) x))
19.                                       (bvor (bvand l y_t) (bvand (bvnot y_t) y)) ) )
20.            ) ) )
21.        (implies (= #b0 ((_ extract 0 0) result_t) )
22.          (forall ( (i (_ BitVec 2)) (j (_ BitVec 2)) (k (_ BitVec 2)) (l (_ BitVec 2)) )
23.            (=
24.              ( (_ extract 0 0) (f_and (bvor (bvand i x_t) (bvand (bvnot x_t) x))
25.                                       (bvor (bvand k y_t) (bvand (bvnot y_t) y)) ) )
26.              ( (_ extract 0 0) (f_and (bvor (bvand j x_t) (bvand (bvnot x_t) x))
27.                                       (bvor (bvand l y_t) (bvand (bvnot y_t) y)) ) )
28.            ) ) )
29.   ) )
30.
31. ; A rule is not precise if there are exists a tainted bit in the result,
32. ; but all possible assignments of tainted input bits do not change the value
33. ; of the resulting bit that was tainted
34. (define-fun isNotPrecise ( (x (_ BitVec 2)) (y (_ BitVec 2)) (x_t (_ BitVec 2)) (y_t (_ BitVec 2))
35.                           (result_t (_ BitVec 2)) ) (Bool)
36.   (or (and (= #b1 ((_ extract 1 1) result_t) )
37.         (forall ( (i (_ BitVec 2)) (j (_ BitVec 2)) (k (_ BitVec 2)) (l (_ BitVec 2)) )
38.           (=
39.             ( (_ extract 1 1) (f_and (bvor (bvand i x_t) (bvand (bvnot x_t) x))
40.                                      (bvor (bvand k y_t) (bvand (bvnot y_t) y)) ) )
41.             ( (_ extract 1 1) (f_and (bvor (bvand j x_t) (bvand (bvnot x_t) x))
42.                                      (bvor (bvand l y_t) (bvand (bvnot y_t) y)) ) )
43.           ) ) )
44.       (and (= #b1 ((_ extract 0 0) result_t) )
45.         (forall ( (i (_ BitVec 2)) (j (_ BitVec 2)) (k (_ BitVec 2)) (l (_ BitVec 2)) )
46.           (=
47.             ( (_ extract 0 0) (f_and (bvor (bvand i x_t) (bvand (bvnot x_t) x))
48.                                      (bvor (bvand k y_t) (bvand (bvnot y_t) y)) ) )
49.             ( (_ extract 0 0) (f_and (bvor (bvand j x_t) (bvand (bvnot x_t) x))
50.                                      (bvor (bvand l y_t) (bvand (bvnot y_t) y)) ) )
51.           ) ) )
52.   ) )
53.
54. (declare-const x (_ BitVec 2))
55. (declare-const y (_ BitVec 2))
56.
57. (push)
58. (assert
59.   (not
60.     (forall ( (x_t (_ BitVec 2)) (y_t (_ BitVec 2)) )
61.       (isSound x y x_t y_t (f_rule x y x_t y_t))
62.     )
63.   )
64. )
65. (check-sat)
66. ;unsat
67. (pop)
68.
69. (assert
70.   (exists ( (x_t (_ BitVec 2)) (y_t (_ BitVec 2)) )
71.     (isNotPrecise x y x_t y_t (f_rule x y x_t y_t))
72.   )
73. )
74. (check-sat)
75. ;unsat
```

Fig. A.2.: SMT2 for verifying the 2-bit `and` rule

To determine whether a rule is *sound*, a function `isSound` (lines 11-29) is defined that ensures that if a bit of the result is untainted, then all possible assignments of tainted

```
(model
  (define-fun y_t!10 () (_ BitVec 2)
    #b10)
  (define-fun y () (_ BitVec 2)
    #b00)
  (define-fun x () (_ BitVec 2)
    #b00)
  (define-fun x_t!11 () (_ BitVec 2)
    #b00)
)
```

Fig. A.3.: Sat model for simple 2-bit `and` rule

input bits will not change the value of the corresponding output bit. A similar function is defined to determine whether a rule is *not precise* (lines 34-52) by querying if there are tainted bits of the output that do not change for all possible assignments of the tainted inputs.

The soundness and precision queries are listed on lines 58-65 and 69-74, respectively. Both queries return *unsat*, meaning that the rule is both sound and precise. The simple rule can also be verified by replacing lines 4-6 with `(bvor x_t y_t)`. This new query returns *unsat* and *sat*, respectively, meaning that the rule is sound, but not precise. The satisfying model is depicted in Figure A.3 and shows this short-circuiting behavior. That is, if bit 1 of `x` is 0 and untainted (bit 1 of `x_t` is also 0), and bit 1 of `y` is tainted, then the simple rule is wrong; bit 1 of the result cannot be tainted.

B. VDF SAMPLE FUZZING RESULTS: SDHCI VIRTUAL DEVICE

QEMU provides a Secure Digital Host Controller Interface [21] (SDHCI) virtual device to allow emulated guest OSes to interact with SD and microSD media device images that exist on the host. This Appendix provides a subset of the raw data captured by VDF during its testing of the SDHCI virtual device. The full test took over ten days to complete, but the presented subset is only for the first 36 hours of testing.

VDF utilized six fuzzing instances executing concurrently during this test. These fuzzers, named `fuzzer01` through `fuzzer06`, all worked from the same initial seed event replay data. `fuzzer01` was configured as a *master* fuzzer instance, while fuzzers `02` though `06` were configured as *slave* instances. The primary difference between the master fuzzer and slave fuzzer instances are that the master attempts to perform smart mutations (bit/byte swaps, setting whole bytes to `0x00` or `0xFF`, etc.) of the seed data to explore paths of interest. Slave fuzzers will only randomly mutate the seed data (known as a *havoc* mutation).

Figure B.1 shows the number of crashes and hangs discovered while fuzz testing the SDHCI virtual device. Figure B.2 shows the number of paths (series of branches) discovered and explored. Periodically, the fuzzer instances will share their results with each other to notify other fuzzers of any newly discovered branches of interest. This produces a distinctive stair-step pattern in the graph data (Figure B.2), which shows the

total and current number of discovered paths suddenly increasing when the master `fuzzer01` synchronizes with the slave instances.

An interesting observation of the data empirically gathered during the SDHCI testing is that slave fuzzer instances tend to discover new branches of interest and crash/hangs faster than the master fuzzer instance does. One hypothesis is that new branches of interest are so easily discovered by random mutation due to VDF's filtering of invalid events and its focusing of the mutated data into the device callback functions.

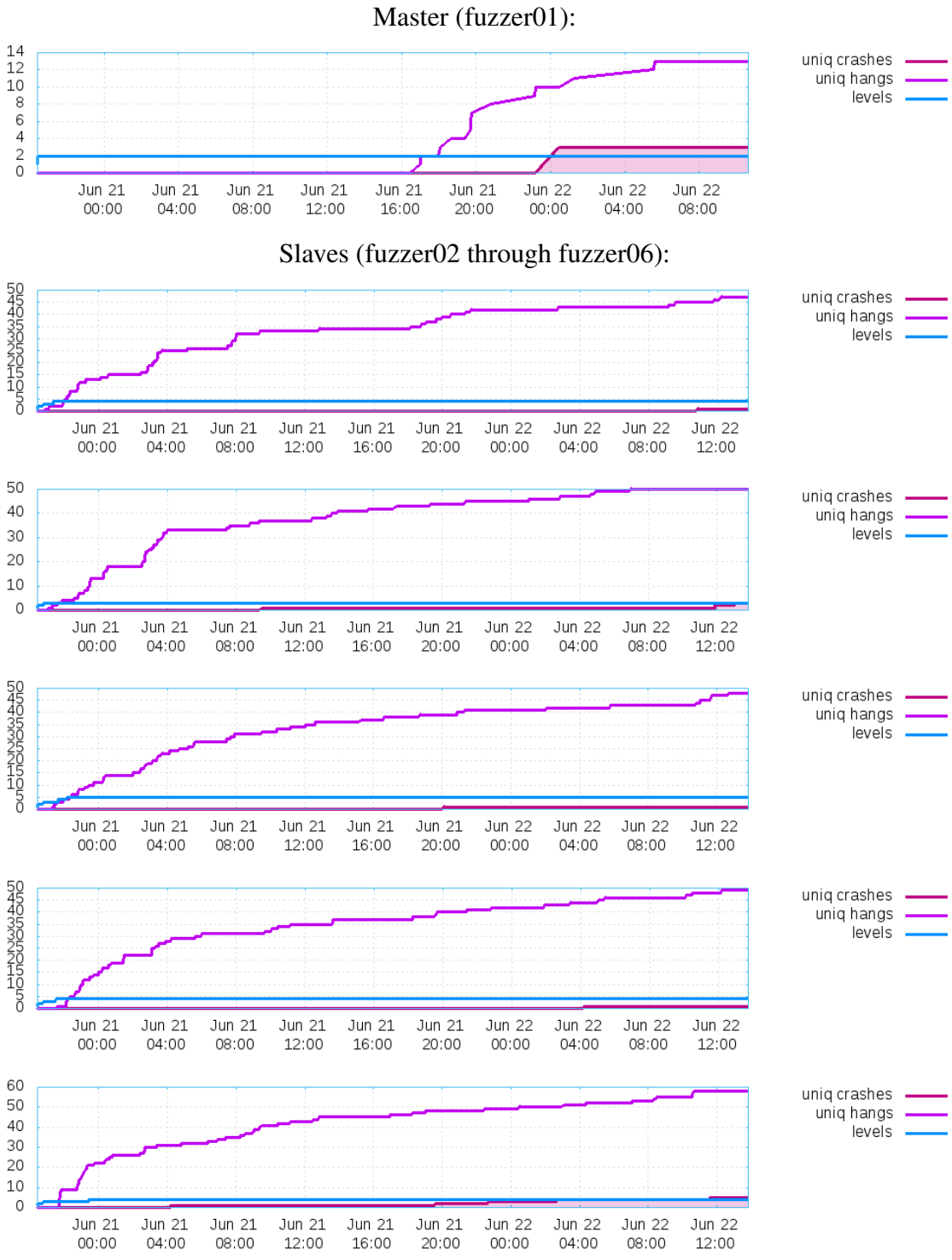


Fig. B.1.: Function call depth and test cases triggering crashes and hangs during the fuzzing of the SDHCI virtual device in QEMU source file `hw/sd/sdhci.c`.

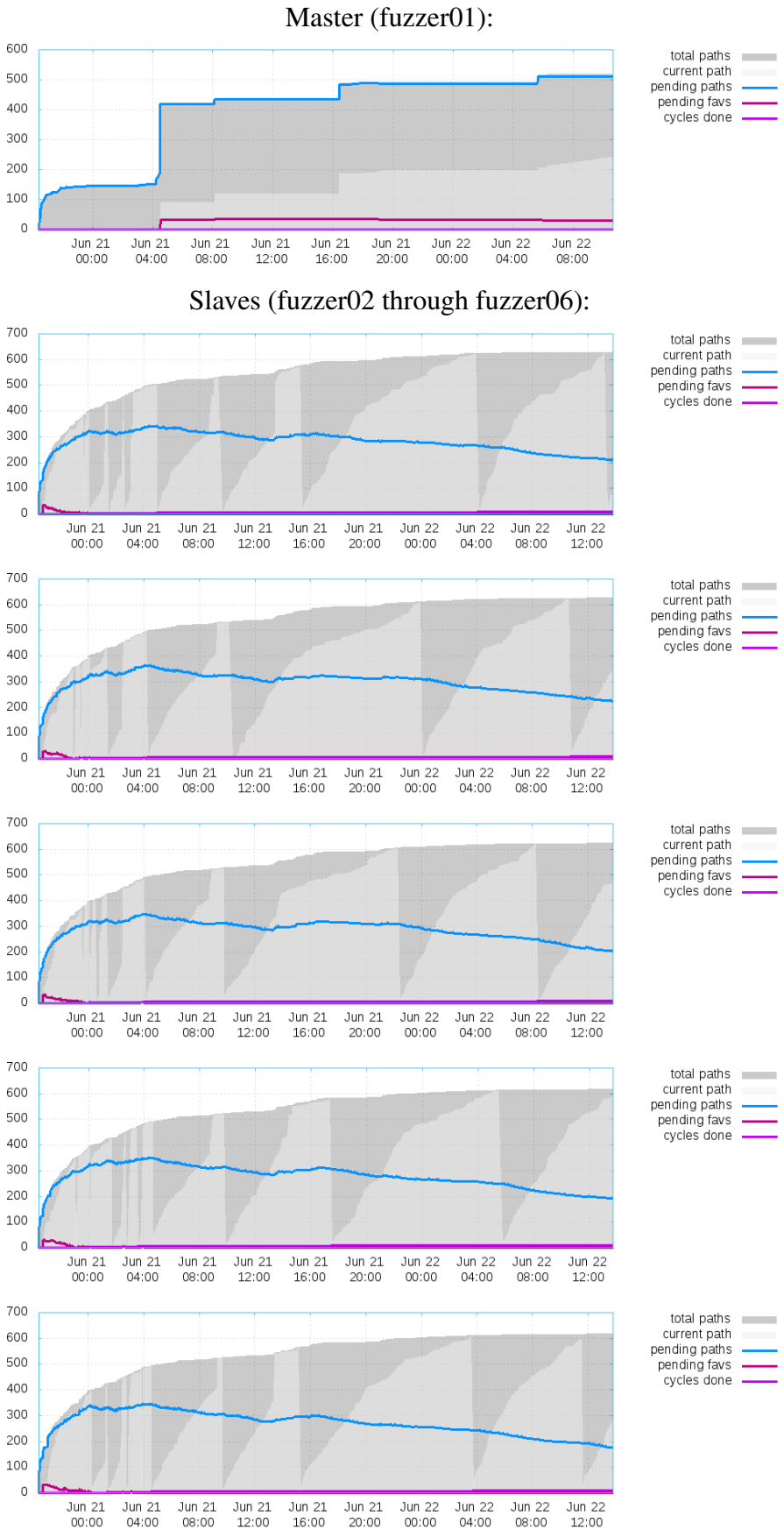


Fig. B.2.: Discovered, explored, and pending paths during the fuzzing of the SDHCI virtual device in QEMU source file `hw/sd/sdhci.c`.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Advanced Linux Sound Architecture (ALSA). URL http://www.alsa-project.org.

[2] Amazon.com, Inc. Form 10-K 2015. URL http://www.sec.gov/edgar.shtml.

[3] Bochs: The Cross Platform IA-32 Emulator. URL http://bochs.sourceforge.net/.

[4] CVE-2014-2894: Off-by-one error in the cmd start function in smart self test in IDE core, . URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2894.

[5] CVE-2015-3456: Floppy disk controller (FDC) allows guest users to cause denial of service, . URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456.

[6] CVE-2015-5279: Heap-based buffer overflow in NE2000 virtual device, . URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5279.

[7] CVE-2015-6855: IDE core does not properly restrict commands, . URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6855.

[8] CVE-2016-1981: Reserved, . URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1981.

[9] [Qemu-devel] [PATCH] e1000: eliminate infinite loops on out-of-bounds transfer start, . URL https://lists.gnu.org/archive/html/qemu-devel/2016-01/msg03454.html.

[10] DECAF Binary Analysis Platform. URL https://github.com/sycurelab/decaf/.

[11] Kernel-Based Virtual Machine. URL http://www.linux-kvm.org/.

[12] Valgrind: Project Suggestions. URL http://valgrind.org/help/projects.html.

[13] PCI - OSDev Wiki. URL http://wiki.osdev.org/PCI.

[14] [Qemu-devel] [PATCH 1/5] Provide support for the CUSE TPM. URL https://lists.nongnu.org/archive/html/qemu-devel/2015-04/msg01792.html.

[15] Qubes OS Project. URL https://www.qubes-os.org/.

[16] TEMU: The BitBlaze Dynamic Analysis Component. URL http://bitblaze.cs.berkeley.edu/temu.html.

[17] VMWare. URL http://www.vmware.com.

[18] [Qemu-devel] [PATCH 1/2] hw/sd: implement CMD23 (SET_BLOCK_COUNT) for MMC compatibility. URL https://lists.gnu.org/archive/html/qemu-devel/2015-12/msg00948.html.

[19] Features/QTest. URL http://wiki.qemu.org/Features/QTest.

[20] TrouSerS - The open-source TCG software stack. URL http://trousers.sourceforge.net.

[21] SD Host Controller Simplified Specification Version 2.00. Technical report, SD Association, 2007. URL https://www.sdcard.org/developers/overview/host_controller/simple_spec/Simplified_SD_Host_Controller_Spec.pdf.

[22] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA : A Low-level Virtual Instruction Set Architecture. In *Proceedings of the 36th annual ACM/IEEE International Symposium on Microarchitecture (MICRO-36)*, San Diego, CA, 2003.

[23] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *Proceedings of PLDI 1990*, 1990.

[24] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1986. ISBN 0-201-10088-6.

[25] A. Alwabel, H. Shi, G. Bartlett, and J. Mirkovic. Safe and Automated Live Malware Experimentation on Public Testbeds. In *7th Workshop on Cyber Security Experimentation and Test (CSET 14)*, 2014. URL https://www.usenix.org/conference/cset14/workshop-program/presentation/alwabel.

[26] T. Avgerinos, S. K. Cha, B. Lim, T. Hao, and D. Brumley. AEG : Automatic Exploit Generation. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2011.

[27] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164, dec 2003. ISSN 01635980. doi: 10.1145/1165389.945462. URL http://portal.acm.org/citation.cfm?doid=1165389.945462.

[28] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard - Version 2.0, 2015.

[29] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, Freenix Track*, pages 41–46, 2005. URL http://static.usenix.org/events/usenix05/tech/freenix/full_papers/bellard/bellard_html/.

[30] S. Berger. libtpms library. URL https://github.com/stefanberger/libtpms.

[31] E. Bosman, A. Slowinska, and H. Bos. Minemu : The World's Fastest Taint Tracker. In *Recent Advances in Intrustion Detection*, pages 1–20. Springer, 2011.

[32] D. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, 2004. URL http://groups.csail.mit.edu/cag/rio/derek-phd-thesis.pdf.

[33] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A Binary Analysis Platform. pages 463–469. 2011. doi: 10.1007/978-3-642-22110-1_37. URL http://link.springer.com/10.1007/978-3-642-22110-1_37.

[34] C. Bryant. [1/4] tpm: Add TPM NVRAM Implementation, 2013. URL https://patchwork.ozlabs.org/patch/288936/.

[35] J. Caballero, N. M. Johnson, S. Mccamant, and D. Song. Binary Code Extraction and Interface Identification for Security Applications. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2010.

[36] C. Cadar, D. Dunbar, and D. Engler. KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th symposium on Operating systems design and implementation*, pages 209–224. USENIX Association, 2008.

[37] C. Carmony, M. Zhang, X. Hu, A. V. Bhaskar, and H. Yin. Extract Me If You Can: Abusing PDF Parsers in Malware Detectors. In *NDSS*, number February, pages 21–24, 2016. ISBN 189156241X. doi: 10.14722/ndss.2016.23483. URL http://dl.acm.org/citation.cfm?id=2664243.2664248http://dl.acm.org/citation.cfm?doid=2664243.2664248.

[38] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394. IEEE, may 2012. ISBN 978-1-4673-1244-8. doi: 10.1109/SP.2012.31. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6234425.

[39] V. Chipounov. *S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems*. PhD thesis, Ecole Polytechnique Federale De Lausanne, 2014.

[40] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective Symbolic Execution. In *Proceedings of Fifth Workshop on Hot Topics in System Dependability*, number June, Lisbon, Portugal, 2009.

[41] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security Symposium*, pages 321–336, 2004.

[42] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conference*, pages 1–14, 2008.

[43] J. Clause, W. Li, and A. Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 international symposium on Software testing and analysis - ISSTA '07*, pages 196–206, New York, New York, USA, 2007. ACM Press. ISBN 9781595937346. doi: 10.1145/1273463.1273490. URL http://portal.acm.org/citation.cfm?doid=1273463.1273490.

[44] K. Cong, F. Xie, and L. Lei. Symbolic Execution of Virtual Devices. In *2013 13th International Conference on Quality Software*, pages 1–10. IEEE, jul 2013. ISBN 978-0-7695-5039-8. doi: 10.1109/QSIC.2013.44. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6605903.

[45] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*. O' Reilly Media, Inc., Sebastopol, CA, third edition, 2005. ISBN 978-0-596-00590-3.

[46] J. Crandall and F. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 221–232. IEEE, 2004. ISBN 0-7695-2126-6. doi: 10.1109/MICRO.2004.26. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1550996.

[47] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, 2008.

[48] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether : Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008. ISBN 9781595938107.

[49] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso : Narrowing the Semantic Gap in Virtual Machine Introspection. In *2011 IEEE Symposium on Security and Privacy*, pages 297–312. IEEE, 2011.

[50] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan. Repeatable Reverse Engineering for the Greater Good with PANDA. Technical report, Columbia University, MIT Lincoln Laboratory, TR CUCS-023-14, 2014.

[51] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, dec 2002. ISSN 01635980. doi: 10.1145/844128.844148. URL http://portal.acm.org/citation.cfm?doid=844128.844148.

[52] J. Elgaard, N. Klarlund, and A. Møller. MONA 1.x: new techniques for WS1S and WS2S. In *Proc. 10th International Conference on Computer-Aided Verification (CAV)*, volume 1427, pages 516–520, 1998. ISBN 3540646086.

[53] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. Mcdaniel, and A. N. Sheth. TaintDroid : An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*. Technical Report NAS-TR-0120-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, USENIX Association, 2010.

[54] Q. Feng, A. Prakash, H. Yin, and Z. Lin. MACE: High-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th Annual Computer Security Applications Conference on - ACSAC '14*, pages 196–205, New York, New York, USA, 2014. ACM Press. ISBN 9781450330053. doi: 10.1145/2664243.2664248. URL http://dl.acm.org/citation.cfm?id=2664243.2664248http://dl.acm.org/citation.cfm?doid=2664243.2664248.

[55] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, jul 1987. ISSN 01640925. doi: 10.1145/24039.24041. URL http://portal.acm.org/citation.cfm?doid=24039.24041.

[56] Y. Fu and Z. Lin. Space Traveling across VM : Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *2012 IEEE Symposium on Security and Privacy*, 2012. doi: 10.1109/SP.2012.40.

[57] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Proceedings of the International Conference in Computer Aided Verification (CAV 2007)*, pages 524–536, Berlin, Germany, 2007.

[58] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2003.

[59] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20, jan 2012. ISSN 15427730. doi: 10.1145/2090147.2094081. URL `http://dl.acm.org/citation.cfm?doid=2090147.2094081`.

[60] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*, pages 1–11. IEEE, apr 1982. ISBN 0-8186-0410-7. doi: 10.1109/SP.1982.10014. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6234468`.

[61] S. Golovanov. Analysis of TDSS rootkit technologies, 2010. URL `https://securelist.com/analysis/publications/36314/tdss/`.

[62] A. Henderson, L. K. Yan, X. Hu, A. Prakash, H. Yin, and S. McCamant. DECAF : A Platform-Neutral Whole-System Dynamic Binary Analysis Platform. *To appear in IEEE Transactions on Software Engineering*.

[63] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, pages 248–258, New York, New York, USA, 2014. ACM Press. ISBN 9781450326452. doi: 10.1145/2610384.2610407. URL `http://dl.acm.org/citation.cfm?doid=2610384.2610407`.

[64] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual, vol. 1-3*. Number 253665, 325383, 325384. 2016. URL `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html`.

[65] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis. A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2012.

[66] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection Through VMM-Based Out-of-the-Box Semantic View Reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138. ACM, 2007. ISBN 9781595937032.

[67] M. G. Kang, P. Poosankam, and H. Yin. Renovo : A Hidden Code Extractor for Packed Executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 46–53. ACM, 2007. ISBN 9781595938862.

[68] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA ++ : Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2011.

[69] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments - VEE '12*, pages 121–132, New York, New York, USA, 2012. ACM Press. ISBN 9781450311762. doi: 10.1145/2151024.2151042. URL http://dl.acm.org/citation.cfm?doid=2151024.2151042.

[70] C.-k. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and K. Hazelwood. Pin : Building Customized Program Analysis Tools. In *Proceedings of PLDI 2005*, volume 40, 2005. ISBN 1595930566.

[71] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU emulators. In *Proceedings of the eighteenth international symposium on Software testing and analysis - ISSTA '09*, page 261, New York, New York, USA, 2009. ACM Press. ISBN 9781605583389. doi: 10.1145/1572272.1572303. URL http://doi.acm.org/10.1145/1572272.1572303http://portal.acm.org/citation.cfm?doid=1572272.1572303.

[72] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-Exploration Lifting : Hi-Fi Tests for Lo-Fi Emulators. In *Proceedings of the 17th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 337–348, 2012. ISBN 9781450307598. doi: 10.1145/2150976.2151012. URL http://dl.acm.org/citation.cfm?id=2151012.

[73] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990. ISSN 00010782. doi: 10.1145/96267.96279. URL http://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf.

[74] N. Nethercote and J. Seward. Valgrind : A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of PLDI 2007*, volume 42, 2007. ISBN 9781595936332.

[75] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection , Analysis , and Signature Generation of Exploits on Commodity Software. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2005.

[76] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-Force: Force-Executing Binary Programs for Security Applications. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 829–844, 2014. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/peng.

[77] G. Portokalidis, A. Slowinska, and H. Bos. Argos : an Emulator for Fingerprinting Zero-Day Attacks for advertised honeypots with automatic signature generation. *ACM SIGOPS Operating Systems Review*, 40(4):15–27, 2006.

[78] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 135–148. IEEE, dec 2006. ISBN 0-7695-2732-9. doi:

10.1109/MICRO.2006.29. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4041842.

[79] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing Seed Selection for Fuzzing. In *23rd USENIX Security Symposium*, 2014. ISBN 9781931971157.

[80] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *18th Annual Computer Security Applications Conference, 2002. Proceedings.*, pages 209–218. IEEE Computer Society, 2002. ISBN 0-7695-1828-1. doi: 10.1109/CSAC.2002.1176292. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1176292.

[81] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference*, pages 17–30, 2005.

[82] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, and P. Poosankam. BitBlaze : A New Approach to Computer Security via Binary Analysis. In *Information Systems Security*, pages 1–25. Springer, 2008.

[83] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of NDSS 2016*, number February, 2016. ISBN 189156241X. doi: 10.14722/ndss.2016.23368.

[84] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *2010 IEEE Symposium on Security and Privacy*, number June, pages 497–512. IEEE, 2010. ISBN 978-1-4244-6894-2. doi: 10.1109/SP.2010.37. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5504701.

[85] J. Wei, L. K. Yan, and M. A. Hakim. MOSE: Live Migration Based On-the-Fly Software Emulation. In *Proceedings of the 31st Annual Computer Security Applications Conference on - ACSAC 2015*, pages 221–230, New York, New York, USA, 2015. ACM Press. ISBN 9781450336826. doi: 10.1145/2818000.2818022. URL http://dl.acm.org/citation.cfm?doid=2818000.2818022.

[86] C. Wu, Z. Wang, and X. Jiang. Taming Hosted Hypervisors with (Mostly) Deprivileged Execution. In *Network and Distributed System Security Symposium*, 2013.

[87] L. K. Yan. *Transparent and Precise Malware Analysis Using Virtualization: From Theory To Practice*. PhD thesis, Syracuse University, 2013.

[88] L. K. Yan and H. Yin. DroidScope : Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[89] L. K. Yan, A. Henderson, X. Hu, H. Yin, and S. McCamant. On Soundness and Precision of Dynamic Taint Analysis. Technical report, Syracuse University, 2014.

[90] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama : Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127. ACM, 2007. ISBN 9781595937032.

[91] H. Yin, Z. Liang, and D. Song. HookFinder : Identifying and Understanding Malware Hooking Behaviors. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2008.

[92] M. Zalewski. American Fuzzy Lop Fuzzer. URL `http://lcamtuf.coredump.cx/afl/`.

VITA

VITA

Andrew W. Henderson was born in Corning, New York, USA. He received his Bachelor of Science degree in Computer Science at Embry-Riddle Aeronautical University (Daytona Beach, Florida, USA). He received his Masters of Business Administration degree from Jacksonville University (Jacksonville, Florida, USA). He received his PhD in Electrical and Computer Engineering from Syracuse University (Syracuse, New York, USA) in December 2016.