PERFORMANT SOFTWARE HARDENING UNDER HARDWARE SUPPORT

A Dissertation
Presented to
The Academic Faculty

By

Ren Ding

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science
College of Computing

Georgia Institute of Technology

May 2021

PERFORMANT SOFTWARE HARDENING UNDER HARDWARE SUPPORT

Thesis committee:

Dr. Taesoo Kim (Advisor)
School of Computer Science
Georgia Institute of Technology

Dr. Brendan D. Saltaformaggio
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Wenke Lee
School of Computer Science
Georgia Institute of Technology

Dr. Yeongjin Jang
School of Electrical Engineering and Computer Science
Oregon State University

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Date approved: April 26, 2021

“To you, 2000 years from now."

Hajime Isayama

For my family and myself.

ACKNOWLEDGMENTS

My Ph.D. journey would have never come to an end so easily without the help from a number of people. First, I would like to express my deepest gratitude to my advisor, Prof. Taesoo Kim, who has guided me and pushed me through the challenges in the past six years. Despite all the ups and downs in research, he has never left me behind, with his technical insights and concrete instructions. He has also shaped me as a person with his work ethics, preparing me for bigger challenges once I step out of school. I am forever grateful for his mentorship, as he is the person who has led me to who I am today.

Second, I would like to thank my thesis committee members for their invaluable comments on my dissertation. Particularly, I would like to thank Prof. Wenke Lee, who has introduced me into the research of computer security, for his academic and career advice throughout my master program. My sincere gratitude also goes to the other committee members: Prof. Alessandro Orso, Prof. Brendan Saltaformaggio, and Prof. Yeongjin Jang. Thank you so much for being by my side towards the end of the journey.

I am also fortunate enough to have met my brilliant colleagues and have the chance to collaborate with some of them while I have been at Georgia Tech. Given this opportunity, I would like to show my special appreciation to: Prof. Byoungyoung Lee, Prof. Chengyu Song, Prof. Kangjie Lu, Prof. Daehee Jang, Prof. Meng Xu, Prof. Sanidhya Kashyap, Prof. Hong Hu, Prof. Insu Yun, Dr. Bill Harris, Dr. Sangho Lee, Dr. Chanil Jeon, Dr. Yang Ji, Dr. Ming-Wei Shih, Dr. Ruian Duan, Dr. Jinho Jung, Chenxiong Qian, Kyuhong Park, Wen Xu, Fan Sang, Seulbae Kim, Hanqing Zhao, Yonghae Kim, and Gururaj Saileshwar. In addition, I am also grateful to our supporting staff, Elizabeth Ndongi, Sue Jean Chae, and Trinh Doan for their help in the administrative work. It has always been a pleasure to work alongside you all.

I would like to thank my parents in China, Hong Ding and Yan Wang, who have always been patient with me and supported me unconditionally. May you rest in peace, mama. To

my girlfriend Qingqing Tan, and my puppies, Udon and Sushi, thank you for the love and your company throughout the years. I owe you everything that I have and will have in the coming future. Finally, to myself: thank you for your persistence and hard work; you are the only person who can perfectly be you.

TABLE OF CONTENTS

Acknowledgments v

List of Tables xi

List of Figures xii

Summary xiv

Chapter 1: Introduction 1

1.1 Problem Statement 1

1.2 Research Outline 4

Chapter 2: PittyPat: Efficient Protection of Path-Sensitive Control Security . 6

2.1 Introduction 6

2.2 Motivation: Precision and Performance 8

2.2.1 Limitations of existing CFI 9

2.2.2 Path-sensitive CFI 12

2.2.3 Intel Processor Trace 13

2.3 Design 14

2.3.1 Overview 14

2.3.2 Efficient branch sharing 16

2.3.3 Online points-to analysis . . . 17
2.4 Implementation . . . 19
2.4.1 Monitor module . . . 19
2.4.2 Analyzer module . . . 20
2.5 Evaluation . . . 21
2.5.1 Methodology . . . 22
2.5.2 Benchmarks . . . 23
2.5.3 Results . . . 24

Chapter 3: SNAP: Hardware Support to Improve Fuzzing Performance and Precision . . . 30

3.1 Introduction . . . 30
3.2 Motivation: Precision and Performance . . . 33
3.2.1 Better feedback in fuzzing . . . 33
3.2.2 Dissecting AFL's tracing overhead . . . 35
3.3 Design . . . 39
3.3.1 Overview . . . 39
3.3.2 Deep-dive: Implementation of Hardware Primitives . . . 40
3.3.3 Micro-architectural Optimizations . . . 43
3.3.4 Edge Encoding . . . 44
3.3.5 Richer Coverage Feedback . . . 46
3.3.6 OS Support . . . 48
3.4 Evaluation . . . 49
3.4.1 Experimental setup . . . 50

3.4.2 Tracing Overhead by SNAP 51

3.4.3 Evaluating Fuzzing Metrics 53

3.4.4 Practicality of SNAP 56

3.5 Limitations and future directions 56

Chapter 4: HYPERSET: Nested Virtualization Framework for Performant Hypervisor Fuzzing 58

4.1 Introduction 58

4.2 Background 59

4.2.1 x86 Virtualization 59

4.2.2 Nested Virtualization 60

4.2.3 Attack surfaces on hypervisors 61

4.2.4 Fuzzing Non-user Applications 62

4.3 Design 63

4.3.1 Overview 64

4.3.2 Snapshot and Restore 65

4.3.3 Minimized Guest OS 67

4.4 Implementation 69

4.5 Evaluation 70

4.5.1 Experimental Setup 70

4.5.2 Improving Fuzzing Throughput 70

4.5.3 Anatomy of Overheads 71

4.5.4 Testing Real-world Hypervisors 75

4.6 Discussion 76

4.6.1 Limitations 77

4.6.2 Future Directions 77

Chapter 5: Conclusion 79

References 80

LIST OF TABLES

1.1 A survey of hardware-assisted software-hardening techniques in recent years. 2

2.1 Control-relevant trace packets from Intel PT. 13

2.2 "Name" contains the name of the benchmark. "KLoC" contains the number of lines of code in the benchmark. Under "Payload Features," "Exp" shows if the benchmark contains an exploit and "Tm (sec)" contains the amount of time used by the program, when given the payload. Under "π-CFI Featues", "HYPERSET Features," and "CETS+SB Features," "Alarm" contains a flag denoting if a given framework determined that the payload was an attack and aborted; "Overhd (%)" contains the time taken by the framework, expressed as the ratio over the baseline time. 28

3.1 The cost of program size and runtime overhead for tracing on x86 platform across the SPEC benchmarks. 36

3.2 New Control and Status Registers (CSRs) in SNAP. 40

3.3 Evaluated BOOM processor configuration. 50

3.4 Tracing overhead from AFL source instrumentation and SNAP with various L1-D cache sizes across the SPEC benchmarks. 51

3.5 Memory request aggregation rates and L1 cache hit rates between the baseline and SNAP across the SPEC benchmarks. 52

3.6 Estimates of area and power consumption. 56

4.1 Components of the nested virtualization framework with their implementation complexity in lines of code. 69

4.2 Assigned CVEs for bugs that have been found by our system. 76

LIST OF FIGURES

2.1 A motivating example that illustrates the advantages of control-path validity. 9

2.2 The architecture of PITTYPAT. P denotes a target program. The *analyzer* and *driver* modules of PITTYPAT are described in §2.3.1. 14

2.3 Control-transfer targets allowed by π-CFI and PITTYPAT over `403.gcc` and `444.namd`. 24

2.4 A program vulnerable to a COOP attack. 27

3.1 An illustrative example for the runtime information gathered by SNAP. The code abstracts demangling in `cxxfilt`. 34

3.2 Source-instrumented assembly inserted at each basic block between compilers. 35

3.3 Overview of SNAP with its CPU design. The typical workflow involves the components from userspace, kernel, and hardware. The architecture highlights the modified pipeline stages for the desired features, including trace decision logic, Bitmap Update Queue (BUQ), and Last Branch Queue (LBQ). 38

3.4 Bitmap update operation in the Bitmap Update Queue. 41

3.5 An example of encoding a basic block ID. 46

3.6 An example of data flow approximation between two runs leveraging the branch predictions stored in LBQ. 47

3.7 The average execution speed from fuzzing with AFL-QEMU, AFL-gcc and AFL-SNAP for 24 hours across the Binutils binaries. The numbers below the bars of AFL-QEMU show the number of executions per second for the mechanism. 53

3.8 The overall covered paths from fuzzing seven Binutils binaries for 24 hours. The solid lines represent the means, and the shades suggest the confidence intervals of five consecutive runs. 54

4.1 An overview of KVM/QEMU for x86 virtualization. 60

4.2 The nested virtualization scheme supported on x86 platforms. 61

4.3 The overview of the proposed system performant hypervisor fuzzing. 63

4.4 VMCSes maintained in the nested virtualization setting. 65

4.5 Management of virtualized memory in KVM/QEMU. 67

4.6 The average throughput improvement across multiple fuzzing schemes when testing ASAN-enabled QEMU. The numbers below the bars indicate the actual throughput (*i.e.*, executions per second) on our evaluating setup. . . . 72

4.7 The breakdown of exit and memory overhead from single and nested virtualization setting across kernels of various complexity. 73

4.8 Performance cost from the memory overhead in the single virtualization setting. 74

4.9 Performance cost from the exit overhead in the nested virtualization setting. 75

SUMMARY

Hypervisors serve as the backbone of cloud-based solutions by providing an abstraction of hardware resources and isolation of guest hosts. While gaining attention, the attack surfaces of hypervisors are broad due to the complexity of the design. The consequences of a compromised cloud service can be disastrous, leading to denial of service, information leakage, or even VM escape, jeopardizing the integrity of the other instances running on the same platform. Recent studies have found success in finding vulnerabilities in hypervisors with formal verification and symbolic execution. While fuzzing remains one of the most effective bug-finding techniques in most real-world programs, the challenges of applying it in the current context can be non-trivial. Particularly to non-user applications, like hypervisors, both efficiency and determinism are required from any proposed fuzzing approaches to be considered effective.

In this thesis, we propose a hardware-assisted nested virtualization framework for fuzzing hypervisors. With fast resets on guest VM, including states of virtualized CPU, memory, and devices, it avoids the primary source of overhead from reboots. Meanwhile, we adopt a customized OS to further reduce the performance costs from VM exits and dirty memory pages in the nested setting. Prototyped as a type-II framework in work with KVM and QEMU, our system demonstrates that it can achieve a 72x higher fuzzing throughput than the existing solutions. Given that, our system has found 14 bugs among real-world hypervisors, such as QEMU and VirtualBox.

CHAPTER 1
INTRODUCTION

1.1 Problem Statement

With a booming number of applications and their end-users in the past decade, software security has been emphasized more than ever. Nonetheless, a consistent increase of security-critical bugs have been observed along the way, mainly due to the variety and complexity of existing software. Based on the numbers provided by GitHub, more than 100 million open-source projects have been created to date, spanning different protection rings, including browsers, kernels, and hypervisors. To mitigate the situation, *software hardening* in the daily development cycle typically involves three stages. When a new code snippet is created, bug-finding techniques involving static or dynamic analyses are applied to find all possible bugs in the existing codebase. Once the effort has been exhausted and the software is deployed, runtime protection schemes with a different security guarantee, such as control flow integrity and memory access control, try to ensure the validity of program execution. In the case where the prior steps have failed and any hidden bugs have been exploited, fault analyses help to pinpoint the root cause of the error, while providing rollback mechanisms for recovery if necessary. The cycle iterates the stages in order, until there is no obvious incident to be handled.

While previous research has proposed various software-hardening techniques across the stages, a considerable number of works have relied on hardware support to achieve their goals. Table 1.1 shows a list of recent samples in chronological order. For example, kAFL [1] and Honggfuzz [2] utilize Intel Processor Trace (PT) [3] for efficient coverage tracking when fuzzing targeted binaries. PHMon [4] and HDFI [5], on the other hand, revise the existing architecture to ensure control and data security. The reasons behind the

Table 1.1: A survey of hardware-assisted software-hardening techniques in recent years.

Bug Finding	Runtime Mitigation	Fault Analysis
SNAP [8] Ptrix [9] PTfuzz [10] kAFL [1] Honggfuzz [2] WinAFL [11]	PHMon [4] libmpk [12] uCFI [13] PITTYPAT [14] HDFI [5]	ARCUS [15] REPT [16] POMP [17]

noticeable trending of hardware-based solutions are three-folded. First, the performance benefit from hardware can be substantial compared to a purely software-based solution. In the case of virtualization, for instance, Intel VMX [6] with the extended page table (EPT) enabled by the hardware extension can outperform the software mechanism of shadow page tables by $3\times$ due to the tremendous cost of the maintenance of memory pages [7]. Second, compatibility and ease of use are also key for more and more solutions to adopt hardware features besides performance gain. While tracing schemes based on source-based instrumentation or binary rewriting are commonly applied under various constraints, tracing by Intel PT can effectively avoid the hassles, such as when applying to a large codebase or legacy binaries without source code. Last, implementation with hardware support can consequentially present a smaller codebase, thus introducing less attack surface for attackers. Even though a technical idea might be sound, the inevitable implementation errors in a large codebase, such as logical or functional bugs, might render the approach useless in extreme cases.

However, leveraging hardware extensions for software hardening can be non-trivial. This is especially true when existing features are not tailored for the intended purposes, resulting in extra but unwanted overhead. One of the example comes from the design and use of Intel PT. Although the hardware extension has been introduced for debugging given its full traces in a highly compressed manner, many works have suggested borrowing it as a substitution for existing tracing techniques. While most use cases, such as runtime security and fuzzing, do not require full traces, the inevitable overhead from decoding is thus forced onto them when applied. In my first work, I presented PittyPat [14], an efficient control

security scheme for online protection against hijacking attacks under the support of Intel PT. To provide the strictly stronger security guarantee with path-sensitive points-to analysis, we adopt the hardware feature in a careful design. Specifically, we offer a parallel trace-sharing mechanism, along with various optimizations, to avoid the major sources of overhead from decoding conditional branches. By maintaining the current program state at the LLVM IR level and identifying control-relevant instructions on the fly, the solution only suffers 12.73% runtime overhead with most indirect call sites posing one precise branch target.

Unlike the previous work, which relies on the exact execution paths for path-sensitive analyses, the use case of fuzzing can find such path information unnecessary. Although works [2, 9] have adopted Intel PT in a similar way as proposed in PittyPat, they can barely catch up with the tracing overhead from source instrumentation (*e.g.*, 60+%), still significantly underperforming based on the standard of hardware-assisted solutions. In my second work, I presented SNAP [8], a customized hardware platform dedicated for performant fuzzing. Since most micro-architectural states, such as executed branch addresses and instruction bytes, are already in the existing CPU pipeline, our solution barely introduces any overhead. Along with proposed micro-architectural optimization for efficient coverage tracing, including memory-request aggregation and opportunistic bitmap updating under free cache bandwidth, we find our platform to be practical, incurring 3.14%, 4.82%, and 6.53% for runtime, area, and power overhead while being compatible with most existing fuzzers without much manual intervention. Compared to fuzzers running off our platform, those running with our hardware support can achieve more coverage under a limited timeframe due to the high fuzzing throughput enabled.

Meanwhile, the overhead of coverage tracing might not necessarily be the only root cause of the slowdown in pursuit of fuzzing speed when testing non-user applications, which are valuable targets due to their role in daily services, but are vulnerable due to their complexity. For example, to test the functionalities of hypervisors serving as the backbone for any cloud-based solutions [18, 19, 20], directly applying fuzzing in the context can be non-trivial. In

particular, fuzzers need to sit at the guest kernel space in a virtualized environment with a clean state of VM for each mutated input. The major overhead thus results from continuous VM reboots after each fuzzing execution, in general, taking seconds with serious limitations on efficiency. To mitigate this problem, Syzkaller [21] sacrifices determinism of unit testing for speed and only reboots periodically. Other approaches [22, 23] propose customized guest OSes as the main executors of fuzzing inputs for faster reboot time compared to the default Linux Kernel configuration, yet still are throttled by the throughput of up to four executions per second. Neither solution is ideal, as they fundamentally lack determinism and efficiency to fulfill the requirement of an effective fuzzing framework.

1.2 Research Outline

To overcome the aforementioned issues, this thesis aims to develop a hardware-based solution for this specialized testing target, *i.e.*, hypervisors. Besides the desired determinism and efficiency, this solution is also designed to be compatible with most existing hypervisors that can run on the Linux platform, such as QEMU and VirtualBox. Specifically, a nested virtualization framework consisting of two major components is proposed, including ① a snapshot and restoring mechanism realized by Intel VMX and ② a minimized OS with little redundancy. As mentioned previously, one source of overhead is a result of the rebooting process itself. For example, QEMU and the booting Linux kernel can each take seconds to initialize all their internal data structures, before a VM can be launched to reach a stable state for fuzzing. This process is expensive in the fuzzing space and inevitable as long as reboots are involved. Rather, in our framework, since the testing hypervisor resides in the L1 VM, the L0 host acts as the snapshot maintainer, taking and reloading the intended snapshot for faster reset. By resetting virtualized CPU, memory, and devices per request, the framework effectively prohibits side effects carried over from previous executions (*i.e.*, determinism) without really rebooting the tested VM (*i.e.*, efficiency). Another source of overhead in the framework comes from the complexity of the compiled kernel image. While the L2 guest

kernel is only expected to execute fuzzing inputs, applying the Linux Kernel can be overkill, downgrading performance through additional VM exiting requests and memory overhead. The minimized OS trims the unnecessary OS features in the current context to perform the snapshot and restore operations in a more efficient manner. Overall, the work serves as the last piece of the thesis by proposing another performant hardware-assisted solution for software hardening. In the following, I will briefly introduce my first two works, PITTYPAT and SNAP, before diving into more of its details.

CHAPTER 2

PITTYPAT: EFFICIENT PROTECTION OF PATH-SENSITIVE CONTROL SECURITY

2.1 Introduction

Attacks that compromise the control-flow of a program, such as return-oriented programming [24], have critical consequences for the security of a computer system. Control-Flow Integrity (CFI) [25] has been proposed as a restriction on the control-flow transfers that a program should be allowed to take at runtime, with the goals of both ruling out control-flow hijacking attacks and being enforced efficiently. A CFI implementation can be modeled as program rewriter that (1) before a target program P is executed, determines feasible targets for each indirect control transfer location in P, typically done by performing an analysis that computes a sound over-approximation of the set of all memory cells that may be stored in each code pointer (i.e., a static *points-to analysis* [26, 27]). The rewriter then (2) rewrites P to check at runtime before performing each indirect control transfer that the target is allowed by the static analysis performed in step (1).

A significant body of work [25, 28, 29] has introduced approaches to implement step (2) for a variety of execution platforms and perform it more efficiently. Unfortunately, the end-to-end security guarantees of such approaches are founded on the assumption that if an attacker can only cause a program to execute control branches determined to be feasible by step (1), then critical application security will be preserved. However, recent work has introduced new attacks that demonstrate that such an assumption does not hold in practice [30, 31, 32]. The limitations of existing CFI solutions in blocking such attacks are inherent to *any* defense that uses static points-to information computed per control location in a program. Currently, if a developer wants to ensure that a program only chooses

valid control targets, they must resort to ensure that the program satisfies *data integrity*, a significantly stronger property whose enforcement typically incurs prohibitively large overhead and/or has deployment issues, such as requiring the protected program being recompiled together with all dependent libraries and cannot be applied to programs that perform particular combinations of memory operations [33, 34, 35, 36].

In this work, we propose a novel, *path-sensitive* variation of CFI that is stronger than conventional CFI (i.e., CFI that relies on static points-to analysis). A program satisfies path-sensitive CFI if each control transfer taken by the program is consistent with the program's *entire* executed control path. Path-sensitive CFI is a stronger security property than conventional CFI, both in principle and in practice. However, because it does not place any requirements on the correctness of data operations, which happen much more frequently, it can be enforced much more efficiently than data integrity. To demonstrate this, we present a runtime environment, named PITTYPAT, that enforces path-sensitive efficiently using a combination of commodity, low-overhead hardware-based monitoring and a new runtime points-to analysis. PITTYPAT addressed two key challenges in building an efficient path-sensitive CFI solution. The first challenge is how to efficiently collect the path information about a program's execution so as to perform the analysis and determine if the program has taken only valid control targets. Collecting such information is not straightforward for dynamic analysis. An approach that maintains information inside the same process address space of the monitored program (e.g., [33]) must carefully protect the information; otherwise it would be vulnerable to attacks [37]. On the other hand, an approach that maintains information in a separate process address space must efficiently replicate genuine and sufficient data from the monitored program. The second key challenge is how to use collected information to precisely and efficiently compute the points-to relationship. Niu et al. [38] have proposed leveraging execution history to dynamically activate control transfer targets. However, since the activation is still performed over the statically computed control-flow graph, its accuracy can degrade to the same as pure static-analysis-based

approach.

PITTYPAT applies two key techniques in addressing these two challenges. First, PITTYPAT uses an event-driven kernel module that collects all chosen control-transfer targets from the *Processor Tracing (PT)* feature available on recent Intel processors [39]. PT is a hardware feature that efficiently records conditional and indirect branches taken by a program. While PT was originally introduced to enable detailed debugging through complete tracing, our work demonstrates that it can also be applied as an effective tool for performing precise, efficient program analysis for security. The second technique is an abstract-interpretation-based incremental points-to analysis. Our analysis embodies two key innovations. First, raw PT trace is highly compressed (see §2.2 for details). As a result, reconstructing the control-flow (i.e., source address to destination address) itself is time consuming and previous work has utilized multiple threads to reduce the decoding latency [40]. Our insight to solve this problem is to sync up our analysis with the execution, so that our analysis only needs to know what basic blocks being executed, not the control transfer history. Therefore, we can directly map the PT trace to basic blocks using the control-flow graph (CFG). The second optimization is based on the observation that static points-to analyses collect and solve a system of constraints over *all pairs* of pointer variables in the program [26, 41]. While this approach has good throughput, it introduces unacceptable latency for online analysis. At the same time, to enforce CFI, we only need to know the points-to information of code pointers. Based on this observation, our analysis eagerly evaluates control relevant points-to constraints as they are generated.

2.2 Motivation: Precision and Performance

In this section, we demonstrates the limitation on traditional CFI and shows the definition path-sensitive CFI with a motivating example. We further describes Intel PT, a commodity, low-overhead hardware extension dedicated for tracing.

```
1  struct request {
2    int auth_user;
3    char args[100];
4  };
5
6  void dispatch() {
7    void (*handler)(struct request *) = 0;
8    struct request req;
9
10   while(1) {
11     // parse the next request
12     parse_request(&req);
13     if (req.auth_user == ADMIN) {
14       handler = priv;
15     } else {
16       handler = unpriv;
17       // NOTE. buffer overflow, which can overwrite
18       //  the handler variable
19       strip_args(req.args);
20     }
21     // invoke the hanlder
22     handler(&req);
23   }
24 }
```

Figure 2.1: A motivating example that illustrates the advantages of control-path validity.

2.2.1 Limitations of existing CFI

Figure 2.1 contains a C program, named `dispatch`, that we will use to illustrate PITTYPAT. `dispatch` declares a pointer `handler` (line `L7`) to a function that takes an argument of a struct `request` (defined at line `L1`–`L4`), which has two fields: `auth_user` represents a user's identity, and `args` stores the arguments. `dispatch` contains a loop (line `L10`–`L23`) that continuously accepts requests from users, and calls `parse_request` (line `12`) to parse the next request. If the request is an administrator (line `L13`), the function pointer `handler` will be assigned with `priv`. Otherwise, `handler` is assigned to `unpriv` (line `L16`), and `dispatch` will call `strip_args` (line `L19`) to strip the request's arguments. At last, `dispatch` calls `handler` to perform relevant behaviors. However, the procedure `strip_args` contains a buffer-overflow vulnerability, which allows an attacker with control over input to `strip_args` to potentially subvert the control flow of a run of `dispatch` by using well-known techniques [42]. In particular, the attacker can provide inputs that overwrite memory outside of the fixed-size buffer pointed to by `req.args` in order to overwrite the address stored in `handler` to be the address of a function of their choosing, such as `execve`.

Protecting `dispatch` so that it satisfies conventional control-flow integrity (CFI) [25]

does not provide strong end-to-end security guarantees. An implementation of CFI attempts to protect a given program P in two steps. In the first step, the CFI implementation computes possible targets of each indirect control transfer in P by running a flow-sensitive points-to analysis[1] [26, 27, 41]. Such an approach, when protecting `dispatch`, would determine that when the execution reaches each of the following control locations L, the variable `handler` may store the following addresses $p(\texttt{L})$:

$$p(\texttt{L7}) = \{0\} \qquad p(\texttt{L14}) = \{\texttt{priv}\}$$
$$p(\texttt{L16}) = \{\texttt{unpriv}\} \qquad p(\texttt{L22}) = \{\texttt{priv}, \texttt{unpriv}\}$$

While flow-sensitive points-to analysis may implement various algorithms, the key property of each such analysis is that it computes points-to information per control location. If there is any run of the program that may reach control location L with a pointer variable `p` storing a particular address `a`, then the result of the points-to analysis must reflect that `p` may point to `a` at L. In the case of `dispatch`, any flow-sensitive points-to analysis can only determine that at line `L22`, `handler` may point to either `priv` or `unpriv`. After computing points-to sets p for program P, the second step of a CFI implementation rewrites P so that at each indirect control-transfer instruction in each run, the rewritten P can only transfer control to a control location that is a points-to target in the target register according to p. Various implementations have been proposed for encoding points-to sets and validating control transfers efficiently [25, 45, 29].

However, all such schemes are fundamentally limited by the fact that they can only validate if a transfer target is allowed by checking its membership in a flow-sensitive points-to set, computed per control location. `dispatch` and the points-to sets p illustrate a case in which any such scheme *must* allow an attacker to subvert control flow. In particular, an attacker can send a request with the identity of anonymous user. When `dispatch` accepts

[1] Some implementations of CFI [29, 43, 44] use a type-based alias analysis to compute valid targets, but such approaches are even less precise.

such a request, it will store `unpriv` in `handler`, and then strip the arguments. The attacker can provide arguments crafted to overwrite `handler` to store `priv`, and allow execution to continue. When `dispatch` calls the function stored in `handler` (line `L22`), it will attempt to transfer control to `priv`, a member of the points-to set for `L22`. Thus, `dispatch` rewritten to enforce CFI must allow the call. Let the sequence of key control locations visited in the above attack be denoted $p_0 = [\texttt{L7}, \texttt{L16}, \texttt{L22}]$.

Per-input CFI (denoted π-CFI) [38] avoids some of the vulnerabilities in CFI inherent to its use of flow-sensitive points-to sets, such as the vulnerability described above for `dispatch`. π-CFI updates the set of valid targets of control transfers of each instruction dynamically, based on operations performed during the current program execution. For example, π-CFI only allows a program to perform an indirect call to a function whose address was taken during an earlier program operation. In particular, if `dispatch` were rewritten to enforce π-CFI, then it would block the attack described above: in the execution of π-CFI described, the only instruction that takes the address of `handler` (line `L14`) is never executed, but the indirect call at `L22` uses `priv` as the target of an indirect call. However, in order for π-CFI to enforce per-input CFI efficiently, it updates valid points-to targets dynamically using simple, approximate heuristics, rather than a precise program analysis that accurately models the semantics of instructions executed. For example, if a function f appears in the static points-to set of a given control location L and has its address taken at any point in an execution, then f remains in the points-to set of L for the rest of the execution, even if f is no longer a valid target as the result of program operations executed later. In the case of `dispatch`, once `dispatch` takes the address of `priv`, `priv` remains in the points-to set of control location `L22` for the remainder of the execution.

An attacker can thus subvert the control flow of `dispatch` rewritten to enforce π-CFI by performing the following steps. **(1)** An administrator sends a request, which causes `dispatch` to store `priv` in `handler`, call it, and complete an iteration of its loop. **(2)** The attacker sends an anonymous request, which causes `dispatch` to set `unpriv` in `handler`. **(3)**

The attacker provides arguments that, when handled by `strip_args`, overwrite the address in `handler` to be `priv`, which causes `dispatch` to call `priv` with arguments provided by the attacker. Because `priv` will be enabled as a control target as a result of the operations performed in step (1), `priv` will be a valid transfer target at line `L22` in step (3). Thus, the attacker will successfully subvert control flow. Let the key control locations in the control path along which the above attack is performed be denoted $p_1 = [\texttt{L7}, \texttt{L14}, \texttt{L22}, \texttt{L16}, \texttt{L22}]$.

2.2.2 Path-sensitive CFI

In this paper, we introduce a path-sensitive version of CFI that addresses the limitations of conventional CFI illustrated in §2.2.1. A program satisfies path-sensitive CFI if at each indirect control transfer, the program only transfers control to an instruction address that is in the points-to set of the target register according to a points-to analysis of the whole executed control *path*. `dispatch` rewritten to satisfy path-sensitive CFI would successfully detect the attacks given in §2.2.1 on existing CFI. One collection of valid points-to sets for `handler` for each control location in subpath p_0 (§2.2.1) are the following:

$$(\texttt{L7}, \{0\}), (\texttt{16}, \{\texttt{unpriv}\}), (\texttt{L22}, \{\texttt{unpriv}\})$$

When execution reaches `L22`, `priv` is not in the points-to set of `handler`, and the program halts. Furthermore, `dispatch` rewritten to satisfy path-sensitive CFI would block the attack given in §2.2.1 on π-CFI. One collection of valid points-to sets for `handler` for each control location in subpath p_1 are the following:

$$(\texttt{L7}, \{0\}) \qquad (\texttt{L14}, \{\texttt{priv}\}) \qquad (\texttt{L22}, \{\texttt{priv}\})$$
$$(\texttt{L16}, \{\texttt{unpriv}\}) \qquad (\texttt{L22}, \{\texttt{unpriv}\})$$

When execution reaches `L22` in the second iteration of the loop in `dispatch`, `priv` is not in the points-to set of `handler`, and the program determines that the control-flow has been

Table 2.1: Control-relevant trace packets from Intel PT.

Packet	Description
TIP.PGE	IP at which the tracing begin
TIP.PGD	Marks the ending of tracing
TNT	Taken/non-taken decisions of conditional branches
TIP	Target addresses of indirect branches
FUP	The source addresses of asynchronous events

subverted.

2.2.3 Intel Processor Trace

Intel PT is a commodity, low-overhead hardware designed for debugging by collecting *complete* execution traces of monitored programs. PT captures information about program execution on each hardware thread using dedicated hardware facilities so that after execution completes, the captured trace data can be reconstructed to represent the exact program flow. The captured control flow information from PT is presented in encoded data packets. The control relevant packet types are shown in Table 2.1. PT records the beginning and the end of tracing through `TIP.PGE` and `TIP.PGD` packets, respectively. Because the recorded control flow needs to be highly compressed in order to achieve the efficiency, PT employs several techniques to achieve this goal. In particular, PT only records the taken/non-taken decision of each conditional branches through `TNT`, along with the target of each indirect branches through `TIP`. A direct branch does not trigger a PT packet because the control target of a direct branch is fixed.

Besides the limited packet types necessary for recovering *complete* execution traces, PT also adopts compact packet format to reduce the data throughput aggressively. For instance, `TNT` packets use one bit to indicate the direction of each conditional branches. `TIP` packets, on the other hand, contain compressed target address if the upper address bytes match the previous address logged. Thus on average, PT tracing incurs less than 5% overhead [40]. Compared to source-based instrumentation, PT tracing is protected with CR3 filtering, while traces are shared from the hardware to the kernel, before delegated to the user space. Even if a vulnerable program is compromised, tampering with collected traces is not straigtforward

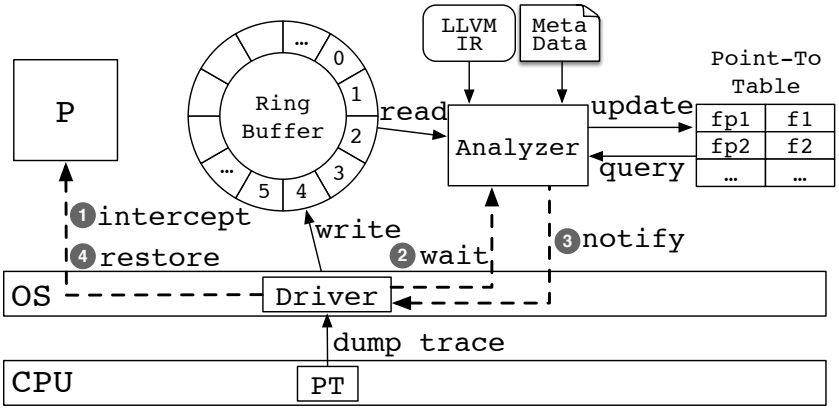


Figure 2.2: The architecture of PITTYPAT. P denotes a target program. The *analyzer* and *driver* modules of PITTYPAT are described in §2.3.1.

in this case. On the other hand, compared to any dynamic binary instrumentation schemes, such as Intel PIN [46] and QEMU [47], PT tracing is much faster due to the hardware support with compressed packets for low bandwidth as mentioned above.

2.3 Design

2.3.1 Overview

The points-to sets for control paths considered in §2.2.2 illustrate that if a program can be rewritten to satisfy path-sensitive CFI, it can potentially satisfy a strong security guarantee. However, ensuring that a program satisfies path-sensitive CFI is non-trivial, because the program must be extended to dynamically compute the results of sophisticated semantic constraints [26] over the exact control path that it has executed. A key contribution of our work is the design of a runtime environment, PITTYPAT, that enforces path-sensitive CFI efficiently. PITTYPAT's architecture is depicted in Figure 2.2. For program P, the state and code of PITTYPAT consist of the following modules, which execute concurrently: **(1)** a user-space process in which P executes, **(2)** a user-space *analysis module* that maintains points-to information for the control-path executed by P, and **(3)** a kernel-space *driver* that sends control branches taken by P to the analyzer and validates system calls invoked by P using the analyzer's results.

Before a program P is monitored, the analysis module is given (1) an intermediate

representation of P and (2) meta data including a map from each instruction address in the binary representation of P to the instruction in the intermediate representation of P. We believe that it would also be feasible to implement PITTYPAT to protect a program given only as a binary, given that the analyzer module only must perform points-to analysis on the sequence of executed instructions, as opposed to inferring the program's complete control-flow graph. As P executes a sequence of binary instructions, the driver module copies the targets of control branches taken by P from PT's storage to a ring buffer shared with the analyzer. PT's storage is privileged: it can only be written by hardware and flushed by privileged code, and cannot be tampered with by P or any other malicious user-space process. The analyzer module reads taken branches from the ring buffer, uses them to reconstruct the sequence of IR instructions executed by P since the last branch received, and updates the points-to information in a table that it maintains for P's current state by running a points-to analysis on the reconstructed sequence. When P invokes a system call, the driver first intercepts P (❶), while waiting for the analyzer module to determine in parallel if P has taken a valid sequence of control targets over the *entire* execution up to the current invocation (❷ and ❸). The analyzer validates the invocation only if P has taken a valid sequence, and the driver allows execution of P to continue only if the invocation is validated (❹).

There are two key challenges we must address to make PITTYPAT efficient. First, trace information generated by PT is highly compressed; e.g., for each conditional branch that a program executes, PT provides only a single bit denoting the value of the condition tested in the branch. Therefore additional post-processing is necessary to recover transfer targets from such information. The approach used by the `perf` tool of Linux is to parse the next branch instruction, extract the offset information, then calculate the target by adding the offset (if the branch is taken) or the length of instruction (if branch is not taken). However, because parsing x86 instructions is non-trivial, such an approach is too slow to reconstruct a path online. Our insight to solve this problem is that, to reconstruct the executed path,

an analysis only needs to know the basic blocks executed. We have applied this insight by designing the analysis to maintain the current basic block executed by the program. The analysis can maintain such information using the compressed information that PT provides. E.g., if PT provides only a bit denoting the value of a condition tested in a branch, then the analysis inspects the conditional branch at the end of the maintained block, and from the branch, updates its information about the current block executed.

The second key challenge in designing PITTYPAT is to design a points-to analysis that can compute accurate points-to information while imposing sufficiently low overhead. Precise points-to analyses solve a system of constraints over all pairs of pointer variables in the program [26, 41]; solving such constraints uses a significant amount of time that is often acceptable in the context of an offline static analysis, but would impose unacceptable overhead if used by PITTYPAT's online analysis process. Other analyses bound analysis time to be nearly linear with increasing number of pointer variables, but generate results that are often too imprecise to provide strong security guarantees if used to enforce CFI [27]. To address the limitations of conventional points-to analysis, we have designed an online points-to analysis that achieves the precision of precise analysis at high performance. The analysis eagerly evaluates control relevant points-to constraints as they are generated, while updating the points-to relations table used for future control transfer validation. The analysis enables PITTYPAT, when analyzing runs of `dispatch` that execute paths p_0 and p_1, to compute the accurate points-to information given in §2.2.2. The following sections explain the details of our solution to each challenge as described.

2.3.2 Efficient branch sharing

PITTYPAT uses the PT extension for Intel processors [39] to collect the control branches taken by P. A naive implementation of PITTYPAT would receive from the monitoring module the complete target address of each branch taken by P in encoded packets and decode the traces offline for analysis. PITTYPAT, given only Boolean flags from PT, decodes

complete branch targets on the fly. To do so, PITTYPAT maintains a copy of the current control location of P. For example, in Figure 2.1, when `dispatch` steps through the path $[$`L10`, `L16`, `L22`$]$, the relevant PT trace contains only two `TNT` packets and one `TIP` packet. A `TNT` packet is a two-bit stream: 10. The first bit, 1, represents the conditional branch at `L10` is taken (i.e., the execution enters into the loop). The second bit, 0, indicates the conditional branch at `L13` is not taken, and the executed location is now in the else branch. The `TIP` packet contains the address of function `unpriv`, which shows an indirect jump to `unpriv`.

PITTYPAT uses the Linux `perf` infrastructure to extract the execution trace of P. In particular, PITTYPAT uses the `perf` kernel driver to **(1)** allocate a ring buffer shared by the hardware and itself and **(2)** mark the process in which the target program executes (and any descendant process and thread) as traced so as to enable tracing when context switching into a descendant and disable tracing when context switching out of a descendant. The driver then transfers the recorded PT packets, together with thread ID and process ID, to the analyzer module through the shared buffer. This sharing mechanism has proved to be efficient on all performance benchmarks on which we evaluated PITTYPAT, typically incurring less than 5% overhead.

PITTYPAT intercepts the execution of a program at security-sensitive system calls in the kernel and does not allow the program to proceed until the analyzer validates all control branches taken by the program. The list of intercepted system calls can be easily configured; the current implementation checks `write`, `mmap`, `mprotect`, `mremap`, `sendmsg`, `sendto`, `execve`, `remap_file_pages`, `sendmmsg`, and `execveat`. The above system calls are intercepted because they can either disable DEP/W$\oplus$X, directly execute an unintended command, write to files on the local host, or send traffic over a network.

2.3.3 Online points-to analysis

The analyzer module executes in a process distinct from the process in which the monitored process executes. Before monitoring a run of the program, the analyzer is given the

monitored program's LLVM IR and meta information about mapping between IR and binary code. At runtime, the analyzer receives the next control-transfer target taken by the protected program from the monitor module, and either chooses to raise an alarm signaling that the control transfer taken would violate path-sensitive CFI, or updates its state and allows the original program to take its next step of execution. The updated states contain two components: (1) the callstack of instructions being executed (i.e., the `pc`'s) and (2) points-to relations over models of memory cells that are control relevant only. The online points-to analysis addresses the limitations of conventional points-to analyses. In particular, it reasons precisely about the calling context of the monitored program by maintaining a stack of register frames. It avoids maintaining constraints over pairs of pointer variables by eagerly evaluating the sets of cells and instruction addresses that may be stored in each register and cell. It updates this information efficiently in response to program actions by performing updates on a single register frame and removing register frames when variables leave scope on return from a function call.

In general, a program may store function pointers in arbitrarily, dynamically allocated data structures before eventually loading the pointer and using it as the target of an indirect control transfer. If the analyzer were to maintain precise information about the points-to relation of all heap cells, then it would maintain a large amount of information never used and incur a significant cost to performance. We have significantly optimized PITTYPAT by performing aggressive analyses of a given program P offline, before monitoring the execution of P on a given input. PITTYPAT runs an analyzer developed in previous work on *code-pointer integrity* (CPI) [33] to collect a sound over-approximation of the instructions in a program that may affect a code pointer used as the target of a control transfer. At runtime, the analyzer only analyzes instructions that are control relevant as determined by its offline phase. Meanwhile, a program may also contain many functions that perform no operations on data structures that indirectly contain code pointers, and do not call any functions that perform such operations. We optimized PITTYPAT by applying an offline analysis based on

a sound approximation of the program's call graph to identify all such functions. At runtime, PITTYPAT only analyzes functions that may indirectly perform relevant operations.

To illustrate the analyzer's workflow, consider the execution path [L10, L12, L16, 19, L22] in Figure 2.1 as an example. Initially, the analyzer knows that the current instruction being executed is L10, and the points-to table is empty. The analyzer then receives a taken TNT packet, and so it updates the pc to L12, which calls a non-sensitive function parse_request. However instead of tracing instructions in parse_request, the analyzer waits until receiving a TIP packet signaling the return from parse_request before continue its analysis. Next, it updates the pc to L16 after receiving a non-taken TNT packet, which indicates that the else branch is taken. Here, the analyzer updates the points-to table to allow handler to point to unpriv when it handles L16. Because the program also calls a non-sensitive function at L19, the analyzer waits again and updates the pc to L22 only after receiving another TIP packet. Finally, at L22, the analyzer waits for a TIP packet at the indirect call, and checks whether the target address collected by the monitor module is consistent with the value pointed by handler in the points-to table. In this case, if the address in the received TIP packet is not unpriv, the analyzer throws an alarm.

2.4 Implementation

2.4.1 Monitor module

PITTYPAT controls the Intel PT extension and collects an execution trace from a monitored program by adapting the Linux v4.4 perf infrastructure. Because perf was originally designed to aid debugging, the original version provided with Linux 4.4 only supports decoding and processing traces offline. In the original implementation, the perf kernel module continuously outputs packets of PT trace information to the file system in user space as a log file to be consumed later by a userspace program. Such a mechanism obviously cannot be used directly within PITTYPAT, which must share branch information at a speed that allows it to be run as an online monitor.

We modified the kernel module of `perf`, which begins and ends collection of control targets taken after setting a target process to trace, allocates a ring buffer in which it shares control branches taken with the analyzer, and monitors the amount of space remaining in the shared buffer. The module also notifies the analyzer when taken branches are available in its buffer, along with how many chosen control targets are available. The notification mechanism reuses the pseudo-file interface of the `perf` kernel module. The analyzer creates one thread to wait (i.e., poll) on this file handler for new trace data. Once woken up by the kernel, it fetches branches from the shared ring buffer with minimal latency.

System calls are intercepted by a modified version of the system-call mechanism provided by the Linux kernel. When the monitored process is created, it—along with each of its sub-processes and threads created later—is flagged with a true value in a `PT_CPV` field of its `task_struct` in kernel space. When the kernel receives a request for a system call, the kernel checks if the requesting process is flagged. If so, the kernel inspects the value in register `rax` to determine if it belongs to the configured list of marked system calls as described in §2.3.2. The interception mechanism is implemented as a semaphore, which blocks the system call from executing further code in kernel space until the analyzer validates all branches taken by the monitored process and signals the kernel.

The driver module and modifications to the kernel consist of approximately 400 lines of C code.

2.4.2 Analyzer module

PITTYPAT's analyzer module is implemented as two core components. The first component consists of a LLVM compiler pass, implemented in 500 lines, that inserts an instruction at the beginning of each basic block before the IR is translated to binary instructions. Such instructions are used to generate a map from binary basic blocks to LLVM IR basic blocks. Thus when PITTYPAT receives a `TNT` packet for certain conditional branch, it knows the corresponding IR basic block that is the target of the control transfer. The inserted

instructions are removed when generating binary instructions; therefore no extra overhead is introduced to the running program.

The second component, implemented in 5,800 lines C++ code, performs a path-sensitive points-to analysis over the control path taken by the monitored process, and raises an error if the monitored process ever attempts to transfer control to a branch not allowed by path-sensitive CFI. Although the analysis inspects only low-level code, it directly addresses several challenges in analyzing code compiled from high-level languages. First, to analyze exception-handling by a C++ program, which unwinds stack frames without explicit calls to return instructions, the analyzer simply consumes the received `TNT` packets generated when the program compares the exception type and updates the `pc` to the relevant exception handler.

To analyze a dynamic dispatch performed by a C++ program, the analyzer uses its points-to analysis to determine the set of possible objects that contain the vtable at each dynamic-dispatch callsite. The analyzer validates the dispatch if the requested control target stored in a given `TIP` packet is one of the members of the object from which the call target is loaded. At each call to `setjmp`, the analyzer stores all possible `setjmp` buffer cells that may be used as arguments to `setjmp`, along with the instruction pointer at which `setjmp` is called, in the top stack frame. At each call to `longjmp`, the analyzer inspects the target `T` of the indirect call and unwinds its stack until it finds a frame in which `setjmp` was called at `T`, with the argument buffer of `longjmp` may have been the buffer passed as an argument to `setjmp`.

2.5 Evaluation

We performed an empirical evaluation to answer the following experimental questions. **(1)** Are benign applications transformed to satisfy path-sensitive CFI less susceptible to an attack that subverts their control security? **(2)** Do applications that are explicitly written to perform malicious actions that satisfy weaker versions of CFI fail to satisfy path-sensitive

CFI? **(3)** Can PITTYPAT enforce path-sensitive CFI efficiently?

To answer these questions, we used PITTYPAT to enforce path-sensitive CFI on a set of benchmark programs and workloads, including both standard benign applications and applications written explicitly to conceal malicious behavior from conventional CFI frameworks. In summary, our results indicate that path-sensitive CFI provides a stronger security guarantee than state-of-the-art CFI mechanisms, and that PITTYPAT can enforce path-sensitive CFI while incurring overhead that is acceptable in security-critical contexts.

2.5.1 Methodology

We collected a set of benchmarks, each described in detail in §2.5.2. We compiled each benchmark with LLVM 3.6.0, and ran them on a set of standard workloads. During each run of the benchmark, we measured the time used by the program to process the workload. If a program contained a known vulnerability that subverted conventional CFI, then we ran the program on inputs that triggered such a vulnerability as well, and observed if PITTYPAT determined that control-flow was subverted along the execution. Over a separate run, at each control branch taken by the program, we measured the size of the points-to set of the register that stored the target of the control transfer.

We then built each benchmark to run under a state-of-the-art CFI framework implemented in previous work, π-CFI [38]. While π-CFI validates control targets per control location, it instruments a subject program so that control edges of the program are disabled by default, and are only enabled as the program executes particular triggering actions (e.g., a function can only be called after its address is taken). It thus allows sets of transfer targets that are no larger than those allowed by conventional implementations of CFI, and are often significantly smaller [38]. For each benchmark program and workload, we observed whether π-CFI determined that the control-flow integrity of the program was subverted while executing the workload and measured the runtime of the program while executed under π-CFI. We compared PITTYPAT to π-CFI because it is the framework most similar to PITTYPAT in

concept: it validates control-transfer targets based not only on the results of a static points-to analysis, but collecting information about the program's dynamic trace.

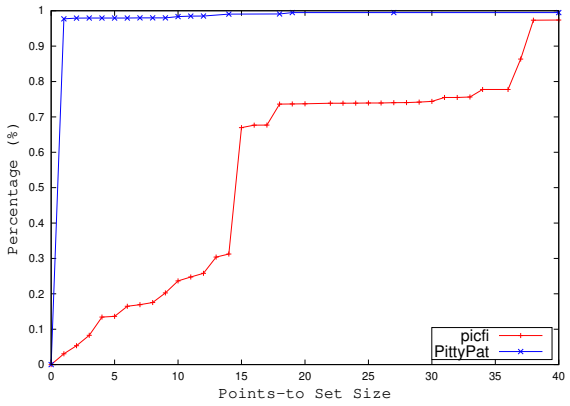
2.5.2 Benchmarks

To evaluate the ability of PITTYPAT to protect long-running, benign applications, and to evaluate the overhead that it incurs at runtime, we evaluated it on the SPEC CPU2006 benchmark suite, which consists of 16[2] C/C++ benchmarks. We ran each benchmark three times over its provided reference workload. For each run, we measured the runtime overhead imposed by PITTYPAT and the number of control targets allowed at each indirect control transfer, including both indirect calls and returns. We also evaluated PITTYPAT on the NGINX server—a common performance macro benchmark, configured to run with multiple processes.

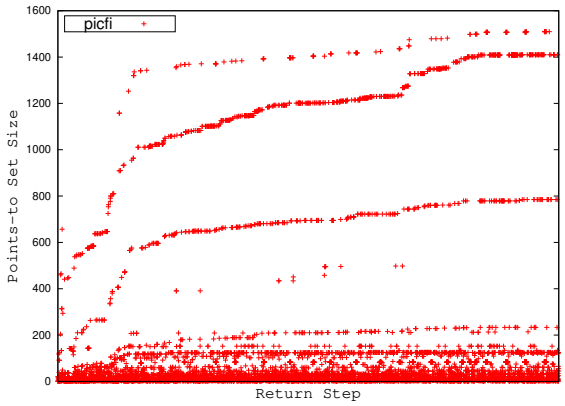
To evaluate PITTYPAT's ability to enforce end-to-end control security, we evaluated it on a set of programs explicitly crafted to contain control vulnerabilities, both as analysis benchmarks and in order to mount attacks on critical applications. In particular, we evaluated PITTYPAT on programs in the RIPE benchmark suite [48], each of which contains various vulnerabilities that can be exploited to subvert correct control flow (e.g. *Return-Oriented Programming* (ROP) or *Jump-oriented Programming* (JOP)). We compiled 264 of its benchmarks in our x64 Linux test environment and evaluated PITTYPAT on each. We also evaluated PITTYPAT on a program that implements a proof-of-concept COOP attack [32], a novel class of attacks on the control-flow of programs written in object-oriented languages that has been used to successfully mount attacks on the Internet Explorer and Firefox browsers. We determined if PITTYPAT could block the attack that the program attempted to perform.

[2]We don't include `447.dealII`, `471.omnetpp`, and `483.xalancbmk` because their LLVM IR cannot be completely mapped to the binary code.

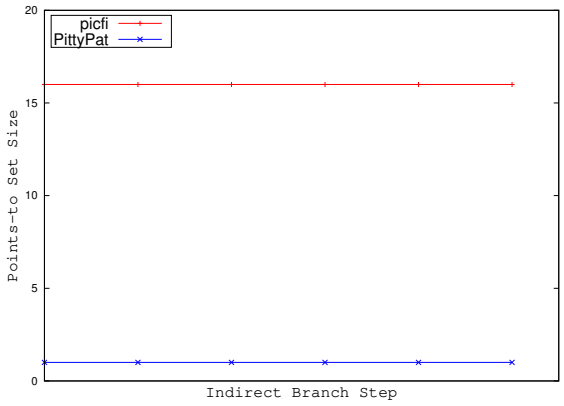
2.5.3 Results



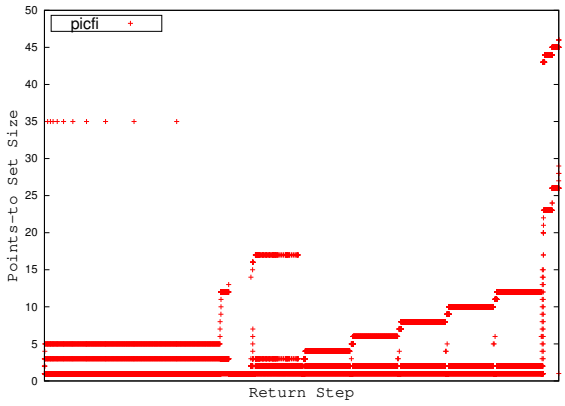
(a) Partial CDF of allowed targets on forward edges taken by `403.gcc`.



(b) π-CFI points-to set of backward edges taken by `403.gcc`.



(c) π-CFI and PITTYPAT points-to sets for forward edges taken by `444.namd`.



(d) π-CFI points-to sets for backward edges taken by `444.namd`.

Figure 2.3: Control-transfer targets allowed by π-CFI and PITTYPAT over `403.gcc` and `444.namd`.

Protecting benign applications. Figure 2.3 contains plots of the control-transfer targets allowed by π-CFI and PITTYPAT over runs of example benchmarks selected from §2.5.2. In the plots, each point on the x-axis corresponds to an indirect control transfer in the run. The corresponding value on the y-axis contains the number of control targets allowed for the transfer. Previous work on CFI typically reports the average indirect-target reduction (AIR) of a CFI implementation; we computed the AIR of PITTYPAT. However, the resulting data does not clearly illustrate the difference between PITTYPAT and alternative approaches, because all achieve a reduction in branch targets greater than 99% out of all branch targets in the program. This is consistent with issues with AIR as a metric established in previous

work [49]. Figure 2.3, instead, provides the absolute magnitudes of points-to sets at each indirect control transfer over an execution.

Figure 2.3a contains a Cumulative Distribution Graph (CDF) of all points-to sets at *forward* (i.e., jumps and calls) indirect control transfers of size no greater than 40 when running `403.gcc` under π-CFI and PITTYPAT. We used a CDF over a portion of the points-to sets in order to display the difference between the two approaches in the presence of a small number of large points-to sets, explained below. Figure 2.3a shows that PITTYPAT can consistently maintain significantly smaller points-to sets for forward edges than that of π-CFI, leading to a stronger security guarantee. Figure 2.3a indicates that when protecting practical programs, an approach such as π-CFI that validates per location allows a significant number of transfer targets at each indirect callsite, even using dynamic information. In comparison, PITTYPAT uses the entire history of branches taken to determine that at the vast majority of callsites, only a *single* address is a valid target. The difference in the number of allowed targets can be explained by the different heuristics adopted in π-CFI, which monotonically accumulates allowed points-to targets without any disabling schemes once targets are taken, and the precise, context-sensitive points-to analysis implemented in PITTYPAT. Similar difference between π-CFI and PITTYPAT can also be found in all other C benchmarks from SPEC CPU2006.

For the remaining 4% of transfers not included in Figure 2.3a, both π-CFI and PITTYPAT allowed up to 218 transfer targets; for each callsite, PITTYPAT allowed no more targets than π-CFI. The targets at such callsites are loaded from vectors and arrays of function pointers, which PITTYPAT's current points-to analysis does not reason about precisely. It is possible that future work on a points-to analysis specifically designed for reasoning precisely about such data structures over a single path of execution—a context not introduced by any previous work on program analysis for security—could produce significantly smaller points-to sets.

A similar difference between π-CFI and PITTYPAT is demonstrated by the number of

transfer targets allowed for other benchmarks. In particular, Figure 2.3c contains similar data for the `444.namd` benchmark. `444.namd`, a C++ program, contains many calls to functions loaded from vtables, a source of imprecision for implementations of CFI that can be exploited by attackers [32]. PITTYPAT allows a *single* transfer target for *all* forward edges as a result of its online points-to analysis. The difference between π-CFI and PITTYPAT are also found for other C++ benchmarks, such as `450.soplex`, `453.povray` and `473.astar`.

π-CFI and PITTYPAT consistently allow dramatically different numbers of transfer targets for return instructions. While monitoring `403.gcc`, π-CFI allows, for some return instructions, over $1,400$ return targets (Figure 2.3b). While monitoring `444.namd`, π-CFI allows, for some return instructions, more than 46 transfer targets (Figure 2.3d). Because PITTYPAT maintains a stack of points-to information during its analysis, it will *always* allow only a single transfer target for each return instruction, over all programs and workloads. PITTYPAT thus significantly improves defense against ROP attacks, which are still one of the most popular attacks software.

Mitigating malicious applications. To determine if PITTYPAT can detect common attacks on control, we used it to monitor selected RIPE benchmarks [48]. For each of the 264 benchmarks that ran in our experimental setup, PITTYPAT was able to successfully detect attacks on the benchmark's control security.

We constructed a proof-of-concept program vulnerable to a COOP [32] attack that corrupts virtual-function pointers to perform a sequence of method calls not possible by a well-defined run of the program. In Figure 2.4, the program defines two derived classes of `SchoolMember` (line `L1`–`L4`), `Student` (line `L5`–`L10`) and `Teacher` (line `L11`–`L16`). Both `Student` and `Teacher` define their own implementation of the virtual function `registration()` (lines `L7`–`9` and `L13`–`15`, respectively). `set_buf()` (line `L17`–`L21`) allocates a buffer `buf` on the stack of size 4 (line `L18`), but does not bound the amount of data that it reads into `buf` (line `L20`). The `main` function (line `L22`–`L37`) constructs instances of `Student` and `Teacher` (lines `L23` and `L24`, respectively), and stores them in `SchoolMember` pointers

```
1  class SchoolMember {
2    public:
3      virtual void registration(void){}
4  };
5  class Student : public SchoolMember{
6    public:
7      void registration(void){
8        cout << "I am a Student\n";
9      }
10 };
11 class Teacher : public SchoolMember{
12   public:
13     void registration(void){
14       cout << "This is sensitive!\n";
15     }
16 };
17 void set_buf(void){
18   char buf[4];
19   //change vptr to that of Teacher's sensitive func
20   gets(buf);
21 }
22 int main(int argc, char *argv[]){
23   Student st;
24   Teacher te;
25   SchoolMember *member_1, *member_2;
26   member_1 = &te;
27   member_2 = &st;
28   //Teacher calling its virtual functions
29   member_1->registration();
30   //Student calling its virtual functions
31   member_2->registration();
32   //buffer overflow to overwrite the vptr
33   set_buf();
34   //Student calling its virtual functions again
35   member_2->registration();
36   return 0;
37 }
```

Figure 2.4: A program vulnerable to a COOP attack.

(lines L26 and 27 respectively). `main` then calls the `registration()` method of each instance (lines L29–L31), reads input from a user by calling `set_buf()` (line L33), and calls `Student::registration()` a second time (line L35). A malicious user can subvert control flow of the program by exploiting the buffer overflow vulnerability in `set_buf` to overwrite the vptr of `Student` to that of `Teacher` and run `Teacher::registration()` at line L35.

Previous work introducing COOP attacks [32] established such an attack cannot be detected by CFI. π-CFI was not able to detect an attack on the above program because it allows a dynamic method as a call target once its address is taken. However, PITTYPAT detected the attack because its analyzer module accurately models the effect of each load of a function pointer used to implement the dynamic calls over the program's well-defined runs.

Table 2.2: "Name" contains the name of the benchmark. "KLoC" contains the number of lines of code in the benchmark. Under "Payload Features," "Exp" shows if the benchmark contains an exploit and "Tm (sec)" contains the amount of time used by the program, when given the payload. Under "π-CFI Featues", "HYPERSET Features," and "CETS+SB Features," "Alarm" contains a flag denoting if a given framework determined that the payload was an attack and aborted; "Overhd (%)" contains the time taken by the framework, expressed as the ratio over the baseline time.

Program Features		**Payload Features**		**π-CFI Features**		**HYPERSET Features**		**CETS+SB Features**	
Name	**KLoC**	**Exp**	**Tm (sec)**	**Alarm**	**Overhd (%)**	**Alarm**	**Overhd (%)**	**Alarm**	**Overhd (%)**
`400.perlbench`	128	No	332	No	8.7%	No	47.3%	Yes	–
`401.bzip2`	6	No	317	No	1.3%	No	17.7%	No	91.4%
`403.gcc`	383	No	179	No	6.2%	No	34.1%	Yes	–
`429.mcf`	2	No	211	No	4.3%	No	32.2%	Yes	–
`433.milc`	10	No	514	No	1.9%	No	1.8%	Yes	–
`444.namd`	4	No	556	No	-0.3%	No	28.8%	Yes	–
`445.gobmk`	158	No	328	No	11.4%	No	4.0%	Yes	–
`450.soplex`	28	No	167	No	-1.1%	No	27.5%	Yes	–
`453.povray`	79	No	100	No	11.9%	No	16.0%	Yes	–
`456.hmmer`	21	No	258	No	0.2%	No	20.2%	Yes	–
`458.sjeng`	11	No	359	No	8.5%	No	6.7%	No	80.1%
`462.libquantum`	3	No	234	No	-1.5%	No	14.1%	Yes	–
`464.h264ref`	36	No	339	No	8.0%	No	11.8%	No	251.7%
`470.lbm`	1	No	429	No	1.4%	No	0.7%	Yes	–
`473.astar`	4	No	289	No	2.2%	No	22.5%	Yes	–
`482.sphinx3`	13	No	338	No	1.7%	No	16.0%	Yes	–
Geo. Mean	15	–	285	–	3.30%	–	12.73%	–	122.60%
`nginx-1.10.2`	122	No	25.41	No	2.7%	No	11.9%	Yes	–

Enforcing path-sensitive CFI efficiently. Table 2.2 contains measurements of our experiments that evaluate performance of PITTYPAT when monitoring benchmarks from SPEC CPU2006 and NGINX server, along with the performance results replicated from the paper that presented π-CFI [38]. A key feature observable from Table 2.2 is that PITTYPAT induces overhead that is consistently larger than, but often comparable to, the overhead induced by π-CFI. The results show that PITTYPAT incurs a geometric mean of 12.73% overhead across the 16 SPEC CPU2006 benchmarks, along with a 11.9% increased response time for NGINX server over one million requests with concurrency level of 50. Overhead of sharing branch targets taken is consistently less than 5%. The remaining overhead, incurred by the analysis module, is proportional to the number of memory operations (e.g., loads, stores, and copies) performed on memory cells that transitively point to a target of an indirect call, as well as the number of child processes/threads spawned during execution of multi-process/-threading benchmarks.

Another key observation from Table 2.2 is that PITTYPAT induces much smaller overhead than CETS [35] and SoftBound [36], which can only be applied to a small selection of the SPEC CPU2006 benchmarks. CETS provides temporal memory safety and SoftBound provides spatial memory safety; both enforce full data integrity for C benchmarks, which entails control security. However, both approaches induce significant overhead, and cannot be applied to programs that perform particular combinations of memory-unsafe operation [33]. Our results thus indicate a continuous tradeoff between security and performance among exisiting CFI solution, PITTYPAT, and data protection. PITTYPAT offers control security that is close to ideal, i.e. what would result from data integrity, but with a small percentage of the overhead of data-integrity protection.

CHAPTER 3
SNAP: HARDWARE SUPPORT TO IMPROVE FUZZING PERFORMANCE AND PRECISION

3.1 Introduction

Historically, bugs have been companions of software development due to limitations of human programmers. Those bugs can lead to unexpected outcomes ranging from simple crashes, which render programs unusable, to exploitation toolchains, which grant attackers partial or complete control of user devices. As modern software evolves and becomes more complex, a manual search for such unintentionally introduced bugs becomes unscalable. Various automated software testing techniques have thus emerged to help find bugs efficiently and accurately, one of which is *fuzzing*. Fuzzing in its essence works by continuously feeding randomly mutated inputs to a target program and watching for unexpected behavior. It stands out from other software testing techniques in that minimal manual effort and pre-knowledge about the target program are required to initiate bug hunting. Moreover, fuzzing has proven its practicality by uncovering thousands of critical vulnerabilities in real-world applications. For example, Google's in-house fuzzing infrastructure *ClusterFuzz* [50] has found more than 25,000 bugs in Google Chrome and 22,500 bugs in over 340 open-source projects. According to the company, fuzzing has uncovered more bugs than over a decade of unit tests manually written by software developers. As more and more critical bugs are being reported, fuzzing is unarguably one of the most effective technique to test complex, real-world programs.

An ideal fuzzer aims to execute mutated inputs that lead to bugs at a high speed. However, certain execution cycles are inevitably wasted on testing the ineffective inputs that do not approach any bug in practice. To save computing resources for inputs that are more likely to

trigger bugs, state-of-the-art fuzzers are coverage-guided and favor mutation on a unique subset of inputs that reach new code regions per execution. Such an approach is based on the fact that the more parts of a program that are reached, the better the chance an unrevealed bug can be triggered. In particular, each execution of the target program is monitored for collecting runtime code coverage, which is used by the fuzzer to cherry-pick generated inputs for further mutation. For binaries with available source code, code coverage information is traced via compile-time instrumentation. For standalone binaries, such information is traced through dynamic binary instrumentation (DBI) [51, 52, 47], binary rewriting [53, 54], or hardware-assisted tracing [3, 55].

Nonetheless, coverage tracing itself incurs large overhead and slows the execution speed, making fuzzers less effective. The resulting waste of computing resources can extend further with a continuous fuzzing service scaling up to tens of thousands of machines [50, 56]. For example, despite its popularity, AFL [57] suffers from a tracing overhead of nearly 70% due to source code instrumentation and of almost 1300% in QEMU mode for binary-only programs [54]. Source code instrumentation brings in additional instructions to maintain the original register status at each basic block, while DBI techniques require dynamic code generation, which is notoriously slow. Although optimized coverage tracing techniques have been proposed to improve performance, especially for binary-only programs, they impose different constraints. RetroWrite [53] requires the relocation information of position-independent code (PIC) to improve performance for binary-only programs. Various existing fuzzers [2, 21, 11] utilize *Intel Processor Trace* (PT) [3], a hardware extension that collects general program execution information. Nevertheless, Intel PT is not tailored for the lightweight tracing required by fuzzing. The ad-hoc use of Intel PT in fuzzing results in non-negligible slowdown caused by extracting useful information (*e.g.*, coverage), with a merely comparable execution speed as source instrumentation in the best effort from the large-scale profiling results [9, 58]. UnTracer [54] suggests *coverage-guided tracing*, which only traces testcases incurring new code paths. However, UnTracer adopts *basic*

block coverage without edge hit count and measures a less accurate program execution trace that misses information about control transfers and loops. The overhead of software-based coverage tracing is inevitable because it requires extra information not available during the original program execution. Moreover, the applicability of fuzzing heavily depends on the availability of source code, given that existing techniques commonly used for fuzzing stand-alone binaries are unacceptably slow and there is a need for faster alternatives.

In this paper, we propose SNAP, a customized hardware platform that implements hardware primitives to enhance the performance and precision of coverage-guided fuzzing. When running on SNAP, fuzzing processes can achieve near-to-zero performance overhead. The design of SNAP is inspired by three key properties observed from the execution cycle of a program in the hardware layer.

First, a hardware design can provide transparent support of fuzzing without instrumentation, as coverage information can be collected directly in the hardware layer with minimal software intervention. By sitting at the bottom of the computer stack, SNAP can assist fuzzers to fuzz any binary efficiently, including third-party libraries or legacy software, regardless of source code availability, making fuzzing universally applicable.

Second, we find that the code tracing routine, including measuring edge coverage and hit count, can be integrated seamlessly into the execution pipeline of the modern CPU architecture, and a near-zero tracing overhead can be achieved without the extra operations inevitable in software-based solutions. To enable such low-cost coverage tracing, SNAP incorporates two new micro-architectural units inside the CPU core: *Bitmap Update Queue (BUQ)* for generating updates to the coverage bitmap and *Last Branch Queue (LBQ)* for extracting last branch records. SNAP further adopts two micro-architectural optimizations to limit the overhead on the memory system from frequent coverage bitmap updates: *memory request aggregation*, which minimizes the number of updates, and *opportunistic bitmap update*, which maximizes the utilization of free cache bandwidth for such updates and reduces their cost.

Third, rich execution semantics can be extracted at the micro-architecture layer. One may think that the raw data gathered at the hardware level largely loses detailed program semantics because the CPU executes the program at the instruction granularity. Counter-intuitively, we find that such low-level information not only enables flexible coverage tracing, but also provides rich execution context for fuzzing without performance penalty. For example, various micro-architectural states are available in the processor pipeline during program execution, such as *lastly executed branches* (which incur higher overhead to extract in software) and *branch predictions* (which are entirely invisible to software). Using such rich micro-architectural information, SNAP is able to provide extra execution semantics, including *immediate control-flow context* and *approximated data flows*, in addition to code coverage. SNAP also supports setting address constraints on execution-specific micro-architectural states prior to execution, providing users the flexibility to selectively trace and test arbitrary program regions. Thus, fuzzers on SNAP can utilize the runtime feedback that describes the actual program state more precisely and make better mutation decisions.

3.2 Motivation: Precision and Performance

In this section, we provide an overview of coverage-guided fuzzing and introduce recent efforts in the research community to improve the quality of coverage feedback. We then dive into the excessive tracing overhead of source instrumentation from AFL, a state-of-the-art coverage guided fuzzer.

3.2.1 Better feedback in fuzzing

Fuzzing has recently gained wide popularity thanks to its simplicity and practicality. Fundamentally, fuzzers identify potential bugs by generating an enormous number of randomly mutated inputs, feeding them to the target program and monitoring abnormal behaviors. To save valuable computing resources for inputs that approach real bugs, modern fuzzers prioritize mutations on such inputs under the guidance of certain feedback metrics, one

```
1  static void demangle_it (char *mangled) {
2    char *cur = mangled;
3    ...
4    while (*cur != '\0') {
5      switch (*cur) {
6        case 'S': ... // static members
7        case 'L': ... // local classes
8        case 'T': ... // G++ templates
9        // more cases...
10     }
11   }
12   // buggy mangled pattern
13   if (has_SLLTS(mangled))  BUG();
14 }
15 int main (int argc, char **argv) {
16   ...
17   for (;;) {
18     static char mbuffer[32767];
19     unsigned i = 0;
20     int c = getchar();
21     // try to read a mangled name
22     while (c != EOF && ISALNUM(c) && i < sizeof(mbuffer)) {
23       mbuffer[i++] = c;
24       c = getchar();
25     }
26     mbuffer[i] = 0;
27     if (i > 0)  demangle_it(mbuffer);
28     if (c == EOF) break;
29   }
30   return 0;
31 }
```

Figure 3.1: An illustrative example for the runtime information gathered by SNAP. The code abstracts demangling in `cxxfilt`.

of which is code coverage. Coverage-guided fuzzers [57, 2, 59, 60] rely on the fact that the more program paths that are reached, the better the chance that bugs can be uncovered. Therefore, inputs that reach more code paths are often favored. Coverage guidance has proved its power by helping discover thousands of critical bugs and has become the design standard for most recent fuzzers [57, 2, 59, 60].

Despite its success, feedback that is solely based on the code coverage reached by the generated inputs can still be coarse grained. Figure 3.1 depicts an example of a buggy `cxxfilt` code snippet that reads an alphanumeric string from *stdin* (line 17-29) before demangling its contained symbols based on the signatures (line 4-11). Specifically, `BUG` in the program (line 13) results from a mangled pattern (*i.e.*, `SLLTS`) in the input. With a seed corpus that covers all the branch transfers within the loop (line 4-11), the coverage bitmap will be saturated even with the help of edge hit count, as shown in Algorithm 1, guiding the fuzzer to blindly explore the bug without useful feedback.

Algorithm 1: Edge encoding by AFL

Input : $BB_{src} \rightarrow BB_{dst}, prevLoc$

1 $curLoc = Random(BB_{dst})$

2 $bitmap[curLoc \hat{} prevLoc] \mathrel{+}= 1$

3 $prevLoc = curLoc \gg 1$

Output : $prevLoc$ – hash value for the next branch

```
# [Basic Block]:
# saving register context
mov    %rdx, (%rsp)
mov    %rcx, 0x8(%rsp)
mov    %rax, 0x10(%rsp)
# bitmap update
mov    $0x40a5, %rcx
callq  __afl_maybe_log
# restoring register context
mov    0x10(%rsp), %rax
mov    0x8(%rsp), %rcx
mov    (%rsp), %rdx
```

(a) AFL-gcc

```
# preparing 8 spare registers
push   %rbp
push   %r15
push   %r14
...
mov    %rax, %r14
# [Basic Block]: bitmap update
movslq %fs:(%rbx), %rax
mov    0xc8845(%rip), %rcx
xor    $0xca59, %rax
addb   $0x1, (%rcx,%rax,1)
movl   $0x652c, %fs:(%rbx)
```

(b) AFL-clang

Figure 3.2: Source-instrumented assembly inserted at each basic block between compilers.

To improve the quality of coverage feedback, much effort has been directed to more accurately approximate program states with extra execution semantics. In particular, to achieve *context awareness*, some fuzzers record additional execution paths if necessary [61, 9, 2, 11], while others track data-flow information [62, 60, 63, 64, 65, 59] that helps to bypass data-driven constraints. Those techniques enrich the coverage feedback and help a fuzzer approach the bugs in Figure 3.1 sooner, yet they can be expensive and thus limited. For example, traditional dynamic taint analysis can under-taint external calls and cause tremendous memory overhead. Although lightweight taint analysis for fuzzing [64] tries to reduce the overhead by directly relating byte-level input mutations to branch changes without tracing the intermediate data flow, it can still incur an additional 20% slowdown in the fuzzing throughput of AFL across tested benchmarks.

3.2.2 Dissecting AFL's tracing overhead

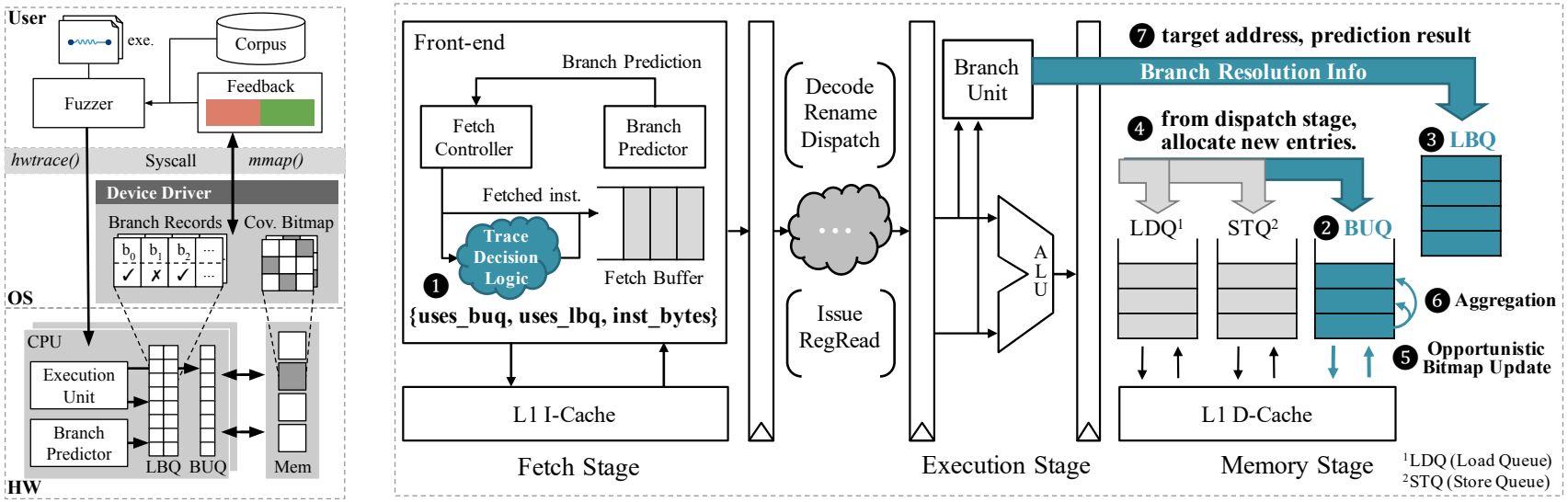
AFL injects the logic into a target program in two different ways based on the scenarios. When source code is available, AFL utilizes the compiler or the assembler to directly instrument the program. Otherwise, AFL relies on binary-related approaches such as DBI

Table 3.1: The cost of program size and runtime overhead for tracing on x86 platform across the SPEC benchmarks.

Name	Size (MB)		Runtime Overhead (%)	
	baseline	**instrumented**	**AFL-clang**	**AFL-QEMU**
perlbench	2.58	6.56	105.79	376.65
bzip2	0.95	1.20	63.66	211.14
gcc	4.51	15.73	57.15	257.76
mcf	0.89	0.95	66.30	92.52
gobmk	4.86	8.11	44.80	224.27
hmmer	1.51	2.57	39.34	340.03
sjeng	1.04	1.38	47.36	261.04
libquantum	1.10	1.23	47.95	186.63
h264ref	1.70	3.43	49.32	542.73
omnetpp	3.72	7.31	48.97	186.35
astar	1.10	1.39	43.57	124.93
xalancbmk	8.49	49.56	107.64	317.63
Mean	2.70	8.29 (207.04%)	60.15	260.14

and binary rewriting. While source code instrumentation is typically preferred due to its significantly lower tracing overhead compared to binary-related approaches, previous research indicates that AFL can still suffer from almost a 70% slowdown under the tested benchmarks [54]. Table 3.1 shows that the situation can worsen for CPU-bound programs, with an average tracing overhead of 60% from source code instrumentation and 260% from DBI (*i.e.*, QEMU). In the worst case, DBI incurs a 5× slowdown. The tracing overhead from DBI mostly comes from the binary translation of all applicable instructions and trap handling for privileged operations. On the other hand, the overhead of source code instrumentation results from the instructions inserted at each basic block that not only profiles coverage but also maintains the original register values to ensure that the instrumented program correctly runs. Figure 3.2a depicts the instrumentation conducted by `afl-gcc`, which requires assembly-level rewriting. Due to the crude assembly insertion at each branch, the instructions for tracing (line 7-8) are wrapped with additional instructions for saving and restoring register context (line 3-5 and line 10-12). Figure 3.2b shows the same processing done by `afl-clang`, which allows compiler-level instrumentation through *intermediate representation* (IR). The number of instructions instrumented for tracing can thus be minimized (line 8-12) thanks to compiler optimizations. Nevertheless, the

instructions for maintaining the register values still exist and blend into the entire workflow of the instrumented program (line 2-6). Table 3.1 lists the increased program sizes due to instrumentation by `afl-clang`, suggesting an average size increase of around $2\times$. The increase of program size and the runtime overhead given by `afl-gcc` can be orders of magnitude larger [66].



(a) Overview of SNAP.

(b) Architecture of the RISC-V BOOM core.

Figure 3.3: Overview of SNAP with its CPU design. The typical workflow involves the components from userspace, kernel, and hardware. The architecture highlights the modified pipeline stages for the desired features, including trace decision logic, Bitmap Update Queue (BUQ), and Last Branch Queue (LBQ).

3.3 Design

3.3.1 Overview

Motivated by the expensive yet inevitable overhead of existing coverage-tracing techniques, we propose SNAP, a customized hardware platform that implements hardware primitives to enhance performance and precision of coverage-guided fuzzing. A fuzzer coached by SNAP can achieve three advantages over traditional coverage-guided fuzzers.

① **Transparent support of fuzzing.** Existing fuzzers instrument each branch location to log the control-flow transfer, as explained in §3.2.2. When source code is not available, the fuzzer has to adopt slow alternatives (*e.g.*, Intel PT and AFL QEMU mode) to conduct coverage tracing, a much less favored scenario compared to source code instrumentation. By sitting in the hardware layer, SNAP helps fuzzers to construct coverage information directly from the processor pipeline without relying on any auxiliary added to the target program. SNAP thus enables transparent support of fuzzing any binaries, including third-party libraries or legacy software, without instrumentation, making fuzzing universally applicable.

② **Efficient hardware-based tracing.** At the hardware level, many useful resources are available with low or zero cost, most of which are not exposed to higher levels and cause excessive overhead to obtain through the software stack. For example, SNAP provides the control-flow information by directly monitoring each branch instruction and the corresponding target address that has already been placed in the processor execution pipeline at runtime, eliminating the effort of generating such information that is unavailable in the original program execution from a software perspective. This allows fuzzers to avoid program size increase and significant performance overhead due to the instrumentation mentioned in Table 3.1. In addition, SNAP utilizes idle hardware resources, such as free cache bandwidth, to optimize fuzzing performance.

③ **Richer feedback information.** To collect richer information for better precision of cov-

Table 3.2: New Control and Status Registers (CSRs) in SNAP.

Name	Permission	Description
BitmapBase	Read/Write	Base address of coverage bitmap
BitmapSize	Read/Write	Size of coverage bitmap
TraceEn	Read/Write	Switch to enable HW tracing
TraceStartAddr	Read/Write	Start address of traced code region
TraceEndAddr	Read/Write	End address of traced code region
LbqAddr[0-31]	Read Only	Target addresses of last 32 branches
LbqStatus	Read Only	Prediction result of last 32 branches
PrevHash	Read/Write	Hash of the last branch target inst.

erage, many existing fuzzers require performance-intensive instrumentation. Surprisingly, we observe that the micro-architectural state information already embeds rich execution semantics that are invisible from the software stack without extra data profiling and processing. In addition to code coverage, SNAP exposes those hidden semantics to help construct better feedback that can more precisely approximate program execution states without paying extra overhead. Currently, SNAP provides the records of lastly executed branches and the prediction results to infer immediate control-flow context and approximated data flows. We leave the support of other micro-architectural states as future work.

Figure 3.3a shows an overview of SNAP in action, which includes underlying hardware primitives, OS middleware for software support and a general fuzzer provided by the user. While running on SNAP, a fuzzer is allowed to configure the hardware and collect desired low-level information to construct input feedback through interfaces exposed by the OS. In addition, the fuzzer coached by SNAP can perform other fuzzing adjustments directly through the hardware level, such as defining code regions for dedicated testing on specific logic or functionalities.

3.3.2 Deep-dive: Implementation of Hardware Primitives

We design and implement SNAP based on the out-of-order BOOM core [67], one of the most sophisticated open-source RISC-V processors to match commercial ones with modern performance optimizations. Figure 3.3b highlights the lightweight modifications on key

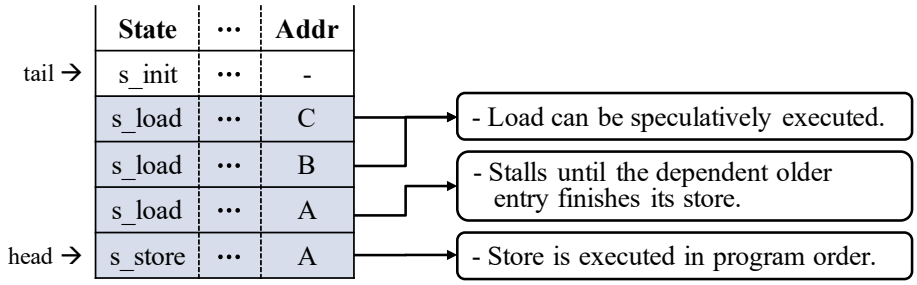


Figure 3.4: Bitmap update operation in the Bitmap Update Queue.

hardware components in SNAP.

Trace Decision Logic. In the front-end of the BOOM core (Figure 3.3b), instructions are fetched from the instruction cache (I-cache), enqueued to the Fetch Buffer, and then sent onward for further execution at every cycle. We extend the Fetch Controller by adding the Trace Decision Logic (❶), which determines whether an instruction inserted into the Fetch Buffer needs to be traced by SNAP. The trace decision results in tagging two types of instructions within the code region to be traced (*i.e.*, between `HwTraceStartAddr` and `HwTraceEndAddr`) using two reserved bits, `uses_buq` and `uses_lbq`. The `uses_buq` bit is used to tag the target instruction of every control-flow instruction (*i.e.*, a branch or a jump) to help enqueued bitmap update operations into the BUQ. Note that we choose to trace the target instruction instead of the control-flow instruction itself for bitmap update due to our newly devised trace-encoding algorithm (described later in §3.3.4). The control-flow instruction itself is also tagged with the `uses_lbq` bit to help enqueue the corresponding branch resolution information (*i.e.*, the target address and the prediction result) into the LBQ for additional contextual semantics (described later in §3.3.5). Overall, the trace decision logic conducts lightweight comparisons within a single CPU cycle in parallel to the existing fetch controller logic and thus does not delay processor execution or stall pipeline stages.

Bitmap Update Queue. The BUQ (❷) is a circular queue responsible for bitmap updates invoked by the instructions following control-flow instructions. A new entry in the BUQ is allocated when such an instruction tagged with `uses_buq` is dispatched during the Execute Stage (❹). Each entry stores the metadata for a single bitmap update operation (❺) and

performs the update sequentially through four states:

1. `s_init`: The entry is first initialized with the bitmap location to be updated, which is calculated using our trace encoding algorithm described in §3.3.4.

2. `s_load`: Subsequently, the current edge count at the bitmap location is read from the appropriate memory address.

3. `s_store`: Then, the edge count is incremented by one and written to the same bitmap location stored in the entry.

4. `s_done`: Once the entry reaches this state, it is deallocated when it becomes the head of the BUQ.

Figure 3.4 depicts the bitmap update operation in the BUQ designed in a manner that imposes minimal overhead. Since the bitmap itself is stored in user-accessible memory, its contents can be read and written via load and store operations with the base address of the bitmap and specific offsets. To ensure the bitmap update operation does not stall the CPU pipeline, the load part of the update operation is allowed to proceed speculatively in advance of the store operation, which is only executed when the corresponding instruction is committed. However, in case there are store operations pending for the same bitmap location from older instructions, such speculative loads are delayed until the previous store completes to prevent reading stale bitmap values. Moreover, each bitmap load and store is routed through the cache hierarchy, which does not incur the slow access latency of the main memory. Note that the cache coherence and consistency of the bitmap updates can be ensured by the hardware in a manner similar to that for regular loads and stores in the shared memory. Last, a full BUQ can result in back pressure to the Execution Stage and cause pipeline stalls. To avoid this, we sufficiently size up the BUQ; our 24-entry BUQ ensures that such stalls are infrequent and incur negligible overhead.

Last Branch Queue. The LBQ (❸) is a circular queue recording the information of the last 32 branches as context-specific feedback used by a fuzzer, as we describe in §3.3.5.

Specifically, each entry of the LBQ stores the target address and the prediction result for a branch (*i.e.*, what was the branch direction and whether the predicted direction from the branch-predictor was correct or not). Such information is retrieved through the branch resolution path from the branch unit (❼), where branch prediction is resolved. To interface with the LBQ, we utilize the CSRs described in Table 3.2. Each LBQ entry is wired to a defined CSR and can be accessible from software after each fuzzing execution using a CSR read instruction.

3.3.3 Micro-architectural Optimizations

Since the BUQ generates additional memory requests for bitmap updates, it may increase cache port contention and cause non-trivial performance overhead. To minimize the performance impact, we rely on the fact that the bitmap update operation is not on the critical path of program execution, independent of a program's correctness. Hence, the bitmap update can be *opportunistically* performed during the lifetime of a process and also *aggregated* with subsequent updates to the same location. Based on the observations, we develop two micro-architectural optimizations.

Opportunistic bitmap update. At the Memory Stage in Figure 3.3b, memory requests are scheduled and sent to the cache based on the priority policy of the cache controller. To prevent normal memory requests from being delayed, we assign the lowest priority to bitmap update requests and send them to the cache only when unused cache bandwidth is observed or when BUQ is full. Combined with the capability of the out-of-order BOOM core in issuing speculative bitmap loads for the bitmap updates, this approach allows us to effectively utilize the free cache bandwidth while also minimizing the performance impact caused by additional memory accesses.

Memory request aggregation. A memory request aggregation scheme (❻) is also deployed to reduce the number of additional memory accesses. When the head entry of the BUQ issues a write to update its bitmap location, it also examines the other existing entries,

which might share the same influencing address for subsequent updates. If found, the head entry updates the bitmap on behalf of all the matched ones with the influence carried over, while the represented entries are marked finished and deallocated without further intervention. This is effective, especially for loop statements, where the branch instruction repeatedly jumps to the same target address across most iterations. In that case, the BUQ can aggregate the bitmap update operations aggressively with fewer memory accesses.

3.3.4 Edge Encoding

Algorithm 1 describes how AFL [57] measures edge coverage, where an edge is represented in the coverage bitmap as the hash of a pair of randomly generated basic block IDs inserted during compile time. To avoid colliding edges, the randomness of basic block IDs plays an important role to ensure the uniform distribution of hashing outputs. Rather than utilizing a more sophisticated hashing algorithm or a bigger bitmap size to trade efficiency for accuracy, AFL chooses to keep the current edge-encoding mechanism, as its practicality is well backed by the large number of bugs found. Meanwhile, software instrumentation for coverage tracing requires excessive engineering effort and can be error-prone, especially in the case of complex COTS binaries without source code. Since it is non-trivial to instrument every basic block with a randomly generated ID, one viable approach is to borrow the memory address of a basic block as its ID, which has been proven effective in huge codebase [21]. Such an approach works well on the x86 architecture where instructions have variable lengths, usually ranging from 1 to 15 bytes, to produce a decent amount of entropy for instruction addresses to serve as random IDs. In the case of the RISC-V architecture, however, instructions are defined with fixed lengths. Standard RISC-V instructions are 32-bit long, while 16-bit instructions are also possible only if the ISA compression extension (RVC) is enabled [68]. As a result, RISC-V instructions are well-aligned in the program address space. Reusing their addresses directly as basic block IDs for edge encoding lacks enough entropy to avoid collisions.

Algorithm 2: Edge encoding by SNAP

Input : $BB_{src} \rightarrow BB_{dst}, prevLoc$

1 $p = Address(BB_{dst})$
2 $inst\_bytes = InstBytes(BB_{dst})$
3 $curLoc = p \hat{} inst\_bytes[15:0] \hat{} inst\_bytes[31:16]$
4 $bitmap[curLoc \hat{} prevLoc] \mathrel{+}= 1$
5 $prevLoc = curLoc \gg 1$

Output : $prevLoc$ – hash value for the next branch

To match the quality of edge encoding in AFL, we devise a new mechanism (Algorithm 2) for SNAP that produces a sufficient amount of entropy with no extra overhead compared to naive employment of memory addresses. Specifically, SNAP takes both the memory address and the instruction byte sequence inside a basic block to construct its ID. A pair of such basic block IDs are then hashed to represent the corresponding edge in the coverage bitmap. By sitting at the hardware level, SNAP is able to directly observe and leverage the instruction bytes as a source of entropy without overhead to compensate the lack of randomness due to the RISC-V instruction alignment. To be compatible with both 16- and 32-bit long RISC-V instructions, SNAP always fetches two consecutive 16-bit sequences starting at the instruction address and performs *bitwise XOR* twice to produce a basic bock ID (line 3 in Algorithm 2, also in Figure 3.5). Therefore, each ID contains the entropy from various data fields, including *opcode*, *registers*, and *immediates*, of either an entire instruction or two compressed ones. In addition, SNAP chooses to operate on the destination instruction of a branching operation to construct a basic block ID, as it provides a larger variety of instruction types (*i.e.*, more entropy) than the branch instruction itself. Similar to that of AFL, the encoding overhead of SNAP is considered minimal, as the operation can be completed within one CPU cycle. Note that Algorithm 2 can be easily converted to trace basic block coverage by discarding `prevLoc` (line 5), which tracks control transfers (*i.e.*, edges), and performing bitmap update (line 4) solely based on `curLoc` (line 3).

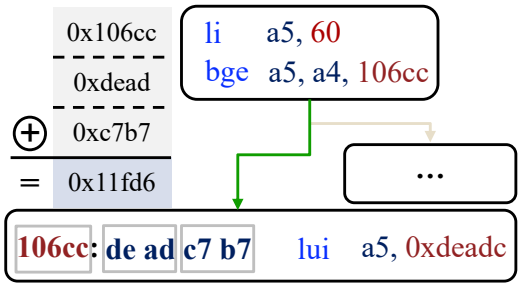


Figure 3.5: An example of encoding a basic block ID.

3.3.5 Richer Coverage Feedback

As discussed in §3.2.1, edge coverage alone can be coarse-grained and does not represent execution states accurately. Meanwhile, collecting additional execution semantics via software-based solutions always incurs major performance overhead. SNAP aims to solve the dilemma from a hardware perspective. With various types of micro-architectural state information available at the hardware level, SNAP helps fuzzers to generate more meaningful feedback that incorporate *immediate control flow context* and *approximated data flow* of a program run without extra overhead.

Capturing immediate control-flow context. Tracking long control-flow traces can be infeasible due to noise from overly sensitive feedback and the performance overhead from comparing long traces. Therefore, SNAP records only the last 32 executed branches of a program run in the circular LBQ by default. Note that SNAP provides the flexibility of configuring branch entry number and address filters through software interfaces so that the hosted fuzzer can decide to track the execution trace of an arbitrary program space, ranging from a loop to a cross-function code region. A unique pattern of the 32 branch target addresses recorded in LBQ captures the *immediate control-flow context* of a program execution, such as the most recent sequence of executed parsing options inside string manipulation loops. When the immediate control-flow context is included in coverage feedback, a fuzzer is inspired to further mutate the inputs that share identical edge coverage but trigger unseen branch sequences within the loop (Figure 3.1 line 4-11) that will otherwise be discarded. As a result, the fuzzer is more likely to generate the input that can reach the

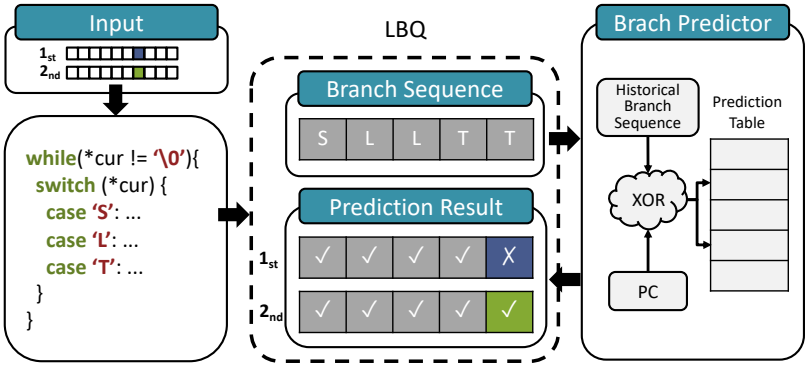


Figure 3.6: An example of data flow approximation between two runs leveraging the branch predictions stored in LBQ.

specific lastly executed branch sequence (*i.e.*, SLLTS) for the buggy constraint (line 13).

Approximating data flow via branch prediction. Data flow analysis has proven to be useful for fuzzers [62, 60, 63] to mutate inputs more precisely (*e.g.*, identify the input byte that affects a certain data-dependent branch condition). However, recent research [64] points out that the traditional data-flow analysis requires too much manual effort (*e.g.*, interpreting each instruction with custom taint propagation rules) and is slow, making fuzzing less effective. Surprisingly, SNAP is able to provide an approximation of data flow without paying extra performance overhead by leveraging the branch prediction results in the LBQ. A typical branch predictor, such as the one used in RISC-V BOOM [69] and shown in Figure 3.6, is capable of learning long branch histories and predicts the current branch decision (*i.e.*, taken vs. not-taken) based on the matching historical branch sequence. Conversely, given the prediction results of the recorded branch sequence in the LBQ, SNAP is able to infer a much longer branch history than the captured one. Therefore, if a mutated input byte causes a change in the prediction result of a certain branch, the branch condition is likely related to the input offset, thus revealing dependency between them with near-zero cost. Since most branch conditions are data-dependent [70, 63, 64], the prediction result thus approximates the data flow from the input to any variables that affect the branch decision. In Figure 3.6, even if the coverage map and the immediate control-flow context remain the same, the fuzzer can still rely on the approximated data flows to mutate further for the sequence of interest (*i.e.*, line 13 in Figure 3.1) when it is not captured by the LBQ.

3.3.6 OS Support

Besides the hardware modification, kernel support is another key for SNAP to work as expected. We generalize it into three components, including configuration interface, process management, and memory sharing between kernel and userspace. Rather than testing the validity of modified OS on the hardware directly, we also provide the RISC-V hardware emulation via QEMU [47], allowing easier debugging of the full system.

Configuration interface. Among the privilege levels enabled by the standard RISC-V ISA [68], we define the newly added CSRs (Table 3.2) in `supervisor-mode` (`S-mode`) with constraints to access through the kernel. To configure the given hardware, SNAP provides custom and user-friendly system calls to trace a target program by accessing the CSRs. For example, one can gather the lastly executed branches to debug a program near its crash site by enabling branch record only. Others might request dedicated fuzzing on specific code regions or program features by setting the address range to be traced. Overall, SNAP is designed to be flexible for various use cases.

Process management. Besides the software interface that allows user programs to configure hardware through system calls, the kernel also manages the tracing information of each traced process. Specifically, we add new fields to the process representation in the Linux kernel (*i.e.*, `task_struct`), including the address and the size of the coverage bitmap, the address and the size of the branch queue, the tracing address range, and the previous hash value. Those fields are initialized with zeros upon process creation and later assigned accordingly by the system calls mentioned before. During a context switch, if the currently executing process is being traced, the kernel disables the tracing and saves the hash value and branch records in the hardware queue. In another case, if the next process is to be traced, the SNAP CSRs will be set based on the saved field values to resume the last execution. Note that when fuzzing a multi-threaded application, existing fuzzers typically do not distinguish code paths from different threads but record them into one coverage bitmap to test the

application as a whole. Although maintaining unique bitmaps is supported, SNAP enables the kernel to copy all the SNAP-related fields of a parent process, except the address of the branch queue, into its newly spawned child process by default. In addition, when a target process exits, either with or without error, SNAP relies on the kernel to clean up the corresponding fields during the exit routine of the process. However, the memory of the coverage information and the branch queue will not be released immediately, as it is shared with the designated user programs (*i.e.*, fuzzers) to construct the tracing information. Instead, the memory will be freed on demand once the data have been consumed in the userspace.

Memory sharing. To share the memory created for the coverage bitmap and the branch queue with the userspace program, SNAP extends the kernel with two corresponding device drivers. In general, the device drivers enable three *file operations*: `open()`, `mmap()`, and `close()`. A user program can create a kernel memory region designated for either of the devices by *opening* the device accordingly. The created memory will be maintained in a kernel array until it is released by the *close* operation. Moreover, the kernel can *remap* the memory to userspace if necessary. The overall design is similar to that of `kcov` [71], which exposes kernel code coverage for fuzzing.

3.4 Evaluation

We perform empirical evaluations on the benefits of SNAP on fuzzing metrics and answer the following questions:

- **Performance.** How much performance cost needs to be paid for tracing on SNAP?
- **Effectiveness.** Can SNAP increase coverage for fuzzing in a finite amount of time? How do branch records and branch predictions provide more context-sensitive semantics?
- **Practicality.** How easy is it to support various fuzzers on SNAP? How much power and area overhead does the hardware modification incur?

Table 3.3: Evaluated BOOM processor configuration.

Clock	75 MHz	**L1-I cache**	32KB, 8-way
LLC	4MB	**L1-D cache**	64KB, 16-way
DRAM	16 GB DDR3	**L2 cache**	512KB, 8-way
Front-end		8-wide fetch 16 RAS & 512 BTB entries gshare branch predictor	
Execution		3-wide decode/dispatch 96 ROB entries 100 int & 96 floating point registers	
Load-store unit		24 load queue & 24 store queue entries 24 BUQ & 32 LBQ entries	

3.4.1 Experimental setup

We prototype SNAP on Amazon EC2 F1 controlled by FireSim [72], an open-source FPGA-accelerated full-system hardware simulation platform. FireSim simulates RTL designs with cycle-accurate system components by enabling FPGA-hosted peripherals and system-level interface models, including a last-level cache (LLC) and a DDR3 memory [73]. We synthesize and operate the design of SNAP at the default clock frequency of `LargeBoomConfig`, which is applicable to existing CPU architectures without significant design changes. While modern commercial CPUs tend to adopt a data cache (L1-D) larger than the instruction cache (L1-I) for performance [74, 75, 76], we mimic the setup with the default data cache size of 64 KB for our evaluation. In general, the experiments are conducted under Linux kernel v5.4.0 on *f1.16xlarge* instances with eight simulated RISC-V BOOM cores, as configured in Table 3.3. Our modified hardware implementation complies with the RISC-V standard, which has been tested with the official RISC-V verification suite. The area and power overhead of the synthesized processor after modification is measured by a commercial EDA tool, Synopsys Design Compiler [77].

We evaluate SNAP on the industry-standardized SPEC CPU2006 benchmark suite to measure its tracing overhead. The reference workload is used on the 12 C/C++ benchmarks compilable by the latest RISC-V toolchain to fully explore the CPU-bound benchmarks for a more accurate measurement. Meanwhile, we test AFL's runtime coverage increase and

Table 3.4: Tracing overhead from AFL source instrumentation and SNAP with various L1-D cache sizes across the SPEC benchmarks.

Name	HyperSet (%)			AFL-gcc (%)
	32 KB	64 KB	128 KB	
perlbench	7.63	4.28	4.20	690.27
bzip2	2.32	2.21	2.10	657.05
gcc	7.85	5.11	4.97	520.81
mcf	1.75	1.54	1.54	349.83
gobmk	16.92	5.25	4.92	742.98
hmmer	0.72	0.60	0.54	749.56
sjeng	7.29	0.68	0.52	703.44
libquantum	0.80	0.67	0.44	546.67
h264ref	10.37	0.27	0.07	251.56
omnetpp	13.88	5.55	5.37	452.89
astar	0.37	0.30	0.30	422.96
xalancbmk	21.24	11.26	11.11	1109.24
Mean	7.59	3.14	3.00	599.77

throughput with Binutils v2.28 [78], a real-world collection of binary tools that have been widely adopted for fuzzing evaluation [79, 80]. In general, we fuzz each binary for 24 hours with the default corpus from AFL in one experimental run and conduct five consecutive runs to average the statistical noise in the observed data.

3.4.2 Tracing Overhead by SNAP

We measure the tracing overhead imposed by SNAP and source instrumentation (*i.e.*, AFL-gcc[1]) across the SPEC benchmarks. Table 3.4 suggests that SNAP incurs barely 3.14% overhead with the default cache size of 64 KB, significantly outperforming the comparable software-based solution (599.77%). While we have excluded the numbers for DBI solutions (*e.g.*, AFL QEMU mode), the resulting overhead is expected to be much heavier than source instrumentation, as explained in §3.2.2. The near-zero tracing overhead of SNAP results from its hardware design optimizations, including optimistic bitmap update and memory request aggregation (§3.3.3). Table 3.5 shows that the bitmap update requests have been reduced by 13.47% on average thanks to aggregation. In the best case, the reduction rate can reach above 40%, which effectively mitigates cache contention from frequent memory

[1]We use AFL-gcc rather than AFL-clang because LLVM has compatibility issues in compiling the SPEC benchmarks.

Table 3.5: Memory request aggregation rates and L1 cache hit rates between the baseline and SNAP across the SPEC benchmarks.

Name	Agg. Rate (%)	L1 Cache Hit Rate (%)		
		Base	HYPERSET	Δ
perlbench	3.32	97.82	96.49	-1.33
bzip2	13.67	91.80	91.32	-0.47
gcc	25.14	68.53	67.42	-1.11
mcf	7.83	44.45	43.89	-0.56
gobmk	8.78	95.51	91.81	-3.70
hmmer	1.36	95.80	95.64	-0.17
sjeng	5.24	98.44	96.18	-2.26
libquantum	41.60	53.97	53.24	-0.73
h264ref	16.23	96.67	95.89	-0.78
omnetpp	4.69	82.10	79.68	-2.42
astar	3.77	87.39	87.09	-0.30
xalancbmk	30.04	82.94	77.17	-5.77
Mean	13.47	82.95	81.32	-1.63

accesses (*e.g.*, array iteration) and avoids unnecessary power consumption.

Further investigation shows that the performance cost of SNAP might also result from cache thrashing at the L1 cache level. In general, applications with larger memory footprints are more likely to be affected. Since bitmap updates by the BUQ are performed in the cache shared with the program, cache lines of the program data might get evicted when tracing is enabled, resulting in subsequent cache misses. Note that this problem is faced by all existing fuzzers that maintain a bitmap. For instance, Table 3.5 points out that `gobmk` and `xalancbmk` both suffer from comparably higher overhead ($\geq 5\%$) caused by reduced cache hit rates of over 3.5%. The impact of cache thrashing can also be tested by comparing the tracing overhead of SNAP configured with different L1 D-cache sizes. Table 3.4 shows that a larger cache exhibits fewer cache misses and can consistently introduce lower tracing overhead across benchmarks. In particular, the overhead can be reduced to 3% on average by increasing the cache size to 128 KB. Alternatively, the extra storage can be repurposed as a dedicated buffer for the coverage bitmap to avoid cache misses due to bitmap update, which we leave for future work.

In addition, Table 3.4 shows that the tracing overhead of AFL-gcc is much larger. With the CPU-bound benchmarks that best approximate the extreme circumstances, the overhead

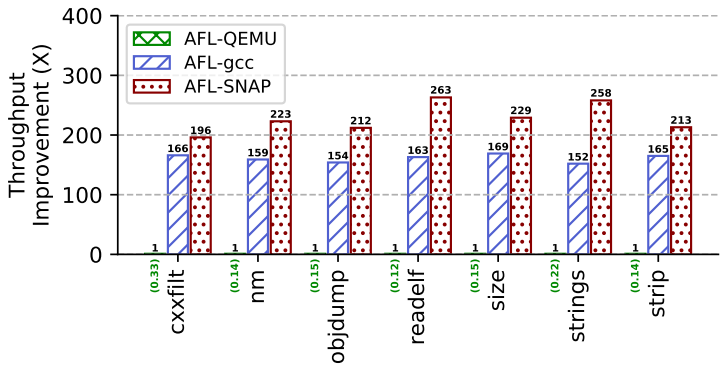


Figure 3.7: The average execution speed from fuzzing with AFL-QEMU, AFL-gcc and AFL-SNAP for 24 hours across the Binutils binaries. The numbers below the bars of AFL-QEMU show the number of executions per second for the mechanism.

is expected [66], as discussed in §3.2.2. This finding is generally consistent with the numbers from the x86 setup, which also incurs an average of 228.09% overhead on the same testing suite by AFL-gcc. The extra slowdown in the current RISC-V experiment is caused by the additional instrumented locations in binaries due to different ISAs. For example, RISC-V specification does not define a specific instruction for backward edges (*i.e.*, `ret`), which are not tracked on x86 binaries when instrumented. Thus, the RISC-V benchmarks have 58.51% more instrumentation than the x86 version, resulting in a 40.03% increase of binary size. Note that the cache size has negligible impact on the tracing overhead for the software-based solution. Although bitmap updates can still cause cache thrashing, the overhead mainly comes from the execution cost of instrumented instructions.

3.4.3 Evaluating Fuzzing Metrics

To understand how SNAP improves fuzzing metrics, we evaluate it on seven Binutils binaries. Given the default corpus, we compare the code coverage and runtime throughput of AFL running for 24 hours under the existing DBI scheme (*i.e.*, AFL-QEMU), source instrumentation (*i.e.*, AFL-gcc), and support of SNAP.

Fuzzing throughput. Figure 3.7 shows the fuzzing throughput across the compared mechanisms. Specifically, AFL on SNAP can achieve 228× higher execution speed than AFL-QEMU, which is limited by the low clock frequency and its inefficient RISC-V support.

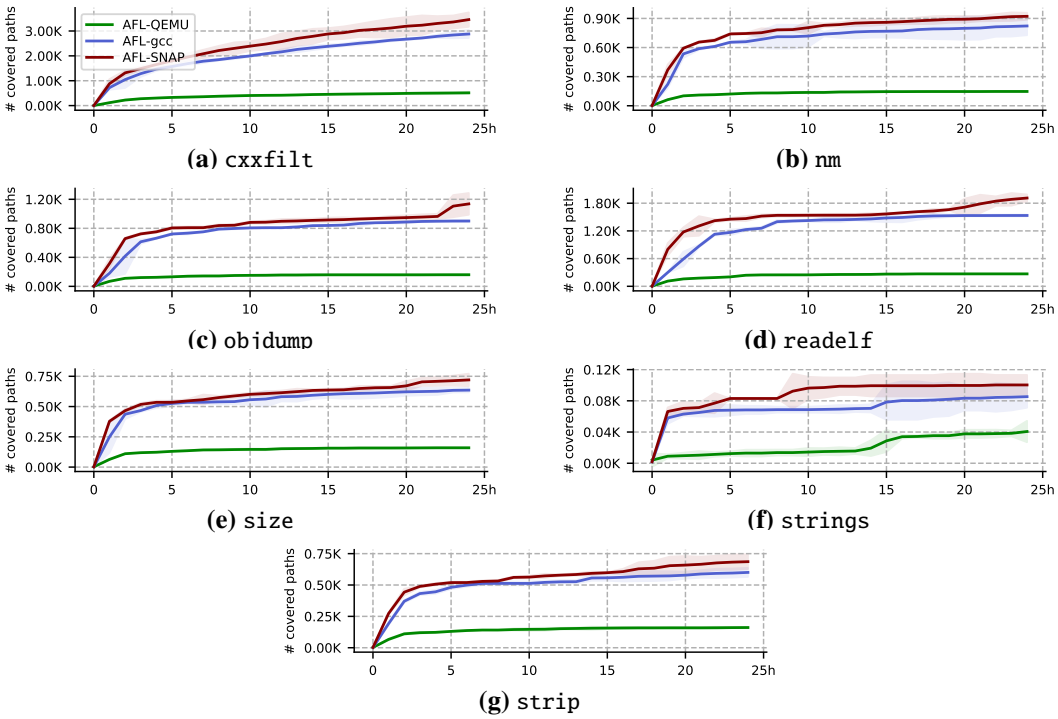


Figure 3.8: The overall covered paths from fuzzing seven Binutils binaries for 24 hours. The solid lines represent the means, and the shades suggest the confidence intervals of five consecutive runs.

The average throughput of AFL-QEMU (*i.e.*, 0.18 exec/s) is consistent with the previous findings in PHMon [4]. Note that SNAP improves the throughput much more significantly than PHMon, which only achieves a 16× higher throughput than AFL-QEMU. Despite that the baseline BOOM core in SNAP is about 50% faster [81] than the baseline Rocket core [82] adopted by PHMon, the difference of 14× more throughput accomplished by SNAP mainly results from its design optimizations (*e.g.*, optimistic bitmap update and memory request aggregation). Compared to AFL-gcc, SNAP can still achieve a 41.31% higher throughput on average across the benchmarks.

Edge coverage. Figure 3.8 depicts the resulting coverage measurement, where the confidence intervals indicate the deviations across five consecutive runs on each benchmark. Given an immature seed corpus and a time limit, AFL with SNAP consistently covers more paths than the others throughout the experiment. Since no change to fuzzing heuristics (*e.g.*, seed selection or mutation strategies) is made, the higher throughput of SNAP is the key contributor to its winning. On average, AFL-QEMU and AFL-gcc have only reached 23.26%

and 84.59% of the paths discovered by AFL-SNAP, respectively. Although larger deviations can be observed when the program logic is relatively simple (Figure 3.8f), SNAP in general can help explore more paths in programs with practical sizes and complexity thanks to its higher throughput. For example, AFL with SNAP manages to find 579 (16.74%), 237 (20.82%), 378 (19.77%) more paths when fuzzing `cxxfilt`, `objdump`, and `readelf`, respectively.

Adopting execution context. Given the lastly executed branches and their prediction results in LBQ, fuzzers on SNAP are equipped with additional program states. To take the feedback, one can easily follow the mechanisms introduced previously [9, 2, 11]. Our prototype of AFL instead adopts a similar feedback encoding mechanism as Algorithm 1 to showcase the usage. Specifically, the highest significant bit (HSB) of each 64-bit branch address is set based on the respective prediction result (*i.e.*, 1/0). To maintain the order of branches, the records are iterated from the least recent to the latest in the circular LBQ and *right circular shift*'ed (*i.e.*, rotated) by N bits based on their relative positions in the sequence before being *bitwise XOR*'ed. The encoded value is finally indexed into a separate comparing bitmap from the one for edge coverage (*i.e.*, *trace_bits*).

Reproducing a known bug. Running on SNAP, the modified AFL is able to trigger `CVE-2018-9138` discovered by the previous work [9], which proposes to use feedback similar to that provided by our platform. As in Figure 3.1, the vulnerability occurs when `cxxfilt` consumes a long *consecutive* input of "F"s, each indicating that the current mangled symbol stands for a function. The corresponding switch case in the loop (line 5-10) tries to further demangle the function arguments (*i.e.*, `demangle_args()`) before running into the next "F" to start a recursive call chain. Luckily, SNAP offers the execution context by capturing branch sequences triggered by mutated inputs. While a vanilla AFL cannot easily reach the faulty program state with only edge coverage feedback, our fuzzer can consistently achieve it within one fuzzing cycle, led by the guidance.

Table 3.6: Estimates of area and power consumption.

Description	Area (mm^2)	Power (mW)
BOOM core	9.2360	36.4707
HYPERSET core	9.6811	38.8513

3.4.4 Practicality of SNAP

To estimate the area and power overhead of SNAP, we synthesize our design using Synopsys Design Compiler at 1GHz clock frequency. To obtain a realistic estimate of the SRAM modules such as L1 D-cache, L1 I-cache, and branch predictor (BPD) tables used in the BOOM core, we black-box all the SRAM blocks and use analytical models from OpenRAM [83]. Our synthesis uses 45nm FreePDK libraries [84] to measure the area and power consumption between the unmodified BOOM core and the modified SNAP core. Table 3.6 shows that SNAP only incurs 4.82% area and 6.53% power overhead, more area-efficient than the comparable solution (16.5%) that enables hardware-accelerated fuzzing [4]. When tracing is disabled, the power overhead can be mostly avoided by clock gating through the switch CSR *TraceEn*.

3.5 Limitations and future directions

While SNAP is carefully desgined not to hamper the maximum clock frequency, we are limited in our evaluation to a research-grade hardware setup with low clock speed. We hope our work motivates future studies, and adoption on more powerful cores [85] and custom ASICs by processor vendors [86]. Besides, while SNAP does not support kernel coverage filtered by the privilege level, leveraging the hardware for tracing kernel space is not fundamentally restricted. Meanwhile, SNAP is not suitable to track dynamic code generation with reused code pages, such as JIT and library loading/unloading, as it affects the validity of the coverage bitmap. If needed, annotations with filters on the program space can be applied to reduce noise. Future work includes repurposing a buffer dedicated for coverage bitmap storage to avoid extra cache misses, leveraging other micro-architectural states from

hardware, such as memory access patterns, to identify dynamic memory allocations (*e.g.*, heap) across program runs, or adopting operands of comparing instructions for feedback as suggested [87]. Alternatively, given filters in the debug unit of ARM's CoreSight extension [88], the practicality of the design can be further demonstrated without relying on custom hardware.

CHAPTER 4
HYPERSET: NESTED VIRTUALIZATION FRAMEWORK FOR PERFORMANT HYPERVISOR FUZZING

4.1 Introduction

Given a total market cap of $130 billion in 2020 [89], Amazon Web Services (AWS) [18], for example, serves as many as 2 million registered parties, including publicly traded companies (*e.g.*, Netflix) and US military services, and still poses a yearly growth of 36.5% despite the potential slowdown caused by the pandemic. Due to the popularity from daily usage, cloud services have become the new battleground between attackers and security researchers. While almost all major providers have their own bug-bounty program, a bug in the off-market can be priced at half a million based on Zerodium [90]. Microsoft has even released the debugging symbols of Hyper-V [91], the hypervisor essential to its cloud security, for gathering the public effort of bug finding from the community. This is because a zero-day vulnerability from any cloud-based solutions can cause severe consequences, including denial of service, information leakage, or VM escape, jeopardizing the integrity of the other instances running on the same platform.

While fuzzing remains one of the most effective bug-finding techniques in most real-world programs, the challenges of applying it in the current context can be non-trivial. Particularly to non-user applications, like hypervisors, the overhead of coverage tracing might not necessarily be the only root cause of the slowdown in pursuit of fuzzing speed. Instead, the major overhead comes from continuous VM reboots after each fuzzing execution to ensure determinism. To mitigate this problem, Syzkaller [21] sacrifices reproducibility of unit testing for speed and only reboots periodically. Other approaches [22, 23] propose customized guest OSes as the main executors of fuzzing inputs for faster reboot time

compared to the default Linux Kernel configuration, yet still are throttled by the throughput of up to four executions per second. Neither solution is thus ideal, as they fundamentally lack either determinism or efficiency to fulfill the requirement of an effective fuzzing framework.

The current work aims to develop a hardware-based solution for this specialized testing target, *i.e.*, hypervisors. Besides the desired determinism and efficiency, it is also designed to be compatible with most existing hypervisors that can run on the Linux platform, such as QEMU and VirtualBox. Specifically, a nested virtualization framework consisting of two major components is proposed, including ① a snapshot and restoring mechanism realized by Intel VMX and ② a minimized OS with little redundancy. In our framework, since the testing hypervisor resides in the L1 VM, the L0 host acts as the snapshot maintainer, taking and reloading the intended snapshot for faster reset. By resetting virtualized CPU, memory, and devices per request, the framework effectively prohibits side effects carried over from previous executions (*i.e.*, determinism) without really rebooting the tested VM (*i.e.*, efficiency). Another source of overhead in the framework comes from the complexity of the compiled kernel image. While the L2 guest kernel is only expected to execute fuzzing inputs, applying the Linux Kernel can be overkill, downgrading performance through additional VM exiting requests and memory overhead. The minimized OS trims the unnecessary OS features in the current context to perform the snapshot and restore operations in a more efficient manner.

4.2 Background

4.2.1 x86 Virtualization

Over the last decade, cloud-based solutions have been flourishing, thanks to the emergence of hardware virtualization extensions to their x86 platforms [6, 92], given the support of root mode and VMCS for CPU virtualization, as well as EPT for efficient memory mapping. Despite the few key differences between type-I and type-II in terms of design, hypervisors in general serve as the coordinator of all, providing an abstraction of hardware resources and

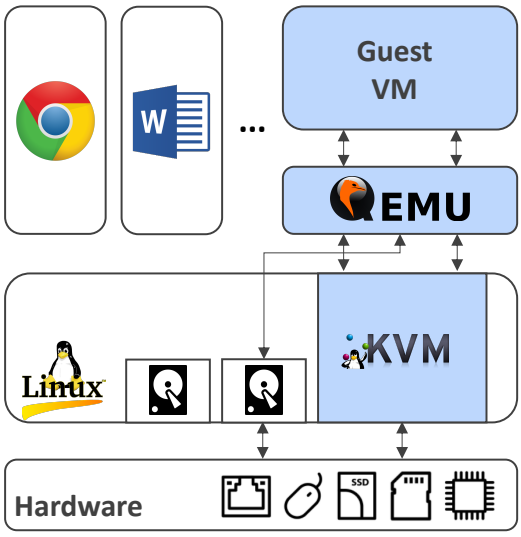
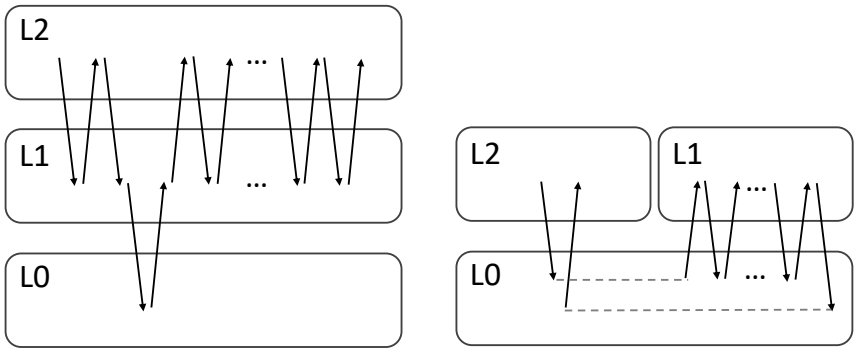


Figure 4.1: An overview of KVM/QEMU for x86 virtualization.

isolation of guest VMs. Figure 4.1 shows an overview of x86 virtualization on the Linux platform. In particular, a VM is launched as a child process of QEMU, which works with KVM in the kernel space to configure the virtualization specifications. While QEMU has direct control of the VM, such as vCPU execution and guest memory, it also serves as the emulator for IO requests by interpreting the IO addresses and the associated virtual devices enabled by the guest machine. More miscellaneous features, including dirty page tracking for live migration of VMs [93], are also supported.

4.2.2 Nested Virtualization

The nested virtualization scheme was initially proposed and developed for testing and benchmarking hypervisors and virtualization setups [94, 95]. KVM [96] was the first hypervisor to support the idea of nested virtualization, which allows running a VM (*i.e.*, L2) inside another VM (*i.e.*, L1) on the same host (*i.e.*, L0). Compared to the single-layer virtualization scheme where L0 serves as the hypervisor of the L1 guest, L1 now also becomes the virtual machine monitor (VMM) of the L2 guest. Ideally, both L0 and L1 should have the dedicated responsibilities for controlling their own guest VMs. Figure 4.2a shows the stacked interactions between the nested layers, where most privileged operations (*i.e.*, VM exit) from L2 are supposed to be trapped by L1, before some are rendered to L0



(a) Logical view. **(b)** Implementationl view.

Figure 4.2: The nested virtualization scheme supported on x86 platforms.

for further processing if they cannot be handled properly.

Nevertheless, the actual implementation of nested virtualization differs from the logical view. This is mainly due to the lack of support from the x86 extensions, which only provide a single-level architectural support, fundamentally limited by the design of the hypervisor mode, VMCS, and so on. In other word, only one hypervisor is allowed to run in the root mode, supporting multiple guests at the same time. To overcome the barriers to the desired goal, L0 *multiplexes* the extensions between L1 and L2 for its virtualization. Instead of letting L1 take over, L0 serves as the intermediate layer to relay the requests from L2 to L1, without L1 being aware of the non-root mode it resides in. Figure 4.2b shows the abstracted handling workflow of a single VM exit from the L2 nested guest. On the other hand, the performance is hurt by the relaying job of L0 due to *exit multiplication*. In extreme cases, a single L2 exit can cause more than 40 L1 exits [94]. Although many works [97, 98, 99] try to improve this situation with virtual passthrough for occasionally allowing direct access hardware, the additional overhead is mostly guaranteed with the design.

4.2.3 Attack surfaces on hypervisors

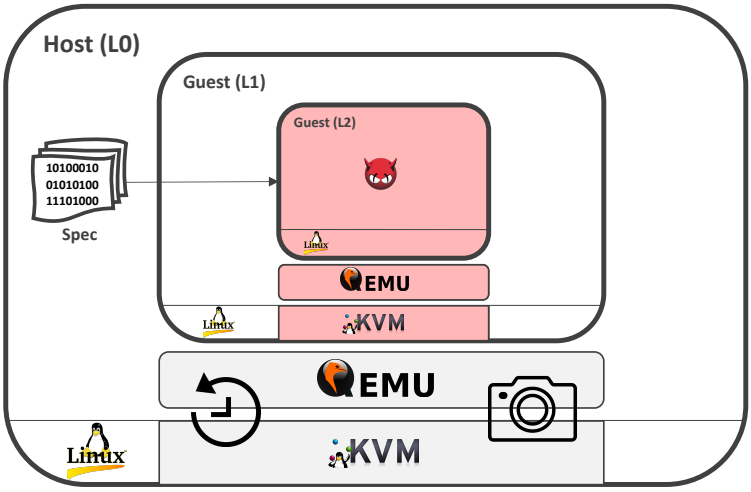
Hypervisors serve as the fundamental backbone for cloud-based solutions these days. Due to the values from daily usage, recent research [23, 100, 101, 102] has focused more on the security of cloud services by major vendors [18, 19, 20]. Yet, the attack surfaces of hypervisors are still wide due to their complexity. The consequences of a compromised

cloud service, of course, can be disastrous, leading to denial of service, information leakage, or even a VM escape [100], jeopardizing the integrity of the other instances running on the same host. Hence, most bounty programs usually rate the cloud-related vulnerabilities among the highest, rewarding up to half a million dollars for a single security-critical bug [102].

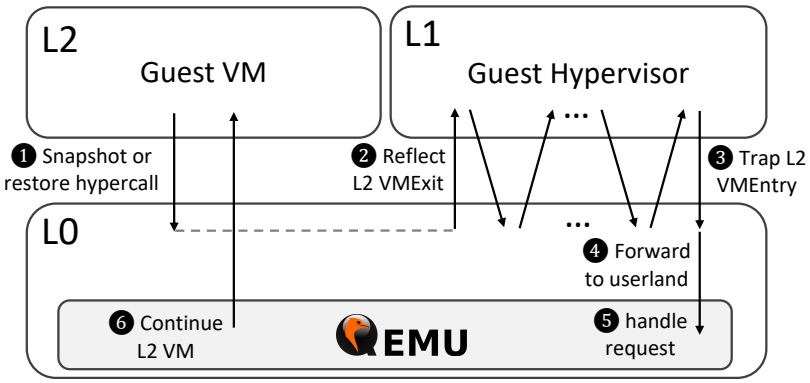
Based on the virtualization target, hypervisors have exposed multiple attack surfaces, including virtualized CPU, memory, device, VM exit handlers, and vendor-specific methods, such as backdoor RPC/HGCM. Among them, some previous studies [103, 104] tried to explore vulnerabilities from the first two surfaces through verification. However, since these efforts necessitate the latest hardware features, it is harder to find a relevant bug that can be considered as security-critical. Meanwhile, emulation of virtual devices occurs in the host userspace, and is triggered by IO accesses (*e.g.*, port and memory-mapped IO) from the guests. As most hypervisor vendors do not implement their own device drivers on the hosts, device emulation through relaying the requests from the guests to the host kernel space can be complex. Given the large codebase of available virtual devices and the lack of maintenance for some that are rarely used in the cloud environment (*e.g.*, sound card), almost all recent vulnerabilities reside in this space per survey.

4.2.4 Fuzzing Non-user Applications

Fuzzing has been proven to be effective as an automatic bug-finding techniques, widely adopted by industry for commercial products [56, 50]. However, directly applying fuzzers in the context of non-user programs can be non-trivial, as reboots are needed for a clean state of VM per mutated fuzzing input. While the expensiveness is expected, various approaches have been taken to mitigate this challenge. Some merge multiple inputs for each fuzzing trial without rebooting the guest VMs [21, 1], while others try to use customized kernels to accelerate the loading processes [22, 23]. Nonetheless, they are not ideal since an effective fuzzer requires both determinism and efficiency without sacrificing one or the other. On the



(a) The nested framework with the snapshot mechanism.



(b) A typical workflow of handling the snapshot and restore requests.

Figure 4.3: The overview of the proposed system performant hypervisor fuzzing.

other hand, recent works have tried to tackle the challenge from a fundamentally different perspective. For example, Intel has introduced a VM forking mechanism [105] for testing Xen [106]. Agamotto [107] has proposed to dynamically checkpoint a VM state for fuzzing kernel drivers. Neither applies directly to the current topic, however, as VM forking only works for hypervisors sitting on the bare metal (*i.e.*, type-I), and fuzzing drivers does not involve a nested framework of multiple type-II hypervisors.

4.3 Design

In this section, we discuss the nested framework with its workflow. Then we dive into the details of the snapshot framework and the customized OS designated for fuzzing.

4.3.1 Overview

To test any hypervisor target, which involves kernel and userspace components for virtualization, the guest OS is often assumed to be malicious and can perform any privileged operations, such as PIO and MMIO. A crafted attack can exploit through the trapped interactions from the guest (*i.e.*, VM exit) to make intended state transitions in the hypervisor, before a security-critical state (*e.g.*, UAF) can be reached. Given the threat model, along with the intended snapshot and restore functionalities, Figure 4.3a shows the logical view of the nested virtualization framework overall. In particular, the L2 guest serves as the executor of fuzzing inputs, testing the hypervisor at L1, while the L0 host plays the role of snapshot maintainer for quick reset (without reboot). The fuzzing inputs contain the self-defined specification similar to that of existing fuzzers targeting non-user applications (*e.g.*, Syzkaller), and are shared from L0 to L2 through a pre-allocated memory. Note that the L1 testing target can be of any hypervisors, as long as they can run on the Linux platform in a nested setting. For L0, KVM/QEMU is chosen to meet the requirement of compatibility, as a type-II hypervisor can be adopted more easily. By restoring anything that has been changed since the last execution, such as registers, memory and devices, the host maintainer ensures the determinism of its guest VM, including both L1 and L2 states.

Figure 4.3b depicts the typical workflow of the snapshot mechanism throughout fuzzing. A snapshot request is created by the L2 guest VM once its kernel boots up before executing any fuzzing inputs (❶). The hypercall request is then trapped to KVM at L0, which reflects to the L1 guest hypervisor (❷). However, since it is a self-defined hypercall and the testing hypervisor is left unmodified, L1 cannot recognize it and finally gives up after multiple communications with L0. Such a sign as VM entry is trapped again in L0 (❸) before further processing in the userspace (❹). Although this process might seem redundant at first glance, as L0 could have handled it without reflection, it is indeed necessary. While L1 is the hypervisor of the L2 guest VM in the nested setting, most privileged requests, such as hypercalls and IOs, will be handled appropriately at L1 instead of L0. Violating the process

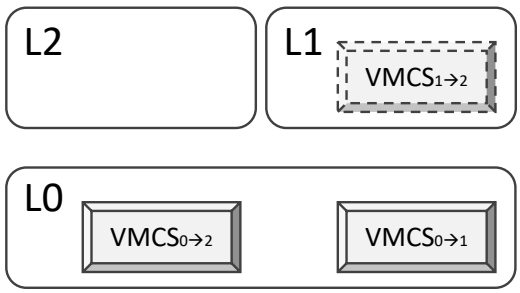


Figure 4.4: VMCSes maintained in the nested virtualization setting.

can cause deadlocks and freeze the VM based on the implementation. Meanwhile, QEMU at L0 makes the snapshot of the entire VM per request (❺), and continues the guest for further execution (❻). After L2 finished executing a fuzzing input, a restore request is issued and handled similarly as that of the snapshot request, and so the details are skipped here.

4.3.2 Snapshot and Restore

In this section, we describe the necessity of the snapshot mechanism. The resets relate to the virtualization features enabled by any hypervisors to run a guest VM, including virtualized CPU, memory, and devices.

CPU reset. To support the virtualized CPU with hypervisor mode (*i.e.*, root), Intel VMX [6] defined the usage of the Virtual Machine Control Structure (VMCS), which contains information about VM entries and exits for handling. More like the program states on context switches, the information includes the register values, page table pointer, exit reason, and so on. However, due to the lack of hardware support for nested virtualization as mentioned in §4.2, the current design multiplexes the extension by having L0 acting as the intermediate layer between L1 and L2. While only one VMCS can be loaded per CPU at once, L0 maintains both $\text{VMCS}_{0\text{->}1}$ and $\text{VMCS}_{0\text{->}2}$ for the purpose, as shown by Figure 4.4. Specifically, the former is used when L0 runs L1, and the latter allows L0 to run L2. On the other hand, since L1 is unaware of the virtualized environment that it lives in, an extra $\text{VMCS}_{1\text{->}2}$ is maintained for its L2 guest. Although it is never loaded into the processor, L0 shadows the information to construct $\text{VMCS}_{0\text{->}2}$ and syncs frequently after

each mode transition [94]. Thus, our approach bookkeeps and resets all the VMCS per request. $VMCS_{0->1}$ and $VMCS_{0->2}$ are saved as part of the live migration feature [93] enabled by QEMU, whereas $VMCS_{1->2}$ is kept by the memory resets, which are described next.

Memory reset. A guest VM is created as a child process in QEMU. Figure 4.5 shows the setup of virtualized memory, where the guest physical memory (GPA) is mapped to the QEMU virtual memory on the host (HVA). Despite the existence of page tables and EPT [6], each page follows the one-to-one mapping as indicated. In the nested setting, the L0 QEMU contains the entire memory used by the L1 guest, which in turn creates its own L2 guest with memory space that still falls into the range. Therefore, resetting the memory in the L0 QEMU should perform a full reset on both L1 and L2. A libc function, such as *memcpy()*, will do the job as expected. However, copying all the memory pages can be time consuming, especially if the VM is configured with gigabytes of memory. In fact, memory reset has a key impact on the performance of the snapshot mechanism as proposed, and we show the related findings in a later section. To reduce the heavyweight cost, dirty page tracking following each restore request is enabled so that L0 can selectively resets memory in use. Particularly, the L0 KVM sets the permission of all memory pages allocated for the guest as *read-only* (RO). Any writes to a fresh page are trapped, causing a bit set in the dirty bitmap shared with the L0 QEMU. Note that the permission of the corresponding page is then changed back to *read&write* (RW), so any subsequent accesses will not trigger more traps with additional overhead. Finally, QEMU refers to the dirty bitmap and only resets pages as needed. In general, this is also partially implemented by the live migration feature [93].

Device reset. Non-memory device states, such as those for sound, display, network, storage, and USB, are important targets to reset, as most of them can be enabled as virtual devices, which are considered the most vulnerable attack surfaces mentioned in §4.2. In the nested virtualization framework, where most device emulation occurs in the L1 guest through QEMU, there are still IO requests that are not reflected and handled in L0 instead. For example, emulated pci buses at L0 are often affected by IOs from L2 consequentially,

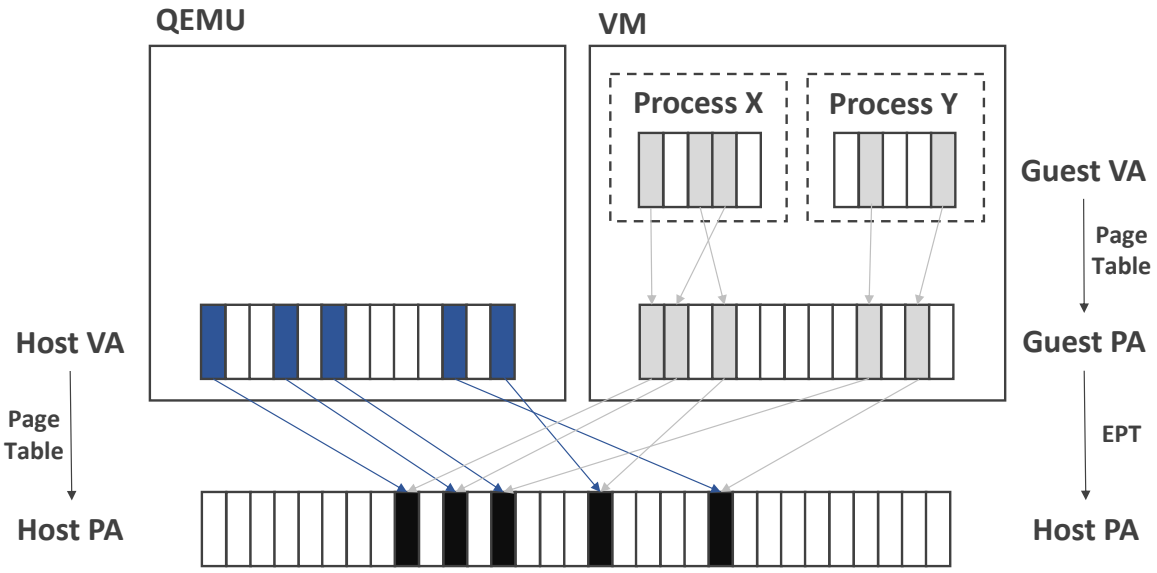


Figure 4.5: Management of virtualized memory in KVM/QEMU.

spanning across multiple interactions between L0 and L1, as suggested in Figure 4.2b. Fortunately, most enabled virtual devices in QEMU have already defined their important data structures that need to be reset, thanks to the live migration feature [93]. The metadata are saved in a JSON array, with each item indicating a device along with its necessary data in the fields. By bookkeeping and restoring them, our approach can ensure the last piece for determinism without sacrificing the efficiency through reboots. Note, however, that the current implementation cannot perform device reset on those that have not defined their critical structures. Any customized or legacy devices that fail the assumption are considered beyond the scope of our current resetting scheme.

4.3.3 Minimized Guest OS

While the nested framework involves two guest VMs, their complexity matters due to the multiplexing design. As discussed in §4.2.2, such a design can cause multiplication on multiple aspects, including VM exits and dirty memory pages, which interleave with one another in terms of their impact on the overhead in general. Although many works [97, 98, 99] have examined ways to reduce the cost, the current work tries to minimize the complexity of OSes based on their roles in the fuzzing context. Since the L1 hypervisor

is the testing target, the L1 kernel only needs to maintain the minimal support of KVM as long as the L2 guest can be run. Specifically, we adopt the minimal configuration of the Linux Kernel with KVM for compatibility, as most hypervisors (*e.g.*, QEMU, VirtualBox) would still need the platform to work properly. On the other hand, since L2 serves as the executor of privileged instructions (*i.e.*, IO) based on the specification shared by L0, it can be fully minimized. Compared to a typical kernel, for example, the L2 kernel can give up the majority of the codebase, such as that for memory management, task scheduling, interrupt handling, and so on. The only required feature left is to scan the available port and memory-mapped regions once launched, which are the target addresses of any effective IO operations. We build upon CrashOS [22] to meet our goal. The following sections highlight the potential impact of a complex OS, such as the Linux Kernel, on the nested setting.

VM exits. One of the potential fallbacks against efficiency is the execution speed affected by the number of VM exits. The corresponding multiplication effect is well depicted in the original proposal of the nested virtualization design [94]. For instance, to handle a single L2 exit, L1 could trigger many more exits to L0, such as reading and writing the VMCS maintained, disable interrupts, and so on. With each VM exit causing considerable overhead, the goal is to minimize the exits from L2 other than the intended IO requests for the purpose of fuzzing. In contrast, if we adopt the Linux Kernel at L2, more interrupts, IOs, MSR read/write will likely occur concurrently (*e.g.*, kthreads) while executing inputs, causing additional exits and thus slowing the fuzzing throughput.

Memory overhead. Another potential downgrade of performance comes from the memory overhead handled by the snapshot mechanism, also suffering from the multiplication effect in the nested setting. Even though only the dirty pages are tracked and reset as mentioned, memory overhead can be significant when a typical kernel is used as L2. For example, the Linux Kernel contains an RCU lock-detection scheme [108] that applies to CPU deadlock if necessary. It can persist different amount of memory footprint based on its implementation on various system specifications, meeting the requirement from real-time servers with a large

Table 4.1: Components of the nested virtualization framework with their implementation complexity in lines of code.

Component	Lines of code (LoC)
Snapshot & restore mechanism (QEMU)	1,317 lines in C
VM exit trapping and handling (KVM)	97 lines in C
Fuzzing executor (L2 kernel)	1,146 lines in C
Fuzzing coordinator (fuzzer)	1,386 lines in Python

number of CPUs to IoT devices with a non-preemptible uniprocessor. Note that the memory overhead could lead to more VM exits from memory management, causing page faults and EPT exceptions trapped to L1. Thus, the impact from both is interleaving, jeopardizing more on the performance gain from the framework.

4.4 Implementation

Our prototype consists of four major components, which are summarized in Table 4.1. Particularly, the majority of the snapshot mechanism is implemented in QEMU, which allows fast resets on states of virtual CPU, memory, and enabled devices. It also tracks dirty memory pages set in KVM, which handles the VM exits in the nested settings by trapping the requests of snapshot and restore from the L2 OS. By modifying QEMU v5.0.0 and Linux 5.0 accordingly and having these two components working together, the desired functionality has been mostly fulfilled.

On the other hand, to fuzz hypervisors more efficiently, we need the minimized OS at L2, which involves scanning IO addresses and parsing self-defined fuzzing specifications before executing the inputs shared. The coordinator, being the only component implemented in Python, serves to randomly generate fuzzing inputs and monitor launched instances for statistics collection and abnormal statuses, such as timeout and ASAN-enabled crashes. Although the current design is Linux-specific by borrowing the existing infrastructure of KVM and QEMU, we believe that the approach is applicable to other type-II hypervisors with similar efforts. Overall, the prototype is written in around 4,000 lines of code.

4.5 Evaluation

We perform empirical evaluations on our system by answering the following questions:

- **Efficiency.** How much performance gain does the system provide in terms of fuzzing throughput? (§4.5.2)
- **Effectiveness.** How much does the improvement benefit from the design choices? (§4.5.3)
- **Practicality.** How easily can we apply the framework to test multiple hypervisors? Can we find real-world vulnerabilities with it? (§4.5.4)

4.5.1 Experimental Setup

To evaluate our system, the experiment has been conducted on a machine of 64 cores and 126GB memory, with the processor of Intel(R) Xeon(R) CPU E7- 4820 @ 2.00GHz. The evaluations on throughput improvement and overhead breakdowns are collected from a single instance running on one physical core, as we believe it would better represent the meaning behind the numbers. For the average throughput comparison, we let the fuzzers of various schemes run for 24 hours as one trial and persist for three consecutive trials to reduce potential statistical noise. QEMU v5.0.0 and Linux 5.0 are involved as parts of the comparing approaches in this study. To hunt zero-day bugs, we test our system on the real-world hypervisors, *i.e.*, the latest QEMU and VirtualBox from the HEAD of their master branches, with all the cores utilized. In general, we fuzzed the targets for a total of 20 days, with most virtual devices enabled.

4.5.2 Improving Fuzzing Throughput

To illustrate the benefits of our system with the snapshot mechanism and the customized OS, we compare its average throughput with that achievable by the other fuzzing schemes. Specifically, we measure the numbers from the approaches of rebooting VM with the Linux Kernel (*miniconfig*), HYPERCUBE [23], and nested snapshot on the Linux Kernel

(*defconfig* and *miniconfig*) in the multiple fuzzing trials. Note that since HYPERCUBE has not released its source code yet, the specific number is borrowed from the published paper without reproducing it on our end.

Figure 4.6 shows the throughput comparison when fuzzing QEMU built with AddressSanitizer (ASAN) [109]. Our system well outperforms the other approaches, enabling a 72x faster execution speed over the baseline, which consistently reboots VM with the Linux Kernel. When comparing to HYPERCUBE, which is the state-of-the-art hypervisor fuzzer from the academics, we still present a 9x throughput improvement due to our snapshot mechanism. Meanwhile, the numbers from the nested setting with snapshot across various kernels demonstrate the benefits from the customized OS at L2. More importantly, a consistent increase of performance gain can be observed when the kernel complexities have been reduced, resulting in less exit and memory overhead as expected (§4.3.3). More detail about the overhead breakdown is discussed in a later section.

Although the AddressSanitizer does have a negative influence on the absolute throughput in general, posing almost half of the execution speed as that from the build without ASAN enabled, it does not have a significant impact on the relative throughput across the schemes based on our observation. Compared to the rebooting approaches, the snapshot mechanism will suffer an extra slowdown from dirty page-tracking and recovering, in addition to the slower execution of the ASAN build itself. However, with the random fuzzing heuristic adopted by the current framework, we do not see much of an increase for dirty pages that may be significant to throttle throughput in most executions. Overall, the findings of throughput improvement can be generalized to testing other hypervisors, such as VirtualBox.

4.5.3 Anatomy of Overheads

In this section, we provide the details of performance cost in the nested setting, and examine the design choices of our framework.

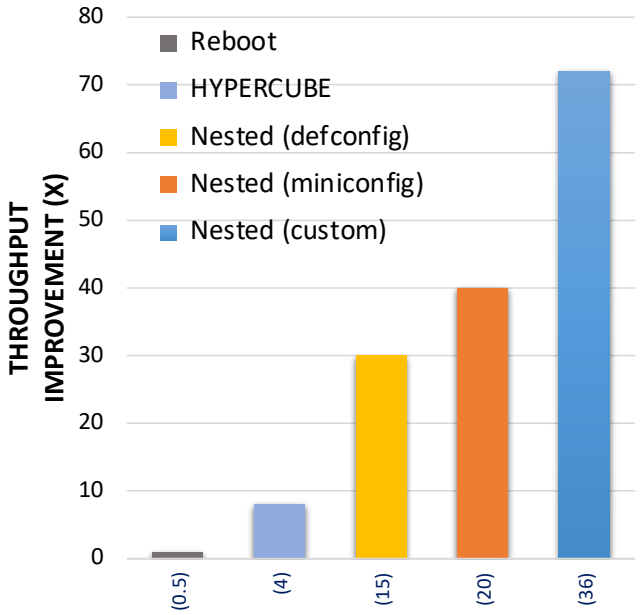
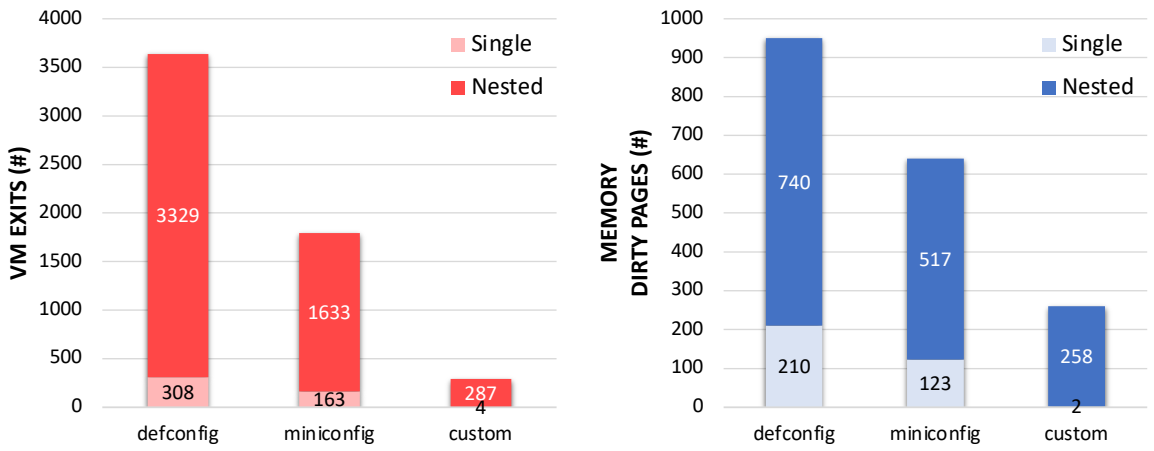


Figure 4.6: The average throughput improvement across multiple fuzzing schemes when testing ASAN-enabled QEMU. The numbers below the bars indicate the actual throughput (*i.e.*, executions per second) on our evaluating setup.

Nested virtualization: the multiplexing effect. The cause of the multiplexing effect from the nested virtualization framework due to the lack of hardware support has been discussed in §4.2.2 and §4.3.3. Figure 4.7 presents the additional VM exits and dirty-memory pages incurred on average across multiple kernels in both the single and nested virtualization setting. While the VM exits slow the execution, the dirty pages stress the snapshot mechanism. Note that to reduce noise, the kernels do not perform any fuzzing-related IO operations. Instead, only the requests of snapshot and restore are issued before the VM states are reset. In this way, we can reason more precisely about the extra overhead within each fuzzing execution. In general, the result shows that both VM exits and dirty-memory pages have been multiplicated from the single to the nested setting as expected. However, with the custom OS that removes all the unnecessary kernel features, our system only triggers 8.00% of VM exits and 27.37% of dirty memory from the Linux Kernel of *defconfig* per reset. Compared to the Linux Kernel of *miniconfig* that significantly improves over *defconfig*, our OS still manages to reduce VM exits and dirt memory by 83.80% and 59.38%, respectively.



(a) Exit multiplication.

(b) Memory multiplication.

Figure 4.7: The breakdown of exit and memory overhead from single and nested virtualization setting across kernels of various complexity.

In terms of VM exits, only EPT violations have occurred in the customized OS when applied in the single virtualization setting. The violations are common and inevitable when a new guest memory region is created and accessed, as they indicate a missing mapping from a guest physical address (GPA) to a host physical address (HPA). In contrast, the Linux Kernel is more complex by orders of magnitude with common features including scheduling, interrupt handling, multi-threading support, and IO subsystems regardless of the building configuration, causing more VM exits in the nested setting. On the other hand, unlike the customized OS, the Linux Kernel also maintains more kernel data structures, such as those for RCU detection explained in §4.3.3. While the exit and memory overhead is interleaving by affecting one another, more memory accesses in L2 could lead to more VM exits, such as EPT violations, waking up the L1 hypervisor more often for handlers and thus creating more dirty pages overall from the perspective of L0. The impact from both types of overhead on performance is addressed in the next section.

Performance costs. We try to identify the performance bottleneck of the snapshot mechanism through FlameGraph [110], a stacked visualizer of overhead sorted by the implemented functions. Based on our profile and the resets enabled, the time needed to recover dirty-memory pages between two consecutive executions drives the majority of the slowdown. The resets on virtual CPUs and devices, on the other hand, are considered minimal. While

Figure 4.8: Performance cost from the memory overhead in the single virtualization setting.

the memory handling could be further divided into two stages, such as dirty-page tracking and recovering, the former has a bigger impact on the performance by changing the page permissions. Nevertheless, the tracker cannot be disabled, as simply recovering all memories per execution could cause a even more significant overhead based on our observation. That being said, memory recovery through *memcpy()* becomes the key contributor that we look forward to optimizing, as the memory overhead could burden the snapshot mechanism in regard to its efficiency.

Figure 4.8 suggests the impact by showing the correlation between the number of dirty pages and the absolute throughput consequentially in the single virtualization setting. Particularly, we see a continuous downgrade of performance when the number of dirty pages increases. The decreasing pattern is almost linear since *memcpy()* dominates. Given this finding, we can conclude that our customized OS with a much smaller memory overhead plays an important role in improving the fuzzing throughput. Meanwhile, the time needed for reboots persists regardless of the memory overhead.

However, in the nested virtualization setting, the increase of VM exits could be more detrimental to the performance overall. Figure 4.9 shows the throughput from the nested settings with the Linux Kernel and the customized OS, applying Figure 4.8 as the background. With the vertical space between each throughput data point and the blue line indicating the memory overhead on snapshot throughput, we can see a significant performance downgrade

Figure 4.9: Performance cost from the exit overhead in the nested virtualization setting.

of 53.88% and 42.59% for the adoption of the Linux Kernel in *defconfig* and *miniconfig*, respectively. Note that the slowdown is irrelevant to the impact from dirty-memory pages, as the numbers on the x-axis stay the same. Rather, the multiplicated VM exits mentioned previously should explain the situation. Although our customized OS also increases the VM exits due to the multiplexing design, it does not overwhelm, and only suffers a bare throughput reduction of 4.07%. In general, our design of adopting the customized OS in place of the Linux Kernel at L2 as the fuzzing executor is considered effective, dramatically reducing the multiplicated overhead from VM exits and dirty memory to achieve the desired fuzzing speed as requested.

4.5.4 Testing Real-world Hypervisors

To examine the practicality of our fuzzing framework, we apply it to multiple real-world hypervisors, including QEMU and VirtualBox, for hunting zero-day vulnerabilities. In preparation for the fuzzing targets, we build the dynamic binaries of hypervisors and their library dependencies, before storing them into a ramfs image [111]. Since QEMU accepts the command line options of kernel and ramfs directly, applying the target is considered

Table 4.2: Assigned CVEs for bugs that have been found by our system.

CVE	Device	Vulnerability
CVE-2020-13361	ES1370 audio	Heap OOB access
CVE-2020-13362	MegaRAID SAS storage manager	Heap OOB access
CVE-2020-13659	MegaRAID SAS storage manager	Null pointer dereference
CVE-2020-13754	Message Signalled Interrupt (MSI-X)	Heap OOB access
CVE-2020-13791	ATI VGA display	Heap OOB access
CVE-2020-13800	ATI VGA display	Infinite recursion

hassle-free. Specifically, we also put the customized OS into the ramfs image so that it can be booted as the L2 executor from the L1 QEMU. VirtualBox, however, does not allow the options from command line, but rather boots on a unique VM image format. To overcome this, we create a Ubuntu image for VirtualBox and replace the default grub option to our customized OS. We then store the VM image into the ramfs image similarly. That being said, adopting other hypervisor targets, including proprietary ones that are closed source, should not be a fundamental challenge to the current system as long as they can run on Linux in a nested virtualized environment provided by KVM/QEMU. We leave it as future work to the study.

In total, our system was able to find 14 bugs, including assertions and crashes triggered by ASAN, from both evaluating targets. Table 4.2 summarizes the 6 CVEs assigned so far across QEMU's emulated devices of various types, such as sound, storage, display, and MSI. At the time of this writing, two more bugs among all the reported ones are still under investigation. Interestingly, one of them is a stateful bug, which requires three IO reads and writes to trigger. Given the blackbox fuzzing heuristic without any feedback and the argument from previous works on the bottleneck of fuzzing non-user applications [21, 23], we credit our bug-finding result to the improved throughput enabled by our framework.

4.6 Discussion

In this section, we address the limitations of our current system, and discuss the potential future directions.

4.6.1 Limitations

While performance is considered the major roadblock for fuzzing non-user applications [23], fuzzing precision based on various feedback can still be beneficial to bug finding in general. A typical fuzzer could rely on coverage guidance, which has proved its power by helping discover thousands of critical bugs, to prioritize inputs that cover rare paths [1, 2, 57, 59, 112]. It may also adopt dataflow information to better approximate program executions and bypass data-driven constraints [62, 60, 63, 64, 65]. However, neither approach is considered lightweight, adding more pressure to the fuzzers targeting non-user applications that are already slow enough. Intel PT [3] is another viable option, especially for testing proprietary hypervisors that are closed-source. Nonetheless, adopting the hardware extension for fuzzing is ad-hoc, which typically involves the decoding of full traces before translating the data to fuzzers as feedback [9, 61]. Although SNAP [8] proposes embedding the tracing algorithm into the existing processor pipeline on custom hardware without incurring extra cost, applying the approach is not straightforward, as it requires the architectural support for virtualization.

Another limitation that we consider is from implementation. Specifically, the current functionality of resetting device states heavily relies on the live migration feature from QEMU [93]. Despite it being the source of the second major overhead in the snapshot mechanism, it is non-trivial to decouple the functionality from the existing QEMU feature and further improve the performance.

4.6.2 Future Directions

Future studies might apply the current system to more hypervisor targets, including those that are closed source. It may even be extended to non-user applications beyond hypervisors, such as core kernel. Meanwhile, adopting various sanitizers [113, 114] and feedback for guidance could be interesting to improve the fuzzing precision. While most security-critical bugs are stateful, various corresponding techniques [115, 116] can also be applicable. Since

device emulation is our main target among the attack surfaces, we might also be able to auto-generate fuzzing harnesses from kernel device drivers, which contain valuable code snippets for communicating with virtual devices in a valid format. Last, more advanced techniques like concolic execution can also be applied to replace pure fuzzers for bug finding.

CHAPTER 5
CONCLUSION

In this dissertation, we present a series of software-hardening solutions with careful design to adopt hardware features for the goals of performance, compatibility, and reliability. PITTYPAT utilizes Intel PT for collecting full execution traces and efficiently enforcing path-sensitive control-flow security at runtime. SNAP, on the other hand, proposes hardware primitives realized on the FPGA platform to enhance the performance and precision of coverage-guided fuzzing. Finally, HYPERSET shows a nested virtualization framework with fast VM resets for performant hypervisor fuzzing.

In the work of HYPERSET, we demonstrate the improvement on fuzzing throughput when testing non-user applications, such as hypervisors. We further dissect the overhead, and break down the benefits of our major design choices, including the snapshot mechanism and the customized OS. Given the hardware support of virtualization on x86 platforms, our system is able to achieve a 72x faster execution speed than that of traditional approaches when fuzzing hypervisors. When applying to the real-world targets, such as QEMU and VirtualBox, it also shows its practicality by successfully finding 14 zero-day vulnerabilities.

REFERENCES

[1] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "KAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[2] Google, *Honggfuzz*, https://github.com/google/honggfuzz, 2021.

[3] James R., *Processor Tracing*, https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html, 2013.

[4] L. Delshadtehrani, S. Canakci, B. Zhou, S. Eldridge, A. Joshi, and M. Egele, "Phmon: A programmable hardware monitor and its security use cases," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[5] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, and W. Lee, "HDFI: Hardware-assisted data-flow isolation," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[6] Intel, *Intel Virtualization Technology (Intel VT)*, https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html, 2005.

[7] VMWare, *Performance Evaluation of Intel EPT Hardware Assist*, https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf, 2009.

[8] R. Ding, Y. Kim, F. Sang, W. Xu, G. Saileshwar, and T. Kim, "Hardware support to improve fuzzing performance and precision," in *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, Seoul, South Korea, Nov. 2021.

[9] Y. Chen, D. Mu, J. Xu, Z. Sun, W. Shen, X. Xing, L. Lu, and B. Mao, "Ptrix: Efficient hardware-assisted fuzzing for cots binary," in *Proceedings of the 14th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Auckland, New Zealand, Jul. 2019.

[10] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "Ptfuzz: Guided fuzzing with processor trace feedback," in *IEEE Access*, 2018.

[11] Google, *WinAFL*, https://github.com/googleprojectzero/winafl, 2021.

[12] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)," in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, Jul. 2019.

[13] H. Hu, C. Qian, C. Yagemann, S. P. Chung, B. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[14] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[15] C. Yagemann, M. Pruett, S. Chung, K. Bittick, B. Saltaformaggio, and W. Lee, "ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual conference, Aug. 2021.

[16] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun, "REPT: Reverse Debugging of Failures in Deployed Software," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, Oct. 2018.

[17] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, "Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[18] Amazon, *AWS – Amazon Web Services*, https://aws.amazon.com/, 2021.

[19] Google, *Google Cloud Platform*, https://cloud.google.com/, 2021.

[20] Microsoft, *Microsoft Azure*, https://azure.microsoft.com, 2021.

[21] Google, *syzkaller – kernel fuzzer*, https://github.com/google/syzkaller, 2018.

[22] A. Gantet, "CrashOS: Hypervisor testing tool," in *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2017.

[23] S. Schumilo, C. Aschermann, A. Abbasi, S. Worner, and T. Holz, "HYPER-CUBE: High-dimensional hypervisor fuzzing," in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[24] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS*, 2007.

[25] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS*, 2005.

[26] L. O. Andersen, “Program analysis and specialization for the c programming language,” PhD thesis, U. Cophenhagen, 1994.

[27] B. Steensgaard, “Points-to analysis in almost linear time,” in *POPL*, 1996.

[28] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, “Opaque control-flow integrity,” in *NDSS*, 2015.

[29] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, “Practical control flow integrity and randomization for binary executables,” in *SP*, 2013.

[30] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, “Control-flow bending: On the effectiveness of control-flow integrity,” in *USENIX Security*, 2015.

[31] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, “Control jujutsu: On the weaknesses of fine-grained control flow integrity,” in *CCS*, 2015.

[32] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, “Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications,” in *SP*, 2015.

[33] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R Sekar, and D. Song, “Code-pointer integrity,” in *OSDI*, 2014.

[34] G. C. Necula, S. McPeak, and W. Weimer, “Ccured: Type-safe retrofitting of legacy code,” in *PLDI*, 2002.

[35] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “Cets: Compiler enforced temporal safety for c.,” in *ISMM*, 2010.

[36] ——, “Softbound: Highly compatible and complete spatial memory safety for c,” in *PLDI*, 2009.

[37] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, “Missing the point (er): On the effectiveness of code pointer integrity,” in *SP*, 2015.

[38] B. Niu and G. Tan, “Per-input control-flow integrity,” in *CCS*, 2015.

[39] J. Reinders, *Processor tracing - Blogs@Intel*, https://blogs.intel.com/blog/processor-tracing/, Accessed: 2016 May 12, 2013.

[40] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," in *ASPLOS*, 2017.

[41] B. Hardekopf and C. Lin, "The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code," in *PLDI*, 2007.

[42] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, 1996.

[43] M. Zhang and R Sekar, "Control flow integrity for COTS binaries.," in *Usenix Sec.*, 2013.

[44] B. Niu and G. Tan, "Modular control-flow integrity," in *PLDI*, 2014.

[45] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *DAC*, 2014.

[46] Intel, *PIN - A Dynamic Binary Instrumentation Tool*, https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html, 2018.

[47] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, Anaheim, CA, Apr. 2005.

[48] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime intrusion prevention evaluator.," in *ACSAC*, 2011.

[49] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz, "Control-flow integrity: Precision, security, and performance," *arXiv preprint arXiv:1602.04056*, 2016.

[50] Google, *ClusterFuzz*, https://github.com/google/clusterfuzz, 2021.

[51] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[52] D. Bruening and S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation," PhD thesis, Massachusetts Institute of Technology, 2004.

[53] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, May 2018.

[54] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[55] Intel, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part 2*, 2011.

[56] Microsoft, *OneFuzz*, https://github.com/microsoft/onefuzz, 2021.

[57] M. Zalewski, *AFL*, https://lcamtuf.coredump.cx/afl/, 2014.

[58] Intel, *Intel processor trace decoder library*, https://github.com/intel/libipt, 2013.

[59] LLVM, *libFuzzer - a library for coverage-guided fuzz testing*, https://llvm.org/docs/LibFuzzer.html, 2021.

[60] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[61] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path Sensitive Fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[62] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2010.

[63] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[64] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GREYONE: Data Flow Sensitive Fuzzing," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[65] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[66] M. Zalewski, *Fast LLVM-based instrumentation for afl-fuzz*, https://github.com/google/AFL/blob/master/llvm_mode/README.llvm, 2019.

[67] U. Berkeley, *The Berkeley Out-of-Order RISC-V Processor*, https://github.com/riscv-boom/riscv-boom, 2017.

[68] SiFive Inc., *The RISC-V Instruction Set Manual*, https://riscv.org//wp-content/uploads/2017/05/riscv-spec-v2.2.pdf, 2017.

[69] U. Berkeley, *The Branch Predictor (BPD) in RISC-V BOOM*, https://docs.boom-core.org/en/latest/sections/branch-prediction/backing-predictor.html, 2019.

[70] R. Thomas, M. Franklin, C. Wilkerson, and J. Stark, "Improving Branch Prediction by Dynamic Dataflow-based Identification of Correlated Branches from a Large Global History," in *Proceedings of the 30th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, San Diego, CA, USA, Jun. 2003.

[71] S. Kagstrom, *Kcov is a FreeBSD/Linux/OSX code coverage tester*, https://github.com/SimonKagstrom/kcov, 2015.

[72] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanović, "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18, Los Angeles, California, 2018.

[73] D. Biancolin, S. Karandikar, D. Kim, J. Koenig, A. Waterman, J. Bachrach, and K. Asanovic, "Fased: Fpga-accelerated simulation and evaluation of dram," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, Seaside, CA, USA, 2019.

[74] Intel, *10th Generation Intel Core Processor Families*, https://www.intel.com/content/www/us/en/products/docs/processors/core/10th-gen-core-families-datasheet-vol-1.html, 2020.

[75] IBM, *IBM z13 Technical Guide*, https://www.redbooks.ibm.com/redbooks/pdfs/sg248251.pdf, 2016.

[76] Intel, *11th Generation Intel Core Processor (UP3 and UP4)*, https://cdrdv2.intel.com/v1/dl/getContent/631121, 2020.

[77] Synopsys, *DC Ultra*, https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html, 2020.

[78] GNU Project, *GNU Binutils*, https://www.gnu.org/software/binutils, 2020.

[79] Google, *OSS-Fuzz - Continuous Fuzzing for Open Source Software*, https://github.com/google/oss-fuzz, 2020.

[80] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[81] J. Zhao, K. Ben, G. Abraham, and A. Krste, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, 2020.

[82] U. Berkeley, *Rocket Chip Generator*, https://github.com/chipsalliance/rocket-chip, 2016.

[83] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "Openram: An open-source memory compiler," in *Proceedings of the 35th International Conference on Computer-Aided Design (ICCAD)*, Austin, Texas, 2016.

[84] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, "Freepdk: An open-source variation-aware design kit," in *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*, 2007.

[85] TechPowerUp, *RISC-V Processor Achieves 5 GHz Frequency at Just 1 Watt of Power*, https://www.techpowerup.com/275463/risc-v-processor-achieves-5-ghz-frequency-at-just-1-watt-of-power, 2020.

[86] M. Kan, *Intel to Build Chips for Other Companies With New Foundry Business*, https://in.pcmag.com/processors/141636/intel-to-build-chips-for-other-companies-with-new-foundry-business, 2021.

[87] lafintel, *LAF LLVM Passes*, https://gitlab.com/laf-intel/laf-llvm-pass, 2016.

[88] ARM, *CoreSight Components Technical Reference Manual*, https://developer.arm.com/documentation/ddi0314/h/, 2009.

[89] F. Richter, *Amazon Leads $130-Billion Cloud Market*, https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers, 2021.

[90] *Zerodium*, https://zerodium.com/, 2021.

[91] MSRC, *Hyper-V Debugging Symbols Are Publicly Available*, https://msrc-blog.microsoft.com/2018/05/03/hyper-v-debugging-symbols-are-publicly-available/, 2018.

[92] AMD, *Secure Virtual Machine Architecture Reference Manual*, https://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf, 2005.

[93] RedHat, *Live Migrating QEMU-KVM Virtual Machines*, https://developers.redhat.com/blog/2015/03/24/live-migrating-qemu-kvm-virtual-machines/, 2015.

[94] M. Ben-Yehuda, M. Day, Z. Dubitzky, M. Factor, N. HarEl, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The Turtle Project: Design and Implementation of Nested Virtualization," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[95] A. Graf and J. Roedel, *Nesting the virtualized world.* Linux Plumbers Conference, 2009.

[96] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, *KVM: the Linux Virtual Machine Monitor*, in Proceedings of the Linux symposium, vol. 1, no. 8, pp. 225-230. 2007.

[97] VMWare, *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*, https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf, 2007.

[98] L. Alon, *Improving KVM x86 Nested Virtualization*, https://events19.linuxfoundation.org/wp-content/uploads/2017/12/Improving-KVM-x86-Nested-Virtualization-Liran-Alon-Oracle.pdf, 2019.

[99] J. T. Lim and J. Nieh, "Optimizing Nested Virtualization Performance Using Direct Virtual Hardware," in *Proceedings of the 25rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, Mar. 2020.

[100] H. Zhao, Y. Zhang, K. Yang, and T. Kim, "Breaking Turtles All the Way Down: An Exploitation Chain to Break out of VMware ESXi," in *Proceedings of the 13rd USENIX Workshop on Offensive Technologies (WOOT)*, Santa Clara, CA, USA, Aug. 2019.

[101] R. Wojtczuk, "Analysis of the Attack Surface of Windows 10 Virtualization Based Security," in *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2016.

[102] N. Joly and J. Bialek, "A Dive in to Hyper V Architecture and Vulnerabilities," in *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2018.

[103] N. Amit, D. Tsafrir, A. Schuster, A. Ayoub, and E. Shlomo, "Virtual CPU Validation," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[104] P. Fonseca, X. Wang, and A. Krishnamurthy, "MultiNyx: a multi-level abstraction framework for systematic analysis of hypervisors.," in *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, Porto, Portugal, Apr. 2018.

[105] T. Kengyel, *VM Forking and Hypervisor-based Fuzzing*, https://github.com/intel/kernel-fuzzer-for-xen-project, 2020.

[106] T. X. Project, *The Hypervisor (x86 & ARM)*, https://xenproject.org/developers/teams/xen-hypervisor/, 2021.

[107] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[108] P. McKenney, *The RCU API, 2019 edition*, https://lwn.net/Articles/777036/, 2019.

[109] LLVM, *AddressSanitizer*, https://clang.llvm.org/docs/AddressSanitizer.html, 2021.

[110] *Flamegraph*, https://github.com/flamegraph-rs/flamegraph, 2019.

[111] *ramfs*, https://wiki.debian.org/ramfs, 2013.

[112] M. Bohme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[113] LLVM, *MemorySanitizer*, https://clang.llvm.org/docs/MemorySanitizer.html, 2021.

[114] ——, *UndefinedBehaviorSanitizer*, https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2021.

[115] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "IJON: Exploring Deep State Spaces via Fuzzing," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, May 2018.

[116] M. Bohme, V. Manes, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," in *Proceedings of the 28th Joint Meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Sacramento, CA, Nov. 2020.