FUZZING WITH PERFORMANCE MONITORING AND TRACING HARDWARE

A Thesis
Presented to
The Academic Faculty

By

Gabriela Lopez

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

May 2022

FUZZING WITH PERFORMANCE MONITORING AND TRACING HARDWARE

ACKNOWLEDGMENTS

I am grateful to have had the opportunity in my graduate program to delve deeper into different fields in information security through my master's thesis research, accompanying courses, and internships. I want to thank my advisor Professor Brendan Saltaformaggio for his ongoing support and guidance. Through Professor Saltaformaggio's profound lectures and guidance, an immense appreciation for reverse engineering and its related fields in information security (like vulnerability discovery) has flourished within me. I will use this newfound appreciation like a compass to guide me in the next steps of my career. I will never forget the research environment you have created, instilling values of collaboration, integrity, and dedication to quality. I will strive to carry with me these values where ever go.

I want to also thank my family, friends, and colleagues for their love and support. Thank you for always believing in me and encouraging me to never give up. While there were many times when I could not see the light at the end of the tunnel, thanks to my support system, I always managed to find the strength to continue forward. I would also like to thank the GEM Fellowship and the Georgia Institute of Technology for their generous support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The field of fuzzing has brought about many new open-source tools, techniques, and insights to improve the state of the art of automated vulnerability discovery systems. However, there are instances where the adoption of such new techniques and tools improves the state of the art of these systems while at the expense of portability, accessibility, and performance. Additionally, while many of the processor platforms used in the fuzzing community already come built with components that observe program execution in the form of performance monitoring and tracing hardware, such hardware is not commonly used by fuzzers. On a similar note, there is currently a lack of evaluations for the usage of such hardware in the fuzzing literature. The most commonly used processor platforms in the fuzzing community are Intel processors. Our work seeks to evaluate the performance impact in using performance monitoring and tracing hardware (specifically Intel Last Record Branch sampling and Intel Branch Trace Store) for coverage feedback in coverage-guided fuzzers. In our evaluation, we seek to learn if the adoption of these specific performance monitoring and tracing hardware in coverage-guided fuzzers can improve the performance of binary-only fuzzing.

CHAPTER 1
INTRODUCTION

As real-world software grows in size and complexity, the demand for an automated approach to vetting their security rises. Manual analysis can not scale with the size of modern software. Additionally, today's real-world software has significantly more access to sensitive data than ever before. Thus, software vendors and developers must utilize highly fast automated approaches to find vulnerabilities in their software before malicious actors.

A popular technique used by academic and industry security researchers in their design of automated vulnerability discovery systems is coverage-guided fuzzing [1, 2, 3, 4, 5, 6]. Like most fuzzing approaches, it tests applications by continuously generating and evaluating several inputs per second while monitoring the output for crashes. Coverage-guided fuzzing uses coverage information (e.g. branch coverage) as feedback to guide its input generation process. This is an attractive characteristic of coverage-guided fuzzing because quality inputs can be generated that discover and test new regions of code for bugs. Coverage-guided fuzzers, however, need to use coverage tracing technologies in order to obtain the coverage information it needs to guide its input generation.

There are four coverage tracing technologies available to coverage-guided fuzzers to obtain coverage information. The first three coverage tracing technologies need to insert instrumentation code at each edge (or basic block) to collect coverage information. The first technology is compiler-inserted instrumentation, this type of instrumentation can accurately insert instrumentation code and easily apply optimizations but can only be used when there is access to the source code of the target application. Two alternative technologies commonly used by the coverage-guided fuzzers to obtain coverage information of closed source binaries are static binary rewriting and dynamic binary instrumentation. Statically rewriting a binary to insert instrumentation is still an ongoing area of research because of the difficulty of accurately and appropriately inserting code statically and the limited ability to apply optimizations to inserted code. These challenges and limitations can impact the quality and accuracy of coverage information generated and the performance of the rewritten binary. Dynamic binary instrumentation can easily and accurately insert code but dynamically translating the binary can have a prohibitively high overhead

An alternative to using static binary rewriting and dynamic binary instrumentation that has the potential to obtain accurate coverage information of closed source binaries is hardware-assisted tracing. Hardware-assisted tracing leverages built-in hardware tracing extensions that directly capture execution flow information (e.g executed branches) at run-time. The most widely used machines in the fuzzing community [2, 3, 7, 8, 9, 10, 11, 12, 13, 14] whether it be server or desktop models are Intel processors. In modern Intel processors, the hardware tracing mechanisms available are Last Branch Record (LBR), Branch Trace Store (BTS), and Intel Processor Trace (PT). Each hardware tracing mechanism has its strengths and limitations that can be respectively improved upon or addressed by the way they are implemented and how they are integrated into fuzzing systems.

The most widely used hardware tracing mechanism in the fuzzing literature is Intel PT [6, 14, 15, 16, 17, 18, 19, 20, 21]. While Intel PT can capture execution information with minimal performance overhead [22], current fuzzing implementations that use Intel PT are still considerably slower than compiler-inserted instrumentation [21]. Using Intel PT requires the use of a decoder to process and interpret PT traces which contain encoded and compressed information of execution flow. Intel PT was originally designed to suit offline processing of PT traces and requires using time expensive procedures such as static analysis of the disassembled traced binary. The bottleneck which results from the decoding PT traces for each executed test case goes against one of the main goals of fuzzing which is evaluating as many test cases as possible.

On the other hand, there are currently limited evaluations of the performance impact to fuzzing in using either Last Record Branch or Brace Trace Store for hardware-assisted tracing. Prior fuzzing works express concern of significant performance overhead in using these tracing mechanisms but typically do not cite any studies evaluating the performance of these tracing mechanisms. To the best of our knowledge honggfuzz [6] is one of the few existing fuzzers that has an implementation that uses the Intel BTS hardware tracing mechanism to obtain coverage information. Additionally in the fuzzing literature, there are very few evaluations that include the Intel BTS implementation of honggfuzz in their evaluations. Prior fuzzing works that include honggfuzz in their evaluations mostly use the compiler-assisted implementation of honggfuzz. In regards to LBR, while there are even fewer fuzzing works that utilize Intel LBR, we notice there are numerous works in another performance-critical domain that have used Intel LBR for Control-Flow Integrity enforcement [23].

Motivated by the limited evaluations of the Intel Branch Trace Store and Last Record Branch in the fuzzing literature, we develop our own evaluation of the two hardware tracing mechanisms by integrating them into our fuzzing implementations based on the well evaluated fuzzer American Fuzzy Lop [4]. In this work, we will answer the following research questions:

- **RQ1:** What is the performance overhead in using Intel LBR and BTS as sources of coverage information in coverage-guided fuzzers?
- **RQ2:** Can using Intel BTS and LBR as coverage tracing mechanisms in fuzzing improve binary-only fuzzing?

One challenge in our evaluations is managing bias due to differences in fuzzing implementation and instrumentation insertion techniques. To address this challenge and answer **RQ1** we develop two fuzzing implementations which use the same fuzzing infrastructure and compiler-assisted instrumentation insertion technique as AFL-Clang-Fast but replace the coverage tracking mechanism with LBR and BTS respectively. We compare the performance of our fuzzing implementations with AFL-Clang-Fast [4]. Another challenge is properly configuring and managing the LBR and BTS facilities to produce accurate coverage information and making this coverage information easily accessible to fuzzers as efficiently as possible. For all our fuzzing implementations we leverage the Linux kernel's perf_events subsystem [24] [25], also used by other works [6] [15], to appropriately monitor our applications and easily obtain the LBR and BTS coverage information. To answer **RQ2**, we minimally statically rewrite target binaries to include a hook to our LBR/BTS monitoring code so it can run before the main function to enable the LBR/BTS coverage tracing mechanism without access to source code. Additionally, to answer **RQ2** we fuzz our rewritten target binaries and we compare the performance of our fuzzing implementations with two currently available binary-only fuzzing implementations, AFL-QEMU [4] and PTfuzz[15].

Our contributions can be summarized as follows:

- Develop a fair and meaningful evaluation of the performance impact of Intel LBR/BTS tracing mechanisms in coverage-guided fuzzing.
- Propose and investigate an approach on how to properly incorporate the LBR/BTS tracing mechanisms in coverage-guided fuzzers to improve binary-only fuzzing.

CHAPTER 2
BACKGROUND

2.1 Fuzzing

2.1.1 Introduction to Fuzzing

Fuzzing is a popular technique used in automated vulnerability finding systems [8]. Fuzzing iteratively and randomly generates input to explore the input space of an application to trigger unintended program behaviors (i.e crashes) [13]. The assumption is that if an application's input validation can not graciously handle unexpected inputs, such inputs could expose vulnerabilities. Why is fuzzing a popular choice of technique in vulnerability finding systems? Fuzzing, unlike other techniques such as symbolic execution [1], can generate concrete inputs quickly, therefore, helping in running more test cases per second, which correlates with finding more bugs in a shorter amount of time (assuming there are no hard checks involved in input validation). As evidenced by the popular state-of-the-art fuzzer AFL [4] when compared with symbolic executor angr, AFL found 76% more bugs in the same amount of testing time (24 hours) than angr [13] [1]. Another attractive feature of fuzzing is the ability to generate interesting test cases (inputs) with very little domain knowledge of the tested application [26]. Fuzzing can therefore be used to test a diverse set of applications. Even with little domain knowledge, the genetic mutation-based input generators of fuzzing can come up with inputs that reach and explore corner cases in program execution paths which at times are hard for humans to generate [26]. Fuzzing has had many successes as a technique in vulnerability finding systems and as a result, major software vendors such as Google and Microsoft have been using it to detect bugs in their software [11].

2.1.2 Alternative to Fuzzing

In addition to fuzzing, the second most common technique [9] considered in automated vulnerability finding systems is symbolic/concolic execution [9]. While fuzzing is generally faster than symbolic/concolic execution at evaluating test cases, fuzzing can become highly inefficient when there are tight branches conditions in an application [9]. Tight branch conditions, such as verifying checksums and magic bytes, require specific inputs that can be computationally difficult to determine [9] [14]. The random mutations in fuzzing may take an exponential time to generate valid inputs for tight branch conditions [1]. Hence, tight branch conditions are roadblocks for fuzzing. Symbolic/concolic executors, on the contrary, can effectively bypass these roadblocks using program interpretation and constraint solving techniques [1]. Additionally, symbolic executors can generate test cases that directly exercise unique execution paths [12], hence efficiently discovering new execution paths in the application. Compare this to fuzzing, where many test cases produced can be redundant and therefore aren't as efficient in exploring different execution paths or increasing code coverage [27].

While symbolic/concolic execution is effective in bypassing tight branch conditions, as a consequence, the majority of its time is spent on program interpretation and constraint solving [12]. In particular, symbolic/concolic execution is especially susceptible to the ”path explosion” problem, therefore, limiting scalability [1]. Additionally, symbolic/concolic executors typically need access to source code, some description of the Operating System environment, and the precise semantics of the desired platform instruction set [14]. Applications that target unwidely instruction sets (with complex extensions), not as popular Operating Systems, or use obscure libraries can therefore be difficult to configure with symbolic/concolic executors [14]. Symbolic/concolic execution can be computationally expensive techniques for automated vulnerability discovery [9][14].

Rather than being standalone solutions to vulnerability discovery systems, symbolic/-concolic execution can be used to complement fuzzing. This approach is known as hybrid fuzzing. Hybrid fuzzing with symbolic/concolic execution performs selective symbolic/-concolic execution to overcome the roadblocks of fuzzing whenever present [9]. Hybrid fuzzing today, however, still lacks the scalability to test large complex real-world applications [9] [14]. Currently is ongoing research in hybrid fuzzing which aims to improve the scalability of hybrid fuzzing by providing new assistive techniques to complement fuzzing as an alternative to the computationally expensive symbolic/concolic execution [14]. Previous, assistive techniques to complement fuzzing include taint tracking test cases [2] [3] and patching away tight branch conditions [28].

2.2 American Fuzzy Lop: A State-Of-The-Art Fuzzer

One of the most commonly used, and evaluated fuzzers in the fuzzing community and our baseline fuzzer of choice is American Fuzzy Lop (AFL) [4]. In this section, we discuss the core internals of AFL and its different variants to show how state-of-the-art fuzzers are typically designed.

2.2.1 Core Internals of AFL and How It Relates to Other Fuzzers

Like in the design of most fuzzers, AFL takes in a seed corpus to use as a starting point for its input generating process. Throughout the fuzzing process, AFL maintains a queue that holds potential candidates. During the candidate selection stage, AFL uses a particular mutation strategy (i.e. genetic mutation) to select the next candidate to mutate. During the mutation stage, AFL selectively uses a wide array of mutators to generate a new input test case. At the evaluation stage, AFL tests the target application with the new input test case and makes observations about its execution. Like most fuzzers, the observations collected are based on the output of the program (most notably crashes) and some form of inside knowledge about the control-flow execution of the program. AFL can observe the branch (edge) coverage of each tested input and maintains a global map of edges seen in previous executions of the target application [4]. These observations are then used by AFL to guide the candidate selection process so to choose tested inputs with the most interesting execution results. AFL in particular prioritizes mutating tested inputs that have shown to reach new regions of code (increase code coverage) or execute a previously seen region of code by a factor of times more [4].

2.2.2 Different Variants of AFL and How It Relates to Different Types of Fuzzing

The kind of insider knowledge about the target application and its execution flow fuzzers like AFL are designed to have access to is the main distinguishing feature between the three major types of fuzzing modes. The three major types of fuzzing modes are black-box fuzzing, white-box fuzzing, and grey-box fuzzing. The AFL project [4] has different variants of AFL showcasing the three major types of fuzzing modes. The lesser-used mode of AFL, the black-box fuzzing mode, refers to fuzzing that does not have access to source code nor insights about runtime information and mostly monitors only program output. Black-box fuzzers are generally designed to evaluate as many input test cases as possible rather than generating quality test cases or testing efficiently. The more commonly used mode of AFL, the white-box fuzzing mode referred to as AFL-Clang-Fast, has full access to the target program's source code and monitors control flow execution via compiler inserted instrumentation to determine coverage. Unlike black-box fuzzers, white-box fuzzers prioritize generating quality inputs. However, white-box fuzzers often have to sacrifice speed in testing inputs because methods typically used to assist in generating quality inputs, such as program analysis, constraint solving, and fine-grained execution flow monitoring [27], often have a non-negligible performance overhead. Intuitively, white-box fuzzers can only be used when source code is available. To fuzz applications when only the binary is available and still prioritize generating quality inputs, finding a way to obtain insights into program behavior and execution flow is required. AFL's grey-box fuzzer, AFL-QEMU, uses dynamic binary instrumentation to obtain insights into runtime information and control flow information without having to rely on access to source code.

2.3 Existing Coverage Tracing Technologies in Coverage-Guided Fuzzing

Another category of fuzzing, regardless of the degree of access to source code and execution flow monitoring, is coverage-guided fuzzing. Coverage-guided fuzzing is any type of fuzzing that centers its input test case generation on maximizing code coverage. The motivation behind coverage-guided fuzzing is that covering more code might correlate with finding more bugs [13]. Coverage-guided fuzzers use coverage tracing technologies during program execution to obtain the coverage information needed to guide its input generation. The coverage tracing technologies currently available to coverage-guided fuzzers are compiler-inserted instrumentation, static binary rewriting, dynamic binary instrumentation, and hardware-assisted tracing.

2.3.1 Compiler-Inserted Instrumentation

In compiler-inserted instrumentation, at compile-time, the desired instrumentation code is inserted by the compiler at every basic block or between basic blocks (edges) to produce an accurate measurement of coverage information. Any additional overhead to the performance of the application is attributed to the amount of instrumentation code and the content of the instrumentation code inserted. Access to the compiler process allows some overhead to be reduced by optimizations and in-lining. However, because source code is required by compiler-inserted instrumentation, this becomes an invalid option for security researchers who are fuzzing closed source applications (e.g. proprietary software) and only have access to binaries.

2.3.2 Static Binary Rewriting

In static binary rewriting, the binary itself is modified to include the desired instrumentation code at each basic block or edge without any access to source code. Unlike compiler-inserted instrumentation, static binary rewriting can be used to assist in tracing coverage of closed source binaries. Static binary tools usually do not have access to as much information about the program to be able to apply optimizations and inlining as well as compilers, this leads to worse performance. Additionally, a classic challenge in static binary rewriting is recovering control flow information from a binary. Statically recovering control flow information is hard because it is difficult to accurately distinguish between scalars and references in a disassembled binary, furthermore, in the general case, this problem is undecidable [29]. In an attempt to solve this problem, existing static binary rewriting tools rely on heuristics [30] and assumptions [7, 31] about the binary to be able to appropriately rewrite the binary[32]. However, such static binary rewriting tools struggle to scale support to rewriting arbitrary binaries [7].

2.3.3 Dynamic Binary Instrumentation

Dynamic binary instrumentation is also able to insert instrumentation code into a binary without access to source code, but dynamically during runtime. As a result of translating the binary as it executes, dynamic binary instrumentation has access to runtime information. Unlike static binary rewriting, dynamic binary instrumentation can use runtime information to easily extract control flow information dynamically and insert instrumentation with accuracy. However, the currently available dynamic binary instrumentation tools have been shown to have a prohibitively high overhead reducing the efficacy of fuzzing [7, 33, 34].

An alternative to static binary rewriting and dynamic binary instrumentation for tracing coverage in closed binaries and the focus of our work is hardware-assisted tracing. Hardware-assisted tracing consists of hardware tracing facilities that can be used to monitor control-flow execution as the program executes. Hardware-assisted tracing does not require access to source code and can be configured to be as minimally intrusive to the program executing. We use the next section to provide background information on hardware tracing and related facilities and how they can be used to monitor control-flow execution to trace coverage.

2.4 Performance Monitoring and Tracing Hardware

Performance monitoring and tracing hardware are built-in hardware observation facilities located in and around the processor core. Performance monitoring hardware can measure performance metrics of the processor, usage of other computing resources (i.e. memory), and the occurrence of certain hardware events (i.e. machine checks). Output from performance monitoring hardware can be used by software tools to identify performance bottlenecks such as sections of code with high rates of cache-misses or mispredicted branches [35] and guide tuning compiler [36] and system performance[37]. Tracing hardware can capture a real-time trace of the program as it executes by recording control-flow execution [22], contextual information about software execution, and memory accesses [38]. The information obtained by tracing hardware can be used by debugging software to find and investigate program failures encountered during program execution [38]. While there are performance monitoring and tracing hardware in various platforms such as ARM [38] and AMD [35], our work focuses on the performance monitoring and tracing hardware on Intel processors [37] [39] because Intel processors are the most commonly used platform for evaluations in the fuzzing literature.

2.4.1 Intel Performance Monitoring

On Intel processors the performance monitoring hardware consists of a series of model specific registers (MSRs) [37]. A subset of these MSRs is referred to as performance monitoring counters. These counters are configured by performance monitoring control MSRs to measure a specific performance metric or count the occurrence of a supported event [37]. In addition to measuring and counting, performance monitoring hardware also supports various interrupt-based sampling facilities. To enable these facilities a performance counter is pre-configured to trigger a performance monitoring interrupt (PMI) upon counter overflows and its value is initially set to MAX-N, where N counts would trigger an overflow [37]. Examples of common events the performance counter could count are instructions retired and branch instructions retired [40]. After N counts occur, a registered PMI service routine is triggered and can take a snapshot of a selection of hardware tracing-related buffers, architectural state registers, and other processor state information [37]. As the last step before returning, the PMI service routine should reset the performance counter to MAX-N to enable future triggering of the PMI.

2.4.2 Hardware Tracing: Last Branch Record

One hardware tracing-related buffer that a PMI can sample is the Last Branch Record (LBR). The Last Branch Record is a hardware ring buffer that captures a running trace of the most recent taken branches by the processor [41]. The LBR consists of sets of MSRs, where for each taken branch a set of MSRs store the branch source (address of the branch instruction) and branch destination address (target of the branch instruction). The LBR can also be configured to store transitions to interrupts, and/or exceptions. One thing to note is that in modern processors each logical core in a hyper-threading enabled processor has a dedicated set of LBR MSRs [41]. A single set of LBR MSRs is commonly referred to as a branch record. The size of the LBR buffer is a fixed architecturally defined length, on modern Intel processors the length is typically 32 and can therefore store up to 32 branch records [41]. To read the LBR, one needs to be at privilege level 0 and use the RDMSR instruction to read each LBR MSR from each record.

There are various considerations concerning the configuration and usage of the LBR. Enabling the LBR tracing facility and reading the LBR MSRs has minimal impact on performance [40]. However, the firing of the PMI interrupt service routine to sample the LBR has a non-negligible impact on performance if the rate at which the PMI fires (sampling frequency of the LBR) is high [40]. Developers need to also consider the small length and ring buffer nature of the LBR when determining an appropriate sampling frequency. One thing to consider when sampling the LBR is that interrupt-based sampling introduces skid [42]. Skid refers to the time delay between the generation of an event (i.e. counter overflow) and the generation of a PMI [42]. In the case of LBR, the LBR facility can technically still record relevant branch records during this delay period which means one needs to increase their frequency of sampling the LBR according to the amount of skid to avoid losing information since the LBR is a ring buffer. Additionally, to reduce redundant branch records and the need for a high sampling frequency, the LBR built-in branch filtering capabilities can be used to filter out branches based on branch privilege level and branch instruction type. Nevertheless, developers wanting to perform LBR sampling need to find a balance between precision and performance.

2.4.3 Hardware Tracing: Brace Trace Store

Another hardware tracing-related buffer that a PMI can sample is the Branch Trace Store (BTS) buffer. The BTS tracing facility is the successor of the LBR tracing facility. When the BTS tracing facility is enabled each LBR branch record is sent out on the system bus [43] and is stored in a memory-resident (cache-as-RAM) BTS buffer that is part of the Debug Store (DS) save area[39]. BTS can give a full history of all taken branches executed [44] without sampling as frequently as in LBR sampling because BTS buffer can hold many more branch records [40]. The buffers residing in the DS save area can have a PMI triggered when the buffer (BTS buffer) is almost full [39]. One important difference between BTS and LBR is that in BTS tracing the BTS buffer being almost full is the source of the PMI whereas in LBR sampling it is a counter overflow [45]. This also means LBR sampling is susceptible to any counting inaccuracies in the performance counter such as undercounting [46] and overcounting [47]. Additionally, because the BTS buffer does not need to be sampled so often, BTS tracing is not as susceptible to sampling bias as LBR sampling is which is caused by the synchronization of the executing program with a fixed sample period [48]. While the frequent firing of PMIs is reduced in BTS, simply enabling

BTS alone can greatly slow down the performance of the processor [39] by an estimate as great as 40x [49]. To reduce the performance impact of BTS, the BTS branch filtering capabilities can be used to filter out branches by branch privilege level [40]. BTS tracing is expected to provide more precise branch tracing than LBR sampling at the expense of performance.

2.4.4 Hardware Tracing: Processor Trace

Another hardware-tracing related buffer that a PMI can sample is a trace buffer handled by the tracing facility Intel Processor Trace (PT). Intel PT can capture an extensive trace of software execution in the form of compressed packets [50]. The packets typically used for determining control-flow execution are packets that contain the encoded instruction pointer value of branch targets (TIP packets) and packets that use 1 bit to indicate Taken/not Taken for conditional branches (TNT packets) [22]. Generally, TIP packets are generated for indirect control flow instructions like indirect branches whose target address is resolved runtime. Intel PT has a specially designed output mechanism called the Table of Physical Addresses (ToPA) to store trace output with consideration to performance [22]. The ToPA maintains a collection of variable-sized regions of physical memory whose stores to these regions bypass caches and TLBs but are not serialized [22]. This is intended to minimize the performance impact of storing trace output during program execution [22]. The ToPA can be configured to trigger a PMI to sample the ToPA memory regions when they are almost full. Similar to the BTS buffer, the ToPA memory regions do not need to be sampled as often if they are considerably large. Additionally, the trace output to be stored to the ToPA memory regions can be minimized by using the Intel PT filtering capabilities such as Address IP, privilege level (CPL), and thread context (C3) filtering [22]. While Intel PT is said to capture traces with only minimal performance perturbation to the software being traced [22], the most variable and potentially performance hindering aspect in using Intel PT is the packet decoding process [16] [17]. Users of Intel PT use a custom or off-theshelf decoder which extracts the control flow information from decoded packets and uses it to walk through the disassembly of the traced application to reconstruct the execution flow [51]. Applications that generate a large output trace and are large (i.e. real-world applications) could be time expensive to process and slow down the delivery of control flow information to consuming software tools (i.e. coverage guided fuzzers).

2.4.5 Inherit Limitation of LBR and BTS: Fall-through Branches

As indicated on the Intel Software Developer's Manual [41], LBR and BTS store branch records for taken branches. In other words, only executed retired branch machine code instructions will be stored. One inherent limitation of LBR and BTS is that fall-through branches that exist at the source code level that are not explicitly translated into dedicated machine code branch instructions will not be visible to LBR and BTS. Therefore LBR (and by extension BTS) will only be able to monitor 50% of source code level branches [52]. It is important to consider this lack of visibility of fall-through branches when evaluating the branch coverage of fuzzers that use LBR and BTS to trace coverage. Other fuzzers such as ALF-Clang-Fast can view fall-through branches because it has access to the Intermediate Representation level where fall-through branches from the source code level are visible. Fuzzers that use Intel PT would be able to view fall-through branches and monitor all source code level branches if the decoder in use analyzes TNT packets in addition to TIP packets.

CHAPTER 3
RELATED WORKS

As mentioned in chapter 1, the hardware-assisted tracing mechanism most commonly selected by prior works in the fuzzing literature is Intel PT [6, 14, 15, 16, 17, 18, 19, 20, 21]. The most commonly cited reasons and motivations for selecting Intel PT are 1.) inefficiencies of dynamic binary instrumentation (particularly with QEMU), 2.) the limited capacity of Last Branch Record [17], and 3.) prior work citing the Intel Software Developer's Manual's description of Intel Brace Trace Store as a mechanism that can greatly reduce the performance of the processor [15, 39]. None of the prior fuzzing works [6, 14, 15, 16, 17, 18, 19, 20, 21] cite specific studies that evaluate the performance of BTS and LBR with Intel PT.

3.0.1 State of Hardware-Assisted Tracing in Fuzzing: Intel PT Based Fuzzers

As mentioned in subsection 2.4.4 while capturing the execution information with Intel PT has minimal performance perturbation to software [22], the decoding of packets and reconstructing execution flow can have a significant overhead [16, 17, 21]. Thus it is important to consider that overhead results in using Intel PT can vary greatly depending on the performance of the selected decoder, benchmarks, and seeds [16]. PTrix [16] has an implementation that reports lower overheads than AFL-QEMU by performing online (parallel) decoding and forgoing the reconstruction process by hashing encoded packets to collect path coverage as opposed to edge coverage. However, their Edge-PT [16] implementation with parallel decoding as well and a decoder that performs reconstruction and collects edge coverage incurred a greater performance overhead compared to AFL-QEMU [16]. Another point to consider in evaluations is that PTrix requires 2X CPU time because it offloads decoding to dedicated cores to perform parallel decoding [16]. Offloading the decoding and reconstruction process to idle logical cores is commonly seen in systems that use Intel PT for Control-Flow Integrity enforcement, another field where performance is critical [23]. However, a processor executing various fuzzing jobs in parallel might have a very limited number of idle cores available. Another prior work is PTfuzz [15] which performs decoding after test cases have finished executing (offline). While PTfuzz [15] reports lower overheads than AFL-QEMU, other fuzzing works [16, 10] report PTfuzz having higher overheads than AFL-QEMU and cite differences in results are due to using different benchmarks and seeds. Kafl [17] and its derivatives [14, 18, 20] also use Intel PT to obtain coverage information and are built on top of modified versions of KVM and QEMU to fuzz operating systems, libraries, and hypervisors in isolation. While Kafl [17] and its derivatives [14, 18, 20] can perform binary-only fuzzing, there are currently limited evaluations that include these fuzzers and comparisons with other [16, 15, 6] Intel PT works. The Github page for one of the derivatives of Kafl, NYX [20], has preliminary evaluations [53] for the performance overhead of different decoders belonging to different Intel PT fuzzing works. Hence it can be difficult to draw comparisons among the different Intel PT fuzzing works and make conclusions about the general performance impact of Intel PT in fuzzing. In the same vain many prior Intel PT fuzzing works [17, 14, 15, 16, 18, 20] do not include the Intel PT implementation of honggfuzz [6] nor of Winafl [19] in their evaluations. It is important to note that honggfuzz has support for in-process fuzzing, which can greatly benefit performance, whereas the other Intel PT fuzzing works do not all have this support. Additionally not all Intel PT fuzzing works can support fuzzing Windows applications like Winafl [19]. One must consider CPU time, used benchmarks, other implementation/design-related factors, and gaps in the Intel PT fuzzing literature to be able to draw fair comparisons between Intel PT and other hardware tracing mechanisms.

3.0.2 State of Hardware-Assisted Tracing in Fuzzing: Lack of Evaluations of LBR/BTS in Fuzzing

Upon reviewing the fuzzing literature there are currently very limited evaluations of the performance impact to fuzzing in using either Brace Trace Store and Last Record Branch for hardware-assisted tracing. As mentioned in chapter 1 honggfuzz [6] is one of the few existing fuzzers that has an implementation that uses the Intel BTS hardware tracing mechanism to obtain coverage information. Honggfuzz [6] has indicated in their GitHub that their Intel PT implementation should be faster than their BTS implementation although there isn't an evaluation yet to show this. In general, though there are very few fuzzing works that include the Intel BTS implementation of honggfuzz [6] in their evaluations. Most fuzzing works that use hongggfuzz [6] use the compiler-assisted implementation of honggfuzz in their evaluations. As mentioned in chapter 1, while there are even fewer fuzzing works that utilize Intel LBR, we notice there are numerous works in another performance-critical domain that have used Intel LBR for Control-Flow Integrity (CFI) enforcement [23]. These CFI enforcement works use LBR as a pseudo shadow stack to mitigate Return-Oriented Programming (ROP) attacks [54]. Walcott-Justice et al. [52] has used LBR sampling in the domain of software testing (i.e. structural testing) as a lower time and space overhead alternative to instrumentation to collect coverage information to perform test coverage analysis in resource-constrained environments. Walcott-Justice et al. has also provided one of the earliest performance evaluations of BTS in software coverage testing before the Linux Kernel had officially supported LBR and BTS monitoring [40]. The recent fuzzing work by Ding et al. [55] implements a customized hardware platform that introduces a new hardware primitive that is similar in principle to Intel LBR and is used to obtain coverage information.

CHAPTER 4
DESIGN

4.1 Overview

Our goal is to develop a fair and meaningful evaluation of the LBR/BTS tracing mechanisms as respective sources of coverage information in coverage-guided fuzzers. In regards to research question **RQ1**, to evaluate the performance implications of using LBR/BTS in coverage-guided fuzzers, we need to see how differently a well-behaved fuzzer performs when using LBR/BTS as an alternative source of coverage information. There are, however, various avenues for bias, specifically differences due to fuzzing implementation and instrumentation insertion techniques. If these biases are unaccounted for our results would produce a misleading assessment of the use LBR/BTS in coverage-guided fuzzing. We develop two fuzzing implementations that use LBR/BTS respectively and are designed to reduce the aforementioned biases. Our fuzzing implementations, LBR-Fuzz and BTS-Fuzz are based on the well evaluated fuzzer AFL [4]. An additional second goal is to implement a LBR/BTS-based coverage feedback mechanism most effectively so that fuzzers can best utilize the aforementioned hardware features to obtain coverage information. A non-trivial challenge, however, is properly configuring and managing the LBR/BTS facilities so to ensure the coverage information generated is accurate and that it is accessible to fuzzers as efficiently as possible. For our solution, we use the Linux Kernel's perf_event(s) (also known as perf) [24] [25] to configure and manage the LBR/BTS facilities. In regards to research question **RQ2**, to determine if LBR/BTS can be used to improve binary-only fuzzing we propose to use a non-compiler assisted minimally invasive method to insert our LBR/BTS monitoring code into target applications to be able to test its effectiveness in binary-only fuzzing.

4.2 Design of Our LBR/BTS Coverage Tracing Mechanisms

4.2.1 Challenges to Configuring LBR/BTS

The first challenge we discuss is how to effectively enable and configure LBR and BTS facilities. There are a few approaches one could take to configure the LBR/BTS facilities. The first series of approaches involve writing a kernel driver that uses the necessary privileged instructions to configure the LBR/BTS control MSRs to enable the LBR/BTS facilities. For LBR sampling, the kernel driver would need to use the RDMSR instruction to individually read LBR MSRs containing the trace of the most recent taken branches. However, as mentioned in subsection 2.4.2, one needs to consider that the depth of the LBR is very small, at the time of this writing on modern processors the LBR ring buffer can hold is a maximum of 32 branch records at any given time. Thus, the sample period for sampling the LBR ring buffer should be carefully calibrated to avoid the additional overhead from excessive sampling while still ensuring all branch records of interest are read. Recall LBR is the predecessor to BTS and that in BTS tracing LBR branch records are continuously logged into a specified large buffer in memory. For BTS tracing the kernel driver would allocate the buffer for BTS and read the BTS buffer when it is almost full. As mentioned in subsection 2.4.3, one thing to consider with BTS is the overhead from writing each LBR branch record to the BTS buffer. A non-trivial challenge for any approach that uses a kernel driver to perform LBR sampling/BTS is to figure out how to monitor a specific process. In other words, how do we configure our system so that the LBR/BTS facilities only trace branches executed by our target application? One thing to note is that each logical core has its own set of LBR MSRs. An initial approach to the challenge would be to restrict the execution of our desired application to a logical core with the LBR/BTS facilities enabled. However, a kernel driver approach to implementing LBR/BTS must also handle context switches. Since context switches are handled by the kernel, the task scheduler of the kernel would also have to be modified to include the saving and restoring of LBR MSRs upon context switches. Such modifications to the kernel's task scheduler would also need to appropriately enable/disable the LBR/BTS facilities during a context switch. Any approach that uses a kernel driver to implement LBR sampling/BTS would require recompiling and rebooting a new kernel that contains modifications mentioned earlier.

4.2.2 Our Approach to Configuring LBR/BTS

We have opted for an easier and portable approach to implement LBR sampling/BTS tracing that does not involve writing a custom kernel driver and does not modify the kernel's task scheduler. We use the Linux Kernel's perf_event_open API to interface with the LBR/BTS facilities to obtain control flow execution information of our target application. The backend configuration of BTS and LBR is abstracted away by the perf_event_open API. One key difference in configuration is that for LBR sampling we interrupt the application after every N LBR_INSERTS, where N can be at most the length of the LBR, and for BTS tracing we interrupt the application when the BTS buffer is almost full. LBR_INSERTS refers to when an entry is added to the LBR, calibrating the number of LBR_INSERTS that would trigger an interrupt in LBR sampling is further discussed in subsection 5.1.1. The Linux Kernel supports the perf_event_open API through its perf_events subsystem and handles for us the saving and restoring of the LBR stack MSRs. Additionally, the perf_event_open API directly allows us to implement LBR/BTS at a per-process granularity, which is not a feature directly offered in hardware. Rather than going through the hassle of modifying the kernel, to specifically monitor our target application we simply need to provide the perf_event_open API with the PID of the running target application. The perf_event_open API allows us to enable LBR/BTS and monitor our target application from a separate process. We write our LBR sampling/BTS code using the perf_event_open API from userspace, making it easy to integrate into any fuzzing implementation.

4.3 Design of Our Fuzzing Implementations

4.3.1 Challenges to Developing a Fair Performance Evaluation of Fuzzers

To develop a fair comparison of fuzzers and evaluate the performance impact of using LBR/BTS in coverage-guided fuzzers, we need to make sure all other variables that could also impact performance are controlled in our assessment. When **RQ1**, one challenge is to reduce bias due to differences in fuzzing implementation. Differences in how new test cases are executed (for example whether or not a new process is created) or how new test cases are generated (for example differences in the overhead of mutation algorithms) can greatly affect our assessment. Another challenge to consider when addressing **RQ1** is the bias due to differences in instrumentation insertion techniques. Different instrumentation techniques vary in runtime, memory overheads [7] and can produce different execution characteristics even when instrumenting the same binary [11].

4.3.2 Baseline Fuzzer

To address the aforementioned challenges, we choose a baseline fuzzer and add the minimal changes necessary to incorporate our LBR/BTS coverage tracing mechanisms. We choose as our baseline fuzzer AFL because it is one of the most widely-adopted and enhanced fuzzers in industry and academia. Aside from the changes to incorporate our LBR/BTS coverage tracing mechanisms, we keep the overall design and implementation of the AFL fuzzing engine the same. Our fuzzing implementations, LBR-Fuzz and BTS-Fuzz, are based on AFL-Clang-Fast and replace the original coverage tracing mechanism with our respective LBR/BTS coverage tracing mechanisms. AFL is commonly included in the evaluation of fuzzers in the fuzzing community. Using AFL as our baseline fuzzer allows our evaluation of the performance impact of LBR sampling/BTS to be relatable to other fuzzing works that have also used AFL as their baseline.

4.3.3 Modifications to the Baseline Fuzzer for Source-Code Available and Binary-Only Fuzzing

To address **RQ1**, when performing source-code available fuzzing we use the same instrumentation insertion technique as AFL-Clang-Fast to insert our LBR/BTS coverage tracing mechanisms. AFL-Clang-Fast uses a Clang wrapper to insert the AFL forkserver and AFL instrumentation into target applications. We modify the Clang wrapper to insert only our customized forkserver equipped with our LBR/BTS coverage tracing mechanisms instead. Thus can fairly compare the performance of our fuzzing implementations directly with AFL-Clang-Fast to answer research question **RQ1**. To address **RQ2**, when performing binary-only fuzzing we statically rewrite our target binaries to include our LBR/BTS coverage tracing mechanisms. The static binary rewriting tool we use is discussed in subsection 4.3.5.

4.3.4 How Our LBR/BTS Coverage Tracing Mechanism Are Incorporated into AFL

For our fuzzing implementations, we modify the AFL forkserver to include the code for our LBR/BTS coverage tracing mechanism, which interfaces with the Linux perf_events subsystem to enable the LBR/BTS facilities in hardware and capture control flow execution. When the forkserver forks a new process to evaluate a test case, our LBR/BTS monitoring code tells perf_events to enable the LBR/BTS facilities at the beginning of program execution. During the execution of our target applications, whenever branch instructions of interest are executed LBR branch records are generated by hardware and captured by perf_events as seen in Figure 4.1. This allows us to monitor the branch activities of our target applications during their execution for each test case with minimal interference with the application's control flow. Due to the infrastructure differences in LBR sampling and BTS tracing, perf_events captures the LBR branch records differently. In both of our fuzzing implementations, perf_events gives us the branch information, see Figure 4.1, and we then process it before updating the AFL shared coverage bitmap appropriately to reflect the cur-

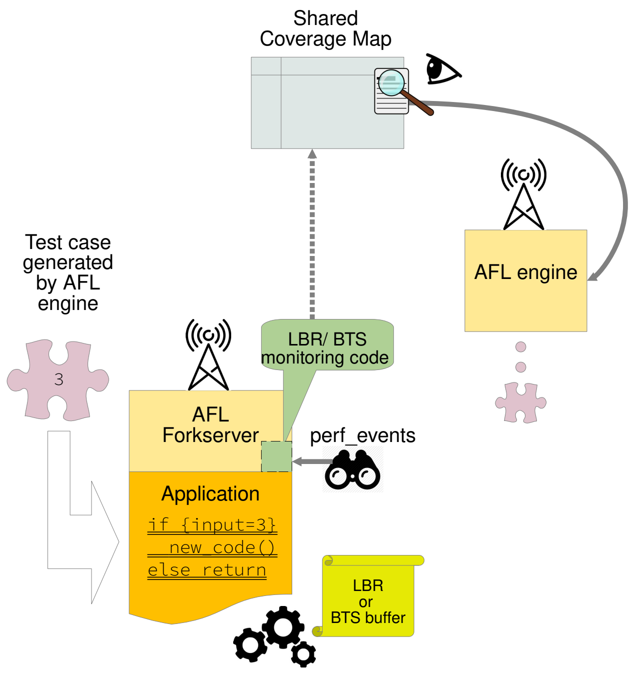

Figure 4.1: Updating shared coverage map with branch activity information from LBR sampling/BTS during execution of target application.

rent branch coverage profile seen at runtime. When the application terminates, the AFL forkserver notifies the AFL engine of the application's termination and exit status. The rest of the fuzzing procedure is offloaded to the original design of AFL. The AFL engine inspects the coverage information we provided in the global coverage map and along with the exit status of the application decides if the test case should be used again for further mutation or discarded.

4.3.5 Binary-Only Fuzzing Challenges

In regards to research question **RQ2**, to determine if LBR/BTS coverage tracing mechanisms can improve binary-only fuzzing, we need a way to enable and configure the LBR/BTS facilities to monitor control flow execution of our target applications with only access to binaries. One challenge in properly enabling LBR/BTS monitoring in binaries for binary-only fuzzing is making sure LBR/BTS is enabled before the target application executes. It is crucial to enable LBR/BTS consistently at the right time to generate accurate and consistent coverage information. If we enable LBR/BTS early, late, or at inconsistent times for each run, the coverage information generated will not be accurate and will not be able to effectively guide input mutation to discover new regions of code. For source-code available fuzzing, we can use the Clang wrapper from AFL-Clang-Fast to ensure that the LBR/BTS facilities are enabled at the right time using compiler constructors to run our code during program initialization. Since we can not use compiler constructors on binaries alone, for binary-only fuzzing we need a solution for not only inserting our LBR/BTS monitoring code into binaries but also ensuring our monitoring code executes at the right time. Additionally, we also need to be able to insert the AFL forkserver into binaries to communicate with the AFL engine and the forkserver must be the first function to execute at program startup.

To address the aforementioned challenges, we use the static binary rewriter tool e9Patch [32] to insert our customized forkserver equipped with our LBR/BTS monitoring code into pre-compiled target application binaries before fuzzing. We choose the e9Patch because of its ease-of-use Plugin API and its ability to insert an initialization function. Additionally, the authors of e9Patch have a project called e9afl [56] that shows e9Patch can be used to insert the AFL forkserver and AFL instrumentation code into pre-compiled binaries before runtime to fuzz with AFL. Unlike other static binary rewriting tools, e9Patch can insert our LBR/BTS monitoring code without moving instructions or re-adjusting the target addresses of jump instructions [32]. By using a static binary rewriter tool to insert our LBR/BTS monitoring code, our fuzzing implementation can be directly compared with AFL-QEMU and other fuzzers that can perform binary-only fuzzing.

CHAPTER 5
IMPLEMENTATION

In this chapter, we discuss various technical implementation details about our fuzzing implementations. We discuss in detail how we use perf_events to configure LBR/BTS, what modifications we make to the AFL forkserver, and how we insert our monitoring code into target applications.

5.1 Using Perf_Events to Configure LBR/BTS

In our fuzzing implementations, from within our customized forkserver, we use the Linux kernel's perf_events subsystem to access the hardware tracing facilities. To interface with perf_events we use the perf_event_open() application programming interface (API). Through perf_event_open(), we give perf_events the PID of the application to monitor and specific configuration values to set up the respective LBR/BTS facilities. We select configuration values based on the perf_event_open() Linux manpage [25]. In both of our fuzzing implementations, we specify in the configurations values that we only want branch information from user space and to exclude information from kernel space and hypervisor environments. Additionally, we specify our desire to have the tracing mechanisms start in disabled mode when the configuration is complete. We do this to have precise control of when the tracing mechanisms start, which is critical for obtaining accurate coverage information. In our fuzzing implementations, we control the enabling and disabling of the tracing mechanisms with ioctl() system calls. The ioctl() system calls require the file descriptor that is returned by perf_event_open() upon successful configuration. Additionally, we use this same file descriptor to obtain the coverage information collected by perf_events. We do this by creating a memory mapping using mmap() and the file descriptor returned by perf_event_open(), the result is a memory mapping to the perf_events internal ring buffer that stores the runtime collected coverage information. Lastly, in our fuzzing implementations, due to the built-in support from the perf_events subsystem, we can poll this file descriptor to be notified when the perf_events internal ring buffer is ready for reading. For LBR monitoring, we specify that perf_events notify us when there have been at least 12,500 perf_events snapshots recorded in the perf_events ring buffer. For BTS monitoring we specify that perf_events notify us when there have been at least 2,097,152 bytes written to the perf_events dedicated auxiliary buffer, which is a separate sample buffer for high-bandwidth data streams [25]. All of the programming related to perf_event_open, as discussed above, is done within our customized version of the AFL forkserver.

5.1.1 Differences in Configuration between LBR and BTS

In perf_event_open() we pass in different values for the type and config subfields of the perf_event_attr data structure [25], these values are listed in Table 5.1. These values are important because they tell perf_events which monitoring feature we want to use.

Table 5.1: Important configuration values that define LBR sampling and BTS tracing passed to perf_event_open(). NA means not applicable

Mechanism	type value	config value
LBR sampling	PERF_TYPE_HARDWARE	PERF_COUNT_HW_BRANCH_INSTRUCTIONS
BTS	value in /sys/bus/event_source/devices/intel_bts/type	NA

For LBR sampling, we specify further that we want to perform event sampling. For this, using the perf_event_attr data structure, we pass into perf_event_open() a sampling period, the type of branches we want to monitor, and a value specifying that we want to sample the LBR ring buffer as indicated by the perf_event_open manpage [25]. For the type of branches to monitor for we choose to monitor for userspace conditional branches. For LBR sampling, we choose to monitor for only one type of branch because for the sake of performance we want to reduce frequent and redundant sampling of the LBR. We choose to monitor for conditional branches because we believe it to be the most important branch information for fuzzing. Additionally, we have perf_events program a performance counter to monitor for LBR_INSERTS and to take a snapshot of the LBR after every N LBR_INSERTS, where N is the sampling period [25]. While ideally, the sampling period should be equal to the length of the LBR, unfortunately, because of the presence of skid we found in our preliminary experiments that we need to sample the LBR more frequently and so our sampling period is 32-MAXSKID=17, where experimentally we determined the MAXSKID to be 15. The effect of skid on our experiments is further discussed in chapter 7. Hence, perf_events will take a snapshot of the LBR after every 17 LBR_INSERTS for which the LBR should only hold 17 retired conditional branches from userspace. perf_events stores the LBR snapshots in the internal perf_events ring buffer that is memory mapped to our main LBR sampling process.

For BTS tracing, using the perf_event_attr data structure, we need to pass to perf_event_open() the value the kernel exports in the file system location as shown in Table 5.1. As a result, perf_events sets up the DS Save Area[45] which will contains the BTS buffer. During runtime, the LBR branch records will be transferred over to the BTS buffer. When the BTS buffer is almost full, perf_events will take a snapshot of the BTS buffer and store it in the perf_events auxiliary buffer. The size of the BTS buffer is configured when memory mapping to the perf_event_open file descriptor by specifying the desired buffer size. For BTS tracing using perf_events, one can only specify to monitor branches based on privilege level. We specify to monitor for userspace branches.

5.2 Structural Modifications to AFL Forkserver

For our fuzzing implementations, we choose to modify the AFL fuzzer version 2.57b, the most up-to-date version at the time of this writing, which was released inn June 2020. As mentioned earlier, all of the programming related to LBR/BTS monitoring is done within our customized version of the AFL forkserver which is part of the AFL runtime component.

It is important to remember that the AFL forkserver is inserted into target binaries in such a way that at runtime the forkserver runs before the target application's main function.

To start our customized version of the AFL forkserver, as in the original implementation, we have the AFL engine execute a target application binary with execv(). The main responsibility of the forkserver is to clone new child target application processes by having a virgin copy of itself in memory and leveraging the Linux copy-on-write implementation of fork to avoid further calls to execv(). Additionally, the forkserver is designed to clone new child processes whose execution is steered to execute the target application's main function.

5.2.1 Initialization Tasks in Forkserver

When the forkserver starts, we have the forkserver parse proc/pid/maps to obtain the virtual address space range of its text segment. We do this because we need an address range to filter out branch records that do not pertain to the executable code of the target application. We only need the forkserver to obtain this address range once before the generation of any children because the entire virtual address space of a parent process is replicated in child processes during a call to fork() [57]. When the forkserver clones a new target application process to fuzz, we have the new target application process raise a SIGSTOP signal to stop itself, meanwhile, the forkserver configures the LBR sampling/BTS facilities discussed in section 5.1. Right before our forkserver sends a continue signal, SIGCONT, to the stopped child process, we have our forkserver enable the LBR sampling/BTS facilities. To avoid race conditions, we have the forkserver perform a waitpid with WUNTRACED beforehand to make sure we send the continue signal only after confirming the child process was indeed stopped. Once the child process receives the continue signal, it's execution is steered to start executing the target application's main function.

5.2.2 Monitoring Tasks of the Forkserver

While the child process executes the original code of the target application, we have the forkserver continuously poll two file descriptors in a while loop to capture two events of interest. One file descriptor captures the SIGCHLD signal which is sent to the forkserver when a child process dies. Just as in the original AFL implementation we send the termination information of the dead child to the AFL engine, afl-fuzz. The other file descriptor, which is the return value from perf_event_open, indicates when the perf_events internal ring buffer is ready for reading. Using a file descriptor to capture SIGCHLD signals allows our forkserver to keep track of queued SIGCHLD signals as a result of dead children even if other events are triggered first. To avoid data race conditions, we process the perf_events ring buffer both when perf_events indicates the buffer is ready to read and when we detect a child has died. We do this because there might be unprocessed branch records in the perf_events internal ring buffer since it was last processed.

5.2.3 Processing Perf_Events Ring Buffer and Updating AFL Bitmap

To process the perf_events ring buffer we first obtain the necessary meta-information such as the offset to the ring buffer head and tail using the data structure provided by perf_events called perf_event_mmap_page that is written to the first page of the perf_events ring buffer. Using the meta-information and memory mapping to the perf_events ring buffer, we copy the contents of the perf_events ring buffer into a temporary buffer. As soon as the contents of the perf_events ring buffer are copied over we immediately update the offset of the ring buffer tail to the head so as to free ring buffer memory to accommodate for incoming branch records. The contents of the perf_events ring buffer are organized in memory as perf_events records, where a single perf_events record contains a header followed by the branch records. To parse the contents of the perf_events ring buffer, in the temporary buffer we iterate over the perf_events records and only further analyze perf_events records of the type PERF_RECORD_SAMPLE for LBR sampling and PERF_RECORD_AUX for BTS tracing. Records of the aforementioned types will contain branch records. To obtain edge-based code coverage information we need to guide our fuzzing implementations, we iterate over these branch records and only consider branch records whose branch source and destination addresses are 1.) within the address space range of the text segment we obtained during the startup of the forkserver, and 2.) to not referring to artificial branches we inserted into the target application. In regards to artificial branches, to ensure perf_events is always triggered to record the last legitimate branch records during a crash and during normal/abnormal program exits we have the target application run 32 artificial branches during a crash or exit. Our customized forkserver uses atexit() to register an exit handler that executes these artificial branches. Since LBR-Fuzz is subjected to the effects of skid, when parsing the contents of the perf_events ring buffer we calculate the skid seen for each LBR snapshot and further restrict our reading window of the perf_events ring buffer to ensure we are correctly reading the LBR branch records in the correct order and not rereading already read branch records. We do not have a skid-detection method in BTS-Fuzz since it is not affected by skid like LBR-Fuzz. To update the AFL shared coverage bitmap, similar to the original AFL implementation we compute the following hash to determine where to write into the bitmap.

$$(\text{src} \gg 1) \oplus \text{dest}$$

Unlike the AFL implementation which uses a random value generated at compile-time for the src variable, for each valid branch record we use the address offset of the branch source address for the src variable. For the dest variable, we use the address offset of the branch destination address.

5.3 How We Insert Our LBR/BTS Monitoring Code

For source-code available fuzzing, we use the clang wrapper provided by AFL-Clang-Fast to insert our customized forkserver equipped with our LBR/BTS monitoring code. When compiling the target application binaries with the clang wrapper, the compilation will include a runtime component that contains our customized forkserver. The runtime component uses compiler constructors to establish a new initialization routine that will run before the main function. The initialization routine contains our customized forkserver. Furthermore, we discard entirely the LLVM pass used originally by ALF-Clang-Fast since we rely on our LBR/BTS monitoring code to get branch information.

For binary-only fuzzing, we insert our monitoring code into target binaries using the static binary rewriter called e9Patch [32]. e9Patch uses a suite of binary rewriting methodologies, such as instruction punning, eviction, and padding, to insert jumps to trampolines with instrumentation code without the need to move over other instructions [32]. e9Patch has a front-end tool called e9Tool with a plugin API to allow for fine-grained control over the generated trampolines [58]. The authors of e9Patch also have a project called e9AF [56]. The e9AF project has open-sourced an e9Tool plugin, supporting scripts and their customized AFL runtime component to aid e9Tool in rewriting closed-source binaries to contain the AFL forkserver and AFL instrumentation. We reuse their plugin and supporting scripts to meet our purposes. We modify their runtime component by modifying their forkserver to match the functionality of our customized forkserver. We also modify their plugin to only insert our customized AFL forkserver. From their plugin, we also remove any code that was previously used for adding AFL instrumentation. As mentioned in their documentation, one important limitation of the e9Patch backend is that patched code cannot use standard libraries directly, including glibc [32]. While the e9Patch backend has a parallel implementation of common libc functions, we had to dynamically load glibc and get the function pointers to the libc functions our customized forkserver needed that were not supported by their parallel libc implementation such as atexit. Additionally, while their parallel libc implementation did have an implementation for malloc(), we ended up using the glibc implementation instead because their malloc() allocated significantly more memory than we needed. With our modifications to e9AFL, we can rewrite closed-source binaries to include our LBR/BTS monitoring code and provide feedback to our fuzzing implementations.

CHAPTER 6
EVALUATION

The goal of our evaluations is to evaluate the performance implications of the LBR/BTS tracing mechanisms in coverage-guided fuzzers. We also seek to assess the inherent drawbacks/weaknesses of using these tracing mechanisms in coverage-guided fuzzers. Our evaluation consists of two categories source-code available fuzzing and binary-only fuzzing. For each category, we compare the appropriate AFL-based fuzzers with our fuzzing implementations, LBR-Fuzz and BTS-Fuzz. From our experimental results, we determine the performance overhead directly attributed to the LBR/BTS tracing mechanisms respectively (which answers research question **RQ1**) and its impact on the performance of binary-only fuzzing (which answers research question **RQ2**).

6.1 Experimental Setup

6.1.1 Fuzzers

As discussed in subsection 4.3.2, our LBR/BTS fuzzing implementations are based on AFL-Clang-Fast and replace the original coverage tracking mechanism with the LBR/BTS tracing mechanisms respectively. In our evaluations, we refer to our fuzzing implementations that use the LBR/BTS tracing mechanisms as LBR-Fuzz and BTS-Fuzz respectively. For our source-code available fuzzing evaluation, we compare LBR-Fuzz and BTS-Fuzz with AFL-Clang-Fast. For our binary-only fuzzing evaluation, we compare LBR-Fuzz and BTS-Fuzz with AFL-QEMU [4] and PTfuzz [15]. AFL-QEMU and PTfuzz are AFL-based fuzzers that use different coverage tracing methods from LBR-Fuzz/BTS-Fuzz. AFL-QEMU runs userspace emulation and performs on-the-fly instrumentation at runtime to obtain control-flow execution of the target application. PTfuzz uses the Intel PT hardware tracing mechanism to monitor software execution of the target application and processes the compressed output stream to determine control-flow execution. In terms of fuzzing works that used Intel PT, we choose to compare LBR-Fuzz and BTS-Fuzz with PTfuzz because aside from using a different hardware mechanism to obtain coverage information, in terms of components and fuzzing architecture this particular PT fuzzing work is most similar to our fuzzing implementations. Honggfuzz PT [6] and Kafl [17] and its derivatives [14, 18, 20] have many more differences aside from just using a different hardware mechanism. Honggfuzz PT, in particular, only uses TIP packets, hence only monitors for indirect control-flow instructions such as indirect branches to perhaps avoid the expensive process of disassembling the output stream [53]. Therefore the coverage information used by honggfuzz PT isn't comparable to the coverage information used by LBR-Fuzz/ BTS-Fuzz which monitors for conditional branches and all userspace branches respectively. PTfuzz, on the other hand, uses both TIP and TNT packets, hence monitoring for both indirect branches and conditional branches. Ptrix [16] uses path coverage instead of edge coverage which is used by our fuzzing implementations, making it hard to compare against our fuzzing implementations.

6.1.2 Benchmark and Seeds

In the fuzzing community there is a lack of a standard, sufficient and agree-upon set of benchmark programs to use when evaluating fuzzers [13], [11], [59], [60], [61]. Typically fuzzers are usually evaluated on hand-picked benchmark programs [11]. Careful consideration must be used when selecting target applications to test new fuzzing technologies because, as demonstrated by Fioraldi et al. [62], results produced are often target-dependent. We take the recommendations of Klees et al. [13] and Li et al. [11] who suggest a benchmark suite should contain target applications that are representative of real-world programs: diverse in size, functionality, and coding styles. We also look through several academic fuzzing papers to find commonly selected targets, including some of which had proposed and released their benchmark suites [11], [8], [59]. We hand select the following real-world applications as seen in Table 6.1 which are a subset of the UNIFUZZ benchmark suite [11].

Table 6.1: Subset of selected real-world programs of the UNIFUZZ benchmark suite. @@ is a placeholder for the name of an input file

Type	Program	Version	Arguments
Image	jhead	3.00	@@
Network	tcpdump	4.8.1 + libpcap 1.8.1	-e -vv -nr @@
Audio	mp3gain	1.5.2-r2	@@
Binary	objdump	binutils 2.28	-S @@

For the selection of initial seeds, we use a subset of seeds provided by the UNIFUZZ benchmark suite [11]. The seeds were collected from the Internet according to the file format type requirements of the selected benchmark programs. We obtain our selected benchmark programs and seeds from the UNIFUZZ Github repository [63]. An important note that we need to consider is that fuzzing implementations based on AFL should have seeds that are each no greater than 1MB, the AFL maximum seed size requirement. It is a common practice in fuzzing to minimize both the number of seeds and the size of the seeds [64], therefore we select a subset of the smallest seeds from the UNIFUZZ seed set. Although the work in [13] has demonstrated the importance of having both empty and non-empty seeds in the evaluation of fuzzers, their work shows this to be most relevant when evaluating the potential improvements to bug finding capabilities. Our work is more focused on evaluating fuzzers based on system performance overhead and reliability, as the primary goal of our work is to determine the performance implications of the LBR/BTS tracing mechanisms in coverage-guided fuzzers. Finally, while Herrera et al. [64] and Rebert et al. [65] suggest that fuzzing efficiency can improve by using distillation and seed pre-processing techniques to remove redundant seeds, this is out of the scope of our work. We are more interested in evaluating the efficiency of our fuzzers per run and not across fuzzing runs.

6.1.3 Evaluation Metrics

We use the following evaluation metrics to compare LBR-Fuzz and BTS-Fuzz with AFL-Clang-Fast, AFL-QEMU, and PTfuzz: branch coverage (edge coverage), total paths discovered, stability, system performance overhead (based on executions per second), and number of unique crashing inputs.

We measure branch coverage to ensure our fuzzing implementations are covering as much code as the baseline fuzzer AFL-Clang-Fast. Additionally, branch coverage is a widely used metric in the fuzzing community to assess fuzzing effectiveness [13]. To calculate branch coverage, we use the afl-cov tool [66] which also can report additional information such as path coverage and line coverage.

For stability, we use AFL's definition of stability which measures the consistency of observed coverage traces. If a target application always behaves the same (produces nearly identical coverage profiles) when a fuzzer under evaluation gives the same input data, such fuzzer will earn a stability rating of 100% [4]. Otherwise, if a target application's behavior differs, the AFL internal system will assign the fuzzer a lower stability rating. We measure stability to approximate the correctness of the coverage information used by the fuzzer under evaluation and hence approximate fuzzing reliability.

We obtain measurements for system performance (executions per second), total paths discovered, number of unique crashing inputs, and stability rating by parsing AFL's output fuzzing statistics reports. For executions per second, we parse for the start-time timestamp, the last update to the fuzzing environment timestamp, and the metric total_executions to manually calculate the executions per second. We manually calculate executions per second because the executions per second posted in the fuzzer statistics report might only be a snapshot of a specific point in time. The evaluation metrics, executions per second and total paths discovered, allow us to evaluate the fuzzers in terms of performance and fuzzing efficiency.

Lastly, as a proxy metric for fuzzing effectiveness, we measure the number of unique crashing inputs. The unique number of crashing inputs is an approximate insight into a fuzzer's ability to find crashing inputs. However, we should note unique crashing inputs do not always directly map to bugs found. The work by Klees et al. [13] discusses the threats to validity when using the metric number of unique crashing inputs as the primary metric to evaluate fuzzing effectiveness. Due to time and computing resource constraints, we did not triage unique crashing inputs to determine if any new vulnerabilities were found.

6.1.4 Platform and Configuration

We conduct our experiments on a machine with a 6-core Intel(R) Core(TM) architecture Coffee Lake, hyper-threading enabled (12 logical cores in total), and 32GB of available RAM running Ubuntu 18.04 LTS 64-bit.

We acknowledge that a fuzzer's bug-finding abilities can vary across runs and even over the course of a single run which has been demonstrated by Klees et al. [13]. The experiment guidelines of Klees et al. [13] are commonly referenced to develop a meaningful evaluation of fuzzers that produces trustworthy results. We note, however, that the study by Klees et al. [13] that produced these guidelines used fuzzers that had the aim of making contributions to improving the state-of-the-art of bug-finding abilities, whereas our work aims to make contributions to the system performance of binary-only fuzzing. Therefore we carefully select the experiment setup guidelines of Klees et al. [13] that is most relevant to our work.

Our work is focused on determining the performance implications of using the LBR/BTS tracing mechanisms in coverage-guided fuzzers per run of the target application. Therefore for our experiments, we justify that a single fuzzing campaign of 24 hours for each of our four benchmark programs is sufficient testing time for our performance-based evaluations. Other works [56, 27, 67] that aim to improve the system performance of fuzzers have also used a similar amount of testing time. We believe the number of executions performed in a 24-hour window, sufficient enough to get an understanding of the system performance characteristics of each fuzzer.

We also note that we do not use Address Sanitizers (ASAN and UBSAN) [68] in the compilation of any benchmark programs. ASAN and UBSAN add dynamic checks for various errors including memory errors [68] [13]. Since we are not triaging crashes and evaluating the bug-finding capabilities of each fuzzer, we regard the use of address sanitizers as out of the scope of our work.

6.2 Results

As mentioned at the beginning of this chapter, our evaluation has two categories: source-code available fuzzing and binary-only fuzzing. Our experimental results for source-code available fuzzing and binary-only fuzzing aim to answer the following research questions:

- **RQ1:** What is the performance overhead in using Intel LBR and BTS as sources of coverage information in coverage-guided fuzzers?
- **RQ2:** Can using Intel BTS and LBR as coverage tracing mechanisms in fuzzing improve binary-only fuzzing?

6.2.1 Source-Code Available Fuzzing

The results for source-code available fuzzing are presented in Table 6.2, Figure 6.1, Table 6.3, and Table 6.4. The evaluation metric *executions per second* in Table 6.2 and its graphical representation in Figure 6.1 will be used to address research question **RQ1**. We can see the *executions per second* in Table 6.2 and its graphical representation in Figure 6.1, where more is better, that both LBR-Fuzz and BTS-Fuzz on average had executed less fuzzing runs than the baseline fuzzer AFL-Clang-Fast for all target applications. BTS-Fuzz had the lowest number of executions per second, with overheads greater than 95% when compared with AFL-Clang-Fast for mp3gain. For some targets, LBR-Fuzz had comparable speeds to AFL-Clang-Fast, with overheads of 5% and 29% for mp3gain and objdump respectively. We can see in Figure 6.1 that LBR-Fuzz was generally faster than

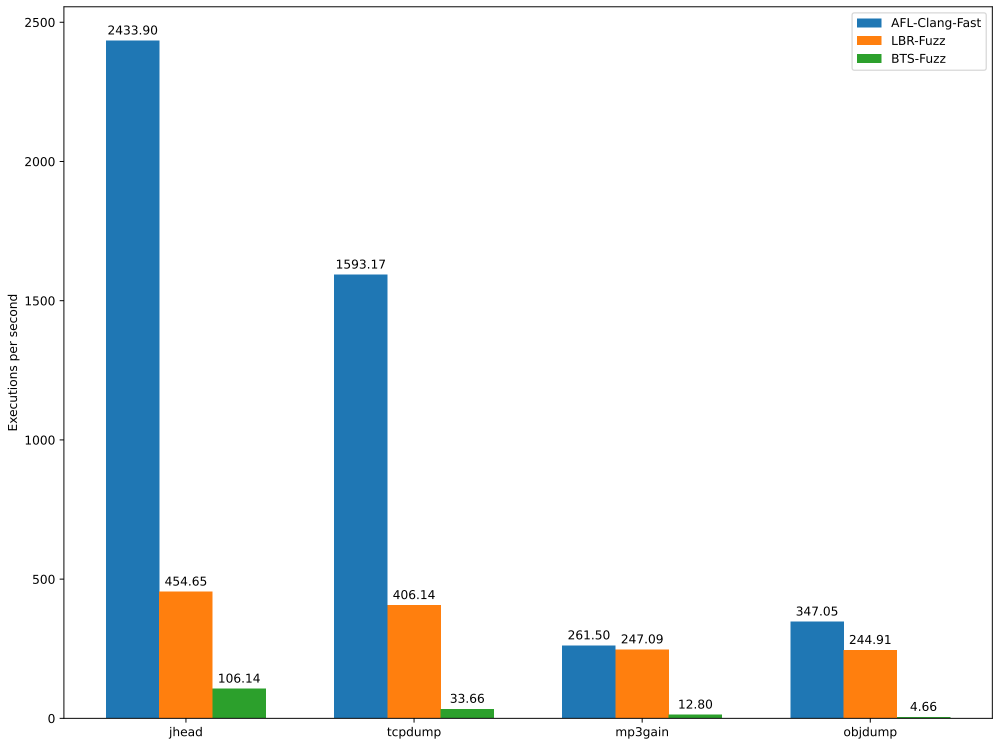

Figure 6.1: Performance of Fuzzers for Source-Code Available Fuzzing

BTS-Fuzz by 4.3x, 12x, 19.3x, and 53x speed for jhead, tcpdump, mp3gain, and objdump respectively.

Table 6.2: Fuzzing efficiency and performance for source-code available fuzzing

	total paths discovered			**executions per second**		
Targets	AFL-Clang-Fast	LBR-Fuzz	BTS-Fuzz	AFL-Clang-Fast	LBR-Fuzz	BTS-Fuzz
jhead	228	86	261	2433.90	454.65	106.14
tcpdump	2088	235	939	1593.17	406.14	33.66
mp3gain	754	116	420	261.50	247.09	12.80
objdump	2516	70	827	347.05	244.91	4.66

The results presented in the *total paths discovered* column in Table 6.2, the *branch coverage* percentages in Table 6.3 and the *stability* ratings column in Table 6.4 provide an insight into how efficient the executions were and how reliable was the coverage feedback for each fuzzer during the 24 hour source-code available fuzzing campaign. While LBR-Fuzz was observed to be faster than BTS-Fuzz as in terms of executions per second, LBR-Fuzz on average had discovered fewer paths during the 24-hour window when compared with BTS-Fuzz and AFL-Clang-Fast for all targets as seen in the *total paths discovered* column in Table 6.2. For instance, in Table 6.2 we can see LBR-Fuzz found 97% fewer paths than AFL-Clang-Fast for objdump. Additionally, LBR-Fuzz also reported the lowest branch coverage percentages as seen in Table 6.3. LBR-Fuzz also had the lowest stability rating for all targets with stability ranging from 57% to as low as 16.57% as seen in Table 6.4. On the other hand, BTS-Fuzz had discovered significantly more paths than LBR-Fuzz by on average 3x more as seen with jhead, tcpdump, and mp3gain in Table 6.2. Additionally, the stability ratings for BTS-Fuzz were close to 100% for all targets as seen in Table 6.4. Interestingly though despite discovering more paths, BTS-Fuzz only had slightly higher branch coverage percentages than LBR-Fuzz as in Table 6.3. In terms of fuzzing effectiveness, BTS-Fuzz found more unique crashing inputs compared to LBR-Fuzz. On the other hand, AFL-Clang-Fast found significantly more crashing inputs than LBR-Fuzz and BTS-Fuzz.

Table 6.3: Branch coverage percentages for source-code available fuzzing

Branch Coverage %	AFL-Clang-Fast	LBR-Fuzz	BTS-Fuzz
jhead	14.93%	14.42%	14.79%
tcpdump	16.16%	5.62%	9.51%
mp3gain	36.38%	20.32%	22.55%
objdump	4.38%	3.37%	3.87%

Table 6.4: Fuzzing reliability and number of unique crashes for source-code available fuzzing

	Stability %			**number of unique crashing inputs**		
Targets	AFL-Clang-Fast	LBR-Fuzz	BTS-Fuzz	AFL-Clang-Fast	LBR-Fuzz	BTS-Fuzz
jhead	100.00%	27.69%	98.56%	0	0	0
tcpdump	100.00%	34.84%	100.00%	0	0	0
mp3gain	100.00%	57.44%	100.00%	59	1	17
objdump	100.00%	16.57%	95.54%	32	1	9

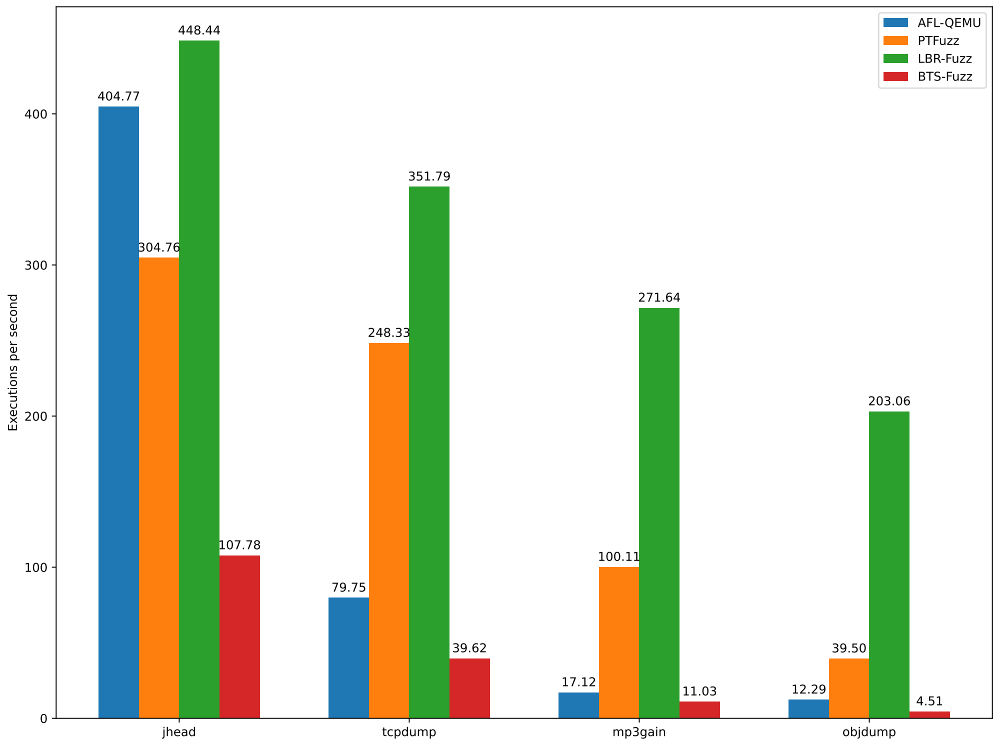

Figure 6.2: Performance of Fuzzers for Binary-Only Fuzzing

6.2.2 Binary-Only Fuzzing

The results for binary-only fuzzing are presented in Table 6.5, Figure 6.2, Table 6.6, and Table 6.7. To help address research question **RQ2**, the *executions per second* column in Table 6.5 along with its graphical representation in Figure 6.2 is used to determine the performance overhead in using LBR-Fuzz and BTS-Fuzz over the state of the art in binary-only fuzzing AFL-QEMU. Additionally, we compare the performance overhead in using LBR-Fuzz and BTS-Fuzz with the performance overhead in using an alternative binary-only fuzzer, PTfuzz, that uses a different hardware tracing technology namely Intel PT.

Table 6.5: Fuzzing efficiency and performance for binary-only fuzzing

	total paths discovered				**executions per second**			
Targets	AFL-QEMU	PTfuzz	LBR-Fuzz	BTS-Fuzz	AFL-QEMU	PTfuzz	LBR-Fuzz	BTS-Fuzz
jhead	325	237	81	209	404.77	304.76	448.44	107.78
tcpdump	1283	1869	26	16	79.75	248.33	351.79	39.62
mp3gain	596	864	123	374	17.12	100.11	271.64	11.03
binutils	1440	2088	83	128	12.29	39.50	203.06	4.51

We observe that the binary-only fuzzing results of *executions per second* in Table 6.5 for both LBR-Fuzz and BTS-Fuzz were very similar from the source-code available fuzzing results in Table 6.2, with BTS-Fuzz varying the least. In other words, the fuzzing speed of LBR-Fuzz and BTS-Fuzz from binary-only fuzzing is nearly identical to that of source-code available fuzzing. As expected, using Table 6.2 and Table 6.5 we see that AFL-QEMU was significantly slower than AFL-Clang-Fast. When comparing BTS-Fuzz with AFL-QEMU according to Table 6.5 we see that BTS-Fuzz had fewer executions per second than AFL-QEMU for all targets. The performance overhead of BTS-Fuzz, when compared to AFL-QEMU, ranged from 35% to 73%. On the other hand, LBR-Fuzz had more executions per second than AFL-QEMU for all targets. In Figure 6.2 we see that for tcpdump, mp3gain, and objdump LBR-Fuzz was 4.4x, 15.9x, and 16.5x times faster than AFL-QEMU. In Figure 6.2 we can also see that for tcpdump, mp3gain, and objdump, PTfuzz was 3.1x, 5.8x, and 3.2x faster than AFL-QEMU. On the other hand, LBR-Fuzz was faster than PTfuzz for all fuzzing targets, ranging from 1.4x for tcpdump and 5.1x for objdump.

Table 6.6: Branch coverage percentages for binary-only fuzzing

Branch Coverage %	AFL-QEMU	PTfuzz	LBR-Fuzz	BTS-Fuzz
jhead	14.79%	14.86%	14.49%	14.86%
tcpdump	10.74%	12.42%	3.60%	3.66%
mp3gain	37.90%	43.87%	20.32%	22.07%
binutils	3.99%	4.11%	3.51%	3.81%

Table 6.7: Fuzzing reliability and number of unique crashes for binary-only fuzzing

	Stability %				**unique number of crashing inputs**			
Targets	AFL-QEMU	PTfuzz	LBR-Fuzz	BTS-Fuzz	AFL-QEMU	PTfuzz	LBR-Fuzz	BTS-Fuzz
jhead	100.00%	91.60%	33.96%	99.61%	0	0	0	0
tcpdump	100.00%	95.66%	30.30%	99.01%	0	0	0	0
mp3gain	100.00%	91.17%	50.00%	100.00%	20	98	1	15
binutils	100.00%	92.99%	17.89%	98.32%	16	10	0	0

To supplement our response to research question **RQ2**, we use the results presented in the *total paths discovered* column of Table 6.5, the *branch coverage* percentages in

Table 6.6, and the stability ratings in Table 6.7 to determine the fuzzing efficiency and reliability for each fuzzer during the 24-hour window binary-only fuzzing campaign. While LBR-Fuzz was observed to be faster than BTS-Fuzz, AFL-QEMU, and PTfuzz, similar to the results from source-code available fuzzing, LBR-Fuzz had discovered the least amount of paths for all targets. Additionally, while BTS-Fuzz generally discovered more paths than LBR-Fuzz, BTS-Fuzz however discovered fewer paths than AFL-QEMU and PTfuzz. BTS-Fuzz had a large variation in the number of paths discovered when compared to AFL-QEMU for different targets. For jhead and mp3gain, BTS-Fuzz discovered roughly 36-37% fewer paths than AFL-QEMU. While for tcpdump and objdump, BTS-Fuzz discovered 91-99% fewer paths than AFL-QEMU. In terms of branch coverage, the binary-only fuzzing results of LBR-Fuzz and BTS-Fuzz are similar to that of the source-code available fuzzing results. Additionally, in terms of branch coverage, AFL-QEMU and PTfuzz are roughly similar and have high branch coverage percentages than LBR-Fuzz and BTS-Fuzz. In terms of stability, AFL-QEMU had 100% stability ratings across all targets. PTfuzz had unexpectedly lower stability ratings than AFL-QEMU and BTS-Fuzz that ranged from 91.2% to 95.7%. Whereas BTS-Fuzz had nearly 100% stability across all targets. On the other hand, LBR-Fuzz has just as low stability ratings as it had for source-code available fuzzing. Lastly, in terms of fuzzing effectiveness AFL-QEMU and PTfuzz had found more unique crashing inputs than LBR-Fuzz and BTS-Fuzz. For both binary-only fuzzing and source-code available fuzzing all the unique crashing inputs were only found for mp3gain and objdump.

6.3 Discussion

6.3.1 Performance Overhead in Using LBR and BTS for Coverage-Guided Fuzzing

In subsection 6.2.1 we saw in source-code available fuzzing BTS-Fuzz and LBR-Fuzz have prohibitively high overheads when compared with the state-of-the-art coverage-guided white-box fuzzer AFL-Clang-Fast and therefore should not be recommended for source-code available fuzzing. However, because BTS-Fuzz and LBR-Fuzz solely rely on hardware mechanisms and are source-code agnostic, the relative performance overheads of BTS-Fuzz and LBR-Fuzz in binary-only fuzzing were nearly identical to that of source-code available fuzzing. While the performance overhead of BTS-Fuzz was still prohibitively high when compared with the state-of-the-art in binary-only fuzzing AFL-QEMU and with PTfuzz, LBR-Fuzz had the lowest performance overhead among all fuzzers under evaluation for binary-only fuzzing. LBR-Fuzz was at least 4x up to 15x faster than the state-of-the-art binary-only fuzzer AFL-QEMU. A reason for BTS being slower than LBR is that in the case of BTS, for every taken branch its corresponding branch record is sent over to the system bus to be stored in the memory-resident (cache-as-RAM or system DRAM) BTS buffer whereas for LBR branch records are directly stored in model specific registers [39]. As shown by the results, the large number of cache or system memory stores by BTS dominate the performance of an application. Additionally, we found that in the hardware vendor's developer forums archives it is mentioned that in order for BTS to maintain the correct branch order in the memory-resident buffer, on every taken branch BTS clears the instruction pipeline [69]. While we anticipated LBR-Fuzz to be faster than BTS-Fuzz, we wanted to verify how much faster it would be because of the lack of cited studies in the fuzzing community regarding the usage of these tracing mechanisms. We observed LBR-Fuzz to be faster than BTS-Fuzz by at least 4x and by at most 53x speed.

6.3.2 Using BTS and LBR in Binary-Only Fuzzing

While LBR-Fuzz had the most executions per second than the state-of-the-art in binary-only fuzzing AFL-QEMU and also PTfuzz, in terms of fuzzing efficiency LBR-Fuzz does not seem to be efficient because LBR-Fuzz also discovered the least amount of execution paths.

One reason why the number of paths discovered by LBR-Fuzz was low is because LBR-Fuzz had low stability ratings. The low stability ratings tell us that there were numerous times where LBR-Fuzz observed inconsistent coverage traces with identical inputs and therefore the AFL fuzzing engine would struggle to discern between meaningful input mutations which legitimately increase code coverage and mutations which do not [70]. Throughout the implementation phase, we struggled with improving the stability rating for LBR-Fuzz. We ultimately found that the stability for LBR-Fuzz is mostly influenced by skid. In chapter 7 we discuss in detail the impact skid had in our fuzzing campaigns for LBR-Fuzz and how it is a threat to the validity of our fuzzing implementation. As we improved our detection and handling of skid, the stability rating of LBR-Fuzz improved. However, we later learned even when we finally achieved a stability rating of over 90% upon the start of a fuzzing job, over the duration of fuzzing an application the stability rating of LBR-Fuzz would always steadily drop. The reason for this is that although our fuzzing infrastructure can handle a generous amount of skid without impacting performance, there is a threshold we support and any skid beyond our threshold leads to variable behavior with identical inputs. As seen in our fuzzing results, when skid is taken out of the equation as is the case with our other hardware tracing-based fuzzing implementation, BTS-Fuzz, which does not experience any skid, detrimental drops in the stability rating do not occur.

Another reason contributing to the lower number of paths discovered by LBR-Fuzz is the lack of visibility of fall-through branches. The fuzzing engine AFL can detect new executions paths taken. For example executing a fall-through branch even if a particular execution path does not hit a new edge but whose sequence of edges hit has not been seen before. However, such an execution path would not be considered by AFL as increasing code coverage and therefore AFL would not further explore the input space of this newly discovered execution path. The consequence of lacking visibility of fall-through branches is that it can potentially limit the discovery of bugs if a bug lies some instructions ahead of where a newly discovered execution path containing a fall-through branch last left off. Bug discovery can be further hindered if a fall-through branch leads to code that does not contain the types of branches (edges) the coverage tracing tool is tracing for. In our case, since we have restricted LBR-Fuzz to only track conditional branches for the sake of performance, if a fall-through branch does not contain a conditional branch before suddenly exiting, the AFL engine of LBR-Fuzz would not further explore the input space that lead to this fall-through branch. The reason for this is because LBR-Fuzz would report there was no increase in code coverage. In contrast, BTS-Fuzz monitors all kinds of branches and hence is more likely to capture coverage information of fall-through branches, not specifically as a fall-through branch, but as an execution path that increases code coverage if it has discovered any new edges. PTfuzz and any Intel-PT based fuzzer can directly detect fall-through branches even if no new edge was discovered during runtime thanks to its ability to generate TNT packets to capture not taken branches.

To potentially increase the number of paths discovered by LBR-Fuzz, one might want to increase the types of branches monitored by LBR-Fuzz. This would not only lead to more complete coverage information but also increase the likelihood of capturing the coverage information of fall-through branches than with just conditional branches alone. As common with most hardware tracing mechanisms, increasing the types of branches to monitor for could impact performance. So one must be carefully selective of the kind of branch information to monitor for. After conditional branch instructions, indirect branch and indirect call instructions are also useful types of branch information to monitor for in fuzzing because machine code can contain jumps that are conditionally based on the dynamic behavior of a program as is the case when there are function pointers or virtual functions in the source code. In Figure 6.3, we see that depending on the dynamic result of line 18, we could either jump to a function containing a bug or another function that leads to more code. In this case, by also tracking for indirect branches and indirect calls, our fuzzers would discover more paths and most definitely reach the function that contains a bug.

```
int foo(void){
        BUG()
        return 0;
 }

int bar(void){
        ret_val = more_code();
        return ret_val;
}

func_ptr[2];

main(){
    func_ptr[0] = &foo;
    func_ptr[1]= &bar;
    int a;
    read(0,&a,1);
    int idx = !!(a-0)
    func_ptr[idx]();
    way_more_code();
    return 0;
}
```

Figure 6.3: Dynamic program behavior determines control flow

CHAPTER 7
THREATS TO VALIDITY AND LIMITATIONS

During the development, debugging, and experimentation process of our fuzzing implementations, we learned a great deal about the capabilities and shortcomings of our fuzzing implementations and of the supporting hardware and software tools our implementations rely on.

7.0.1 Child Processes and Execve

One limitation of our fuzzing implementations is that we can not monitor the branch execution of the child processes of our target applications. This is because the perf kernel source code does not allow children to inherit counters managed by perf if the parent process already has used `mmap()` to create a memory mapping to the perf ring buffer as this would create a performance issue if all children also update the same perf ring buffer [71]. Even if perf did not have this limitation, just like our baseline fuzzer AFL-Clang-Fast, our fuzzing implementations were not designed to support target applications that fork child processes because there could be multiple forkservers spawned and there could be races conditions with multiple threads updating the AFL coverage bitmap. Additionally, our fuzzing implementations, as well as AFL-Clang-Fast, can not support fuzzing all target applications that perform an `execve()` because if the new program's binary image does not have inserted the forkserver code and coverage tracking code then coverage-guided fuzzing with the AFL engine is not possible.

7.0.2 The Effect of Skid

One threat to validity is the effect of skid in LBR sampling in LBR-Fuzz. While we were aware of the existence of skid, we did not anticipate seeing such a large amount of skid when sampling the LBR on our processor when running LBR-Fuzz, our processor is based on the Coffee Lake architecture. If there were no skid, for the sake of performance one should sample the LBR after every 32 LBR inserts and not have to worry about losing branch records. However, in our experiments with LBR-Fuzz, we observed a general maximum skid of 15, which means we had to sample the LBR at least every 32-MAXSKID=17 LBR inserts to avoid older branch records from being overwritten by new incoming branch records during the delay of the firing of the PMI. The existence of skid minimally affects BTS-Fuzz because the BTS buffer does not need to be read by a PMI interrupt so often.

7.0.3 Over-Counting in Performance Counters

Another threat to validity is the known issue of over-counting in performance counters [47, 46]. We noticed in our experiments with LBR-Fuzz, after improving our skid detection and handling capabilities that our PMI was firing a bit early due to the performance counter for LBR_INSERTS over-counting. To be precise, instead of firing at our requested rate of after every 17 LBR inserts the PMI would sometimes fire after 12 LBR inserts. In our implementation of LBR-Fuzz, we labeled this behavior as negative ”skid” and adjusted our skid detection and handling to handle a minimum negative skid of MAXSKID-32. It is important to note that unhandled negative skid only affects our ability to accurately sample the LBR without processing multiple times already seen branch records. The issue of over-counting in performance counters was observed not when running BTS-Fuzz.

7.0.4 Missing Branch Records

An anomaly we observed which could be a threat to validity is the processor not reporting to the LBR all the executed branches. We noticed this anomaly when we were investigating the low stability rating of LBR-Fuzz. In our initial investigation, we occasionally got the warning ”instrumentation output varies across runs” which meant the coverage feedback we got from sampling the LBR differed even when identical inputs were evaluated.

To further investigate the warning, we wrote a script that would run a small program with the same input repeatedly, comparing coverage profiles of consecutive runs, until a different coverage profile was detected. For our script, we used a small program with known branches instructions, a high sampling frequency, and no branch filtering to ensure the coverage profiles contained all the branch records appearing in the LBR. Consequently, we discovered that the anomaly coverage profile was always missing a branch record seen in previous runs even though it was visually factual that the anomaly coverage profile contained all the entries ever appearing in the LBR during that particular run. Furthermore, we saw that a branch record could be missing even in low skid conditions. However, this anomaly rarely happens and inconsistently occurs one out of every hundred runs sometimes even one out of every couple of thousands of runs. One possible explanation for this anomaly is that modern CPUs might execute too fast to feasibly view every executed branch [72]. We also note that the stability rating is also largely affected by positive skid that is outside the bounds of our detection.

7.0.5 Targets That Dynamically Load Glibc

One limitation of our fuzzing implementations is that when performing binary-only fuzzing we require target applications to dynamically load glibc. This is because the tool we use to insert our LBR/BTS monitoring code, e9patch, uses a parallel implementation of libc that does not have support for the libc functions we need such as `atexit` or the desired functionality of functions such as `malloc`. Instead, we use `dlopen`/`dlsym` to dynamically load glibc and get the function pointers to libc functions we need. Another potential limitation seen when performing binary-only fuzzing is that our rewritten target binaries require a large amount of virtual memory upon program startup. This is largely due to a memory optimization feature of e9patch called physical page grouping that reduces memory fragmentation and output binary size bloat [32]. However, it should be noted that while the virtual memory usage is significantly higher than the original binary, the physical memory usage is not [32]. The side effect of this feature is that when running e9patch rewritten binaries one must carefully tune the memory limit to be sufficient (the smallest possible memory limit) or else the binary will crash immediately. Additionally, running with no memory limit could run the risk of consuming all system memory.

CHAPTER 8
FUTURE WORK

As seen in section 6.2, in terms of speed, our LBR-based fuzzer had the most promising results for improving performance in binary-only fuzzing. However based on our results and investigation, using the current widely available interrupt-based sampling hardware support to sample the LBR can be prone to unforgivable amounts of skid which could lead to inconsistent coverage traces. One area of improvement for future work for any LBR-based fuzzer is to reduce skid and further reduce performance overhead from storing/retrieving coverage information. At the time of this writing, there are three future architecture features on the horizon that can potentially reduce the impact of skid and or reduce the performance overhead. The Linux Kernel has already accepted patches that would support these future features inside of the Linux perf subsystem [73], [74] [75].

8.1 Extended PEBS and Adaptive PEBS

Extended Processor Event-Based Sampling (Extended PEBS) and Adaptive PEBS will enhance the legacy Intel PEBS facility by respectively allowing any [76] performance monitoring event to trigger PEBS record upon counter overflows and the ability to include a snapshot of the LBR in a PEBS record [77]. Extended PEBS introduces the opportunity to use the performance monitoring event LBR_INSERTS to trigger a PEBS record upon counter overflows. With Adaptive PEBS the PEBS hardware can be further customized to solely collect LBR snapshots. According to the supporting Linux Kernel patches [73], adaptive PEBS could mean faster sampling of the LBR as a result of performance overhead reduction by avoiding the need to fire an expensive PMI interrupt for every LBR snapshot [73]. This could reduce the impact of skid as skid would mostly be due to the delay in firing microcode[78] rather than a PMI interrupt [42]. Microcode collects a snapshot of LBR and stores it in a buffer in virtual memory. A PMI interrupt is issued only when the buffer is almost full. Extended PEBS and Adaptive PEBS debuts on Ice Lake microarchitectures [77].

8.2 Architectural LBRs with XSAVE Support

With the Architectural LBR feature the LBR ring buffer will now reside within processor state registers as opposed to model specific registers [79]. This new feature also comes with XSAVE support. The XSAVE instruction set was originally designed to allow fast saving and restoring of processor state information upon context switches [80], the state of the Architectural LBR will now also be saved and restored [79]. According to supporting Linux Kernel patches, the Linux Kernel is planning to use the XSAVE support to also perform a faster general reading of the LBR ring buffer when the previously mentioned new PEBS facility isn't being used to sample the LBR [74]. This is said to lower the overhead of the PMI interrupt because using the XSAVE instruction to read the LBR entries all at once is said to be faster [75] than using the rdmsr instruction to individually read the LBR entires, especially since modern processors come with a large number of LBR entries. Architectural LBRs with XSAVE support will debut on Adler Lake microarchitectures [79].

CHAPTER 9
CONCLUSION

Motivated by the lack of evaluations in the usage of hardware tracing mechanisms in the fuzzing literature, we have provided an evaluation of the hardware tracing mechanisms Intel LBR and Intel BTS which are resident in the most commonly used processor platforms in the fuzzing community. The aim of our work was to evaluate the performance impact of using these hardware tracing mechanisms in fuzzing and to see if these hardware tracing mechanisms can improve performance in binary-only fuzzing.

We have implemented two fuzzers (LBR-Fuzz and BTS-Fuzz) based on the state-of-the-art coverage-guided fuzzer AFL and integrated with these respective hardware tracing mechanisms using the Linux perf API to provide coverage feedback to the AFL fuzzing engine. In our evaluations, we have compared our fuzzers to AFL-Clang-Fast to determine the general performance overhead in using these hardware tracing mechanisms as sources of coverage information in coverage-guided fuzzers. We have also compared our fuzzers to the state-of-the-art binary-only fuzzing AFL-QEMU and another existing hardware tracing based fuzzer, PTfuzzz, to evaluate its potential in improving performance in binary-only fuzzing.

Our evaluations prove that while Intel BTS can provide comprehensive and accurate coverage information, its performance overhead is too high to be used effectively in binary-only fuzzing when there are better performance-friendly options currently available to gather coverage information. Despite Intel BTS and Intel LBR using the same technology to dynamically capture coverage information (the LBR ring buffer), our evaluations showed that their differences in storing and retrieving this coverage information lend them to having vastly different performance results. In terms of speed, our Intel LBR based fuzzer was able to outperform AFL-QEMU and PT-Fuzz. However, our evaluations showed, due to its retrieval methods, Intel LBR struggled to produce consistent coverage traces which significantly impacted AFL's ability to discover as many meaningful execution paths as possible and hence limit bug discovery. On the horizon, there are future storage and retrieval architecture enhancements that when paired with Intel LBR would not only further reduce performance overhead but also potentially reduce inconsistencies in coverage traces for consuming software such as fuzzers.

REFERENCES

[1] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution.," in *NDSS*, vol. 16, 2016, pp. 1–16.

[2] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing.," in *NDSS*, vol. 17, 2017, pp. 1–14.

[3] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 711–725.

[4] AFL, *American fuzzing lop (afl)*, https://lcamtuf.coredump.cx/afl/, 2020.

[5] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *2016 IEEE Cybersecurity Development (SecDev)*, IEEE, 2016, pp. 157–157.

[6] R. Swiecki, "Honggfuzz," *Available online a t: http://code. google. com/p/honggfuzz*, 2016, Accessed: 2021-07-18.

[7] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 1497–1511.

[8] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "{mopt} : Optimized mutation scheduling for fuzzers," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1949–1966.

[9] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{qsym}: A practical concolic execution engine tailored for hybrid fuzzing," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 745–761.

[10] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang, "Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting," 2021.

[11] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, *et al.*, "Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers," in *30th USENIX Security Symposium (USENIX Security 21). USENIX Association*, 2021.

[12] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[13] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[14] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence.," in *NDSS*, vol. 19, 2019, pp. 1–15.

[15] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "Ptfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, vol. 6, pp. 37 302–37 313, 2018.

[16] 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 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 633–645.

[17] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "Kafl: Hardware-assisted feedback fuzzing for OS kernels," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 167–182.

[18] T. Blazytko, M. Bishop, C. Aschermann, J. Cappos, M. Schlögel, N. Korshun, A. Abbasi, M. Schweighauser, S. Schinzel, S. Schumilo, *et al.*, "GRIMOIRE: Synthesizing structure while fuzzing," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1985–2002.

[19] I. Fratric, *Winafl: A fork of afl for fuzzing windows binaries*, 2017.

[20] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[21] 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 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2214–2228.

[22] Intel, "Intel 64 and ia-32 architectures software developer's manual, volume 3b: System programming guide, chapter 35 intel processor trace," vol. 3B, 2020.

[23] S. Canakci, L. Delshadtehrani, B. Zhou, A. Joshi, and M. Egele, "Efficient context-sensitive cfi enforcement through a hardware monitor," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2020, pp. 259–279.

[24] *Linux kernel perf architecture*, https://terenceli.github.io/%E6%8A%80%E6%9C%AF/2020/08/29/perf-arch, Accessed: 2021-07-18, 2020.

[25] V. Weaver, *Perf event open linux manpage*, https://www.man7.org/linux/man-pages/man2/perf_event_open.2.html, Accessed: 2021-07-18, 2015.

[26] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding bugs in file systems with an extensible fuzzing framework," *ACM Transactions on Storage (TOS)*, vol. 16, no. 2, pp. 1–35, 2020.

[27] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 787–802.

[28] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 697–710.

[29] R. N. Horspool and N. Marovac, "An approach to the problem of detranslation of computer programs," *The Computer Journal*, vol. 23, no. 3, pp. 223–229, 1980.

[30] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, IEEE, 2010, pp. 175–183.

[31] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic binary recompilation," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 133–147.

[32] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 151–163.

[33] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again.," in *NDSS*, 2017.

[34] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[35] AMD, "Amd64 architecture programmer's manual, volume 2: System programming, chapter 13 software debug and performance resources," vol. 2, 2021.

[36] 216, *Automatic feedback directed optimizer*, https://gcc.gnu.org/wiki/AutoFDO, Accessed: 2021-12-12, 2012.

[37] Intel, “Intel 64 and ia-32 architectures software developer’s manual, volume 3b: System programming guide, chapter 18 performance monitoring,” vol. 3B, 2020.

[38] ARM, “Arm cortex-a series: Programmer’s guide for armv8-a, chapter 18.2.1 coresight,” version 1.0, 2015.

[39] Intel, “Intel 64 and ia-32 architectures software developer’s manual, volume 3b: System programming guide, chapter 17.4.5 branch trace store (bts),” vol. 3B, 2020.

[40] K. R. Walcott-Justice, “Testing in resource-constrained environment,” Ph.D. dissertation, University of Virginia, 2012.

[41] Intel, “Intel 64 and ia-32 architectures software developer’s manual, volume 3b: System programming guide, chapter 17.4 last branch, interrupt, and exception recording overview,” vol. 3B, 2020.

[42] D. Bakhvalov, *Advanced profiling topics, pebs and lbr*, https://easyperf.net/blog/2018/06/08/Advanced-profiling-topics-PEBS-and-LBR, Accessed: 2022-01-01.

[43] K. R. Walcott-Justice, “Chapter 2 - exploiting hardware monitoring in software engineering,” in *Advances in Computers*, A. Memon, Ed., vol. 93, Elsevier, 2014, pp. 53–101.

[44] B. Strong Sr, “Debug and fine-grain profiling with intel processor trace,” *Intel IDF14, San Francisco*, 2014.

[45] Intel, “Intel 64 and ia-32 architectures software developer’s manual, volume 3b: System programming guide, chapter 17.4.9.2 setting up the ds save area,” vol. 3B, 2020.

[46] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, “Sok: The challenges, pitfalls, and perils of using hardware performance counters for security,” in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 20–38.

[47] V. M. Weaver, D. Terpstra, and S. Moore, “Non-determinism and overcount on modern hardware performance counter implementations,” in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2013, pp. 215–224.

[48] S. Eranian, “The perfmon2 interface specification,” *HP Labs Technical Report, HPL-2004-200R1*, 2005.

[49] M. L. Soffa, K. R. Walcott, and J. Mars, “Exploiting hardware advances for software testing and debugging (nier track),” in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 888–891.

[50] J. Reinders, *Processor trace*, https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html, Accessed: 2021-07-18, 2013.

[51] A. Kleen and B. Strong, "Intel processor trace on linux," *Tracing Summit*, vol. 2015, 2015, Accessed: 2021-07-18.

[52] K. Walcott-Justice, J. Mars, and M. L. Soffa, "Theme: A system for testing by hardware monitoring events," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 12–22.

[53] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, *Nyx-fuzz: Test data for libxdc*, https://github.com/nyx-fuzz/libxdc_experiments, Accessed: 2021-12-28, 2020.

[54] L. Delshadtehrani, S. Canakci, B. Zhou, S. Eldridge, A. Joshi, and M. Egele, "Phmon: A programmable hardware monitor and its security use cases," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 807–824.

[55] R. Ding, "Performant software hardening under hardware support," Ph.D. dissertation, Georgia Institute of Technology, 2021.

[56] X. Gao, G. J. Duck, and A. Roychoudhury, "Scalable fuzzing of program binaries with e9afl," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 1247–1251.

[57] M. Kerrisk, *Fork linux manpage*, https://www.man7.org/linux/man-pages/man2/fork.2.html, Accessed: 2021-07-18, 2015.

[58] G. J. Duck, *E9patch*, https://github.com/GJDuck/e9patch, 2020.

[59] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, 2020.

[60] J. Bundt, A. Fasano, B. Dolan-Gavitt, W. Robertson, and T. Leek, "Evaluating synthetic bugs," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 716–730.

[61] R. Natella and V.-T. Pham, "Profuzzbench: A benchmark for stateful protocol fuzzing," *arXiv preprint arXiv:2101.05102*, 2021.

[62] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: Combining incremental steps of fuzzing research," in *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*, 2020.

[63] Unifuzz, *Unifuzz*, https://github.com/unifuzz, 2021.

[64] A. Herrera, H. Gunadi, L. Hayes, S. Magrath, F. Friedlander, M. Sebastian, M. Norrish, and A. L. Hosking, "Corpus distillation for effective fuzzing: A comparative evaluation," *arXiv e-prints*, arXiv–1905, 2019.

[65] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 861–875.

[66] M. Rash and M. Heuse, *Afl-cov: Afl fuzzing code coverage*, https://github.com/vanhauser-thc/afl-cov, Accessed: 2021-07-18, 2015.

[67] C.-C. Hsu, C.-Y. Wu, H.-C. Hsiao, and S.-K. Huang, "Instrim: Lightweight instrumentation for coverage-guided fuzzing," in *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[68] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address sanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[69] *Intel developer software forums-software archive: Branch trace store performance*, https://community.intel.com/t5/Software-Archive/Branch-Trace-Store-performance/m-p/995654?profile.language=en, Accessed: 2022-01-18, 2005.

[70] AFL, *American fuzzing lop (afl) documentation: Understanding the status screen*, https://github.com/google/AFL/blob/master/docs/status_screen.txt, 2020.

[71] T. Gleixner, I. Molnar, P. Zijlstra, and P. Mackerras, *Performance events core code*, https://elixir.bootlin.com/linux/latest/source/kernel/events/core.c#L6182, Accessed: 2022-01-01.

[72] A. kleen, *An introduction to last branch records*, https://lwn.net/Articles/680985/, Accessed: 2022-01-01.

[73] K. Liang, *[patch v5 05/12] perf/x86/intel: Support adaptive pebsv4*, https://lore.kernel.org/lkml/20190402194509.2832-6-kan.liang@linux.intel.com/, Accessed: 2021-07-18.

[74] ——, *[patch 00/21] support architectural lbr*, https://lore.kernel.org/all/3492fcad-344d-174e-7e38-46f2e543b065@linux.intel.com/T/, Accessed: 2021-07-18.

[75] ——, *[patch v2 23/23] perf/x86/intel/lbr: Support xsaves for arch lbr read*, https://lore.kernel.org/lkml/1593195620-116988-24-git-send-email-kan.liang@linux.intel.com/, Accessed: 2021-07-18.

[76] Intel, “Intel 64 and ia-32 architectures software developer’s manual, volume 3b: System programming guide, chapter 18.9.1 extended pebs,” vol. 3B, 2020.

[77] ——, “Intel 64 and ia-32 architectures software developer’s manual, volume 3b: System programming guide, chapter 18.9.2 adaptive pebs,” vol. 3B, 2020.

[78] S. Eranian. (Nov. 17, 2019). “Hardware performance monitoring landscape,” The International Coference for High Performance Computing, Networking, Storage and Analysis, (visited on 03/04/2021).

[79] Intel, “Intel architecture instruction set extensions and future features,” 2021.

[80] M. Gorny, *How debuggers work: Getting and setting x86 registers, part 2: Xsave*, https://www.moritz.systems/blog/how-debuggers-work-getting-and-setting-x86-registers-part-2/, Accessed: 2021-07-18, 2020.