Hardware Support to Improve Fuzzing Performance and Precision

Ren Ding*
Georgia Institute of Technology
rding@gatech.edu

Yonghae Kim*
Georgia Institute of Technology
yonghae@gatech.edu

Fan Sang
Georgia Institute of Technology
fsang@gatech.edu

Wen Xu
Georgia Institute of Technology
wen.xu@gatech.edu

Gururaj Saileshwar
Georgia Institute of Technology
gururaj.s@gatech.edu

Taesoo Kim
Georgia Institute of Technology
taesoo@gatech.edu

ABSTRACT

Coverage-guided fuzzing is considered one of the most efficient bug-finding techniques, given its number of bugs reported. However, coverage tracing provided by existing software-based approaches, such as source instrumentation and dynamic binary translation, can incur large overhead. Hindered by the significantly lowered execution speed, it also becomes less beneficial to improve coverage feedback by incorporating additional execution states.

In this paper, we propose SNAP, a customized hardware platform that implements hardware primitives to enhance the performance and precision of coverage-guided fuzzing. By sitting at the bottom of the computer stack, SNAP leverages the existing CPU pipeline and micro-architectural features to provide coverage tracing and rich execution semantics with near-zero cost regardless of source code availability. Prototyped as a synthesized RISC-V BOOM processor on FPGA, SNAP incurs a barely 3.1% tracing overhead on the SPEC benchmarks while achieving a 228× higher fuzzing throughput than the existing software-based solution. Posing only a 4.8% area and 6.5% power overhead, SNAP is highly practical and can be adopted by existing CPU architectures with minimal changes.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; **Domain-specific security and privacy architectures**.

KEYWORDS

Hardware-assisted fuzzing; Feedback-driven fuzzing; RISC-V BOOM

ACM Reference Format:
Ren Ding*, Yonghae Kim*, Fan Sang, Wen Xu, Gururaj Saileshwar, and Taesoo Kim. 2021. Hardware Support to Improve Fuzzing Performance and Precision. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3460120.3484573

*Both authors contributed equally to this work.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484573

1 INTRODUCTION

Historically, bugs have been companions of software development due to the limitations of human programmers. Those bugs can lead to unexpected outcomes ranging from simple crashes, which render programs unusable, to exploitation toolchains, which grant attackers partial or complete control of user devices. As modern software evolves and becomes more complex, a manual search for such unintentionally introduced bugs becomes unscalable. Various automated software-testing techniques have thus emerged to help find bugs efficiently and accurately, one of which is *fuzzing*. Fuzzing in its essence works by continuously feeding randomly mutated inputs to a target program and watching for unexpected behavior. It stands out from other software-testing techniques in that minimal manual effort and pre-knowledge about the target program are required to initiate bug hunting. Moreover, fuzzing has proved its practicality by uncovering thousands of critical vulnerabilities in real-world applications. For example, Google's in-house fuzzing infrastructure *ClusterFuzz* [24] has found more than 25,000 bugs in Google Chrome and 22,500 bugs in over 340 open-source projects. According to the company, fuzzing has uncovered more bugs than over a decade of unit tests manually written by software developers. As more and more critical bugs are being reported, fuzzing is unarguably one of the most effective techniques to test complex, real-world programs.

An ideal fuzzer aims to execute mutated inputs that lead to bugs at a high speed. However, certain execution cycles are inevitably wasted on testing the ineffective inputs that do not approach any bug in practice. To save computing resources for inputs that are more likely to trigger bugs, state-of-the-art fuzzers are coverage-guided and favor mutation on a unique subset of inputs that reach new code regions per execution. Such an approach is based on the fact that the more parts of a program that are reached, the better the chance an unrevealed bug can be triggered. In particular, each execution of the target program is monitored for collecting runtime code coverage, which is used by the fuzzer to cherry-pick generated inputs for further mutation. For binaries with available source code, code coverage information is traced via compile-time instrumentation. For standalone binaries, such information is traced through dynamic binary instrumentation (DBI) [4, 11, 44], binary rewriting [15, 48], or hardware-assisted tracing [31, 36].

Nonetheless, coverage tracing itself incurs large overhead and slows the execution speed, making fuzzers less effective. The resulting waste of computing resources can extend further with a

continuous fuzzing service scaling up to tens of thousands of machines [24, 47]. For example, despite its popularity, AFL [61] suffers from a tracing overhead of nearly 70% due to source code instrumentation and of almost 1300% in QEMU mode for binary-only programs [48]. Source code instrumentation brings in additional instructions to maintain the original register status at each basic block, while DBI techniques require dynamic code generation, which is notoriously slow. Although optimized coverage-tracing techniques have been proposed to improve performance, especially for binary-only programs, they impose different constraints. RetroWrite [15] requires the relocation information of position-independent code (PIC) to improve performance for binary-only programs. Various existing fuzzers [22, 26, 28] utilize *Intel Processor Trace* (PT) [36], a hardware extension that collects general program execution information. Nevertheless, Intel PT is not tailored for the lightweight tracing required by fuzzing. The ad-hoc use of Intel PT in fuzzing results in non-negligible slowdown caused by extracting useful information (*e.g.*, coverage) from encoded traces, allowing a merely comparable execution speed as source instrumentation in the best effort per large-scale profiling results [13, 33]. UnTracer [48] suggests *coverage-guided tracing*, which only traces testcases incurring new code paths. However, UnTracer adopts *basic block coverage* without edge hit count and measures a less accurate program execution trace that misses information about control transfers and loops. The overhead of software-based coverage tracing is inevitable because it requires extra information not available during the original program execution. Moreover, the applicability of fuzzing heavily depends on the availability of source code, given that existing techniques commonly used for fuzzing standalone binaries are unacceptably slow and there is a need for faster alternatives.

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

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

Second, we find that the code tracing routine, including measuring edge coverage and hit count, can be integrated seamlessly into the execution pipeline of the modern CPU architecture, and a near-zero tracing overhead can be achieved without the extra operations inevitable in software-based solutions. [1] To enable such low-cost coverage tracing, SNAP incorporates two new micro-architectural units inside the CPU core: *Bitmap Update Queue (BUQ)* for generating updates to the coverage bitmap and *Last Branch Queue (LBQ)* for extracting last branch records (§4.2). SNAP further adopts two micro-architectural optimizations to limit the overhead on the memory system from frequent coverage bitmap updates: *memory request aggregation*, which minimizes the number of updates, and *opportunistic bitmap update*, which maximizes the utilization of free cache bandwidth for such updates and reduces their cost (§4.3).

Third, rich execution semantics can be extracted at the micro-architecture layer. One may think that the raw data gathered at the hardware level largely loses detailed program semantics because the CPU executes the program at the instruction granularity. Counter-intuitively, we find that such low-level information not only enables flexible coverage tracing, but also provides rich execution context for fuzzing without performance penalty. For example, various micro-architectural states are available in the processor pipeline during program execution, such as *last-executed branches* (which incur higher overhead to extract in software) and *branch predictions* (which are entirely invisible to software). Using such rich micro-architectural information, SNAP is able to provide extra execution semantics, including *immediate control-flow context* and *approximated data flows*, in addition to code coverage (§4.5). SNAP also supports setting address constraints on execution-specific micro-architectural states prior to execution, providing users the flexibility to selectively trace and test arbitrary program regions. Thus, fuzzers on SNAP can utilize the runtime feedback that describes the actual program state more precisely and make better mutation decisions. SNAP hosts clean software interfaces for adoption by the existing AFL-based fuzzers with a minimal change of less than 100 LoCs.

We prototype SNAP on top of the RISC-V BOOM core [6], which has one of the most sophisticated designs among the open-source processors. We also utilize a real hardware FPGA platform to evaluate the performance of SNAP. In particular, the tracing overhead of SNAP across the SPEC benchmarks is 3.1% on average, significantly outperforming the software-based tracing method adopted by AFL and its descendants. In addition, we fuzz a real-world collection of binary tools, Binutils v2.28 [21], with AFL assisted by SNAP. The evaluation results show that SNAP manages to achieve 228× higher fuzzing throughput compared to that of the existing DBI scheme and outperforms the vanilla AFL in discovering new paths by 15.4% thanks to the higher throughput. Furthermore, by improving coverage feedback with the rich execution semantics provided by SNAP, we demonstrate that the modified AFL running on SNAP is capable of triggering a bug that can be barely reached by the vanilla AFL. Last, our synthesized FPGA is practical, posing only a 4.8% area and 6.5% power overhead.

In summary, this paper makes the following contributions:

- We propose hardware primitives to provide transparent support of tracing and additional execution semantics for fuzzing with minimal overhead.
- We develop a prototype, SNAP, which implements the designed primitives on a real hardware architecture.
- We evaluate the system on the benefits of fuzzing performance and precision, and demonstrate its ease of adoption.

SNAP is available at https://github.com/sslab-gatech/SNAP.

2 BACKGROUND

In this section, we provide an overview of coverage-guided fuzzing. We also introduce recent efforts in the research community to improve the quality of coverage feedback. Finally, we provide a brief introduction to the typical workflow of modern processors.

[1] While PHMon [14] as a security monitor also provides hardware-based tracing, we significantly outperform it with an optimized design more customized for fuzzing. See §6 for more details.

2.1 Coverage-Guided Fuzzing

Fuzzing has recently gained wide popularity thanks to its simplicity and practicality. Fundamentally, fuzzers identify potential bugs by generating an enormous number of randomly mutated inputs, feeding them to the target program and monitoring abnormal behaviors. To save valuable computing resources for inputs that approach real bugs, modern fuzzers prioritize mutations on such inputs under the guidance of certain feedback metrics, one of which is code coverage. Coverage-guided fuzzers [26, 43, 52, 61] rely on the fact that the more program paths that are reached, the better the chance that bugs can be uncovered. Therefore, inputs that reach more code paths are often favored. Coverage guidance has proved its power by helping to discover thousands of critical bugs and has become the design standard for most recent fuzzers [26, 43, 52, 61].

The common practice of measuring the code coverage of an input is to count the number of reached *basic blocks* or *basic block edges* at runtime. To retrieve the coverage information, fuzzers either leverage software instrumentation accomplished during compile-time or use other techniques such as dynamic binary instrumentation (DBI) [4, 11, 44], binary rewriting [15, 48], or hardware-assisted tracing [31, 36] when source code is unavailable. For example, AFL instruments every conditional branch and function entry while compiling the target program and relies on QEMU assistance for standalone binaries. The collected information is then stored in a *coverage bitmap*, allowing efficient comparison across various runs. Although coverage feedback allows fuzzers to approach bugs more efficiently, coverage tracing itself incurs large overhead and slows the execution speed, making fuzzers less effective. For instance, AFL encounters a 70% performance overhead due to source code instrumentation and a daunting 1300% performance overhead in QEMU mode, making it unrealistic to fuzz large-scale binary-only programs. To unleash the true power of coverage-guided fuzzing, we aim to minimize the overhead caused by coverage tracing without any constraint. Given the sizable computing resources used by fuzzing services [24, 47], optimizing the performance allows more tests against a buggy program in finite time and renders an immediate return in the form of a substantial cost reduction.

2.2 Better Feedback in Fuzzing

Feedback in fuzzing aims to best approximate the program execution states and capture the state changes affected by certain input mutations. The more accurate the feedback is in representing the execution states, the more useful information it can provide to guide the fuzzer toward bugs. Despite the success achieved by coverage-guided fuzzing, feedback that is solely based on the edges reached by the generated inputs can still be coarse grained. Figure 1 depicts an example of a buggy `cxxfilt` code snippet that reads an alphanumeric string from *stdin* (line 17-29) before demangling its contained symbols based on the signatures (line 4-11). Specifically, `BUG` in the program (line 13) results from a mangled pattern (*i.e.*, `SLLTS`) in the input. With a seed corpus that covers all the branch transfers within the loop (line 4-11), the coverage bitmap will be saturated even with the help of edge hit count, as shown in Algorithm 1, guiding the fuzzer to blindly explore the bug without useful feedback.

To improve the quality of coverage feedback, much effort has been directed to more accurately approximate program states

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

Figure 1: An illustrative example for the runtime information gathered by SNAP. The code abstracts demangling in `cxxfilt`.

Algorithm 1: Edge encoding by AFL

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

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

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

3 $prevLoc = curLoc \gg 1$

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

with extra execution semantics. In particular, to achieve *context awareness*, some fuzzers record additional execution paths if necessary [13, 19, 26, 28], while others track data-flow information [3, 12, 18, 43, 52, 60] that helps to bypass data-driven constraints. These techniques enrich the coverage feedback and help a fuzzer approach the bugs in Figure 1 sooner; yet they can be expensive and are thus limited. For example, traditional dynamic taint analysis can under-taint external calls and cause tremendous memory overhead. Although lightweight taint analysis for fuzzing [18] tries to reduce the overhead by directly relating byte-level input mutations to branch changes without tracing the intermediate data flow, it can still incur an additional 20% slowdown in the fuzzing throughput of AFL across tested benchmarks.

2.3 Typical CPU Workflow

To motivate how hardware support can minimize the overhead of fuzzing, we explain the typical CPU workflow. When a CPU runs a program, it fetches and executes the instructions stored in memory and constantly updates its program counter (`PC`), which points to the current location being executed. A typical CPU core consists of multiple pipeline stages, such as fetch, decode, execute, and memory stages. Every instruction is processed in a specific way throughout the pipeline stages based on its type. Among various instruction types, branch and jump instructions are notable since they can change the control flow of a program and thus alter the execution order of instructions. To handle the control-flow instructions

```
1  # [Basic Block]:
2  # saving register context
3  mov    %rdx, (%rsp)
4  mov    %rcx, 0x8(%rsp)
5  mov    %rax, 0x10(%rsp)
6  # bitmap update
7  mov    $0x40a5, %rcx
8  callq  __afl_maybe_log
9  # restoring register context
10 mov    0x10(%rsp), %rax
11 mov    0x8(%rsp), %rcx
12 mov    (%rsp), %rdx
```

(a) AFL-gcc

```
1  # preparing 8 spare registers
2  push   %rbp
3  push   %r15
4  push   %r14
5  ...
6  mov    %rax, %r14
7  # [Basic Block]: bitmap update
8  movslq %fs:(%rbx), %rax
9  mov    0xc8845(%rip), %rcx
10 xor    $0xca59, %rax
11 addb   $0x1, (%rcx,%rax,1)
12 movl   $0x652c, %fs:(%rbx)
```

(b) AFL-clang

Figure 2: Source-instrumented assembly inserted at each basic block between compilers.

while achieving high performance, modern computer architectures adopt speculative execution, which allows the CPU to predict the branch target instruction and proceed with the program execution based on the prediction result instead of stalling the pipeline until a destination directed by a control-flow instruction is decided. If the branch prediction turns out to be wrong, the CPU flushes its pipeline to discard the execution on the wrong path and restores the previous architecture states. Such a design reveals that every control-flow divergence during program execution is observed and appropriately managed inside the CPU pipeline. Considering that one essential task of fuzzing is to monitor control-flow transfer and manage code-coverage information, we discuss how to view fuzzing from a hardware perspective and benefit from possible advantages available at the hardware level in §4.

3 DISSECTING AFL'S TRACING OVERHEAD

In this section, we provide a detailed examination of AFL, a state-of-the-art coverage-guided fuzzer, on its excessive coverage tracing overhead as a motivating example. Among the existing coverage-guided fuzzers, AFL [61] is the most iconic one and has inspired numerous fuzzing projects [10, 17, 45]. Despite the differences in the adopted strategies for prioritizing seeds and generating testcases, coverage-guided fuzzers mostly choose to monitor code coverage at edge granularity. In general, edge coverage is preferred over basic block coverage because of the additional semantics it embeds to represent the program space. Specifically, a piece of code will be injected at each branch location of the program to capture the control-flow transfer between a pair of basic blocks, along with coarse-grained hit counts (*i.e.*, number of times the branch is taken) for repetitive operations (*e.g.*, loops) if necessary. Algorithm 1 depicts the tracing logic adopted by AFL.

AFL injects the logic into a target program in two different ways based on the scenarios. When source code is available, AFL utilizes the compiler or the assembler to directly instrument the program. Otherwise, AFL relies on binary-related approaches such as DBI and binary rewriting. While source code instrumentation is typically preferred due to its significantly lower tracing overhead compared to binary-related approaches, previous research indicates that AFL can still suffer from almost a 70% slowdown under the tested benchmarks [48]. Table 1 shows that the situation can worsen for CPU-bound programs, with an average tracing overhead of 60% from source code instrumentation and 260% from DBI (*i.e.*, QEMU). In the worst case, DBI incurs a 5× slowdown. The tracing overhead from DBI mostly comes from the binary translation of all applicable instructions and trap handling for privileged operations. On the other hand, the overhead of source code instrumentation results from the instructions inserted at each basic block that not only profiles coverage but also maintains the original register values to ensure that the instrumented program correctly runs. Figure 2a depicts the instrumentation conducted by `afl-gcc`, which requires assembly-level rewriting. Due to the crude assembly insertion at each branch, the instructions for tracing (line 7-8) are wrapped with additional instructions for saving and restoring register context (line 3-5 and line 10-12). Figure 2b shows the same processing done by `afl-clang`, which allows compiler-level instrumentation through *intermediate representation* (IR). The number of instructions instrumented for tracing can thus be minimized (line 8-12) thanks to compiler optimizations. Nevertheless, the instructions for maintaining the register values still exist and blend into the entire workflow of the instrumented program (line 2-6). Table 1 lists the increased program sizes resulting from instrumentation by `afl-clang`, suggesting an average size increase of around 2×. The increase of program size and the runtime overhead given by `afl-gcc` can be orders of magnitude larger [62].

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

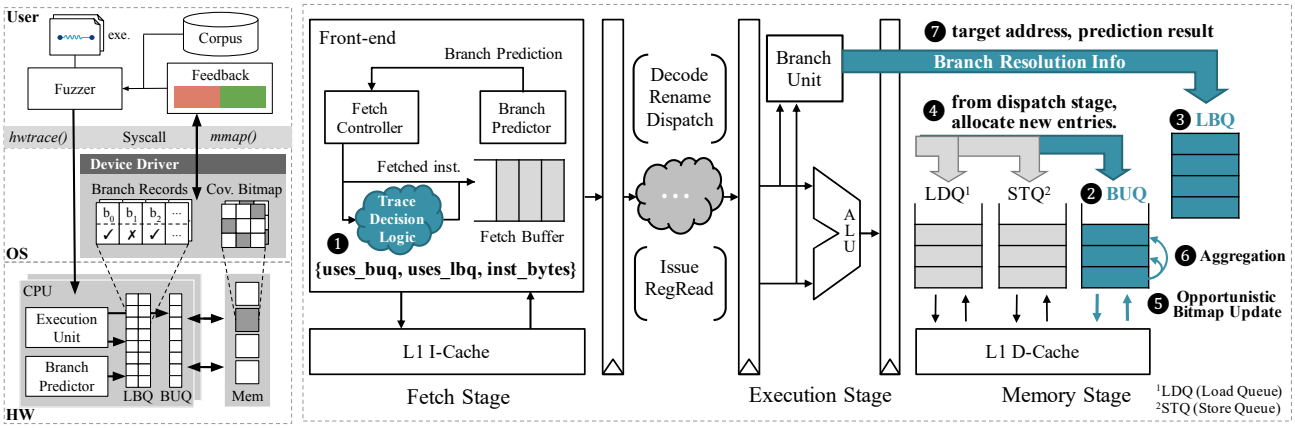
Table 1: The cost of program size and runtime overhead for tracing on an x86 platform across the SPEC benchmarks.

4 SNAP

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

① **Transparent support of fuzzing.** Existing fuzzers instrument each branch location to log the control-flow transfer, as explained in §3. When source code is not available, the fuzzer has to adopt slow alternatives (*e.g.*, Intel PT and AFL QEMU mode) to conduct coverage tracing, a much less favored scenario compared to source code instrumentation. By sitting in the hardware layer, SNAP helps fuzzers to construct coverage information directly from the processor pipeline without relying on any auxiliary added to the target program. SNAP thus enables transparent support of fuzzing any binary, including third-party libraries or legacy software, without instrumentation, making fuzzing universally applicable.

② **Efficient hardware-based tracing.** At the hardware level, many useful resources are available with low or zero cost, most of



(a) Overview of SNAP.

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

Figure 3: Overview of SNAP with its CPU design. The typical workflow involves the components from userspace, kernel, and hardware. The architecture highlights the modified pipeline stages for the desired features, including trace decision logic, Bitmap Update Queue (BUQ), and Last Branch Queue (LBQ).

which are not exposed to higher levels and cause excessive overhead to be obtained through the software stack. For example, SNAP provides the control-flow information by directly monitoring each branch instruction and the corresponding target address that has already been placed in the processor execution pipeline at runtime, eliminating the effort of generating such information that is unavailable in the original program execution from a software perspective. This allows fuzzers to avoid program size increase and significant performance overhead due to the instrumentation mentioned in Table 1. In addition, SNAP utilizes idle hardware resources, such as free cache bandwidth, to optimize fuzzing performance.

③ **Richer feedback information.** To collect richer information for better precision of coverage, many existing fuzzers require performance-intensive instrumentation. Surprisingly, we observe that the micro-architectural state information already embeds rich execution semantics that are invisible from the software stack without extra data profiling and processing. In addition to code coverage, SNAP exposes those hidden semantics to construct better feedback that can more precisely approximate program execution states without paying extra overhead. Currently, SNAP provides the records of last-executed branches and the prediction results to infer immediate control-flow context and approximated data flows. We leave the support of other micro-architectural states as future work.

Figure 3a shows an overview of SNAP in action, which includes underlying hardware primitives, OS middleware for software support, and a general fuzzer provided by the user. While running on SNAP, a fuzzer is allowed to configure the hardware and collect desired low-level information to construct input feedback through interfaces exposed by the OS. In addition, the fuzzer coached by SNAP can perform other fuzzing adjustments directly through the hardware level, such as defining code regions for dedicated testing on specific logic or functionalities. Although SNAP considers fuzzing a first-class citizen, it is designed with versatility in mind for all use cases that demand rich information about program execution states. We provide further discussion in §6.

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

Table 2: New control and status registers (CSRs) in SNAP.

4.1 Overview of Hardware Primitives

SNAP provides hardware primitives to transparently trace program execution and maintain fuzzing metadata (*e.g.*, coverage bitmap), avoiding the overhead of existing software-based techniques.[2] Figure 3b shows an overview of the proposed hardware platform, highlighting the three new primitives blended into the processor pipeline: ❶ *trace decision logic* at the *Fetch Stage* to identify and tag instructions that need to be traced (§4.2.1), ❷ *bitmap update queue* (BUQ) to manage coverage bitmap updates based on the tagged instructions passing *Execute Stage* (§4.2.2), and ❸ *last branch queue* (LBQ) to collect information about the last-executed branches from the *Branch Unit* for additional contextual feedback (§4.2.3). We elaborate on each of the hardware units in §4.2.

SNAP also introduces new *control and status registers* (CSRs) as configuration interfaces between the hardware and the OS. Table 2 contains the list of the new CSRs, each of 8 bytes in size, and their access permissions. `BitmapBase` and `BitmapSize` are used to set the base address and size of the coverage bitmap. The tracing in hardware is enabled by `TraceEn`. `TraceStartAddr` and `TraceEndAddr` serve to define region-specific feedback when needed. `LbqAddr[0-31]` and `LbqStatus` store the branch target addresses and prediction results of the last 32 branches by default. Note that these two CSRs are read-only, as the kernel is not required to modify

[2] All of our hardware primitives use information originated from the existing CPU pipeline – *e.g.*, the *Program Counter* and the raw bytes of an instruction available in the *Fetch Stage*, the branch-target address and branch-prediction results from the Branch Unit in the *Execute Stage*, etc.

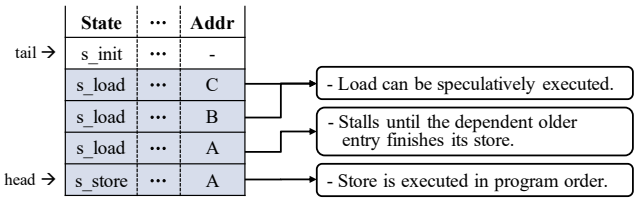


Figure 4: Bitmap update operation in the Bitmap Update Queue.

their contents. Finally, `PrevHash` stores the hash of the last branch target instruction to index the coverage bitmap, as described in §4.4. During a context switch, the OS saves and restores this value to ensure the correctness of the first bitmap update after the switch.

4.2 Implementation of Hardware Primitives

We design and implement SNAP based on the out-of-order BOOM core [6], one of the most sophisticated open-source RISC-V processors to match commercial ones with modern performance optimizations. Figure 3b highlights the lightweight modifications on key hardware components in SNAP.

4.2.1 Trace Decision Logic. In the front-end of the BOOM core (Figure 3b), instructions are fetched from the instruction cache (I-cache), enqueued to the Fetch Buffer, and then sent onward for further execution at every cycle. We extend the Fetch Controller by adding the Trace Decision Logic (❶), which determines whether an instruction inserted into the Fetch Buffer needs to be traced by SNAP. The trace decision results in tagging two types of instructions within the code region to be traced (*i.e.*, between `TraceStartAddr` and `TraceEndAddr`) using two reserved bits, `uses_buq` and `uses_lbq`. The `uses_buq` bit is used to tag the target instruction of every control-flow instruction (*i.e.*, a branch or a jump) to help enqueue bitmap update operations into the BUQ. Note that we choose to trace the target instruction instead of the control-flow instruction itself for the bitmap update due to our newly devised trace-encoding algorithm (described later in §4.4). The control-flow instruction itself is also tagged with the `uses_lbq` bit to help enqueue the corresponding branch resolution information (*i.e.*, the target address and the prediction result) into the LBQ for additional contextual semantics (described later in §4.5). Overall, the trace decision logic conducts lightweight comparisons within a single CPU cycle in parallel to the existing fetch controller logic and thus does not delay processor execution or stall pipeline stages.

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

(1) `s_init`: The entry is first initialized with the bitmap location to be updated, which is calculated using our trace encoding algorithm described in §4.4.
(2) `s_load`: Subsequently, the current edge count at the bitmap location is read from the appropriate memory address.
(3) `s_store`: Then, the edge count is incremented by one and written to the same bitmap location stored in the entry.
(4) `s_done`: Once the entry reaches this state, it is deallocated when it becomes the head of the BUQ.

Figure 4 depicts the bitmap update operation in the BUQ designed in a manner that imposes minimal overhead. Since the bitmap itself is stored in user-accessible memory, its contents can be read and written via load and store operations with the base address of the bitmap and specific offsets. To ensure the bitmap update operation does not stall the CPU pipeline, the load part of the update operation is allowed to proceed speculatively in advance of the store operation, which is only executed when the corresponding instruction is committed. However, in case there are store operations pending for the same bitmap location from older instructions, such speculative loads are delayed until the previous store completes to prevent reading stale bitmap values. Moreover, each bitmap load and store is routed through the cache hierarchy, which does not incur the slow access latency of the main memory. Note that the cache coherence and consistency of the bitmap updates can be ensured by the hardware in a manner similar to that for regular loads and stores in the shared memory. Last, a full BUQ can result in back pressure to the Execution Stage and cause pipeline stalls. To avoid this, we sufficiently increase the BUQ; our 24-entry BUQ ensures that such stalls are infrequent and incur negligible overhead.

4.2.3 Last Branch Queue. The LBQ (❸) is a circular queue recording the information of the last 32 branches as context-specific feedback used by a fuzzer, as we describe in §4.5. Specifically, each entry of the LBQ stores the target address and the prediction result for a branch (*i.e.*, what was the branch direction and whether the predicted direction from the branch-predictor was correct or not). Such information is retrieved through the branch resolution path from the branch unit (❼), where branch prediction is resolved. To interface with the LBQ, we utilize the CSRs described in Table 2. Each LBQ entry is wired to a defined CSR and can be accessible from software after each fuzzing execution using a CSR read instruction.

4.3 Micro-architectural Optimizations

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

Opportunistic bitmap update. At the Memory Stage in Figure 3b, memory requests are scheduled and sent to the cache based on the priority policy of the cache controller. To prevent normal memory requests from being delayed, we assign the lowest priority to bitmap update requests and send them to the cache only when unused cache bandwidth is observed or when BUQ is full. Combined with the capability of the out-of-order BOOM core in issuing speculative bitmap loads for the bitmap updates, this approach allows us to effectively utilize the free cache bandwidth while also minimizing the performance impact caused by additional memory accesses.

Algorithm 2: Edge encoding by SNAP

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

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

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

Memory request aggregation. A memory request aggregation scheme (❻) is also deployed to reduce the number of additional memory accesses. When the head entry of the BUQ issues a write to update its bitmap location, it also examines the other existing entries, which might share the same influencing address for subsequent updates. If found, the head entry updates the bitmap on behalf of all the matched ones with the influence carried over, while the represented entries are marked finished and deallocated without further intervention. This is effective, especially for loop statements, where the branch instruction repeatedly jumps to the same target address across iterations. The BUQ can thus aggregate the bitmap update operations aggressively with fewer memory accesses.

4.4 Edge Encoding

Algorithm 1 describes how AFL [61] measures edge coverage, where an edge is represented in the coverage bitmap as the hash of a pair of randomly generated basic block IDs inserted during compile time. To avoid colliding edges, the randomness of basic block IDs plays an important role to ensure the uniform distribution of hashing outputs. Rather than utilizing a more sophisticated hashing algorithm or a bigger bitmap size to trade efficiency for accuracy, AFL chooses to keep the current edge-encoding mechanism, as its practicality is well backed by the large number of bugs found. Meanwhile, software instrumentation for coverage tracing requires excessive engineering effort and can be error-prone, especially in the case of complex COTS binaries without source code. Since it is non-trivial to instrument every basic block with a randomly generated ID, one viable approach is to borrow the memory address of a basic block as its ID, which has proved effective in huge codebase [22]. Such an approach works well on the x86 architecture where instructions have variable lengths, usually ranging from 1 to 15 bytes, to produce a decent amount of entropy for instruction addresses to serve as random IDs. In the case of the RISC-V architecture, however, instructions are defined with fixed lengths. Standard RISC-V instructions are 32-bit long, while 16-bit instructions are also possible only if the ISA compression extension (RVC) is enabled [54]. As a result, RISC-V instructions are well-aligned in the program address space. Reusing their addresses directly as basic block IDs for edge encoding lacks enough entropy to avoid collisions.

To match the quality of edge encoding in AFL, we devise a new mechanism (Algorithm 2) for SNAP that produces a sufficient amount of entropy with no extra overhead compared to the naive employment of memory addresses. Specifically, SNAP takes both the memory address and the instruction byte sequence inside a basic block to construct its ID. A pair of such basic block IDs are then hashed to represent the corresponding edge in the coverage bitmap. By sitting at the hardware level, SNAP is able to directly observe

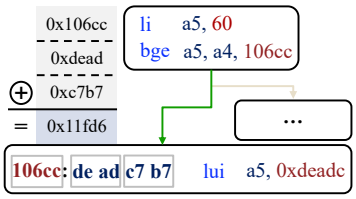


Figure 5: An example of encoding a basic block ID.

and leverage the instruction bytes as a source of entropy without overhead to compensate the lack of randomness due to the RISC-V instruction alignment. To be compatible with both 16- and 32-bit long RISC-V instructions, SNAP always fetches two consecutive 16-bit sequences starting at the instruction address and performs *bitwise XOR* twice to produce a basic bock ID (line 3 in Algorithm 2, also in Figure 5). Therefore, each ID contains the entropy from various data fields, including *opcode*, *registers*, and *immediates*, of either an entire instruction or two compressed ones. In addition, SNAP chooses to operate on the destination instruction of a branching operation to construct a basic block ID, as it provides a larger variety of instruction types (*i.e.*, more entropy) than the branch instruction itself. Similar to that of AFL, the encoding overhead of SNAP is considered minimal, as the operation can be completed within one CPU cycle. Note that Algorithm 2 can be easily converted to trace basic block coverage by discarding `prevLoc` (line 5), which tracks control transfers (*i.e.*, edges), and performing bitmap update (line 4) solely based on `curLoc` (line 3).

4.5 Richer Coverage Feedback

As discussed in §2.2, edge coverage alone can be coarse-grained and does not represent execution states accurately. Meanwhile, collecting additional execution semantics via software-based solutions always incurs major performance overhead. SNAP aims to solve this dilemma from a hardware perspective. With various types of micro-architectural state information available at the hardware level, SNAP helps fuzzers to generate more meaningful feedback that incorporates the *immediate control flow context* and *approximated data flow* of a program run without extra overhead.

Capturing immediate control-flow context. Tracking long control-flow traces can be infeasible due to noise from overly sensitive feedback and the performance overhead from comparing long traces. Therefore, SNAP records only the last 32 executed branches of a program run in the circular LBQ by default. Note that SNAP provides the flexibility of configuring branch entry number and address filters through software interfaces so that the hosted fuzzer can decide to track the execution trace of an arbitrary program space, ranging from a loop to a cross-function code region. A unique pattern of the 32 branch target addresses recorded in LBQ captures the *immediate control-flow context* of a program execution, such as the most recent sequence of executed parsing options inside string manipulation loops. When the immediate control-flow context is included in coverage feedback, a fuzzer is inspired to further mutate the inputs that share identical edge coverage but trigger unseen branch sequences within the loop (Figure 1 line 4-11) that will otherwise be discarded. As a result, the fuzzer is more likely to generate the input that can reach the specific last-executed branch sequence (*i.e.*, `SLLTS`) for the buggy constraint (line 13).

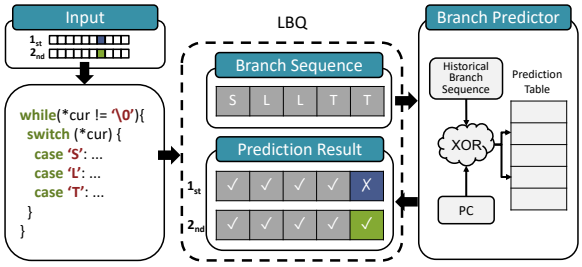


Figure 6: An example of data flow approximation between two runs leveraging the branch predictions stored in LBQ.

Approximating data flow via branch prediction. Data flow analysis has proved to be useful for fuzzers [12, 52, 60] to mutate inputs more precisely (*e.g.*, identify the input byte that affects a certain data-dependent branch condition). However, recent research [18] points out that traditional data-flow analysis requires too much manual effort (*e.g.*, interpreting each instruction with custom taint propagation rules) and is slow, making fuzzing less effective. Surprisingly, SNAP is able to provide an approximation of data flow without paying extra performance overhead by leveraging the branch prediction results in the LBQ. A typical branch predictor, such as the one used in RISC-V BOOM [7] and shown in Figure 6, is capable of learning long branch histories and predicts the current branch decision (*i.e.*, taken vs. not-taken) based on the matching historical branch sequence. Conversely, given the prediction results of the recorded branch sequence in the LBQ, SNAP is able to infer a much longer branch history than the captured one. Therefore, if a mutated input byte causes a change in the prediction result of a certain branch, the branch condition is likely related to the input offset, thus revealing dependency between them with near-zero cost. Since most branch conditions are data-dependent [12, 18, 59], the prediction result thus approximates the data flow from the input to any variables that affect the branch decision. In Figure 6, even if the coverage map and the immediate control-flow context remain the same, the fuzzer can still rely on the approximated data flows to mutate further for the sequence of interest (*i.e.*, line 13 in Figure 1) when it is not captured by the LBQ.

4.6 OS Support

Besides the hardware modification, kernel support is also critical for SNAP to work as expected. We generalize it into three components, including configuration interface, process management, and memory sharing between kernel and userspace. Rather than testing the validity of modified OS on the hardware directly, we also provide the RISC-V hardware emulation via QEMU [4], allowing easier debugging of the full system.

Configuration interface. Among the privilege levels enabled by the standard RISC-V ISA [54], we define the newly added CSRs (Table 2) in `supervisor-mode` (`S-mode`) with constraints to access through the kernel. To configure the given hardware, SNAP provides custom and user-friendly system calls to trace a target program by accessing the CSRs. For example, one can gather the last-executed branches to debug a program near its crash site by enabling the branch record only. Others might request dedicated fuzzing on specific code regions or program features by setting the address range to be traced. Overall, SNAP is designed to be flexible for various use cases.

Process management. Besides the software interface that allows user programs to configure hardware through system calls, the kernel also manages the tracing information of each traced process. Specifically, we add new fields to the process representation in the Linux kernel (*i.e.*, `task_struct`), including the address and the size of the coverage bitmap, the address and the size of the branch queue, the tracing address range, and the previous hash value. Those fields are initialized with zeros upon process creation and later assigned accordingly by the system calls mentioned before. During a context switch, if the currently executing process is being traced, the kernel disables the tracing and saves the hash value and branch records in the hardware queue. In another case, if the next process is to be traced, the SNAP CSRs will be set based on the saved field values to resume the last execution. Note that when fuzzing a multi-threaded application, existing fuzzers typically do not distinguish code paths from different threads but record them into one coverage bitmap to test the application as a whole. Although maintaining unique bitmaps is supported, SNAP enables the kernel to copy all the SNAP-related fields of a parent process, except the address of the branch queue, into its newly spawned child process by default. In addition, when a target process exits, either with or without error, SNAP relies on the kernel to clean up the corresponding fields during the exit routine of the process. However, the memory of the coverage information and the branch queue will not be released immediately, as it is shared with the designated user programs (*i.e.*, fuzzers) to construct the tracing information. Instead, the memory will be freed on demand once the data have been consumed in the userspace.

Memory sharing. To share the memory created for the coverage bitmap and the branch queue with the userspace program, SNAP extends the kernel with two corresponding device drivers. In general, the device drivers enable three *file operations*: `open()`, `mmap()`, and `close()`. A user program can create a kernel memory region designated for either of the devices by *opening* the device accordingly. The created memory will be maintained in a kernel array until it is released by the *close* operation. Moreover, the kernel can *remap* the memory to userspace if necessary. The overall design is similar to that of `kcov` [37], which exposes kernel code coverage for fuzzing.

5 EVALUATION

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

- **Performance.** How much performance cost needs to be paid for tracing on SNAP? (§5.2)
- **Accuracy.** How well can SNAP preserve traces against other approaches of comparable CPU cycles throughout the lifetime of processes? (§5.3)
- **Effectiveness.** Can SNAP increase coverage for fuzzing in a finite amount of time? How do branch records and branch predictions provide more context-sensitive semantics? (§5.4)
- **Practicality.** How easy is it to support various fuzzers on SNAP? How much power and area overhead does the hardware modification incur? (§5.5)

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

Table 3: Evaluated BOOM processor configuration.

5.1 Experimental setup

We prototype SNAP on Amazon EC2 F1 controlled by FireSim [39], an open-source FPGA-accelerated full-system hardware simulation platform. FireSim simulates RTL designs with cycle-accurate system components by enabling FPGA-hosted peripherals and system-level interface models, including a last-level cache (LLC) and a DDR3 memory [8]. We synthesize and operate the design of SNAP at the default clock frequency of `LargeBoomConfig`, which is applicable to existing CPU architectures without significant design changes. While modern commercial CPUs tend to adopt a data cache (L1-D) larger than the instruction cache (L1-I) for performance [30, 34, 35], we mimic the setup with the default data cache size of 64 KB for our evaluation. In general, the experiments are conducted under Linux kernel v5.4.0 on *f1.16xlarge* instances with eight simulated RISC-V BOOM cores, as configured in Table 3. Our modified hardware implementation complies with the RISC-V standard and has been tested with the official RISC-V verification suite. The area and power overhead of the synthesized CPU with our modification are measured by a commercial EDA tool, Synopsys Design Compiler [56].

We evaluate SNAP on the industry-standardized SPEC CPU2006 benchmark suite to measure its tracing overhead. We use the reference (*ref*) dataset on the 12 C/C++ benchmarks compilable by the latest RISC-V toolchain. To profile the encoding collisions, we collect full traces from the benchmark as the ground truth for uniquely executed edges before comparing with the encoded bitmaps. In particular, we enable user emulation of QEMU v4.1.1 in *nochain* mode to force the non-caching of translated blocks so that the entire execution trace of each run is emitted. Meanwhile, we test AFL's runtime coverage increase and throughput with Binutils v2.28 [21], a real-world collection of binary tools that have been widely adopted for fuzzing evaluation [27, 40]. In general, we fuzz each binary for 24 hours with the default corpus from AFL in one experimental run and conduct five consecutive runs to average the statistical noise in the observed data.

5.2 Tracing Overhead by SNAP

We measure the tracing overhead imposed by SNAP and source instrumentation (*i.e.*, AFL-gcc[3]) across the SPEC benchmarks. Table 4 shows that SNAP incurs a barely 3.14% overhead with the default cache size of 64 KB, significantly outperforming the comparable software-based solution (599.77%). While we have excluded the numbers for DBI solutions (*e.g.*, AFL QEMU mode), the resulting overhead is expected to be much heavier than source instrumentation, as explained in §3. The near-zero tracing overhead of SNAP results from its hardware design optimizations, including opportunistic bitmap update and memory request aggregation (§4.3). Table 5 shows that the bitmap update requests have been reduced by 13.47% on average thanks to aggregation. In the best case, the reduction rate can reach above 40%, which effectively mitigates cache contention from frequent memory accesses (*e.g.*, array iteration) and avoids unnecessary power consumption.

Further investigation shows that the performance cost of SNAP might also result from cache thrashing at the L1 cache level. In general, applications with larger memory footprints are more likely to be affected. Since bitmap updates by the BUQ are performed in the cache shared with the program, cache lines of the program data might get evicted when tracing is enabled, resulting in subsequent cache misses. Note that this problem is faced by all existing fuzzers that maintain a bitmap. For instance, Table 5 points out that `gobmk` and `xalancbmk` both suffer from comparably higher overhead ($\geq 5\%$) caused by reduced cache hit rates of over 3.5%. The impact of cache thrashing can also be tested by comparing the tracing overhead of SNAP configured with different L1 D-cache sizes. Table 4 shows that a larger cache exhibits fewer cache misses and can consistently introduce lower tracing overhead across benchmarks. In particular, the overhead can be reduced to 3% on average by increasing the

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

Table 4: Tracing overhead from AFL source instrumentation and SNAP with various L1-D cache sizes across the SPEC benchmarks.

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

Table 5: Memory request aggregation rates and L1 cache hit rates between the baseline and SNAP across the SPEC benchmarks.

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

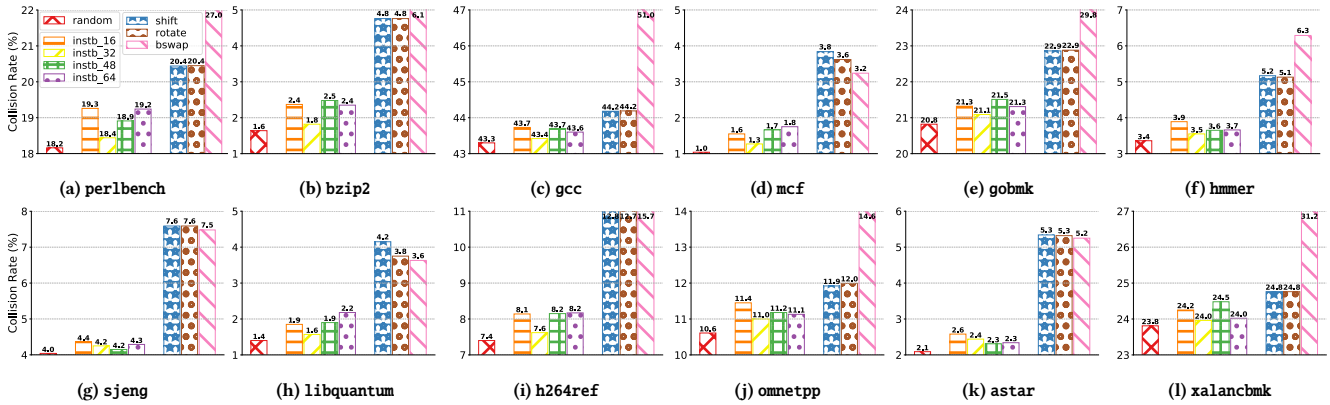


Figure 7: Collisions from encoded trace edges among benchmarks. `random` indicates the baseline by AFL with random basic block IDs. The `instb_*` family by SNAP adopts both memory addresses and instruction bytes as IDs. `shift`, `rotate` and `bswap` present the best collision rates achieved by bitwise operations on memory addresses alone.

cache size to 128 KB. Alternatively, the extra storage can be repurposed as a dedicated buffer for the coverage bitmap to avoid cache misses due to bitmap update, which we leave for future work.

In addition, Table 4 shows that the tracing overhead of AFL-gcc is much larger. With the CPU-bound benchmarks that best approximate the extreme circumstances, the overhead is expected [62], as discussed in §3. This finding is generally consistent with the numbers from the x86 setup, which also incurs an average of 228.09% overhead on the same testing suite by AFL-gcc. The extra slowdown in the current RISC-V experiment is caused by the additional instrumented locations in binaries due to the difference in ISAs. For example, RISC-V does not define a specific instruction for backward edges (*i.e.*, `ret`), which are often not tracked on instrumented x86 binaries. Thus, the RISC-V benchmarks have 58.51% more instrumentation than the x86 version, resulting in a 40.03% increase in the binary size. Note that the cache size has negligible impact on the tracing overhead for the software-based solution. Although bitmap updates can still cause cache thrashing, the overhead mainly comes from the execution cost of instrumented instructions.

5.3 Preserving Traces

Figure 7 shows the collision rates of various edge-encoding schemes on the SPEC benchmarks with the reference workload. The collisions are confirmed by comparing full execution traces against their resulting bitmaps accordingly. A lower collision rate implies that a more complete code trace is preserved when a binary is fully tested. Together with the result of `gcc` from a limited bitmap size (*i.e.*, 64 KB), Algorithm 1 by AFL consistently generates the lowest collision rates (11.47%) among all. With random basic block IDs inserted at compile time, true randomness is introduced into the algorithm to avoid collisions. In comparison, the approaches that directly adopt the memory address of a basic block as its ID perform significantly worse. Even with the bitwise operations (*i.e.*, *logical shifts*, *circular shifts*, or *endian swaps* N bits of the block addresses before *bitwise XORing* them), the impact from well-aligned RISC-V instructions cannot be reduced. In particular, the most effective one, *circular shift*, produces an average of 337 more colliding edges than AFL per benchmark, with a worst case of a 5.29% increase (*i.e.*, `h264ref`).

Name	#Block	Collision Rate (%)				
		random	instb_16	instb_32	instb_48	instb_64
perlbench	18,612	13.06	16.91	14.61	14.60	14.44
bzip2	2,069	1.61	2.32	2.23	2.15	1.96
gcc	55,222	32.39	34.99	32.55	32.71	32.61
mcf	1,493	0.87	1.67	1.34	1.34	1.70
gobmk	19,762	13.58	16.08	14.40	13.80	14.35
hmmer	3,360	2.47	3.21	3.02	3.57	3.33
sjeng	3,351	2.45	3.49	3.01	3.28	3.07
libquantum	1,397	1.15	2.29	1.50	1.72	2.05
h264ref	7,365	5.53	6.93	6.19	5.90	5.94
omnetpp	10,321	7.04	8.41	7.84	8.05	7.82
astar	2,441	1.90	2.58	2.12	2.11	2.60
xalancbmk	27,169	18.06	18.82	18.51	18.45	18.47
Mean	12,713	8.34	9.81	8.94	8.97	9.03

Table 6: Collisions from encoded trace blocks among benchmarks.

On the other hand, Algorithm 2 by SNAP can achieve collision rates (11.70%) similar to those of AFL. By leveraging the entropy of RISC-V instruction(s) at the start of a basic block, SNAP significantly outperforms the aforementioned approaches that solely rely on memory addresses for encoding. Meanwhile, we cannot find a consistent pattern of including more instruction bytes for fewer collisions beyond 32 bits, as shown in Figure 7. Since most instructions are 32-bit long, gathering data fields (*e.g.*, opcode, registers, and immediates) beyond one instruction might not be helpful in practice.

Besides edge coverage, basic block coverage also serves as a metric adopted by existing fuzzers [43, 52] to measure code coverage. Table 6 shows the collisions from SNAP using basic block coverage (§4.4) across the same benchmarks. The mechanism proposed by SNAP (*i.e.*, `instb_32`) reaches a collision rate of 8.94% on average, similar to the rate by AFL (8.34%). Therefore, SNAP is considered equally accurate in terms of preserving either block or edge traces.

5.4 Evaluating Fuzzing Metrics

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

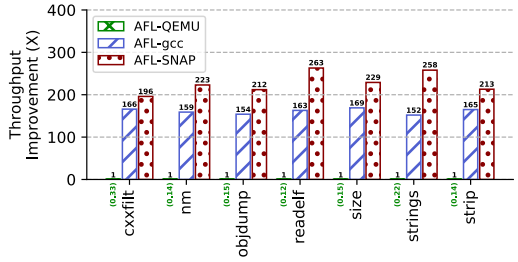


Figure 8: The average execution speed from fuzzing with AFL-QEMU, AFL-gcc and AFL-SNAP for 24 hours across the Binutils binaries. The numbers below the bars of AFL-QEMU show the number of executions per second for the mechanism.

Fuzzing throughput. Figure 8 shows the fuzzing throughput across the compared mechanisms. Specifically, AFL on SNAP can achieve 228× higher execution speed than AFL-QEMU, which is limited by the low clock frequency and its inefficient RISC-V support. The average throughput of AFL-QEMU (*i.e.*, 0.18 exec/s) is consistent with the previous findings in PHMon [14]. Note that SNAP improves the throughput much more significantly than PHMon, which only achieves a 16× higher throughput than AFL-QEMU. Despite that the baseline BOOM core in SNAP is about 50% faster [65] than the baseline Rocket core [5] adopted by PHMon, SNAP achieves a 14× higher throughput-increase in comparison mainly due to its design optimizations (*e.g.*, opportunistic bitmap update and memory request aggregation). Compared to AFL-gcc, SNAP can still achieve a 41.31% higher throughput on average across the benchmarks. More throughput comparisons on x86 platforms are shown in Appendix A.

Edge coverage. Figure 9 depicts the resulting coverage measurement, where the confidence intervals indicate the deviations across five consecutive runs on each benchmark. Given an immature seed corpus and a time limit, AFL with SNAP consistently covers more paths than the others throughout the experiment. Since no change to fuzzing heuristics (*e.g.*, seed selection or mutation strategies) is made, the higher throughput of SNAP is the key contributor to its outperformance. On average, AFL-QEMU and AFL-gcc have only reached 23.26% and 84.59% of the paths discovered by AFL-SNAP, respectively. Although larger deviations can be observed when the program logic is relatively simple (Figure 9f), SNAP in general can help explore more paths in programs with practical sizes and complexity thanks to its higher throughput. For example, AFL with SNAP manages to find 579 (16.74%), 237 (20.82%), and 378 (19.77%) more paths when fuzzing `cxxfilt`, `objdump`, and `readelf`, respectively.

Adopting execution context. Given the last-executed branches and their prediction results in LBQ, fuzzers on SNAP are equipped with additional program states. To take the feedback, one can easily follow the mechanisms introduced previously [13, 26, 28]. Our prototype of AFL instead adopts a feedback encoding mechanism similar to that in Algorithm 1 to showcase the usage. Specifically, the highest significant bit (HSB) of each 64-bit branch address is set based on the respective prediction result (*i.e.*, 1/0). To maintain the order of branches, the records are iterated from the least recent to the latest in the circular LBQ and *right circular shift*'ed (*i.e.*, rotated)

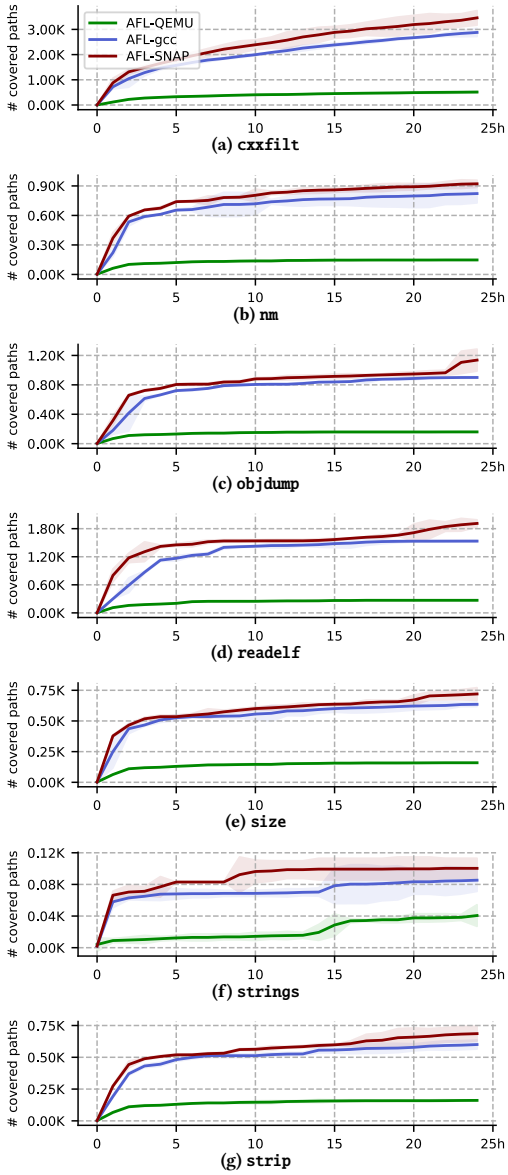


Figure 9: The overall covered paths from fuzzing seven Binutils binaries for 24 hours. The solid lines represent the means, and the shaded areas suggest the confidence intervals of five consecutive runs.

by N bits based on their relative positions in the sequence before being *bitwise XOR*'ed. The encoded value is finally indexed into a separate bitmap from the one for edge coverage (*i.e.*, *trace_bits*).

Reproducing a known bug. Running on SNAP, the modified AFL is able to trigger `CVE-2018-9138` discovered by the previous work [13], which proposes using feedback similar to that provided by our platform. As in Figure 1, the vulnerability occurs when `cxxfilt` consumes a long *consecutive* input of "F"s, each indicating that the current mangled symbol stands for a function. The corresponding switch case in the loop (line 5-10) tries to further demangle the function arguments (*i.e.*, `demangle_args()`) before running into the next "F" to start a recursive call chain. Luckily,

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

Table 7: Estimates of area and power consumption.

SNAP offers the execution context by capturing branch sequences triggered by mutated inputs. While a vanilla AFL cannot easily reach the faulty program state with only edge coverage feedback, our fuzzer can consistently achieve it within one fuzzing cycle, led by the guidance.

5.5 Practicality of SNAP

Easy adoption. To show how SNAP can be easily adopted, we have integrated a variety of fuzzers from FuzzBench [25], including AFL [61], AFLFast [10], AFLSmart [51], FairFuzz [42], MOpt [45], and WEIZZ [16]. The others are excluded from the list, *not* because of fundamental challenges to adopt SNAP but due to the incompatibility of RISC-V. For example, HonggFuzz [26], libFuzzer [43], Entropic [9], laf-intel [41], and Ankou [46] fail to compile on Fedora/RISC-V due to the lack of support from LLVM, GO, and their dependent libraries (*e.g.*, libunwind). Otherwise, the adoption of SNAP is straightforward, requiring only a change of less than 100 LoCs consistently. Around 55 LoCs are C code that issues the system calls for creating shared bitmap and branch records, as well as comparing execution context per testcase. The others are assembly that compiles RISC-V binaries to work with forkserver (*i.e.*, `afl-gcc`).

Area and power overhead. To estimate the area and power overhead of SNAP, we synthesize our design using Synopsys Design Compiler at 1GHz clock frequency. To obtain a realistic estimate of the SRAM modules such as L1 D-cache, L1 I-cache, and branch predictor (BPD) tables used in the BOOM core, we black-box all the SRAM blocks and use analytical models from OpenRAM [29]. Our synthesis uses 45nm FreePDK libraries [55] to measure the area and power consumption between the unmodified BOOM core and the modified SNAP core. Table 7 shows that SNAP only incurs 4.82% area and 6.53% power overhead, more area-efficient than the comparable solution (16.5%) that enables hardware-accelerated fuzzing [14]. When tracing is disabled, the power overhead can be mostly avoided by clock gating through the switch CSR *TraceEn*.

6 DISCUSSION

Comparison with PHMon. PHMon [14] is a recently proposed hardware-based security monitor that enforces expressive policy rules. Despite its demonstration of basic hardware-assisted tracing for fuzzing, PHMon is not specifically designed for this purpose. In comparison, SNAP outperforms PHMon by a 14× higher fuzzing throughput thanks to the optimizations dedicated to lightweight tracing, including opportunistic updates to utilize free cache bandwidth, issuing speculative load operations to avoid delays, and memory request aggregation to reduce operations. Moreover, SNAP enables additional execution semantics as context-aware fuzzing feedback without extra performance cost by providing last-executed branches and their branch prediction results. Together with the cleverly encoded bitmap of low collision rates, SNAP helps fuzzers explore more program states for more interesting mutations.

Usage beyond fuzzing. Although fuzzing is a first-class citizen targeted by SNAP, other applications are also welcomed by the general design. For example, SNAP can provide an efficient coverage estimation for unit testing, which incurs significant hassle and overhead with existing mechanisms such as gcov [20] and Intel PT [36]. The information can also serve as an execution fingerprint for logging and forensic purposes. Last, partial feedback, such as branch prediction results, can serve as approximated performance metrics with profiled cache misses in a specific code region.

Limitations and future directions. While SNAP is carefully designed not to hinder the maximum clock frequency, we are limited in our evaluation to a research-grade hardware setup with low clock speed. We hope our work motivates future studies and adoption on more powerful cores [58] and custom ASICs by processor vendors [38]. Additionally, while SNAP does not support kernel coverage filtered by the privilege level, leveraging the hardware for tracing kernel space is not fundamentally restricted. SNAP is also not suitable to track dynamic code generation with reused code pages, such as JIT and library loading/unloading, as it affects the validity of the coverage bitmap. If needed, annotations with filters on the program space can be applied to reduce noise. Future work could include repurposing a buffer dedicated for coverage bitmap storage to avoid extra cache misses, leveraging other microarchitectural states from hardware, such as memory access patterns, to identify dynamic memory allocations (*e.g.*, heap) across program runs, or adopting operands of comparing instructions for feedback as suggested [41]. Alternatively, given filters in the debug unit of ARM's CoreSight extension [2], the practicality of the design can be further demonstrated without relying on custom hardware.

7 RELATED WORK

Binary-only fuzzing. Runtime coverage tracing can be costly and becomes even more complicated when handling closed-source targets, such as COTS binaries. In particular, a typical software-based solution falls into either static or dynamic binary instrumentation, each limited by different constraints. For example, DynInst [57] is not widely adopted, as the binary rewriting mechanism is error-prone due to its complexity and thus cannot be applied to many real-world use cases [15]. RetroWrite [15] requires relocation information of position-independent code to soundly instrument binaries. While most of the dynamic binary instrumentation schemes [4, 11, 44, 49, 63] are more accessible to fuzzers thanks to their ease of use, they typically suffer from significant overhead due to runtime translation or callback routines. Although UnTracer [48] suggests coverage-guided tracing to achieve near-native execution speed for most of the *non-interesting* fuzzing testcases, its current design and evaluation are based on basic block coverage, which represents a less accurate program execution trace in regard to branch transfers and loops. Despite that a revised edge coverage tracker (without edge count) has been proposed, the performance impact of switching to the new solution is unclear due to the potential increase of interesting testcases. In contrast, SNAP avoids such hassles by tracing at the hardware level. It removes the gap between source-based and binary-only tracing while providing richer execution feedback with near-zero performance overhead.

Hardware-assisted fuzzing. Besides the software-based solutions, existing fuzzers [13, 26, 28, 53, 64] turn to available hardware extensions [31, 32, 36] for guidance when fuzzing binaries without source code. Intel PT [36] has been the most commonly adopted one, exposing the full trace of an execution in a highly compressed fashion for efficiency. Despite its generality, the use of Intel PT for fuzzing can be ad-hoc, as the feature was originally designed for helping debug a program execution with accurate and detailed traces without worrying about performance impact. Therefore, it already incurs at least 20-40% combined overhead for tracing and decoding before a fuzzer can incorporate the feedback for further mutation [33, 48, 64]. Although PTrix [13] utilizes Intel PT to gather traces under a parallel scheme without recovering the exact conditional branches for edge coverage to avoid major decoding overhead, it merely achieves a comparable execution speed as source instrumentation. Similarly, since PHMon [14] is designed to suit different use cases, such as providing shadow stack and watchpoints for a debugger, its usage for tracing is not optimized for fuzzing either. In comparison, SNAP adopts a highly optimized design for fuzzing and shows its advantage over the other approaches in §5.4. Despite the barrier to entry for a customized architecture, the benefits of SNAP under minimal changes to an existing CPU pipeline can be intriguing to commodity hardware. Motivated by the existing hardware-accelerated infrastructures dedicated for machine learning [1, 23, 50], along with the increasing industrial demand of fuzzing services [24, 47], SNAP demonstrates the feasibility of performance boost by hardware-assisted fuzzing, complementing Intel PT for various use cases.

8 CONCLUSION

We present SNAP, a customized hardware platform that implements hardware primitives to enhance performance and precision of coverage-guided fuzzing. SNAP is prototyped as a full FPGA implementation together with the necessary OS support. By leveraging micro-architectural optimizations in the processor, our prototype enables not only transparent tracing but also richer feedback on execution states with near-zero performance cost. Adopted fuzzers, such as AFL, can achieve 41% and 228× faster execution speed (and thus higher coverage) running on SNAP than with existing tracing schemes, such as source instrumentation and DBI. The hardware design only poses a 4.8% area and 6.5% power overhead and thus is applicable to existing CPU architectures without significant changes.

9 ACKNOWLEDGMENT

We thank the anonymous reviewers, and our shepherd, David Chisnall, for their helpful feedback. This research was supported, in part, by the NSF award CNS-1563848 and CNS-1749711 ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA AIMEE HR00112090034 and SocialCyber HR00112190087, ETRI IITP/KEIT[2014-3-00035], and gifts from Facebook, Mozilla, Intel, VMware and Google.

REFERENCES

[1] Apple. 2020. M1: Apple Neural Engine. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/.

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

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[4] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*. Anaheim, CA.

[5] UC Berkeley. 2016. Rocket Chip Generator. https://github.com/chipsalliance/rocket-chip.

[6] UC Berkeley. 2017. The Berkeley Out-of-Order RISC-V Processor. https://github.com/riscv-boom/riscv-boom.

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

[8] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanovic. 2019. FASED: FPGA-Accelerated Simulation and Evaluation of DRAM. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*.

[9] Marcel Bohme, Valentin Manes, and Sang Kil Cha. 2020. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *Proceedings of the 28th Joint Meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Sacramento, CA.

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.

[11] Derek Bruening and Saman Amarasinghe. 2004. *Efficient, transparent, and comprehensive runtime code manipulation*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[12] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[13] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. 2019. Ptrix: Efficient hardware-assisted fuzzing for cots binary. In *Proceedings of the 14th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. Auckland, New Zealand.

[14] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. 2020. PHMon: A Programmable Hardware Monitor and Its Security Use Cases. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

[15] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2018. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*.

[16] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. WEIZZ: Automatic Grey-box Fuzzing for Structured Binary Formats. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. Los Angeles, CA.

[17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*.

[18] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

[19] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[20] GNU Compiler Collection (GCC). 2012. Gcov is a test coverage program. https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html.

[21] GNU Project. 2020. GNU Binutils. https://www.gnu.org/software/binutils.

[22] Google. 2018. syzkaller – kernel fuzzer. https://github.com/google/syzkaller.

[23] Google. 2020. Cloud TPU: Empowering businesses with Google Cloud AI. https://cloud.google.com/tpu.

[24] Google. 2020. ClusterFuzz. https://google.github.io/clusterfuzz.

[25] Google. 2020. FuzzBench: Fuzzer benchmarking as a service. https://github.com/google/fuzzbench.

[26] Google. 2020. Honggfuzz. https://github.com/google/honggfuzz.

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

[28] Google Project Zero. 2020. WinAFL. https://github.com/googleprojectzero/winafl.

[29] Matthew R. Guthaus, James E. Stine, Samira Ataei, Brian Chen, Bin Wu, and Mehedi Sarwar. 2016. OpenRAM: An Open-Source Memory Compiler. In *Proceedings of the 35th International Conference on Computer-Aided Design (ICCAD)*.

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

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

[32] Intel. 2011. Intel BTS: Real time instruction trace. https://www.intel.com/content/dam/www/public/us/en/documents/reference-guides/real-time-instruction-trace-atom-reference.pdf.

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

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

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

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

[37] Simon Kagstrom. 2015. Kcov is a FreeBSD/Linux/OSX code coverage tester. https://github.com/SimonKagstrom/kcov.

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

[39] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. 2018. FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*.

[40] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, ON, Canada.

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

[42] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Corum, France.

[43] LLVM Project. 2020. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[44] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[45] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA.

[46] Valentin Manes, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding Grey-box Fuzzing towards Combinatorial Difference. In *Proceedings of the 42th International Conference on Software Engineering (ICSE)*. Seoul, South Korea.

[47] Microsoft Security Response Center (MSRC). 2020. OneFuzz. https://github.com/microsoft/onefuzz.

[48] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[49] Anh-Quynh Nguyen and Hoang-Vu Dang. 2014. Unicorn: Next Generation CPU Emulator Framework. In *Black Hat USA Briefings (Black Hat USA)*. Las Vegas, NV.

[50] Nvidia. 2020. NVIDIA DRIVE AGX ORIN: Advanced, Software-Defined Platform for Autonomous Machines. https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware/.

[51] Van-Thuan Pham, Marcel Bohme, Andrew Santosa, Alexandru Caciulescu, and Abhik Roychoudhury. 2019. Smart Greybox Fuzzing. In *IEEE Transactions on Software Engineering*.

[52] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[53] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. KAFL: Hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the 26th USENIX Security Symposium (Security)*. Vancouver, BC, Canada.

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

[55] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal. 2007. FreePDK: An Open-Source Variation-Aware Design Kit. In *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*.

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

[57] Cisco Talos. 2014. AFL-Dyninst: AFL fuzzing blackbox binaries. https://github.com/Cisco-Talos/moflow/tree/master/afl-dyninst.

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

[59] Renju Thomas, Manoj Franklin, Chris Wilkerson, and Jared Stark. 2003. Improving Branch Prediction by Dynamic Dataflow-based Identification of Correlated Branches from a Large Global History. In *Proceedings of the 30th ACM/IEEE International Symposium on Computer Architecture (ISCA)*. San Diego, CA, USA.

[60] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*. Oakland, CA.

[61] Michal Zalewski. 2014. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.

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

[63] Google Project Zero. 2020. TinyInst: A lightweight dynamic instrumentation library. https://github.com/googleprojectzero/TinyInst.

[64] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. 2018. PTfuzz: Guided fuzzing with processor trace feedback. In *IEEE Access (vol. 6)*.

[65] Jerry Zhao, Korpan Ben, Gonzalez Abraham, and Asanovic Krste. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*.

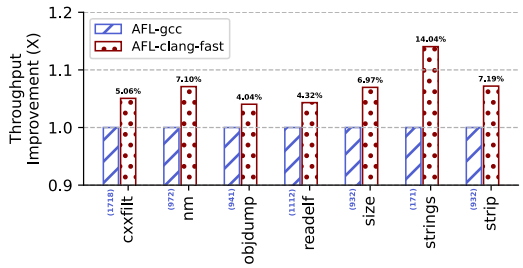


Figure 10: The average execution speed from fuzzing with AFL-gcc and AFL-clang-fast for 12 hours across the Binutils binaries. The numbers below the bars of AFL-gcc show the number of executions per second for the mechanism.

A APPENDIX

A.1 AFL throughput on x86 platforms

Although the fuzzing throughput from AFL-clang on the RISC-V platform is left out due to technical difficulties, a similar comparison for the numbers on x86 platforms is gathered instead. In particular, we compile the Binutils binaries through AFL-gcc and AFL-clang, and conduct five consecutive fuzzing runs of 12 hours to reduce the statistical noise. Figure 10 shows that AFL-clang takes consistent advantage of the compiler-based optimizations over AFL-gcc, which manually instruments at the assembly-level, and outperforms in all evaluating cases by an average of 6.96% faster execution speed.

The finding generally aligns with that of the AFL whitepaper [62], suggesting the performance gain of less than 10% for most binaries other than CPU-bound benchmarks. The only exception occurs when fuzzing *strings* (14.04%). This is because AFL's feedback adopts edge counters, driving the fuzzer to to search for longer inputs with more printable strings, while the gain is magnified due to more iterations and branch encounters consequentially. Thus, given the relative edges over AFL-gcc (Figure 8 and Figure 10) and the posing overhead on the SPEC benchmarks (Table 4 and Table 1), we would expect SNAP to outperform AFL-clang on the RISC-V platform with higher fuzzing throughput.