Efficiently Rebuilding Coverage in Hardware-Assisted Greybox Fuzzing

Tai Yue
National University of Defense Technology, China
Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China
yuetai17@nudt.edu.cn

Yibo Jin
Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China
jinyb@mail.sustech.edu.cn

Fengwei Zhang
Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China
zhangfw@sustech.edu.cn

Zhenyu Ning
Hunan University, China
zning@hnu.edu.cn

Pengfei Wang
National University of Defense Technology, China
pfwang@nudt.edu.cn

Xu Zhou
National University of Defense Technology, China
zhouxu@nudt.edu.cn

Kai Lu
National University of Defense Technology, China
kailu@nudt.edu.cn

ABSTRACT

Coverage-based greybox fuzzing (CGF) is an efficient technique for detecting vulnerabilities, but its coverage-feedback mechanism introduces significant overhead in binary-only fuzzing. Although hardware-assisted greybox fuzzing (HGF) has been proposed to address this issue, existing approaches struggle to achieve a balance between the efficiency and sensitivity of coverage, as well as to cope with trace buffer overflow.

In this paper, we review the typical HGF tools and identify several challenges in their coverage-feedback mechanisms, including efficiency, sensitivity, and stability. Taking Arm CoreSight as an example, we present an efficient tool called Stalker to address these challenges. To achieve high-speed execution while maintaining a branch-sensitivity coverage, we propose two coverage strategies with different overheads and sensitivities and design a novel double-layer coverage mechanism that maximizes the benefits of these strategies. We further accelerate Stalker by conducting many optimizations in the decoder and kernel. To mitigate the imprecision and instability in coverage introduced by trace buffer overflow, we propose an adaptive CPU frequency modulation mechanism that adjusts the bandwidth of the trace units. We implement Stalker on an Arm Juno R2 development board and thoroughly evaluate the efficiency and sensitivity of coverage-feedback mechanisms in existing tools. Our comprehensive evaluations demonstrate that Stalker outperforms other state-of-the-art (SOTA) tools in addressing these challenges. Compared with Armored-Edge, Armored-Path, and μAFL, Stalker accelerates the execution speed by 2.81×, 1.74×, and 1.4× and covers 23.9%, 66.1%, and 3.5% more branches, as well as 66.4%, 323.3%, and 19.2% more paths, respectively.

ACM Reference Format:
Tai Yue, Yibo Jin, Fengwei Zhang, Zhenyu Ning, Pengfei Wang, Xu Zhou, and Kai Lu. 2024. Efficiently Rebuilding Coverage in Hardware-Assisted Greybox Fuzzing. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

CGF has become an efficient technique for detecting software vulnerabilities [31, 32]. A prominent tool is AFL [53]. By generating numerous inputs at a high speed, AFL and its extensions have discovered numerous vulnerabilities [11, 12, 23, 42, 46, 53]. However, this high speed is supported by lightweight compile-time instrumentation, which is unavailable for binaries without the source code [14]. For *binary-only fuzzing*, the main performance bottleneck arises from the collection and reconstruction of coverage [14, 36].

Recently, modern processors have been equipped with tracing techniques[1], such as Intel Processor Trace (Intel PT) [28] or Arm CoreSight [5], which efficiently trace executed instructions and encode control flow information into trace packets. Researchers have leveraged these trace packets to reconstruct coverage and proposed HGF to improve binary-only fuzzing [2, 14, 24, 30, 40, 41, 55].

[1]In this paper, we only focus on the hardware tracing techniques employed in fuzzing. Some other techniques, such as the Last Branch Record (LBR), are not under our consideration and beyond the scope of this paper.

Compared to other techniques employed in binary-only fuzzing, such as dynamic binary instrumenting (DBI) or static binary rewriting [9, 15, 19, 21, 25, 33, 36, 56], hardware tracing offers efficient control flow tracing with a negligible runtime overhead (2-5%) [45]. It relies solely on hardware platform without requiring additional prerequisites for the binary, making it highly practical [2, 8, 14, 24, 30, 40, 41, 55]. Moreover, the powerful tracing ability makes HGF effective in fuzzing the code that is challenging to instrument, such as the secure applications running in the Trust Execution Environment (TEE) or macOS kernel [41, 43, 44, 48].

Though existing hardware-assisted tools have been employed in detecting vulnerabilities in real-world applications [14, 24, 30, 40, 41, 48, 55], there remains some challenges in their basement, namely coverage-feedback mechanism. **First, it is challenging for existing tools to rebuild coverage with high efficiency and moderate sensitivity** (we regard the sensitivity of popular branch-granularity coverage as moderate sensitivity in this paper). The branch-count coverage of AFL has been proven to be highly efficient and is followed by many fuzzers [11, 15, 34, 36, 41, 50, 51]. However, tracing techniques like Intel PT and Arm CoreSight usually record whether a branch is taken or not and the destinations of indirect branches, lacking the ability to record the source addresses of branches (i.e., edges). Some tools [2, 38, 41, 55] rebuilt the branch-count coverage by disassembling the binary to recover the entries of executed blocks, but this approach incurred significant overhead. μAFL [30] utilizes the **Branch Broadcasting (BB)** mode of ETM, which enables ETM to trace the destinations of direct branches, to capture more addresses of branches and efficiently rebuild a branch coverage similar to the branch-count coverage of AFL. However, the BB mode increases the trace data and introduces more workload to the decoder than the default mode. To reduce the overhead in rebuilding coverage, PTrix [14] and Armored-Path[1] [2] rebuild coverage directly from the trace packets without disassembling code but introduced a more sensitive path coverage than that in AFL. The sensitive coverage makes the fuzzer select and add superfluous seeds into the seed queue, defined as *seed explosion*. The seed explosion puts heavy pressure on the scheduling algorithm, limiting the fuzzing efficiency. We will demonstrate this phenomenon in Section 5.3. **Second, the trace buffer overflow in hardware tracing introduces imprecision and instability in rebuilding coverage, affecting the fuzzing efficiency.** A stable fuzzer should consistently follow the same path of a deterministic program when given the same input repeatedly. However, modern CPUs utilize the delicate on-chip buffer to store the trace data temporarily. The trace buffer occurs overflow frequently when the trace data is generated at a high bandwidth [18, 57]. The overflow incurs trace data loss, impacts the precision and stability of coverage, and weakens the efficiency of fuzzing. Experiments in Section 5.4 will prove this.

In this paper, we explore the problems in rebuilding coverage by hardware tracing and argue that an efficient HGF should satisfy two conditions: 1) **Building branch coverage with low overhead to avoid the seed explosion.** 2) **Alleviating the trace buffer overflow.** We present solutions to these challenges and develop an efficient hardware-assisted fuzzer. Considering that the prevalence of Arm-based devices and servers calls for efficient fuzzers [2, 30], we take Arm CoreSight as examples to implement the fuzzer. Additionally, the universality of the problems we tackle and the similarities between Intel PT and Arm CoreSight allow for applying some of our methods to the tools based on Intel PT. Moreover, since our focus lies solely on enhancing the coverage-feedback mechanism in HGF, optimizing mutation strategies and scheduling algorithms or exploring parallel fuzzing are out of our scope.

[1]Armored-CoreSight refers to libxdc and incorporates two types of coverage: branch-count coverage, referred to as Armored-Edge, and PTrix-type path coverage, denoted as Armored-Path.

We present Stalker, *an efficient hardware-assisted greybox fuzzer* designed to achieve above two criteria: 1) To balance the efficiency and sensitivity in rebuilding coverage (Criteria 1), we propose a novel *double-layer coverage mechanism.* The core design of this mechanism is executing test cases under the *lightweight path coverage* to preliminarily discover the coverage-increased seeds and filtering them under the *moderate branch coverage.* To reduce the overhead, we design the lightweight path coverage under the default mode of ETM, optimize the CoreSight driver to disable the unnecessary formatter, and develop a light and stable decoder based on ptm2human [26]. To select a new seed with branch coverage and avoid seed explosion, we develop the moderate branch coverage under the BB mode of ETM without expensive disassembling. 2) To alleviate the overflow (Criteria 2), we propose an *adaptive CPU frequency modulation mechanism* to retain a high execution speed while preventing the overflow. We also illustrate how to configure ETM efficiently to reduce its bandwidth of generating trace data and trace the necessary coverage information during fuzzing.

We implement Stalker based on AFL and conduct comprehensive experiments to evaluate it against SOTA tools, including AFL [53], Armored-CoreSight [2], μAFL [30] (emulated by Stalker-Branch-Fmt), PTrix [14], and libxdc [38]. On 10 real-world programs, Stalker outperforms other Arm-based tools in executing the test cases and rebuilding the same granularity coverage (2.81×, 1.74×, and 1.4× faster than Armored-Edge, Armored-Path, and μAFL, respectively) and covers 10.0%, 23.9%, 66.1%, and 3.5% more branches and 13.7%, 66.4%, 323.3%, and 19.2% more paths than AFL-QEMU++, Armored-Edge, Armored-Path, and μAFL, respectively. Compared with PTrix and libxdc, Stalker is 2.2× and 1.6× faster than them in rebuilding coverage, respectively. We also evaluate the effectiveness and efficiency of the mechanisms in Stalker. The results show that our double-layer coverage mechanism enables Stalker to execute most of test cases under the bottom-layer path coverage with only 9.9% repeated executions and avoid the seed explosion under the top-layer branch coverage. Our frequency modulation mechanism significantly reduces the trace buffer overflow.

The contributions of this paper are as follows:

- We review the development and technical details of existing hardware-assisted fuzzers and highlight the challenges in coverage-feedback mechanisms, such as the efficiency and sensitivity in rebuilding coverage and the effects of trace buffer overflow.
- We design a *double-layer coverage mechanism*, consisting of *lightweight branch coverage* and *moderate path coverage.* This mechanism leverages the strengths of both coverage layers, achieving low overhead while avoiding seed explosion.
- We propose a novel *adaptive CPU frequency modulation mechanism*, alleviating the trace buffer overflow.

- We implement the prototype of Stalker on the Arm Juno R2 and comprehensively evaluate it against SOTA tools, including Arm and Intel platforms. Results show that our techniques have better effectiveness and efficiency than other tools.
- We will open the source of Stalker upon the publication of this paper.

2 BACKGROUND

2.1 Coverage-Based Greybox Fuzzing

CGF is a widely used technique for vulnerability detection [11, 22, 39]. A fundamental principle of CGF is that as more code is covered, the likelihood of triggering bugs increases [35]. To continually improve coverage, current CGF tools employ genetic algorithms to discover and select test cases that increase coverage in a metric [53].

Previous research has highlighted the impact of different coverage metrics on fuzzing efficiency [47]. Metrics with higher sensitivity can result in a larger number of inputs available for seed selection, potentially leading to seed explosion and the fuzzer becoming trapped in exploring specific code regions [14, 47]. Consequently, popular tools often follow the branch-count coverage of AFL, which captures basic block transitions (i.e., branches or edges) and keeps a record of their hit counts in a coverage bitmap [53].

2.2 Arm CoreSight Technique

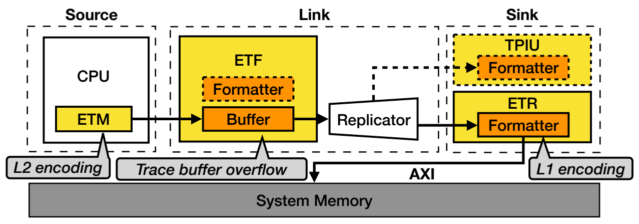


Figure 1: The CoreSight architecture on Arm Juno R2.

Arm proposed CoreSight architecture for debugging and tracing of complex system on chip (SoC), which contains a series of hardware components [5]. Fig. 1 shows the CoreSight architecture implemented on Arm Juno R2 [6]. As a trace source, *Embedded Trace Macrocell* (ETM) traces instructions and data by monitoring buses with negligible overhead. To reduce the size of trace data, the ETM encodes the trace elements into packets [7] (i.e., **L2 encoding** [54]). Both *Embedded Trace FIFO* (ETF) and *Embedded Trace Router* (ETR) can store the trace data [4]. However, on Juno R2, ETF utilizes a 64KB dedicated FIFO buffer to store data with low latency, while ETR routes the data over an AXI bus to memory with some latency but up to 4 GB from DRAM. Therefore, on many high-end SoCs (e.g., Juno R2), users usually configure ETR as a sink to stores all trace data, and ETF as a link to avoid data loss if a delay occurs when accessing the DRAM via ETR [4].

There are two details in this configuration: 1) The formatter in ETR embeds source ID signals into the trace streams to stamp the trace source of each stream (referred to as **L1 encoding**), which introduces unnecessary **L1 decoding** when only one ETM is utilized. 2) When the buffer in ETF becomes full, ETF sends a signal to ETM. ETM then halts until the overflow is resolved (generating an Overflow packet to record this event), resulting in the loss of data during this period. We record this as *trace buffer overflow*.

2.3 Embedded Trace Macrocell

Similarly to Intel PT, the ETM[1] is embedded into the CPU to trace instructions and generate trace elements, including Atom and Address elements [7]. An Atom element indicates whether a branch is taken (E) or not taken (N), while an Address element represents the destination address of a branch. By default, the ETM only records the addresses of indirect branches, such as ret and blr. In the **Branch Broadcasting (BB)** mode, direct branch destinations are also captured [7]. Furthermore, users can configure the context ID and address range comparators of the ETM to trace a specific process within the assigned address range.

Fig. 3 presents a code snippet along with its assembly code and the essential trace elements recorded by the ETM in different modes. The function test_func encompasses four paths. Upon receiving the input $s1 = (*, *, *)$, the function takes a direct branch from `0x400638` to `0x400650`. ETM records this branch as an Atom element E by default and address `0x400650` as an Address element under the BB mode. For the input $s2 = ('b', *, *)$, which satisfies the conditional statement in line 3, the function bypasses the branch at `0x400638`. ETM denotes this with an Atom element N. Regardless of the mode, an execution path can be represented by a sequence of Address and Atom elements, such as $(N, E, \texttt{0x400650})$, starting from an entry. Hence, we can construct coverage based on these elements without disassembling the code.

3 REBUILDING COVERAGE IN HGF

3.1 Coverage-Feedback Mechanism

The coverage-feedback mechanism plays an essential role in HGF, which evaluates test cases but introduces the highest overhead. To better understand the challenges in HGF, we conduct a comprehensive review of its development, qualitatively compare existing tools, and list the details of their coverage-feedback mechanisms in Fig. 2.

From Fig. 2, the coverage-feedback mechanisms of existing tools can be categorized into three groups: 1) **Building branch-count coverage with disassembling code.** Early tools like PTFuzz [55] decode the trace data and use the disassembly code to recover the addresses of executed basic blocks for rebuilding the branch-count coverage. However, this approach incurs significant overhead and reduces the throughput. Though libxdc [38] and Armored-Edge [2] accelerate this process by implementing many micro-optimizations, such as heavy caching and branchless code, this way introduces more overhead than directly rebuilding coverage from trace elements. 2) **Building path coverage directly from the trace elements.** PTrix [14] proposed to build the path coverage by hashing trace elements without heavy disassembling. Armored-Path [2] followed this approach. However, the overly sensitive path coverage leads to a large number of selected seeds and the seed explosion problem, which poses heavy pressure on the scheduling algorithm. This causes the sensitive coverage to be less practical than branch coverage in certain scenarios. Moreover. 3) **Building branch coverage directly from the trace elements based on the hardware**

[1]Due to the diversity in the versions and implementations of ETMs on different SoCs, we focus on the ETM-v4 of Juno R2 in this paper.

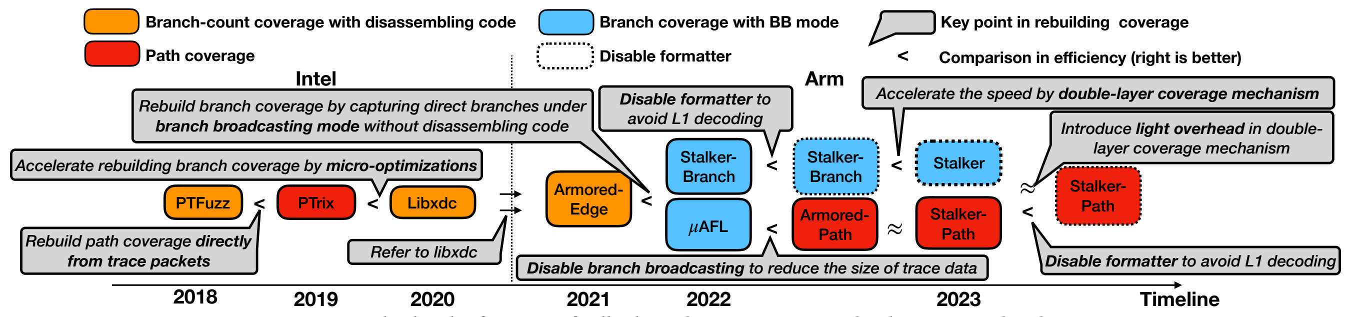
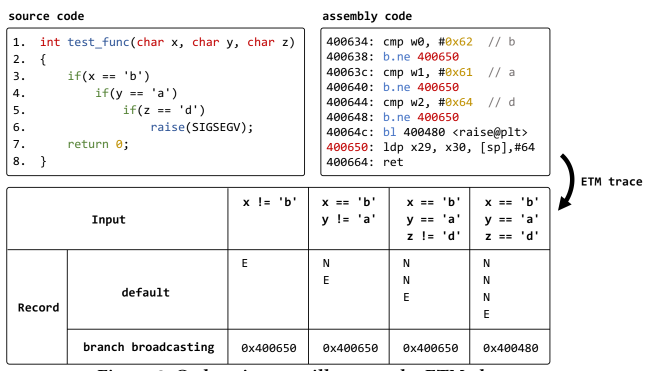

Branch-count coverage with disassembling code
Path coverage
Branch coverage with BB mode
Disable formatter
Key point in rebuilding coverage
< Comparison in efficiency (right is better)
Intel
Arm
Rebuild branch coverage by capturing direct branches under branch broadcasting mode without disassembling code
Disable formatter to avoid L1 decoding
Accelerate the speed by double-layer coverage mechanism
Introduce light overhead in double-layer coverage mechanism
Accelerate rebuilding branch coverage by micro-optimizations
PTFuzz
PTrix
Libxdc
Armored-Edge
Stalker-Branch
μAFL
Stalker-Branch
Armored-Path
Stalker
Stalker-Path
Stalker-Path
Rebuild path coverage directly from trace packets
Refer to libxdc
Disable branch broadcasting to reduce the size of trace data
Disable formatter to avoid L1 decoding
2018
2019
2020
2021
2022
2023
Timeline

Figure 2: The details of coverage-feedback mechanisms in existing hardware-assisted tools.



source code

```
int test_func(char x, char y, char z)
{
    if(x == 'b')
        if(y == 'a')
            if(z == 'd')
                raise(SIGSEGV);
    return 0;
}
```

assembly code

```
400634: cmp w0, #0x62  // b
400638: b.ne 400650
40063c: cmp w1, #0x61  // a
400640: b.ne 400650
400644: cmp w2, #0x64  // d
400648: b.ne 400650
40064c: bl 400480 <raise@plt>
400650: ldp x29, x30, [sp],#64
400664: ret
```

Input		x != 'b'	x == 'b' y != 'a'	x == 'b' y == 'a' z != 'd'	x == 'b' y == 'a' z == 'd'
Record	default	E	N E	N N E	N N N E
	branch broadcasting	0x400650	0x400650	0x400650	0x400480

Figure 3: Code snippet to illustrate the ETM elements.

features. μAFL [30] rebuilds branch coverage based on the BB mode of ETM to avoid heavy code disassembling. However, under the BB mode, the size of trace data increases, leading to heavier decoding overhead compared to the default mode.

Upon analysis and comparison of these tools, it is evident that the fastest approach to rebuild coverage is (2). However, it also introduces the seed explosion. Therefore, **the main challenge in efficiently rebuilding coverage in HGF is to strike a balance between efficiency and sensitivity.**

Furthermore, the presence of trace buffer overflow impairs the accuracy and stability of coverage rebuilt by many hardware-assisted fuzzers [2, 30], which weaken the fuzzing efficiency. For example, AFL-type tools [5, 14, 30, 55] calculate the checksum of the local bitmap to represent an execution path. The checksum is used in many mutation strategies. The trim strategy prunes the seed by selecting a smaller one with the same checksum to maintain high execution speed [53]. The deterministic strategies identify the unnecessary mutations on specific bytes by observing the checksum in conducting bitflip strategy [53]. The imprecise and unstable coverage may impact the calculation of checksum, limiting the efficiency of these strategies or causing the seed explosion. Hence, **it is important to prevent trace buffer overflow to ensure the effectiveness and stability of HGF.**

In conclusion, we argue that efficient HGF should meet two criteria: 1) **Rebuilding coverage with lightweight overhead while avoiding seed explosion;** 2) **Alleviating the trace buffer overflow.** To achieve (1), we propose a novel *double-layer coverage mechanism*. This mechanism employs a lightweight path coverage for executing test cases with high throughput, and a moderate branch coverage for filtering and selecting the seeds into the seed queue to prevent seed explosion. To achieve (2), we propose the *adaptive CPU frequency modulation mechanism*. This mechanism modulates the bandwidth of ETM by adjusting the CPU frequency, effectively avoiding trace buffer overflow.

3.2 Double-Layer Coverage Mechanism

Inspired by above analysis, we propose the double-layer coverage mechanism. **The fundamental design of this mechanism is to assign executing test cases and select the seeds to distinct coverages**. To efficiently execute the test cases, we employ a *lightweight path coverage* in the bottom layer. And we propose a *moderate branch coverage* in the top layer to select and add the seeds into the queue without introducing the seeds explosion. We will illustrate how to design these two coverages in Section 3.3.

Algorithm 1 Double-Layer Coverage Mechanism

```
repeat
    testcase = Mutation(seed)
    path_trace_bits = RunTarget(PATH, testcase)
    if Filter(path_trace_bits, useless_bitmap) then
        if HasNewBits(path_bitmap, path_trace_bits) then
            branch_trace_bits = RunTarget(BRANCH, testcase)
            if HasNewBits(branch_bitmap, branch_trace_bits) then
                AddToQueue(testcase)
                Update(useful_path_bitmap, path_trace_bits)
            else
                Update(useless_bitmap, path_trace_bits)
            end if
        end if
    end if
    if Purge(path_bitmap) then
        path_bitmap = useful_path_bitmap
    end if
until fuzzer exit
```

Algorithm 1 elaborates on this mechanism. Stalker executes each test case under the lightweight path coverage in the bottom layer. By comparing the path_trace_bits and path_bitmap of the path coverage, if a test case is identified as a new seed (defined as a **path seed**), Stalker proceeds to re-execute the test case using the top-layer moderate branch coverage and examine the corresponding branch_trace_bits and branch_bitmap. Finally, a test case will be considered useful and added to the seeds queue (defined as a **branch seed**) if it triggers new states under both layers of coverage.

Purge strategy. A test case marked as a path seed but not a branch seed may not contribute to covering more branches. We define such case as **useless path seed** and incorporate its path_trace_bits into the useless_bitmap to filter out these useless cases in subsequent testing. Since the path_bitmap is contaminated with useless path seeds in the function HasNewBits, we periodically remove these useless entries from the path_bitmap via the useful_path_bitmap, which records the bitmap of branch seeds under path coverage. This strategy can retain the sensitivity of the path coverage.

By employing the double-layer coverage mechanism, STALKER can execute test cases rapidly under the path coverage while preventing seed explosion by filtering out excessive path seeds and selecting only the branch seeds based on the branch coverage (Criteria (1)). This approach ensures efficient execution and effective seed selection, improving the overall performance of STALKER.

3.3 Rebuilding Coverage by Arm ETM

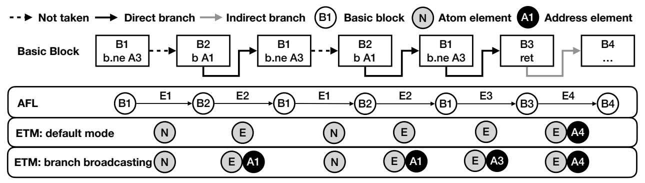


Figure 4: The runtime information tracing of AFL and ETM.

To design efficient and stabilized coverage, we begin by examining the control flow captured by AFL and ETM. Fig. 4 lists four basic blocks (B1-B4). The test case executes a path that covers seven basic blocks, including a loop. AFL captures all the blocks and calculates the hash values for six edges (E1, E2, E1, E2, E3, E4). In contrast, ETM generates an element sequence as $(N, E, N, E, E, E, A4)$ by default and $(N, E, A1, N, E, A1, E, A3, E, A4)$ under the BB mode. However, neither sequence contains all basic block addresses. Though we can recover these addresses by disassembling the code, this yields unacceptable overhead [14, 30].

Notice that the basic blocks are separated by branch instructions, while ETM generates an Atom element for each branch, regardless of whether it is taken or not. An Atom element N indicates the program executing consecutive basic blocks without taking a branch. We divide the element sequence into several slices by the Address elements and utilize the Atom elements within each slice to denote the edges. While the BB mode of ETM records direct branches, it also incurs additional decoding costs compared to the default mode. Considering this, we design two coverage strategies with different sensitivities and overheads for the two modes.

Moderate branch coverage. ETM records the destinations of all branch instructions under the BB mode. This results in an element sequence of the form $(..., E, AX, N, ..., N, E, AY)$. Within the slice between two Address elements, the program executes M consecutive basic blocks, where the number of Atom element N is $(M - 1)$. We can utilize the count of N elements to represent the number of edges. Notably, slices without any N elements, such as $(A1, E, A3)$, indicate that the program takes a branch from one block to another.

Inspired by this, we design a moderate branch coverage under the BB mode. As shown in Algorithm 2, each slice is assigned a hash value calculated as the sum of the destination address and the count of the N elements present in the slice. To ensure coverage is limited to the assigned address range (specifically, the text sections of the binary), we filter out addresses that fall outside of this range.

Algorithm 2 Moderate Branch Coverage
```
trace_bits = Malloc(MAP_SIZE)
prev_slice = 0
N_elements_nums = 0
repeat
    if packet.type == Atom then
        atom_sequence = Decode(packet)
        N_elements_nums += Count(atom_sequence)
    end if
    if packet.type == Address then
        addr = Decode(packet)
        if Filter(addr) then
            if N_elements_nums > MAX_NUM then
                N_elements_nums = MAX_NUM
            end if
            cur_slice = addr + N_elements_nums
            index = cur_slice ⊕ prev_slice
            trace_bits[index] += 1
            prev_slice = cur_slice ≫ 1
            N_elements_nums = 0
        end if
    end if
until decoder exit
```

Furthermore, the presence of loops is captured by ETM in the BB mode through repeated slices, as exemplified by $(A1, N, E, A1)$, with a counter indicating the repetition. As a result, this coverage is not highly sensitive to loops. However, it should be noted that recording all branches in this mode increases the decoding overhead. Therefore, we propose a lightweight coverage in the default mode.

Lightweight path coverage. Under the default mode, the Atom sequence within a slice may include multiple E elements. To uniquely identify each slice, we utilize a hash function to calculate the hash value of the Atom sequence.

We propose a lightweight path coverage in the default mode of ETM (Algorithm 3). Specifically, we utilize one bit to represent an Atom element (N is 0 and E is 1) and denote this slice as the sum of an Address element and the hash value calculated from the Atom sequence. This coverage strategy operates at a path granularity, making it more sensitive than branch coverage, as even slight differences in Atom sequences result in distinct hash values. Fortunately, the overhead associated with reconstructing this coverage is relatively light compared to the top-layer branch coverage.

Algorithm 3 Lightweight Path Coverage
```
trace_bits = Malloc(MAP_SIZE)
prev_slice = 0
hash_value = 0
repeat
    if packet.type == Atom then
        atom_sequence = Decode(packet)
        hash_value = Hash(atom_sequence, hash_value)
    end if
    if packet.type == Address then
        addr = Decode(packet)
        if Filter(addr) then
            cur_slice = addr + hash_value
            index = cur_slice ⊕ prev_slice
            trace_bits[index] += 1
            prev_slice = cur_slice ≫ 1
            hash_value = 0
        end if
    end if
until decoder exit
```

Branchless design. PTrix implemented a similar path coverage [14]. However, it executes a significant amount of conditional code, as the hash function is called for every 64 TNT bits or for every TIP packet. In our testing, we found that the numerous conditional branches introduce considerable overhead. In contrast to PTrix [14], our path coverage implements a branchless design that performs the hash operations after decoding each Atom packet. This design can significantly reduce the number of conditional branches and accelerate coverage rebuilding.

Filtering noisy packets. To ensure the stability of Stalker, we apply a filtering process to the Address elements. This step is necessary because certain noisy packets from ETM can introduce unrelated addresses among the tracing packets [30]. For example, an exception brings two Address packets of the entry and exit points, which may not be the branch destinations. Without filtering noisy packets, the same test case could produce different element sequences with each execution and be regarded as different seeds, referred to as *instability* in AFL [1]. *In our testing, Stalker remains stable in rebuilding coverage under 100K repeated executions.*

Disable formatter. As mentioned in Section 2.2, the L1 encoding of the ETR formatter can introduce additional decoding costs. Since we bind on a specific core to perform fuzzing, identifying the ID of ETM is unnecessary. To mitigate this redundant overhead, we disable the formatter. *Evaluation in Section 5.1 shows that it accelerates Stalker for* 1.4×.

3.4 Adaptive CPU Frequency Modulation Mechanism

The ETF buffer can potentially overflow due to the high bandwidth of trace packets generated by ETM. This bandwidth is determined by factors, including the executed instructions, the ETM configuration, and the program's execution speed.

To address this issue, we configure ETM to trace specific processes and limit the range of addresses in the text section to exclude unrelated instructions. Additionally, we set up ETM to focus on the program flow elements without other unrelated elements like timestamps. This configuration meets the requirements for rebuilding coverage and reduces the number of trace packets, minimizing the overhead of decoding data and slowing down the ETM bandwidth.

Furthermore, decreasing the ETM bandwidth can be achieved by reducing the CPU frequency to slow down program execution. While it is feasible to run programs at a minimal frequency to avoid overflow, this approach may hinder fuzzing speed. It is worth noting that some programs and test cases may not cause trace buffer overflow even under maximum frequency. Hence, we propose an *adaptive CPU frequency modulation mechanism (ACFMM)* to maintain a high CPU frequency and alleviate the overflow.

Our mechanism operates as follows: during testing, if there is frequent buffer overflow over time, the CPU frequency is reduced to prevent overflow. Conversely, when no overflow occurs for an extended period, the frequency is increased to accelerate execution. We adjust the frequency using the kernel driver interface, which introduces minimal overhead (about 64us-160us in Juno R2).

Algorithm 4 outlines the mechanism. It begins by checking the overflow_flag during each execution. If an overflow occurs, the CPU frequency (denoted as cur_cpu_freq_cov) is decreased, and the

Algorithm 4 ACFMM

```
1: repeat
2:     testcase = Mutation(seed)
3:     overflow_flag = RunTarget(COV, testcase)
4:     if COV == PATH then
5:         path_execs += 1
6:     end if
7:     if overflow_flag == TRUE then
8:         if COV == PATH then
9:             overflow_nums_path += 1
10:        end if
11:        no_overflow_num = 0
12:        Decrease(cur_cpu_freq_mode)
13:    else
14:        no_overflow_num += 1
15:        if no_overflow_num == INTERVAL then
16:            Increase(cur_cpu_freq_cov)
17:            no_overflow_num = 0
18:        end if
19:    end if
20:    INTERVAL = (path_execs / overflow_nums_path) / 5
21:    Limit(INTERVAL)
22: until fuzzer exit
```

no_overflow_num is reset to 0. Once the no_overflow_num reaches a predefined threshold value, the frequency is increased to enhance efficiency. In the default mode, the amount of trace data generated by ETM is greater compared to the BB mode. Consequently, the threshold for triggering overflow in the default mode is higher than that in the BB mode. To control the CPU frequency in these two modes and coverages, we use cur_cpu_freq_path and cur_cpu_freq_branch, respectively (uniformly shown as cur_cpu_freq_cov in Algorithm 4). Moreover, the threshold value INTERVAL influences the frequency of increasing the CPU frequency. A longer INTERVAL leads to a slower increase in CPU frequency, while a shorter INTERVAL maintains a high CPU frequency, potentially resulting in frequent overflows. To address this, we use the overflow rate to adaptively adjust the INTERVAL and limit its upper and lower bounds.

Overall, by balancing the efficiency and stability, the ACFMM is flexible in adjusting the CPU frequency to maintain high throughput and avoid overflow, which satisfies the Criteria (2).

4 DESIGN AND IMPLEMENTATION

4.1 Overview

We built Stalker on AFL [53]. As shown in Fig. 5, Stalker follows the framework of AFL, including the fuzzing engine, mutation strategies, and scheduling algorithms, but incorporates the coverage-feedback mechanism through CoreSight. In the user space, we employ the *double-layer coverage mechanism* to reconstruct coverage and select branch seeds (Section 3.2). To avoid trace buffer overflow, we take the *adaptive CPU frequency modulation mechanism* (Section 3.4). Moreover, we implement a decoder based on ptm2human [26] to decode the trace data. In the kernel space, the registers of CoreSight are mapped to the user space through the driver. This enables convenient control of ETM and ETR for Stalker.

Workflow. From Fig. 5, Stalker begins by configuring ETM and setting up the fork server. Then it selects a seed from the seed queue and enters the fuzzing loop. Before executing each test case, Stalker notifies the fork server to fork a child process and retrieves the process ID (①②③). After enabling ETM to trace this process under the default mode, Stalker notifies this process to execute the

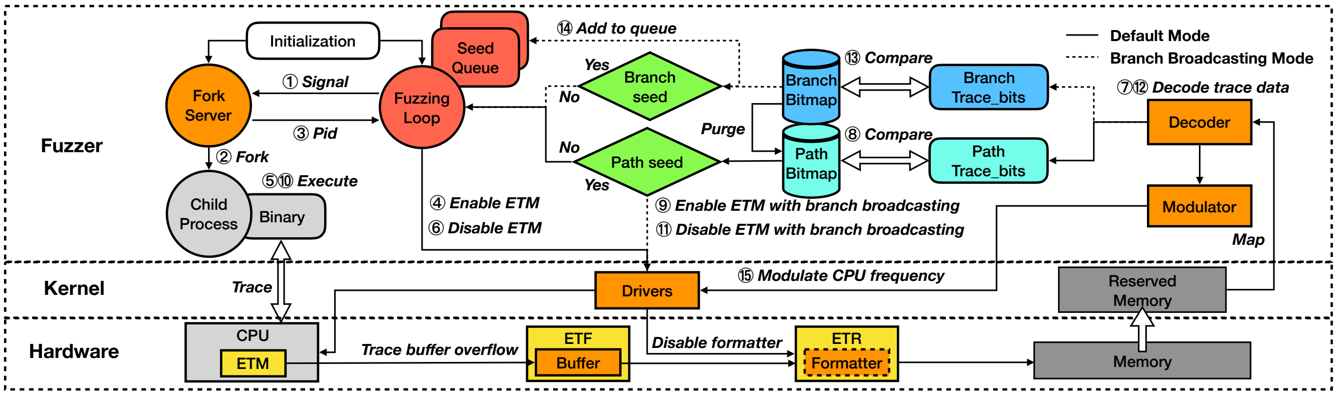


Figure 5: The architecture and workflow of Stalker.

program (④⑤). Once the execution is completed, Stalker disables ETM and reads and decodes the trace data from memory (⑥⑦). Then, Stalker rebuilds the coverage using the *lightweight path coverage* in bottom layer and compares the path bitmap (⑧). If a test case is regarded as a path seed, Stalker will enable the BB mode of ETM, re-execute the case, and examine it using the top-layer *moderate branch coverage* (⑨⑩⑪⑫⑬). Only the branch seed is added to the seed queue (⑭). Stalker also examines the overflow and adaptively modulates the CPU frequency by ACFMM(⑮).

4.2 Technical Details

Fork server. Efficient fuzzers employ a fork server to ensure the fuzzed program only goes through `execve()`, linking, and library initialization only once [52], which can avoid those overhead significantly. Similar to PTrix and Armored-CoreSight [2, 14], we patch this mechanism in the tested binary by patchelf [17].

CoreSight driver. For faster retrieval of trace data, we reserve a 256MB physical memory region during the kernel initialization and force ETR to copy the trace data from the buffer to this region. Stalker can then swiftly fetches trace data through shared memory mapped from this reserved space. To disable the formatter, we set the EnFt and EnTI bits of the formatter control register to 0.

Decoder. The decoder of Stalker is developed based on a open-source tool, namely ptm2human [26]. To improve its efficiency, we employ a hash table for rapid identification of trace packets, and skip the decoding of unrelated packets. Specifically, according to the ETM technical manual [7], the first byte of a trace packet determines its category. We utilize a table to establish the mapping relationship between them. Since only Atom, Address, and Exception packets contain the Address and Atom elements, we focus on decoding these three packet types while excluding other unrelated packets. Atom packets have a size of one byte and may contain multiple Atom elements, so we create a table to map bytes to Atom sequences.

Finally, we implement Stalker on the Arm Juno R2 development board (an official platform released by Arm) [6]. Our modifications to AFL, ptm2human, and Linux kernel (including CoreSight driver) consist of approximately 1, 220, 871, and 41 LoCs, respectively.

5 EVALUATION

We conducted systematic experiments to evaluate the effectiveness and efficiency of Stalker in addressing the following questions:

Q1: How about the efficiency of rebuilding coverage by Stalker compared with other hardware-assisted tools?

Q2: What is the overall performance of Stalker in terms of throughput and coverage compared to other tools?

Q3: Does the double-layer coverage mechanism enhance the efficiency of Stalker while avoiding seed explosion?

Q4: How about the effects brought by trace buffer overflow? Does the ACFMM effectively avoid overflow?

Q5: Do the other strategies in Stalker enhance the efficiency and stability of Stalker?

We evaluated Stalker on Arm Juno R2 development board running Linux-5.3 (8GB RAM) with a Cortex-A72 processor cluster (0.6GHz-1.2GHz, 2MB L2 cache) and a Cortex-A53 cluster (0.45GHz-0.95GHz, 1MB L2 cache) [6]. To compare Stalker with PT-based tools, we utilize a Centos server with 48 cores (Intel Xeon E5-2650 with 2.20GHz) and 64GB RAM.

5.1 Efficiency of Rebuilding Coverage

Compared tool. We verify the efficiency of existing methods in rebuilding coverage reported in Fig. 2. We evaluated the two types of coverage in Stalker, namely Stalker-Branch and Stalker-Path, and compared them with AFL-QEMU [53], AFL-QEMU++ [20], Armored-CoreSight (Edge and Path) [2], μAFL [30], PTrix [14], and libxdc [38]. AFL-QEMU is a popular binary-only fuzzer. Some researchers have improved its performance and integrated it into AFL++[1] [10, 20]. We applied this update to AFL (AFL-QEMU++). Armored-CoreSight and μAFL are SOTA Arm-based hardware-assisted tools. To ensure a fair comparison, we adapted the rebuilding-coverage component of Armored-CoreSight (CoreSight-decoder [3]), which uses edge and path coverage, to work with Stalker on our platform. For μAFL, which is based on ETM-v3.5 and focuses on testing drivers, we emulated it by enabling only moderate branch coverage and the formatter in Stalker as it takes the similar way as our branch coverage to rebuild coverage, denoted as Stalker-Branch-Fmt. Since neither Armored-CoreSight nor μAFL explicitly propose to turn off the formatter, we enable it while running them.

[1] AFL++ also has an immature Frida mode, but it fails in our platform.

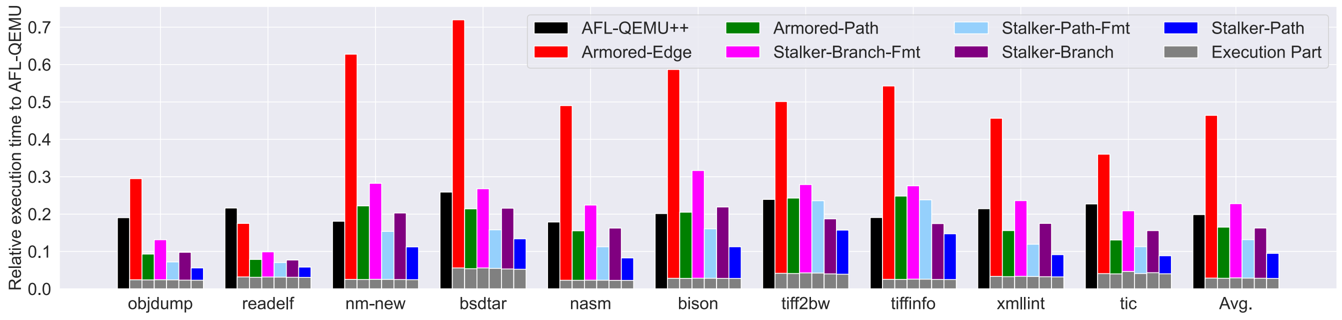


Figure 6: Normalized execution time of all tools with AFL-QEMU as the baseline (lower is better). The grey and remaining bars denote the execution and decoding part of hardware-assisted tools in executing test cases, respectively.

We also explore the overhead brought by formatter via comparing Stalker with Stalker-Fmt (i.e., Stalker with the formatter).

Since PTrix and libxdc cannot be directly deployed on Arm Juno R2 [14, 38], we followed the experiments conducted in libxdc [38] and ported the lightweight path coverage of Stalker in libxdc[1] to compare the decoding time on the Centos server.

Tested programs. We conducted experiments on 10 real-world binaries selected by following the recent binary-only fuzzing works [14, 15, 36], listed in Table 1. They were selected according to the following considerations: 1) High level of diversity to evaluate the applicability of Stalker; 2) Numerous branches to distinguish the performance of fuzzers; 3) Running in a real environment where ETM can directly trace them. Moreover, we use the same binaries patched with the fork server for the Arm-based tools to conduct fair experiments. For the PT-based tools, we follow the benchmark from libxdc [38], which includes 8 programs.

Table 1: Target binaries evaluated in our evaluation.

Program	Version	Size	Format
objdump –dwarf-check -C -g -f -dwarf -x @@	binutils-v2.37	11MB	elf
readelf -a @@	binutils-v2.37	4MB	elf
nm-new -C @@	binutils-v2.37	5.8MB	elf
bsdtar -xf @@ /dev/null	libarchive-3.5.2	3.4MB	tar
nasm -f elf -o sample @@	nasm-2.15.05	3.0MB	text
bison @@	bison-3.8	2.6MB	text
tiff2bw @@ /dev/null	tiff-4.3.0	1.5MB	tiff
tiffinfo @@	tiff-4.3.0	1.6MB	tiff
xmllint @@	libxml2-2.9.10	100KB	xml
tic @@	ncurses-6.3	260KB	text

Setup. Referring to the evaluation in PTrix [14], we fuzzed the 10 binaries by AFL-QEMU++ for 24 hours, collected the discovered seeds as the input corpus, and reran them to measure the execution speed. The execution time for each test case was defined as the time taken to execute the test case and rebuild coverage. We ran all tools except PT-based on tools under the maximal frequency of one Cortex-A53 core, repeated this experiment 5 times, and filtered out the overflow cases for a fair comparison[2].

For PTrix and libxdc, to explore their efficiencies in different sizes of trace data, we varied the sizes from 512KB to 2GB, conducted the experiments on our Centos server, and repeated each trial 5 times. Considering the cold caches in libxdc when running a single input, we iteratively ran libxdc 26 rounds. We recorded the first running time as libxdc-cold and calculated the average time of the remaining 25 rounds as libxdc-hot.

Compared with Arm-based and DBI-based tools. We calculated the average execution time of these tools and normalized the result using AFL-QEMU as the baseline, presented in Fig. 6. From Fig. 6, we can conclude that:

1) **Disabling the formatter has a significant positive impact on the efficiency of rebuilding coverage.** By disabling the formatter, the execution time of Stalker-Branch and Stalker-Path decreases by approximately 1.4× compared to when the formatter is enabled. 2) **Stalker is faster than Armored-CoreSight in rebuilding coverage with the same granularity.** Stalker-Path and Stalker-Path-Fmt are 1.74× and 1.25× faster than Armored-Path, respectively. These improvements are attributable to disabling the formatter mechanism and micro-optimizations in our decoder (e.g., branchless design), bringing 1.39× and 1.25× acceleration, respectively. Stalker-Branch and Stalker-Branch-Fmt also outperform Armored-Edge by 2.81× and 2.04×, respectively. 3) **Rebuilding branch coverage by BB mode is faster than disassembling code.** Even with enabling the formatter, Stalker-Branch-Fmt remains 2.04× faster than Armored-Edge. This highlights the advantage of Stalker-Branch in rebuilding branch coverage directly from the trace data generated by ETM under BB mode, compared to Armored-Edge's reliance on heavy code disassembling. 4) **Rebuilding coverage directly from trace data without enabling BB mode or disassembling code is the fastest way.** Though Stalker-Path and Stalker-Branch rebuild coverage directly from the trace data, the former is 1.7× faster than the latter. The reason is that Stalker-Branch needs to decode more trace data in rebuilding coverage. 5) **Stalker is significantly faster than QEMU-based fuzzers on our platform.** Stalker-Path is 10.5× and 2.09× faster than AFL-QEMU and AFL-QEMU++, respectively. Similarly, Stalker-Branch outperforms AFL-QEMU and AFL-QEMU++ by 6.14× and 1.22×, respectively.

Compared with PTrix and libxdc. We report the average results in Fig. 7(c) and some detailed results in Fig. 7(a)(b). From Fig. 7(c), Stalker-Path is 1.63× and 2.2× faster than libxdc-hot and PTrix with the shortest decoding time, respectively. This supports our analysis in Section 3.1. Libxdc-cold performs worse than PTrix on small files (less than 8MB) but better than PTrix on large files (more than 8MB). This is because the cache mechanism in libxdc

[1]Since Intel PT does not support the BB mode in ETM to capture the addresses of direct branches, we only port the path coverage in Stalker to libxdc.
[2]Considering the code caches in Armored-Edge when running the first test case, we also filter out the first case to ensure fairness in evaluation.

Table 2: Average throughput, path coverage, and branch coverage of six tools. T: throughput. P: path coverage. B: branch coverage.

Programs	**AFL-QEMU**			**AFL-QEMU++**			**Armored-Edge**			**Armored-Path**			**Stalker-Branch-Fmt**			**Stalker**		
	T	P	B	T	P	B	T	P	B	T	P	B	T	P	B	T	P	B
objdump	3.69M	5,483	12,303	7.05M	8,418	12,875	7.99M	5,277	12,502	18.0M	2,199	9,027	17.01M	7,627	**14,415**	**21.6M**	**9,240**	14,038
readelf	4.89M	5,222	7,426	9.27M	6,106	7,561	16.64M	6,358	8,598	17.28M	1,715	5,711	22.46M	6,639	8,910	**31.32M**	**7,271**	**9,380**
nm-new	4.28M	1,065	3,198	10.52M	2,295	4,395	4.95M	1,058	3,249	19.79M	591	2,798	14.86M	2,478	4,765	**20.09M**	**5,043**	**6,218**
bsdtar	3.55M	1,627	4,373	8.78M	**2,233**	**5,013**	6.95M	1,701	4,752	11.75M	193	2,738	8.62M	1,879	4,812	**15.22M**	2,084	4,999
nasm	0.56M	3,679	5,193	1.75M	5,291	5,368	0.93M	1,706	4,533	**3.83M**	1,279	3,925	1.33M	4,477	5,429	2.2M	**5,423**	**5,492**
bison	1.44M	1,465	5,730	1.92M	2,235	6,995	0.77M	805	2,848	3.74M	814	2,785	2.46M	2,197	7,004	**5.71M**	**2,286**	**7,071**
tiff2bw	5.7M	1,459	3,533	15.37M	1,939	4,233	6.1M	1,377	3,419	11.24M	547	2,665	18.28M	1,717	4,593	**24.69M**	**2,055**	**4,820**
tiffinfo	4.69M	2,072	3,936	12.13M	**2,654**	4,174	7.26M	2,152	4,171	11.56M	626	3,083	13.34M	2,295	4,223	**20.17M**	2,466	**4,289**
xmllint	2.95M	2,554	5,493	5.58M	2,972	5,776	5.6M	2,598	5,745	**14.84M**	1,192	4,898	7.81M	3,021	**5,883**	12.31M	**3,063**	5,876
tic	2.72M	2,303	2,699	5.03M	2,715	2,959	6.35M	2,163	2,878	**13.84M**	753	1,660	6.96M	2,843	3,016	11.16M	**2,979**	**3,088**
Avg.	3.45M	2,692	5,388	7.74M	3,685	5,934	6.35M	2,519	5,269	12.59M	990	3,929	11.31M	3,517	6,305	**16.45M**	**4,191**	**6,527**
%-Chg	+376.8%	+55.7%	+21.1%	+112.5%	+13.7%	+10.0%	+159.1%	+66.4%	+23.9%	+30.7%	+323.3%	+66.1%	+45.4%	+19.2%	+3.5%	-	-	-

provides less improvement when decoding fewer trace data. However, even after iterative running, libxdc-hot is still 1.63× slower than Stalker-Path on all sizes of trace data.

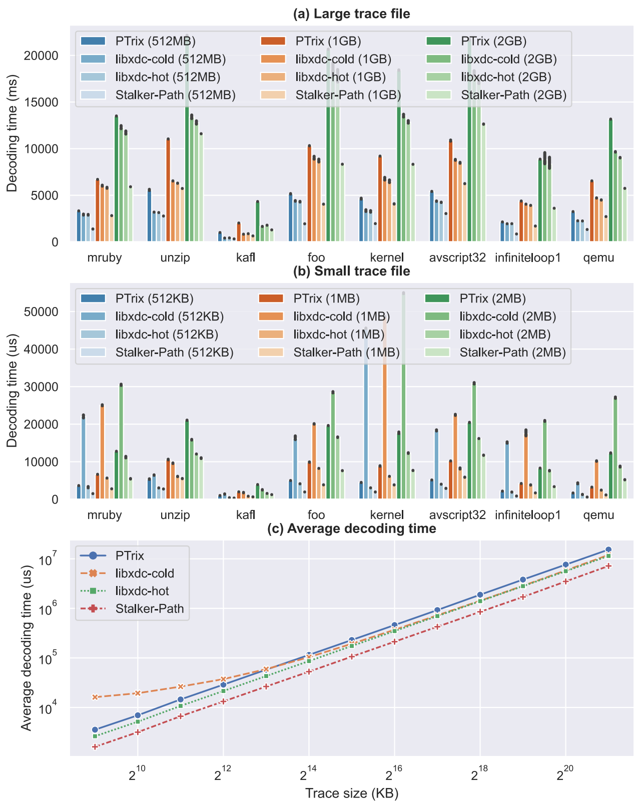


Figure 7: Decoding time of PTrix, libxdc-cold, libxdc-hot, and Stalker-Path on different trace file sizes (lower is better).

Notably, even building the path coverage without disassembling code, Stalker-Path is about 2.2× faster than PTrix. This is due to our micro-optimizations, particularly the branchless design.

Response to Q1: By rebuilding coverage directly from the trace data and disabling formatter, Stalker rebuilds the same granularity coverage more efficiently, 2.81×, 1.74×, and 1.4× faster than Armored-Edge, Armored-Path, and μAFL, respectively. Stalker also outperforms PTrix and libxdc by 2.2× and 1.63×, respectively.

5.2 Performance of Stalker

Metric. Since we test these binaries in a development board with limited computation resources, it is difficult to detect crashes. We refrain from using the number of crashes as a metric, aligning with the evaluation in SNAP [16]. Instead, we evaluate all the tools with the throughput, path coverage, and branch coverage. To calculate the coverage in the unified metrics, we have retained all the seeds discovered by these tools and re-executed them using AFL-QEMU.

Setup. Referring to [13, 29, 51], we fuzzed each program in Table 1 for 5 rounds of 24 hours on a single Cortex-A53 core to reduce the impact of randomness, with the initial seed provided by AFL. Considering the relatively lower performance of our board compared to servers and the effectiveness of AFL's random strategies in achieving broader coverage in a shorter time [49, 51], we disabled deterministic mutations while running all the tools to reach the coverage saturation point in 24 hours as soon as possible. In addition, we ran AFL-QEMU and AFL-QEMU++ at the maximal CPU frequency and ran Arm-based tools with ACFMM.

Results. We report the detailed results in Table 2. Stalker conducts an average of 16.38M test cases, which is 4.75×, 2.12×, 2.58×, 1.30×, and 1.45× more than AFL-QEMU, AFL-QEMU++, Armored-Edge, Armored-Path, and Stalker-Branch-Fmt, respectively. Benefiting from the high throughput, Stalker explores the greatest number of paths (4, 191) and branches (6, 527) across the 10 programs. Even utilizing the same branch coverage to filter seeds, Stalker outperforms Stalker-Branch-Fmt due to its higher speed, which is achieved by disabling formatter and incorporating a double-layer coverage mechanism. Although Armored-Path executes numerous test cases, it achieves the least coverage in terms of 990 paths and 3, 929 branches. This can be attributed to seed explosion caused by its sensitive path coverage, which traps Armored-Path in exploring localized code regions. Armored-Edge performs worse than Stalker due to its heavy overhead in rebuilding coverage.

We also depicts the throughput and branch coverage graphs of Stalker and five other tools on 10 programs over 24 hours in Fig. 8. From the bar charts, the blue bar, which denotes Stalker, is higher than others on most programs. Only on nasm, xmllint, and tic, the throughput of Stalker is less than that of Armored-Path. We deeply analyze the reason in Appendix **??**. After analyzing the results, the reason is that Armored-Path is trapped by seed explosion and generates numerous test cases, which leads to that Armored-Path is unable to trigger the logical codes as deeply as Stalker and executes more lightweight test cases than Stalker. As a result, Armored-Path covers the least coverage among these tools. In the term of branch coverage, as can be seen from the line charts, the blue line is higher than other lines on almost all programs except objdump, indicating that STALKER covers the most branches among the six tools and reaches the upper coverage significantly faster than the others on most binaries.

Response to Q2: STALKER surpasses AFL-QEMU, AFL-QEMU++, Armored-Edge, Armored-Path, and μAFL with higher throughput and covering 55.7%, 13.7%, 66.4%, 323.3%, and 19.2% more paths and 21.1%, 10.0%, 23.9%, 66.1%, and 3.5% branches, respectively.

5.3 Double-Layer Coverage Mechanism

To better understand the sensitivities of the coverage metrics and evaluate the efficiency and effectiveness of the double-layer coverage, we collected the seeds in the queue found by hardware-assisted tools in Section 5.2, and filtered them by AFL-QEMU. Then we counted the number of seeds before and after the filtering as N_q and N_f, respectively, and defined the **sensitive ratio** as N_q/N_f to evaluate the coverage sensitivity. Particularly for STALKER, we also count the number of path seeds and calculate the ratio between the path seeds and branch seeds. Table 3 lists the detailed results. By analyzing these results, we can conclude that:

1) **Double-layer coverage can keep STALKER executing most of test cases under lightweight path coverage.** As per our design, each path seed found by STALKER is re-executed under the moderate branch coverage, which constitutes the primary overhead of our double-layer mechanism. From Table 3 and Table 2, STALKER selects an average of $1.62M$ path seeds from $16.38M$ test cases, implying that only 9.9% of the test cases are repeatedly executed under the heavy branch coverage. 2) **Double-layer coverage can avoid the seed explosion.** Using this mechanism, STALKER filters $6,124$ branch seeds from the $1.62M$ path seeds and adds them into the queue, effectively avoiding the seed explosion. 3) **The sensitivity of our branch coverage is comparable to that of the branch-count coverage.** These $6,124$ branch seeds are regarded as $4,038$ unique seeds under the branch-count coverage of AFL, with a sensitive ratio of only 152%. In comparison, while Armored-Edge employs the same branch-count coverage as AFL, its sensitive ratio (212%) exceeds that of AFL and STALKER due to instability within its rebuilding-coverage algorithm. To verify this, we collect the original seeds found by Armored-Edge and rerun them by Armored-Edge to filter them once again. Armored-Edge averagely selects $2,811$ seeds from $5,343$ original seeds, approximately to that of AFL-QEMU ($2,519$). This evidence supports the existence of seed explosion introduced by the instability of Armored-Edge.

Response to Q3: Our double-layer coverage mechanism enables STALKER to execute most of test cases under the lightweight path coverage and select seeds in a branch-granularity coverage, effectively avoiding the seed explosion.

5.4 Effectiveness of ACFMM

In this section, we outline some experiments to explore the negative effects of trace buffer overflow in fuzzing and prove the effectiveness of ACFMM.

Negative effects of overflow. As stated in Section 3.4, the trace buffer overflow may impair the deterministic mutation strategies of AFL, such as introducing some unnecessary mutations. Therefore, we disabled the ACFMM and ran STALKER under several configurations of CPU cores (e.g., Cortex-A53-0.45GHz) to test tiff2bw with one initial seed for one cycle of all deterministic strategies (i.e., from bitflip to auto extras). We reported the results in Table 4.

Table 3: Numbers of selected and filtered seeds of the hardware-assisted tools, respectively. The right column denotes the number of seeds filtered by AFL in the branch-count coverage. The two left columns in STALKER, and one in others represent the number of seeds selected by them.

Binary	Armored-Edge	Armored-Path	STALKER-Branch-Fmt	STALKER (Path Seed/Branch Seed)
objdump	8,820/5,277	51.55K/2,199	9,850/7,627	0.42M/10.63K/8,476
readelf	9,271/6,358	67.05K/1,715	9,335/6,639	1.54M/10.4K/7,345
nm-new	3,202/1,058	56.24K/591	3,781/2,478	0.35M/7,576/4,850
bsdtar	5,750/1,701	64.91K/193	3,034/1,879	4.75M/3,225/2,043
nasm	4,156/1,706	83.29K/1,279	7,324/4,477	0.48M/8,164/4,889
bison	3,367/805	66.16K/814	3,388/2,197	1.11M/3,627/2,282
tiff2bw	2,924/1,377	60.17K/547	2,371/1,717	2.12M/2,566/1,842
tiffinfo	4,135/2,152	59.86K/626	3,353/2,295	1.93M/3,513/2,422
xmllint	6,641/2,598	0.2M/1,192	5,221/3,021	2.18M/5,509/3,115
tic	5,164/2,163	0.26M/753	5,330/2,843	1.28M/6,032/3,118
Avg.	5,343/2,519	97.37K/990	5,298/3,517	1.62M/6,124/4,038
N_q/N_f	212%	9835%	151%	26392%/152%

Config	Time(s)	Speed(/s)	Throughput	Overflow	Skipped
A53-0.45GHz	506	132.17	66,878	0	153,371
A53-0.8GHz	307	217.84	66,878	0	153,371
A53-0.95GHz	266	251.42	66,878	0	153,371
A72-0.6GHz	399	271.62	108,378	105,875	118,414
A72-1.0GHz	233	439.17	102,326	100,046	118,376
A72-1.2GHz	193	517.06	99,793	97,581	118,376

Table 4: Average testing time, execution speed, the number of executed test cases and overflow test cases, and skipped mutations of STALKER under different configurations in 5 trials.

From Table 4, the execution speed of STALKER on Cortex-A72 is faster than that on Cortex-A53 due to the higher CPU frequency and larger L2 cache. Hence, the overflows when STALKER runs on Cortex-A53 are much less than those on Cortex-A72. STALKER skips fewer mutations and executes more test cases on Cortex-A72. Though it executes test cases faster on A72-0.6GHz than on A53-0.95GHz and A53-0.8GHz, the time to finish the deterministic stage of STALKER on A72-0.6GHz is $399s$, longer than on A53-0.95GHz and A53-0.8GHz. In conclusion, the buffer overflow introduces inaccuracy in rebuilding coverage and impacts the efficiency of the mutation strategies in AFL.

Effectiveness of ACFMM. To demonstrate the effectiveness of our mechanism in improving throughput and avoiding overflow, we tested nasm using the same configuration as described in Section 5.2 but on one Cortex-A72 core. As a control, we ran STALKER without ACFMM at the minimal and maximal CPU frequency of Cortex-A72 (i.e., A72-0.6GHz and A72-1.2GHz, respectively).

Table 5 lists the detailed results of five 24-hour trials. STALKER achieves the highest coverage and reaches a high throughput with the least overflows (2.46%). In contrast, STALKER-A72-1.2GHz executes the most test cases but suffers 40.92% overflows, resulting in the lowest coverage. Our ACFMM reduces the overflow percentage from 40.92% to 2.46%. Intuitively, STALKER-A72-1.2GHz should reach the highest coverage. However, affected by the imprecise and unstable coverage rebuilt under numerous overflows, it mistakenly

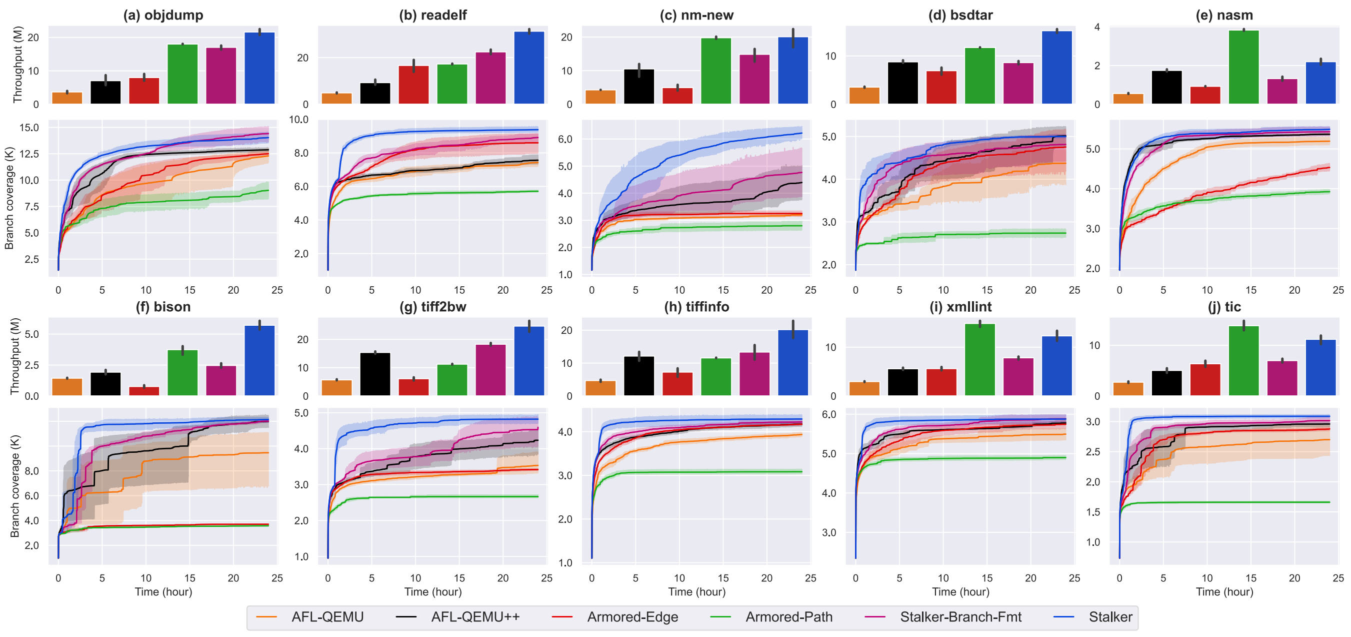


Figure 8: Throughput and branch coverage of six tools over 24 hours. Bar: mean of throughput. Solid lines: mean of the coverage. Line shadows: 95% confidence intervals for 5 fuzzing rounds.

Table 5: Average branches, number of branch and path seeds (B/P seeds), throughput, and overflows of Stalker, Stalker-A72-0.6GHz, and Stalker-A72-1.2GHz on nasm. Numbers in the brackets of Throughput: percentages of the test cases executed under 0.6GHz, 1.0GHz, and 1.2GHz.

Config	Branch	B/P Seeds	Throughput(0.6/1.0/1.2)	Overflow
ACFMM	5,649	9,202/890K	3.93M(58%/22%/21%)	96,909(2.46%)
A72-0.6GHz	5,435	8,465/507K	2.23M(100%/-/-)	0.23M(10.34%)
A72-1.2GHz	5,188	22,073/1.17M	4.12M(-/-/100%)	1.68M(40.92%)

selects more seeds and adds 22, 073 branch seeds to the queue. This leads to seed explosion. To prove this, we filtered these branch seeds by Stalker-Branch under Cortex-A53-0.45GHz to avoid overflow. Stalker-Branch only regards these 22, 073 seeds as 4, 676 seeds.

Since Stalker-A72-0.6GHz runs at the minimal frequency, it is expected to encounter the least overflows. However, it is interesting to note that overflows in Stalker-A72-0.6GHz are more than those in Stalker with ACFMM. Upon further analysis of the seeds, we have discovered that the branch seeds discovered by Stalker-A72-0.6GHz are much heavier than those of Stalker. By enabling the ACFMM, Stalker executes lightweight test cases under a higher frequency than 0.6GHz. Consequently, the execution time of some cases under Stalker is shorter than under Stalker-A72-0.6GHz. Since the scheduling algorithm in AFL prioritizes faster seeds [53], Stalker allocates more energy to these seeds and generates a greater number of lightweight test cases compared to Stalker-A72-0.6GHz, which leads to fewer overflows.

We analyze the average execution speed and the number of overflow and present the results in Fig. 9. From Fig. 9(a), the average execution speed of Stalker-A72-0.6GHz slows significantly after 5 hours, which illustrates that Stalker-A72-0.6GHz generates lots of heavy test cases in the later stage. Therefore, as can be seen in Fig. 9(a), its number of overflow is negligible in the early stage but rapidly increases after 5 hours. In contrast, as can be seen in Fig. 9(b),

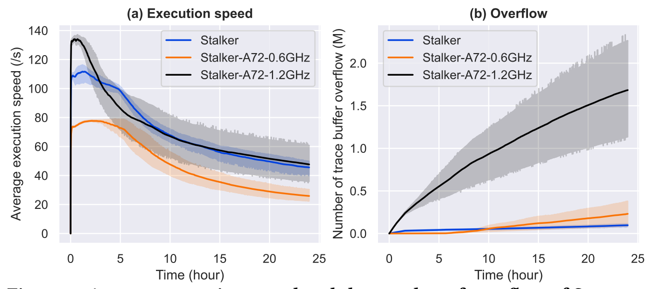


Figure 9: Average execution speed and the number of overflow of Stalker under different configurations on nasm. Solid lines: mean. Line shadows: 95% confidence intervals.

the number of the overflow of Stalker with ACFMM increases very slowly. Moreover, Stalker maintains a high execution speed by generating more lightweight test cases and running under a higher CPU frequency than Stalker-A72-0.6GHz. Therefore, the overflow rate of Stalker with ACFMM is lower than that of Stalker-A72-0.6GHz.

Response to Q4: The overflows impact the precision and stability of coverage, incurring seed explosion. Our ACFMM effectively alleviates the overflows and improves the performance of Stalker.

5.5 Other Strategies

We also evaluated other strategies to prove their efficiency and stability, including the purge strategy and filtering noisy packets.

Filter mechanism in rebuilding coverage. We introduce one critical point in decoding the trace data and rebuilding coverage: filtering the noisy information. To illustrate the stability ensured by it, we built Stalker-Branch-DF and Stalker-Path-DF with disabling the filtering mechanism, ran them by one test case, and repeated

100K times under Cortex-A53-0.45GHz. Then we recorded the number of recognized paths, IRQ exceptions, and the execution speed (times/sec) in Table **??**. Considering that xmllint and tic are stateful programs, we skip testing them. From Table **??**, Stalker-Branch and Stalker-Path are stable in rebuilding coverage without mistakenly detecting any other paths. Without this mechanism, the more exceptions, the more noisy paths are introduced.

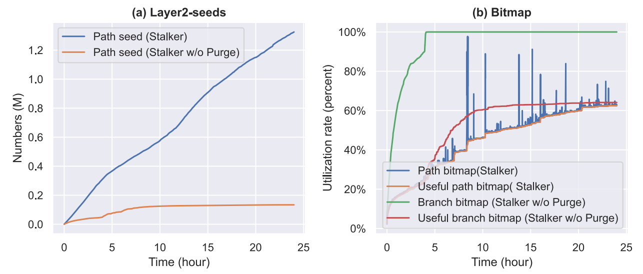


Figure 10: Utilization rate of path bitmap and number of path seeds of Stalker and Stalker w/o Purge on readelf.

Purge strategy. To prove the effectiveness of the purge strategy, we disabled this strategy (Stalker w/o Purge) and compared it against Stalker on readelf under the configurations in Section 5.2. The branch coverage of Stalker w/o Purge is significantly less than that of Stalker and remains stagnant after 3 hours, covering an average of 7,412 branches, significantly less than Stalker with covering 9,380 branches.

Since the coverage is calculated by re-executing the seeds, we analyzed the number of path seeds of Stalker and Stalker w/o Purge during a 24-hour trial in Fig. 10(a). The number of Stalker w/o Purge increases slowly in the later stage and is less than that of Stalker. As outlined in Section 3.2, to maintain the sensitivity of the path_bitmap, we purge it with the useful_path_bitmap.

Thus, we plot the utilization rate of these bitmaps (i.e., the percentage of non-zero bytes) in Fig. 10(b). Since Stalker w/o Purge disables the purge strategy, the path_bitmap is polluted by useless path seeds, and its entries are almost occupied 100% after 4 hours. In contrast, the utilization rate of the path_bitmap in Stalker is usually consistent with the useful_path_bitmap, making Stalker always sensitive to discovering path seeds. In conclusion, the purge strategy maintains the sensitivity of Stalker for discovering new seeds and improves the coverage.

Response to Q5: The purge strategy and filtering noisy packets can enhance the efficiency and stability of Stalker.

6 DISCUSSION

Rebuilding coverage in HGF. During our experiment, we observed certain performance differences among the different methods in various scenarios. For instance, when dealing with lightweight test cases, the overheads of Stalker and Armored-CoreSight were quite similar. However, when faced with heavy cases, reconstructing path coverage directly from trace data (e.g., Stalker-Path) proved to be significantly faster compared to other methods, including μAFL [30] and libxdc [38]. Nevertheless, the discrepancy in overheads between rebuilding path coverage and branch coverage forms the foundation of our double-layer mechanism. If the overhead of rebuilding path coverage is comparable to that of branch coverage, the fuzzer could directly focus on rebuilding branch coverage without implementing the double-layer coverage mechanism for efficient testing. However, since such cases are relatively rare, the double-layer coverage mechanism remains highly applicable and efficient. Furthermore, in comparing Stalker-Branch and Armored-Edge, we recommend that developers prioritize hardware features over code disassembling when aiming to rebuild branch coverage.

Generality. The implementation of CoreSight can vary across different SoCs. For example, the version of ETM in μAFL[30] is v3.5, which utilizes one bit in the P-header packet to denote whether an instruction is executed or not. Consequently, directly deploying Stalker on other Arm platforms may not be straightforward. However, in this paper, we focus on addressing the summarized challenges in HGF and present Stalker as a template to showcase the design of a SOTA hardware-assisted fuzzer. We believe that the design philosophy of Stalker can be referred to by some other tools, including those based on Intel PT, to solve common issues like low throughput, seed explosion, or buffer overflow. For example, Armored-CoreSight [2] can potentially enhance speed by disabling the formatter. Though Intel PT is unable to record the direct branches, it may be feasible for PTrix [14] to incorporate branch coverage of libxdc [38] and implement a double-layer coverage mechanism to mitigate seed explosion. This is based on Intel PT and requires additional work, which is out of our scope.

Parallelism. In this paper, we focus on enhancing HGF in single-core fuzzing. Limited by the implementation of CoreSight on specific devices where the trace data generated by multiple ETMs is integrated into only one ETF and ETR, both Stalker and Armored-CoreSight may currently unsupport parallel fuzzing[2]. It does not mean that the practicability of Stalker or the other Arm-platform tools may be limited in the multi-cores platform. Customizing CoreSight with multi-ETRs may provide the possibility to enhance Stalker. It is also feasible to improve Stalker with the parallel decoding architecture proposed in PTrix[14] for utilizing the multi-cores computation resources, which requires additional design and lots of engineering works and remains our future work.

7 RELATED WORK

Dynamic binary instrumentation in fuzzing. Recent works have improved the performance of DBI in AFL++ with an updated QEMU mode and Frida mode [10, 20]. Nevertheless, our evaluation shows that Stalker surpasses the updated QEMU mode in single-core fuzzing. While DBI techniques support parallel fuzzing and can leverage higher-performance platforms to utilize more computational resources, hardware tracing holds the advantage of extracting control flow with strong ability. For example, recent works employ ETM to directly trace the drivers even trusted applications on some devices [30, 37, 43, 44, 48]. Consequently, for specific targets, utilizing hardware tracing can yield higher efficiency and convenience compared to DBI techniques. Moreover, Jiang *et al.* [27] proposed anti-emulation technique that utilized the inconsistent instructions to mitigate QEMU-based fuzzing. However, hardware tracing is not impeded by this technique.

Static binary rewriting. Several studies have applied static binary rewriting to achieve compiler-level performance for binary-only fuzzing [15, 19, 21, 36, 56]. However, many of these approaches are either unsound or constrained by strict prerequisites of the target binaries [56]. e9patch [19] requires the absence of inline data in the binary, while RetroWrite [15] relies on available relocation information. These restrictions greatly limit the practicality of binary rewriting in fuzzing. Compared with binary rewriting, HGF may be more practical on large-scale binaries without these restrictions.

Avoiding trace buffer overflow. To address the buffer overflow, HART [18] leverages the Performance Monitoring Unit (PMU) to generate interrupts after executing certain instructions, ensuring that the trace data size never exceeds the capacity of the trace buffer. However, this technique takes a conservative way to raise the interrupts frequently in a low threshold. Employing this method in fuzzing will introduce heavy overhead and split the control flow information into multiple slices, making reconstructing precise and stable coverage challenging. In contrast, moderating the CPU frequency may be a more practical and efficient approach for fuzzing.

8 CONCLUSION

In this paper, we reviewed existing tools in HGF and summarized the challenges in rebuilding coverage by hardware tracing. We presented Stalker as an efficient hardware-assisted greybox fuzzing technique, using Arm CoreSight as an illustrative example. We proposed a novel coverage mechanism to achieve high-speed fuzzing with a moderate branch coverage close to AFL and a frequency modulation mechanism to alleviate trace buffer overflow. Optimizing the CoreSight driver and decoder, we built Stalker on the Arm Juno R2 development board and conducted systematic evaluations. The results show that our technique addresses the pointed challenges and performs better than existing SOTA tools.

REFERENCES

[1] 2018. Stability problem in PTFuzz. https://github.com/hunter-ht-2018/ptfuzzer/issues/2.

[2] Yuichi Sugiyama Akira Moroo. 2021. ARMored CoreSight: Towards Efficient Binary-only Fuzzing. https://ricercasecurity.blogspot.com/2021/11/armored-coresight-towards-efficient.html.

[3] Yuichi Sugiyama Akira Moroo. 2021. CoreSight-decoder. https://github.com/RICSecLab/coresight-decoder.

[4] Arm. 2011. CoreSight Trace Memory Controller Technical Reference Manual. https://developer.arm.com/documentation/ddi0461/b/?lang=en.

[5] Arm. 2016. ARM CoreSight SoC-400 Technical Reference Manual. https://developer.arm.com/documentation/100536/latest/.

[6] Arm. 2016. Juno r2 Development Platform SoC. https://developer.arm.com/documentation/100114/0200.

[7] Arm. 2021. Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETM4.6. https://developer.arm.com/documentation/ihi0064/latest/.

[8] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.

[9] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. Califor-nia, USA, 10–5555.

[10] Andrea Biondo. 2018. Improving AFL's QEMU mode performance. https://abiondo.me/2018/09/21/improving-afl-qemu-mode/.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.

[12] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. {MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security 20)*. 2325–2342.

[13] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[14] 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 2019 ACM Asia Conference on Computer and Communications Security*. 633–645.

[15] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1497–1511.

[16] 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*. 2214–2228.

[17] Eelco Dolstra. 2004. Patchelf. https://github.com/NixOS/patchelf.

[18] Yunlan Du, Zhenyu Ning, Jun Xu, Zhilong Wang, Yueh-Hsun Lin, Fengwei Zhang, Xinyu Xing, and Bing Mao. 2020. Hart: Hardware-assisted kernel module tracing on arm. In *European Symposium on Research in Computer Security*. Springer, 316–337.

[19] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 151–163.

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*.

[21] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*. 1075–1092.

[22] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. {GREYONE}: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2577–2594.

[23] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.

[24] Xinyang Ge, Ben Niu, Robert Brotzman, Yaohui Chen, HyungSeok Han, Patrice Godefroid, and Weidong Cui. 2021. HyperFuzzer: An Efficient Hybrid Fuzzer for Virtual CPUs. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 366–378.

[25] Marc Heuse. 2018. AFL-Dyninst. https://github.com/vanhauser-thc/afl-dyninst.

[26] CC HWANG. [n. d.]. ptm2human. https://github.com/hwangcc23/ptm2human.

[27] Muhui Jiang, Tianyi Xu, Yajin Zhou, Yufeng Hu, Ming Zhong, Lei Wu, Xiapu Luo, and Kui Ren. 2022. EXAMINER: automatically locating inconsistent instructions between real devices and CPU emulators for ARM. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 846–858.

[28] Andi Kleen and Beeman Strong. 2015. Intel processor trace on linux. *Tracing Summit* 2015 (2015).

[29] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[30] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. 2022. μAFL: Non-intrusive Feedback-driven Fuzzing for Microcontroller Firmware. *arXiv preprint arXiv:2202.03013* (2022).

[31] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.

[32] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. 2018. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 562–566.

[33] 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.

[34] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1949–1966.

[35] Charlie Miller. 2008. Fuzz by number: More data about fuzzing than you ever wanted to know. *Proceedings of the CanSecWest* (2008).

[36] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. 2021. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*. 1683–1700.

[37] Zhenyu Ning and Fengwei Zhang. 2019. Understanding the security of arm debugging features. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 602–619.

[38] nyx fuzz. 2023. libxdc. https://github.com/nyx-fuzz/libxdc.

[39] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.

[40] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*. 2597–2614.

[41] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kafl: Hardware-assisted feedback fuzzing for {OS} kernels. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 167–182.

[42] Kostya Serebryany. 2017. {OSS-Fuzz}-Google's continuous fuzzing service for open source software. (2017).

[43] Haoqi Shan, Moyao Huang, Yujia Liu, Sravani Nissankararao, Yier Jin, Shuo Wang, and Dean Sullivan. 2023. CROWBAR: Natively Fuzzing Trusted Applications Using ARM CoreSight. *Journal of Hardware and Systems Security* (2023), 1–11.

[44] Haoqi Shan, Sravani Nissankararao, Yujia Liu, Moyao Huang, Shuo Wang, Yier Jin, and Dean Sullivan. 2023. LightEMU: Hardware Assisted Fuzzing of Trusted Applications. *arXiv preprint arXiv:2311.09532* (2023).

[45] Suchakrapani Datt Sharma and Michel Dagenais. 2016. Hardware-assisted instruction profiling and latency detection. *The Journal of Engineering* 2016, 10 (2016), 367–376.

[46] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 999–1010.

[47] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*. 1–15.

[48] Qinying Wang, Boyu Chang, Shouling Ji, Yuan Tian, Xuhong Zhang, Binbin Zhao, Gaoning Pan, Chenyang Lyu, Mathias Payer, Wenhai Wang, et al. 2023. SyzTrust: State-aware Fuzzing on Trusted OS Designed for IoT Devices. *arXiv preprint arXiv:2309.14742* (2023).

[49] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One Fuzzing Strategy to Rule Them All. In *Proceedings of the International Conference on Software Engineering*.

[50] Tai Yue, Yong Tang, Bo Yu, Pengfei Wang, and Enze Wang. 2019. Learnafl: Greybox fuzzing with knowledge enhancement. *IEEE Access* 7 (2019), 117029–117043.

[51] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2307–2324.

[52] Michal Zalewski. 2014. Fuzzing random programs without execve(). https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html.

[53] Michal Zalewski. 2017. American fuzzy lop.

[54] Seyed Mohammad Ali Zeinolabedin, Johannes Partzsch, and Christian Mayr. 2020. Real-time hardware implementation of arm coresight trace decoder. *IEEE Design & Test* 38, 1 (2020), 69–77.

[55] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. 2018. Ptfuzz: Guided fuzzing with processor trace feedback. *IEEE Access* 6 (2018), 37302–37313.

[56] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. 2021. STOCHFUZZ: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 659–676.

[57] Zhiqiang Zuo, Kai Ji, Yifei Wang, Wei Tao, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. JPortal: precise and efficient control-flow tracing for JVM programs with Intel processor trace. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1080–1094.