TATOO: A Flexible Hardware Platform for Binary-Only Fuzzing

Jinting Wu [1,2,*], Haodong Zheng [1,2,*], Yu Wang [1,2], Tai Yue [1,2], Fengwei Zhang [2,1,**]
{wujt,zhenghd,12032879,yuet2021}@mail.sustech.edu.cn,zhangfw@sustech.edu.cn
[1] *Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China*
[2] *Department of Computer Science and Engineering, Southern University of Science and Technology, China*

ABSTRACT

Hardware-based tracing, being efficient, can be a good alternative to the computationally-expensive software-based instrumentation in binary-only greybox fuzzing. However, it only records all branches within a specified address range, lacking the flexibility to re-filter them. To overcome these limitations, this paper introduces TATOO, a hardware platform that employs tagged architectures and hardware tracing to enable users to perform instruction-level tagging, which can significantly reduce the volume of traced data and improve fuzzing efficiency. TATOO also supports recording the dataflow information for smart mutations. Implemented on a real hardware FPGA platform, TATOO demonstrates a mere 8.7% performance overhead.

ACM Reference Format:
Jinting Wu [1,2,*], Haodong Zheng [1,2,*], Yu Wang [1,2], Tai Yue [1,2], Fengwei Zhang [2,1,**]. 2024. TATOO: A Flexible Hardware Platform for Binary-Only Fuzzing . In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3655946

1 INTRODUCTION

Fuzzing is one of the well-known techniques to effectively and automatically find bugs. As an essential variant of fuzzing, Coverage-Guided Fuzzing (CGF [7]) has been widely used in industry and research. The keys to CGF are monitoring each program execution and obtaining coverage information and dataflow information. In many cases, programs are binary-only, as developers are unlikely to release their source code. For such scenarios, code coverage is collected through dynamic binary translation [3], static binary rewriting [16, 18], and hardware tracing [5, 19, 23]. Hardware tracing can provide transparent support for fuzzing because coverage information can be collected directly at the hardware layer. Compared to dynamic binary translation, hardware tracing introduces small overheads [23]. Besides, static binary rewriting is highly limited [25]. Hardware tracing is practical compared to static binary rewriting because it does not require prerequisites of target binaries.

Existing hardware tracing tools such as Intel Processor Trace (PT) [27] and ARM CoreSight [1] suffer from inflexibility. While they are capable of tracing instructions within a configured address range, they lack the ability to filter and differentiate these instructions based on their importance within that range. This limitation becomes evident during hardware fuzzing, where it is often necessary to trace all data within the code segments of the binaries. Consequently, existing hardware tracing tools generate an excessive amount of data, leading to several problems:

They trace a plethora of irrelevant data. Dealing with a massive amount of data incurs significant overhead. Current fuzzing processes typically involve static optimizations that do not require tracing every basic block, such as single successor and dominator-based instrumentation pruning [15, 18]. If hardware tracing supports instruction-level filtering, we can selectively trace basic blocks through static analysis instead of tracing all blocks in the code. This will reduce the number of instrumented instructions, improving fuzzing efficiency.

They encounter difficulties in tracing dataflow. Simultaneously tracing control flow and dataflow results in a high volume of data that can overwhelm the trace buffer, making it challenging to trace dataflow effectively. Existing works confirm that techniques such as taint analysis [10], symbolic execution [13], and taint inference [2, 17] can effectively assist fuzzers in achieving more accurate mutation. Therefore, an ideal hardware-assisted fuzzing approach should aim to minimize traced data and flexibly trace essential dataflow for fuzzing.

In this paper, we present TATOO, a custom hardware platform with dataflow tracing to enhance CGF on the RISC-V architecture. We leverage instruction tagging to provide flexible hardware tracing for fuzzing, which allows us to effectively utilize static analysis to reduce the number of instrumented instructions. TATOO has the following features:

Flexibility. TATOO supports users to add tags to specified instructions at the instruction-level based on program semantics. According to the different tags of instructions, TATOO can perform specific processing operations, such as excluding some low-priority instructions from tracing or tracing the dataflow generated by some specific library functions.

Dataflow Tracing. During program execution, TATOO collects the execution data and captures branch-related dataflow information, which is then committed to the coprocessor. The coprocessor processes the dataflow information, allowing the fuzzer to use taint inference techniques to assist in mutating the input data. This technique enables TATOO to address challenges such as implicit flow and magic bytes within the fuzzing process.

By leveraging these capabilities, TATOO enhances the effectiveness and efficiency of CGF by providing flexible instruction-level tracing and enabling the utilization of dataflow information for more targeted and effective mutation strategies. We implement our prototype based on lowRISC [22] and PHMon [12] and utilize a real hardware FPGA platform to evaluate the overhead of

* Co-first authors. ** The corresponding author.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3655946

TATOO. The performance overhead of TATOO is 8.7% in tested real-world programs, and its area overhead is 33.8%. Compared to AFL, a state-of-the-art CGF, TATOO has achieved a 29.03% improvement in throughput and an 8.2% improvement in coverage. We open-source the code of TATOO on `github.com/Compass-All/TATOO`.

2 PRELIMINARY

Hardware Tracing. Intel PT and ARM CoreSight serve as tools for capturing debugging information during processor operations. To optimize trace data volume, both Intel PT and ARM CoreSight employ a highly compressed encoding format for original program execution data, subsequently storing these encoded packets in memory. Furthermore, Intel PT and ARM CoreSight provide the capability to discern whether a specific address range should be traced or not through the configuration of address range comparators. Despite these features, both Intel PT and ARM CoreSight are susceptible to overflow issues. Intel PT addresses this challenge by incorporating the Table of Physical Addresses mechanism, enabling the redirection of trace output data to alternative locations in the event of overflow. Additionally, users can mitigate the probability of overflow by lowering the frequency of the main CPU and employing sampling techniques [26].

Programmable Hardware Monitor (PHMon) is an open-source hardware tracing tool on RISC-V. Our work is built upon PHMon. PHMon [12] can monitor user-defined events and perform the corresponding action. The hardware design of PHMon consists of three units: Trace Unit (TU), Match Unit (MU), and Action Unit (AU). TU collects the execution data from the processor during the CPU write-back stage. MU checks the data sent by TU, finds the data matching the event, and writes it to the queue. Each MU pairs with the CFU in AU, where the CFU stores the action sequence once the MU matches successfully. AU takes follow-up actions according to the programmer's configuration. The actions include ALU operation, memory access operation, and interrupt. By employing PHMon, users are able to perform the fuzzing process by customizing the hardware monitoring and action execution.

Common Fuzzing Roadblocks. Two common fuzzing roadblocks are magic bytes and checksums, as shown in Listing 1. Researchers circumvent these roadblocks by obtaining dataflow information in programs [2]. For solving the magic bytes, GreyOne [17] employs taint inference to mutate the input at the byte level and monitor the program variables. Once the value of a variable has changed, GreyOne can conclude that it depends on the specific byte of the mutated input and identify all variables that use a direct copy of the input bytes. GreyOne records the values of variables used in branch constraints by instrumentation.

One approach to solving the nest checksum is to remove the hard checks and fix them later [2]. It ignores the hard checks so that the fuzzed program can reach deeper branches. Then, it fixes it and verifies whether the path is reachable.

```
/* magic bytes example */
if(u64(input) == u64("MAGICHDR")){bug(1);}
/* nested checksum example */
if(u64(input) == sum(input+8,len-8)){
  if(u64(input+8) == sum(input+16,len-16)){
    if(input[16]=='R' && input[17]=='Q'){bug(2);}}}
/* implicit flow example */
if(input[16] == 'R' ) {var1=0;}
if(var1 == 0){bug(3);}
```

Listing 1: Roadblocks of Fuzzing

3 TATOO DESIGN

3.1 Architecture and Workflow

As Figure 1(a) shows, we utilize a coprocessor as the trace component and implement instruction tagging in the main core. Instruction tagging allows users to control data collection and assign distinct actions in trace components. When users execute the program, the tags and the values in source registers are committed to the trace component. The trace component assists the fuzzer in collecting coverage and dataflow information. The detailed modification of hardware is illustrated in Section §4.3.

The workflow of TATOO consists of two stages: preparation and fuzzing. The specific workflow shown in Figure 1(b) is as follows:

Preparation Stage: ① Users perform static analysis on a `target program` (e.g., real-world program, dynamic link library), find the instructions to tag, and write them to a `tag file`. ② Users use `Tagger` to create a `tagged program` with instruction tagging. ③ Users configure the programmable coprocessor by `Configurer`. Users use tags and instruction types to distinguish different events, and the coprocessor performs different actions based on the specific events. (1)*Tag=0:* The coprocessor filters instructions, so the instructions do not enter the queue. (2)*Tag=1 and the jump instruction:* The coprocessor updates the bitmap and writes it to the memory. TATOO adopts PHMon's edge encoding algorithm to generate pseudo-random IDs for each edge. (3)*Tag=1 and the conditional jump instruction:* The coprocessor updates the bitmap and records the dataflow information based on taint inference. The method of updating data flow is similar to updating the bitmap. (4)*Tag=2 and Tag=3:* The tag values are retained for future use.

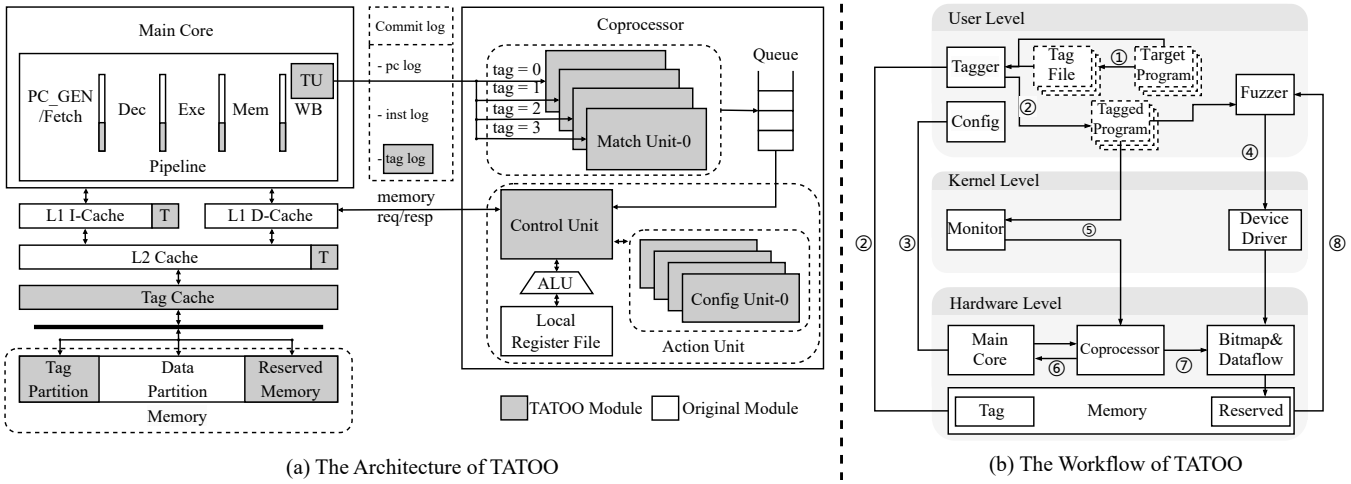
Fuzzing Stage: ④ The kernel reserves continuous physical memory for traced data by `Device Driver` when the fuzzer process starts. ⑤ The fuzzer executes the tagged program to be fuzzed. The `Monitor` determines the program that the kernel should monitor. The coprocessor needs to be enabled or disabled during kernel context switches. ⑥ When a monitored program executes, the processor commits the execution data, including the tag and dataflow information, to the coprocessor. ⑦ The coprocessor collects relevant information and writes it into memory. ⑧ The fuzzer analyzes the data and schedules and mutates the input files according to the collected dataflow. ⑨ When the fuzzer process exits, the kernel frees the memory.

3.2 Trace Component

We employ a coprocessor as a trace component and extend its capabilities to gather branch-related information, enabling us to collect information through taint inference. In this subsection, we delve into the design considerations of the trace component.

There are two design alternatives to collect the traced data in the hardware: in-core [14] and off-core design [6]. If the in-core design is adopted, the hardware manufacturer must redesign all the CPUs to collect trace data, which is impractical. Therefore, we adopt an off-core design using a coprocessor.

Traditional software dataflow-assisted fuzzing is usually implemented using dynamic taint analysis or taint inference. Although dynamic taint analysis [10] can be applied in many scenarios, such as malware detection and information flow leakage, for the application scenario of fuzzing, the implementation of dynamic taint



(a) The Architecture of TATOO

(b) The Workflow of TATOO

Figure 1: The Architecture and Workflow of TATOO

analysis on hardware is deficient in three points compared to taint inference.

Taint inference can tackle implicit flow issues. Existing hardware-assisted dynamic taint analysis [6] struggles with implicit flows due to neglecting tainted data's impact on other data. Despite the dedicated efforts of Jangseop Shin et al. [11] to design new hardware solutions for this issue, their approach still encounters challenges, leading to the presence of false positives and negatives. In contrast, taint inference [17] demonstrates minimal undertaint and only a slight amount of overtaint, establishing itself as a more effective choice due to its provision of more accurate data.

Taint inference offers superior efficiency. Dynamic taint analysis introduces high overhead, as the coprocessor must handle numerous instructions for taint propagation, leading to frequent shadow memory loading and storing, thus hampering performance [6]. In contrast, taint inference only processes branch instructions, making it more efficient.

Taint inference demands minimal manual effort. Dynamic taint analysis involves a plethora of instructions, such as vector, atomic, and customized instructions in RISC-V, making manual summarization tedious. Moreover, propagation rules are diverse, making manual rule summarization impractical for context-based rule specification [24]. Taint inference, on the other hand, simply records source register values for branch instructions, avoiding the need for extensive manual configuration.

3.3 Instruction Tagging

TATOO differentiates and filters the instructions by tagging them.

Differentiation. Different tags necessitate distinct treatments. For Tag = 1, we employ GreyOne's approach to collect dataflow. While GreyOne effectively addresses the magic bytes issue shown in Listing 1, it struggles when using functions like `memcmp` to collect magic bytes information. Specifically, GreyOne, similar to AFL, the data updated later overwrites the previous data in loop comparison, leading to data loss. To mitigate this, we assign varied tags for different dataflow types, enabling TATOO to process each uniquely. This approach also allows for sequential recording of current magic bytes information. In addition, REDQUEEN [2] addresses checksum challenges by initially removing hard checks and fixing them later. Theoretically, tagging certain instructions to trigger a processor exception when not jumping to the desired location could further aid in resolving checksum issues.

Filter. Existing works [4, 18] show that graph reducibility techniques can elide instrument some basic blocks in fuzzing. This is because many basic blocks do not affect code coverage during fuzzing, such as some loops and library functions. However, previous hardware-based binary-only fuzzing usually instruments the branch instructions in the text segments. Such coarse-grained instrumentation leads to low performance and increases the probability of edge collisions. To solve this problem, we tag the instructions so that the coprocessor distinguishes how to process them with the tagging instructions. Although instruction tagging introduces a small amount of overhead, it reduces the number of instrumentation from instrumentation pruning and thus optimizes performance. Notice that the instruction tagging overhead only exists when running the monitored program. When running other programs, we can disable memory tagging by setting the Control State Register (CSR) registers. In this way, TATOO takes advantage of static analysis to optimize the number of instrumentation with high efficiency, which has the following benefits.

TATOO effectively mitigates the probability of buffer overflow. Off-core devices (e.g., Intel PT, ARM CoreSight) require an on-chip buffer to temporarily store the data executed by the hardware. However, if the traced data is generated too fast or consumed too slowly, the buffer overflows, resulting in the loss of traced data. The overflow adversely impacts the accuracy of coverage collection. Due to TATOO's reduction of traced data, it effectively alleviates the buffer overflow probability.

TATOO can effectively impede the anti-fuzzing technique [9]. Existing anti-fuzzing confuses existing coverage feedback mechanisms by inserting fake code, introducing a lot of conditional branches and indirect jumps into the original program. Thus, it creates a meaningless path explosion and wastes most fuzzing time on invalid inputs. TATOO can easily filter the invalid input if static analysis analyzes the meaningless basic blocks. Besides, anti-fuzzing also converts explicit dataflow to implicit dataflow, preventing dataflow tracing through taint analysis. Nevertheless, we collect dataflow by taint inference.

4 IMPLEMENTATION

We implement our prototype based on lowRISC [22] and PHMon. lowRISC implements memory tagging. PHMon is a programmable hardware monitor used to build prototype systems rapidly. They are both open-source and flexible. We synthesize and map our prototype to the Xilinx Kintex-7 FPGA KC705 evaluation board using Xilinx Vivado 2015.4. We present the necessary modifications to the user-level program, kernel, and hardware to support Tatoo. These modifications are essential for nearly all hardware-based binary fuzzing approaches.

4.1 User-level Program Modification

Static Analysis. We utilize the commercial software IDApro for offline static analysis, scripted in Python using IDAPython. The main objective of the static analysis is to identify functions that are not instrumented by AFL, enabling Tatoo to filter out these functions during tracing. The analysis procedure involves the following steps: First, we prepare two binaries: one is instrumented using AFL and the other is compiled normally without instrumentation. We collect line numbers for each function in both binaries and compare them, revealing functions with AFL instrumentation. Subsequently, we record branch and jump instructions in functions initially instrumented by AFL in the second binary. After static analysis, the instructions that require tagging are obtained and stored to a file.

Note that our static analysis is a prototype module. Our intention is to explore whether instruction tagging could assist hardware tracing tools in achieving more flexible tracing capabilities. The choice of specific tags depends on user-driven static analysis. We acknowledge that other existing studies [4, 18] have significantly contributed to binary-only fuzzing, employing techniques such as single successor and dominator-based instrumentation pruning. However, static analysis in binary-only fuzzing, as a distinct research contribution, falls beyond the scope of our article.

Tagger. To create a file with instruction tagging, `Tagger` starts by determining the offset of the text section in the original file. Subsequently, `Tagger` tags the instructions based on the static analysis-generated file.

Configurer. In order to facilitate the programmer to configure PHMon, PHMon provides the software interface. PHMon uses custom RISC-V instructions to configure the MUs and CFUs of PHMon and communicate with PHMon. To notify the coprocessor to trace the dataflow, we modify the software interface of PHMon. Tatoo adopts PHMon's edge encoding algorithm to generate pseudo-random IDs for each edge. We can also configure Tatoo to be compatible with SNAP's edge encoding algorithm, demonstrating the programmable coprocessor's flexibility.

AFL Code Modification. We provide a software interface so that the fuzzer can find the virtual address of the shared memory. Thus, we can use the dataflow information to identify critical bytes.

4.2 Linux Kernel and Bootloader Modification

We modify lowRISC's bootloader and the Linux kernel (version 4.6) to support Tatoo. The bootloader is modified to support custom instructions and delegate interrupt handling. The modifications of Linux kernel are as follows:

Device Driver. Tatoo uses the device driver to collect the bitmap and dataflow information. Upon fuzzer process initiation, the device allocates a dedicated kernel memory region for the coverage bitmap and dataflow, sharing the physical address with the coprocessor. Then, the device driver supports memory remapping to userspace for data processing during fuzzing. The kernel frees the previously allocated memory upon fuzzer process closure.

Monitor. We manually specify the name of processes in Linux to decide about the monitored program. The kernel dynamically enables or disables the coprocessor upon the start or exit of the monitored program.

Context Switch. If the fuzzer is only fuzzing one tested program, the kernel only needs to enable and disable the coprocessor during context switching. If fuzzing multiple programs under test, for example, to detect different processes running on the same core, the kernel needs to save the information of Tatoo when a context switch into another process occurs.

Interrupt Handler. After the coprocessor's queue overflows, the coprocessor sends an interrupt to the processor, prompting the kernel to wait until the coprocessor processes the queue before resuming.

4.3 Hardware Modification

To facilitate implementation, we incorporate the PHMon coprocessor into the lowRISC processor. Because lowRISC processor lacks support for compression instructions in RISC-V, the current prototype does not support compression instructions. The hardware modification consists of modifications to the main core and the coprocessor.

Main Core Modification. We introduce a TU to transmit information to the coprocessor. Unlike PHMon, Tatoo sends the values of source registers and tag values to the coprocessor. Besides, we add memory tagging. First, we add tag propagation in the pipeline and cache to fulfill tag propagation. Second, a tag cache is added between the last-level cache and the memory controller, serving as an assembler. When the last-level cache needs to load the memory with tags, it accesses the memory and tag memory respectively, assembles them, and sends them back to the last-level cache. When the last-level cache needs to store the memory with tags, it stores the memory and tag memory separately. Moreover, we also modify some exception trigger conditions in lowRISC.

Coprocessor Modification. We modify the MU to match the values of source registers and tag values. In addition, the CFU is adjusted to allow user configuration for handling different tags. Besides, in the AU, PHMon configures memory type to indicate the memory access byte width in one AU. As Tatoo requires updating the coverage bitmap and dataflow information within one AU, it allows configuring the memory access byte width in one specific action.

5 EVALUATION

In this section, we focus on measuring the performance and hardware overhead of Tatoo introduced by instruction tagging and hardware tracing to answer the following questions:

RQ1: What is the performance overhead of Tatoo?

RQ2: How effective is Tatoo compared to AFL in enhancing throughput and edge coverage?

RQ3: What is the hardware resource cost of Tatoo?

Config. The hardware configuration includes the Rocket Privileged Specification set at version 1.7, utilizing the RV64IMAFD

Table 1: Target Binaries Evaluated in Our Evaluation

Subjects	Size Change
objdump –dwarf-check -C -g -f -dwarf -x @@	10.08 M to 12.79 M
readelf -a @@	5.02 M to 7.92 M
size @@	5.27 M to 5.34 M
nasm -f elf -o sample @@	5.80 M to 8.04 M
bison @@	5.22 M to 9.27 M
tiff2bw @@ /dev/null	1.29 M to 1.35 M
tiffnfo @@	1.36 M to 1.40 M

instruction set architecture extensions with support for M, S, and U modes. The lowRISC configuration adopts TagL2Config, indicating a specific setup for the system. Additionally, the coprocessor is configured with 4 match units and features a 1024-entry on-chip buffer, contributing to the overall functionality of the prototype.

5.1 Performance Overhead

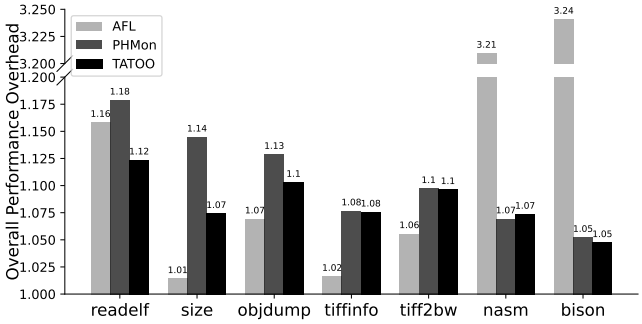


Figure 2: The Overall Performance Measured by Real-World Programs

Setup. To better understand the overall performance overhead of TATOO in real-world programs, we select 7 programs listed in Table 1 for our experiments. Table 1 shows that the program size changed after running LLVM ModulePass provided by AFL. The instrumentation is performed using the `afl-clang-fast` mode.

To measure our overall overhead, we run four sets of comparison experiments: the original programs, the programs instrumented by AFL, the original programs running on PHMon, and the original programs running on TATOO. All the programs are processed to communicate with the forkserver. The first two experiments are run on the unmodified kernel and unmodified hardware. The last two experiments are run on the modified kernel. We configure PHMon to match all jump and branch instructions in text segments, and PHMon only records coverage. We configure TATOO records coverage and dataflow information like GreyOne. TATOO only matches the jump and branch instructions in the function that AFL instruments. We run AFL on another machine to get 2,000 seeds for each program and then run these different seeds within the forkserver. In the end, we count the running time of each seed and took the average. The baseline, which is standardized as 1, is the result of the original program.

Result. In Figure 2, we find that the overall performance overhead introduced by TATOO is around 5% to 12%, and the average performance overhead of AFL, PHMon, and TATOO is 60%, 11.55%, and 8.7%, respectively. Compared to AFL, TATOO runs slightly slower when AFL performs lightweight instrumentation (e.g., `tiffinfo`, `tiff2bw`) because TATOO requires instruction tagging and tracing dataflow. TATOO is faster than AFL when AFL performs heavyweight instrumentation (e.g., `nasm`, `bison`). In Table 1, we notice that the size of `nasm` and `bison` instrumented by AFL increased by 39% and 77% compared with the original size of the program. The heavyweight instrumentation seriously affects the efficiency of AFL. Compared to AFL, TATOO accelerates coverage collection through additional hardware when running these programs. Compared to PHMon, TATOO has better performance because PHMon needs to trace all the branch and jump instructions in the text segment. In addition, PHMon spends more time processing interrupts introduced by overflow than TATOO. For example, on `objdump`, PHMon overflows 681 times, while TATOO overflows only 20 times. The experimental results show that TATOO runs faster than PHMon, which proves that although instruction tagging introduces overhead, it reduces overall performance overhead with the help of instruction tagging.

Table 2: Throughput in Our Evaluation

Binary	AFL	PHMON	AFL_QEMU	TATOO
readelf	213,532	230,181	44,753	232,856
objdump	194,865	213,353	26,648	215,445
size	192,172	209,527	23,887	214,242
nasm	68,677	147,442	7,386	158,351
bison	158,394	185,359	24,190	201,095
tiffinfo	225,086	231,928	56,156	243,002
tiff2bw	227,444	234,146	53,124	242,670

5.2 Throughput and Coverage

Throughput. We executed 7 binary programs targeting different fuzzing objectives. Table 2 demonstrates the fuzzing throughput achieved by TATOO and the compared tools over a 24-hour period for 7 programs. TATOO demonstrates significantly faster progress in achieving upper coverage compared to other tools, surpassing AFL and AFL_QEMU on the majority of programs. In particular, on average, TATOO shows an improvement of approximately 29.03% over AFL, a remarkable 769.88% over AFL_QEMU, and a 4.10% improvement over PHMON. Notably, TATOO achieves a remarkable increase in throughput compared to other tools, primarily attributed to the optimization of the instrumentation quantity.

Coverage. Previous studies [2, 17] have demonstrated the effectiveness of dataflow analysis. Due to the high time overhead and limited availability of FPGA development boards for experimentation, we replay the coverage on different tools. The experimental results reveal that TATOO achieves an 8.2% higher edge coverage compared to AFL, which achieves a higher edge coverage than both PHMon and AFL_QEMU, except for `tiff2bw`. The subpar performance of `tiff2bw` can be attributed to the hardware's edge-encoding algorithm. This experiment highlights that TATOO can improve coverage.

5.3 Hardware Resource Cost

Table 3 illustrates hardware resource costs for systems with and without TATOO. We observe an extra usage of 33.9% slice Look-Up Tables (LUTs) and 33.8% slice registers across the systems. TATOO introduces memory overhead in memory tagging (15%) and hardware tracing (18%). Taking into consideration that future devices will

Table 3: Hardware Resource Cost of Tatoo

	Whole Systems		Power
	Slice LUTs	Slice Registers	
Without Tatoo	73,784	39,035	3.324W
With Tatoo	98,807	52,210	3.309W
%	+ 33.9%	+ 33.8%	- 0.5%

implement memory tagging extensions and hardware tracing, the result shows that the hardware resource cost of Tatoo is acceptable.

6 RELATED WORK

Hardware-assisted Fuzzing. Some researchers utilize architectural extensions like Intel PT and ARM CoreSight for fuzzing. PTfuzz [5] employs Intel PT to collect program branch information but introduces performance overhead due to data packet decoding and coverage calculation based on branch jump addresses. PTrix [23] avoids decoding, directly converting data packets into coverage information to reduce collection overhead. μAFL[21] uses ARM Coresight for transparent execution info collection in IoT devices, transmitting it for MCU firmware fuzz testing. It filters irrelevant tracking information by specifying tracing regions or tracking specific memory instructions. PHMon [12] and SNAP [14] are the existing work of customizing hardware to assist binary-only fuzzing on the RISC-V platform. SNAP integrates new microarchitectural units in the CPU to update the coverage bitmap and provide richer feedback to the fuzzer. Besides, It uses control flow information and approximate dataflow provided by branch prediction to assist fuzzing.

Previous approaches trace the instructions in a range of configured addresses. In contrast, Tatoo utilizes instruction tagging, employing binary analysis to filter out unnecessary instructions and mitigating buffer overflow issues. Unlike traditional methods, which largely lacked dataflow assistance except for SNAP, Tatoo traces additional dataflow using taint inference. In contrast to SNAP, which relies on approximate dataflow from branch prediction, Tatoo utilizes accurate dataflow information, providing more effective assistance in fuzzing.

Memory Tagging. Memory tagging [6, 8, 20], part of Armv8.5 and present in newer Armv9 CPUs like Cortex-X2, is expected to be integrated into future Armv9-based CPUs for enhanced memory security. Memory tagging involves assigning a unique tag to each memory allocation, requiring all memory accesses to be conducted through a pointer with the correct tag. Incorrect tags can be reported by the operating system to the user or logged for further analysis. Memory tagging serves several purposes: improving memory safety [8], information-flow control [20] and dynamic information flow tracing [6]. In contrast to traditional memory tagging, we employ instruction tagging to better complement hardware tracing tools, enhancing their capabilities for fuzzing assistance. Beyond fuzzing, Tatoo extends its applicability to fields like binary debugging like other hardware tracing tools. Tatoo allows easy configuration of breakpoints and observation points by memory tagging. Unlike existing hardware with limited breakpoints, Tatoo's instruction tagging-based breakpoints offer enhanced convenience and flexibility for binary debugging. In essence, Tatoo's combination of memory tagging and hardware tracing renders it a versatile platform adaptable to diverse domains.

7 CONCLUSION

In this paper, we present Tatoo, a customized hardware-assisted fuzzing platform on RISC-V. Tatoo utilizes a coprocessor to collect bitmap and valuable dataflow information. Additionally, we incorporate fine-grained tracing through instruction tagging, enabling optimized binary-only fuzzing using static analysis techniques. Experimental results demonstrate that Tatoo achieves efficient tracing, exhibiting an improvement of 769.88% over AFL_QEMU, the current state-of-the-art fuzzer.

8 ACKNOWLEDGMENTS

We would like to thank the reviewers for their insightful comments. This work is partly supported by the National Natural Science Foundation of China under Grant No.62372218 and Shenzhen Science and Technology Program under Grant No.SGDX20201103095408029.

REFERENCES

[1] Arm 2016. *Arm coresight soc-400 technical reference manual.* Arm. https://developer.arm.com/documentation/100536/latest/
[2] A. Cornelius et al. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *Proc. NDSS' 19*, Vol. 19. 1–15.
[3] A. Fioraldi et al. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proc. USENIX WOOT' 20.*
[4] C. Hsu et al. 2018. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Proc. NDSS' 18, Workshop on Binary Analysis Research.*
[5] G. Zhang et al. 2018. PTfuzz: Guided Fuzzing With Processor Trace Feedback. *IEEE Access* 6 (2018), 37302–37313.
[6] H. Kannan et al. 2009. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *IEEE/IFIP DSN' 09.* IEEE, 105–114.
[7] H. Liang et al. 2018. Fuzzing: State of the Art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.
[8] H. Xia et al. 2019. CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety. In *Proc. MICRO' 52.* 545–557.
[9] J. Jung et al. 2019. Fuzzification:{Anti-Fuzzing} Techniques. In *Proc. USENIX Security' 19).* 1913–1930.
[10] J. Kim et al. 2014. Survey of dynamic taint analysis. In *Proc. IEEE IC-NIDC' 14.* IEEE, 269–272.
[11] J. Shin et al. 2016. A hardware-based technique for efficient implicit information flow tracking. In *Proc. IEEE/ACM ICCAD' 16.* 1–7.
[12] L. Delshadtehrani et al. 2020. PHMon: A Programmable Hardware Monitor and Its Security Use Cases. In *Proc. USENIX Security' 20.* 807–824.
[13] N. Stephens et al. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *Proc. NDSS' 16*, Vol. 16. 1–16.
[14] R. Ding et al. 2021. Hardware Support to Improve Fuzzing Performance and Precision (to appear). In *Proc. ACM CCS' 21.* Seoul, South Korea.
[15] S. Canakci et al. 2021. DirectFuzz: Automated Test Generation for RTL Designs using Directed Graybox Fuzzing. In *Proc. ACM/IEEE DAC' 21.* 529–534.
[16] S. Dinesh et al. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *Proc. IEEE S&P' 20.* IEEE, 1497–1511.
[17] S. Gan et al. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *Proc. USENIX Security' 20.* USENIX Association, 2577–2594.
[18] S. Nagy et al. 2021. Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing. In *Proc. USENIX Security' 21.* 1683–1700.
[19] S. Schumilo et al. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proc. USENIX Security' 17.* 167–182.
[20] S. Weiser et al. 2019. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v.. In *Proc. NDSS' 19.*
[21] W. Li et al. 2022. μAFL: Non-Intrusive Feedback-Driven Fuzzing for Microcontroller Firmware. In *Proc. ICSE'22.* 1–12.
[22] Y. Bradbury et al. 2014. Tagged memory and minion cores in the lowRISC SoC. *Memo, University of Cambridge* (2014).
[23] Y. Chen et al. 2019. Ptrix: Efficient hardware-assisted fuzzing for cots binary. In *Proc. ACM Asia CCS' 19.* 633–645.
[24] Z. Chua et al. 2019. One Engine To Serve'em All: Inferring Taint Rules Without Architectural Semantics.. In *Proc. NDSS' 19.*
[25] Z. Zhang et al. 2021. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *Proc. IEEE S&P' 21.* IEEE, 659–676.
[26] Z. Zheng et al. 2022. Detecting Process Hijacking and Software Supply Chain Attacks Using Intel® Threat Detection Technology. (2022).
[27] Andi Kleen and Beeman Strong. 2015. Intel processor trace on linux. *Tracing Summit* 2015 (2015).