MULCOTAINT: Towards Efficient Multi-tag Dynamic Taint Analysis via Hardware/Software Co-design

Bing Qi[1,2,†] Yi Yang[1,2,3,†] Xiangkun Jia[1,2,3] Zhengpin Qian[1,2] Huafeng Huang[1,2]

Purui Su[1,2,3,✉]

[1]*University of Chinese Academy of Sciences*
[2]*Institute of Software, Chinese Academy of Sciences*
[3]*Key Laboratory of System Software (Chinese Academy of Sciences)*
Email: {qibing2019,yangyi,xiangkun,zhengpin2021,huafeng,purui}@iscas.ac.cn

Abstract

Multi-tag dynamic taint analysis (M-DTA) is critical in fine-grained analysis scenarios such as vulnerability analysis. However, current software solutions have serious performance problems. Although hardware solutions are promising, they are single-tag and difficult to extend to M-DTA. We propose an efficient M-DTA framework named MULCOTAINT via hardware/software co-design. We decouple the taint analysis from the normal execution with the coprocessor architecture and solve several challenges, such as designing taint calculation as vectorized calculation, managing taint tags with page tables, and providing functionality interfaces of the taint analysis engine. We build a dataset of 32 programs with 5 types and conduct the performance evaluation and vulnerability analysis experiments. The results show that MULCOTAINT has high performance and acceptable memory usage with abilities of detailed vulnerability analysis. MULCOTAINT outperforms the software solutions (TaintRabbit and PANDA) and hardware solutions (HardTaint, RAFT, and FineDIFT). The maximum difference of overhead increase based on the respective baselines could be '1.14x vs. 4409.09x' for 'MULCOTAINT vs. PANDA', while HardTaint's average overhead increase is 19.57 times that of MULCOTAINT. Although the prototype of MULCOTAINT 's hardware cost is higher than embedded-oriented works RAFT and FineDIFT, it is acceptable due to M-DTA's complex logic.

1 INTRODUCTION

Dynamic taint analysis (DTA) clarifies the data processing procedure and the program logic by setting taint tags for sensitive data and tracking the taint tag calculation during execution. It has become a fundamental methodology for various security analysis tasks [1, 17, 22, 30, 41, 47, 50, 55, 56]. For example, as shown in Figure 1, by setting the read buffer as tainted, we can find that the return address is tainted when overflow happens, i.e., an attacker can modify the return address. Further, researchers want to know how to modify the input for vulnerability exploitation, thus propose Multi-tag DTA (M-DTA) to distinguish the effect of different input bytes (as the colored content in Figure 1).

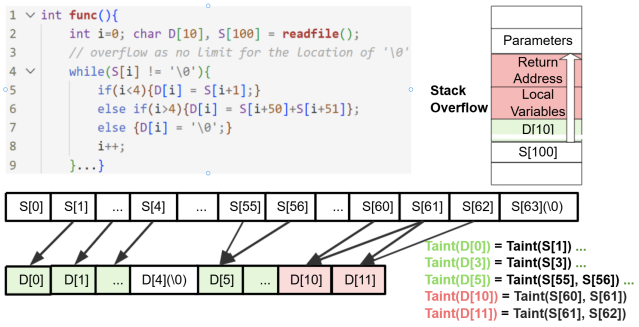

Figure 1: Requirement of multi-tag taint analysis.

The choice between S-DTA (Single-tag DTA) and M-DTA depends on the application scenario rather than the vulnerability type. S-DTA is superior in coarse-grained scenarios (e.g., violation detection) and M-DTA excels in fine-grained vulnerability analysis and taint-tracking scenarios [16, 18, 26, 29, 36, 37]. In vulnerability fix and exploit, distinguishing the impacts of different bytes is essential (mentioned in Penumbra [11], TaintPipe [37], HOTracer [26], etc.). For example, in the buffer overflow case above, S-DTA identifies that the input overwrites the return address but cannot tell how to control it. M-DTA locates critical bytes, and modifying them instead of the entire input can jump to shellcode. Such detailed information is also useful when fixing, as it clarifies vulnerability paths and minimizes audited code, avoiding reviewing all code.

Although M-DTA appears to provide stronger analytical capabilities, it comes at a cost. Currently, M-DTA faces multiple practical challenges as DTA, including source/sink selection and over- and under-tainting, among which performance is one of the major bottlenecks (mentioned in TaintPipe [37], podft [50], AirTaint [45], etc.). Performance is a

† co-leading authors. ✉ corresponding authors.

fundamental problem that it may cause direct analysis failure (e.g., podft [50] mentions Triton's [46] timeout for SPEC programs), and M-DTA faces more serious performance issues due to the complexity of M-DTA calculation logics (the comparison tool PANDA [16] consumes all 32G memory when testing SPEC2017 [49] 641.leela in our experiments).

Reviewing the DTA/M-DTA implementations, most solutions are implemented at the software level, thus have unavoidably performance issues introduced by the instrumentation and execution framework. For instance, the static instrumentation methods, including instrumenting source code during compilation [20, 31, 58] or instrumenting binaries via binary rewriting techniques [8, 45], suffer the overhead of instrumented code for complex taint calculation. The dynamic instrumentation methods, e.g., instrumenting taint logics during the instruction translation based on virtual simulators [12, 14, 16, 23, 26], face the overhead of the dynamic execution framework.

Researchers turn to explore implementing DTA at the hardware level [6, 40, 57, 59]. For example, FineDIFT [6] uses coprocessors to decouple the taint analysis from normal execution. However, as targeting runtime detection, they are designed as single-tag and difficult to extend as M-DTA due to the particularity of hardware solutions, such as customized hardware or limited resources.

Although hardware-based implementation is promising, it faces the following challenges. The first challenge is how to implement an effective taint calculation logic of M-DTA at the hardware level. Generally, the taint analysis is treated as a linear logic deductive process [26]. For example, if Register-A is tainted by two input bytes and Register-B is tainted by three input bytes, we have to determine each byte of the destination operand by checking all the bytes of Register-A and Register-B again and again when conducting M-DTA. The linear logic clearly wastes time and limits us from leveraging the advantages of hardware acceleration.

Second, it is difficult for hardware solutions to handle massive taint tags. The issue exists in S-DTA but becomes more serious in M-DTA as we set different tags for different bytes that a byte of registers or memory may hold multiple tags. Software solutions can allocate a shadow memory for the whole memory space [16], but hardware solutions have limited resources. Besides, the taint tags should be accessed quickly to ensure the efficiency of the whole solution.

Third, how to support real-world fine-grained analysis tasks is challenging. Users often define customized checking rules based on their goals. For instance, different types of vulnerabilities may require distinct DTA checkpoints. However, the hardware implementation has lower applicability, that generally the taint tracking and violation checking are tightly coupled. RAFT [57] is unable to handle complex heap usage and does not support adjusting detection strategies.

In this paper, we propose an efficient M-DTA framework named **MULCOTAINT** via hardware/software co-design. Basically, we decouple the taint analysis from the normal execution with the coprocessor architecture. To solve the above challenges, we first re-model the logic-intensive problem of linear taint calculation into a compute-intensive vectorized calculation problem. When representing the taint tags as vectors, the results can be obtained by calculating the vectors at once, instead of checking byte by byte. The taint calculation logics are then represented as microcodes with corresponding circuit implementations.

Further, building on the concept of vectorized computation, we design each taint tag as a large-width bit vector, where each bit corresponds to a specific byte of a taint source. We leverage the virtual memory of the target program and construct a multi-level page table-based structure for taint address mapping and management. In addition, we design a dedicated wide-byte read/write channel between the program and the coprocessor to support fast taint tag access.

To accommodate diverse taint analysis requirements, we develop a configurable and extensible system interface by introducing a set of extended instructions. Furthermore, to support both source-available and binary-only scenarios, we implement two integration mechanisms: source-level embeddable functions and binary-level dynamic library injection. This design enables efficient taint propagation with high scalability and compatibility across deployment environments.

We implemented a prototype of MULCOTAINT on a RISC-V open-source SOC called RocketChip [2] and completed the development of the taint calculation engine based on the PHMon coprocessor framework [15]. For evaluation, we build a dataset of 32 programs with 5 types, i.e., the CPU spec benchmark and big real-world programs (i.e., PHP and Nginx) for performance evaluation, and the Juliet test suite [44], CTF subjects, and real-world programs with CVE vulnerabilities for both performance evaluation and vulnerability analysis. We compare with both software solutions (i.e., TaintRabbit [18] and PANDA [16]) and hardware solutions (i.e., HardTaint [59], RAFT [57], and FineDIFT [6]). In the performance evaluation, MULCOTAINT only brings 5.8x overhead on average on the basis of PHMon (1.07-15.81x), while TaintRabbit brings 8.08-1037.03x overhead and PANDA brings 401.26-14018.11x overhead. HardTaint's average overhead increase is 19.57 times that of MULCOTAINT. In vulnerability analysis, MULCOTAINT can detect all the vulnerabilities in our dataset with detailed information on how input bytes trigger the vulnerabilities, while FineDIFT [6] and RAFT [57] cannot provide such information and cannot handle programs with complex heap operations. Furthermore, MULCOTAINT has an acceptable hardware cost and memory usage to support M-DTA, even compared to the single-tag methods, i.e., FineDIFT [6] and RAFT [57]. As we target dedicated analysis coprocessor for general-purpose CPUs, the prototype's hardware cost is higher than these two embedded-oriented works due to M-DTA's complex logic. In design, we minimize hardware modifications to CPU itself for future dedicated

hardware.

The contributions to this article are listed below:

- We propose an efficient M-DTA tool named MULCOTAINT via hardware/software co-design. Specifically, we redesign taint calculation, manage taint tags, and provide interfaces to the taint engine, i.e., we modify the M-DTA policy (including taint tag structures and corresponding calculations) and do not change the source and sink selection in vulnerability analysis as it is task-specific.
- We implemented a prototype tool on RocketChip and completed the development of the taint calculation engine based on the PHMon coprocessor framework.
- We evaluate MULCOTAINT based on a dataset of 32 programs. The results show that MULCOTAINT has high performance and abilities of detailed vulnerability analysis with an acceptable hardware cost and memory usage. It outperforms the software solutions (TaintRabbit and PANDA) and hardware solutions (HardTaint, RAFT, and FineDIFT). More details are open-sourced in the Zenodo repository (https://doi.org/10.5281/zenodo.17939551).

2 BACKGROUND

2.1 Dynamic Taint Analysis

Dynamic taint analysis (DTA) is a data flow analysis technique. It sets interesting data with taint tags (i.e., taint source), traces the taint propagation, monitors registers or memory affected by the taint tags during program execution, and finally checks the taint-tagged status at some critical points (i.e., taint sinks) [39]. Single-tag dynamic taint analysis (S-DTA) can only identify whether memory and registers have been tagged by the taint source [6, 8, 23, 29, 45, 57], but cannot identify the specific bytes they come from, as shown in Figure 2 (a).

Based on S-DTA, researchers proposed multi-tag dynamic taint analysis (M-DTA), which provides fine-grained dataflow analysis to distinguish each byte of inputs [10, 18], as shown in Figure 2 (b). It is helpful for vulnerability discovery and analysis [3, 6, 7, 19, 52, 57]. For example, based on the multi-tag ability, researchers accurately determine which bytes of the input should be mutated to bring more coverage during fuzzing [7, 34].

2.2 Existing Solutions

Reviewing the development from S-DTA to M-DTA, the performance problem always exists and restricts the applications. Software schemes inevitably bring additional overhead. With source code available, the schemes use static analysis and instrumentation during compilation [31]. For binary programs, the schemes [9, 18, 29] rely on binary rewriting techniques [35, 38] to perform instrumentation. As claimed in Airtaint [45], the instrumented codes will bring significant overhead. Besides, dynamic instrumentation, such as virtualized schemes [12, 14, 23, 26] or instrumentation frameworks [27, 29, 46, 50], introduces taint analysis during the execution, thus has to face the overhead of instruction translation or execution frameworks.

Therefore, researchers explored hardware-based implementations to improve taint analysis. FlowMatrix [25] accelerates offline taint calculation for program segments with GPU. HardTaint [59] utilizes Intel Processor Trace (PT) to selectively monitor critical data and adopts a parallel strategy for taint analysis. FineDIFT [6] uses a Content Addressable Memory (CAM)-like structure as tag storage to reduce storage demands, while RAFT [57] reduces the demand for tag storage by applying a coarser granularity for the heap.

We believe that hardware-based approaches hold significant promise for addressing the performance challenges of multi-tag taint analysis; however, existing solutions are designed for single-tag [48, 57] or small-sized-tag [6, 13, 28, 42, 51] taint analysis. It is difficult to extend them to support multi-tag analysis for traceability of its source due to limited tag space and inefficient computing units. Further, they suffer from two additional limitations. The first issue is that existing taint propagation models are constrained by limited hardware resources and fail to fully exploit the parallelism and computational efficiency of modern hardware. The second issue is that current analysis approaches suffer from poor hardware scalability and rely heavily on strong assumptions. For example, RAFT [57] can only handle programs that do not use complex heaps, such as embedded programs, and are unable to analyze actual applications. Moreover, fixed functionality and a lack of configurable interfaces limit their adaptability to varying analysis needs.

2.3 Our Solution

In this paper, we propose **MULCOTAINT** to make M-DTA efficient, and the overview is shown in Figure 3. We first decouple the taint calculation from normal execution by utilizing a coprocessor architecture. When taint analysis is needed, relevant instructions from the CPU are forwarded to the coprocessor for processing (highlighted in red in Figure 3).

Moreover, we observe that the current taint calculation logic is inefficient. As shown in Figure 2 (b), taint calculation is performed byte-by-byte for each source operand, leading to significant overhead. Our idea is to set taint tags as vectors, and thus the logic-intensive problem of linear taint calculation turns into a compute-intensive vectorized calculation problem. The vectorized calculation logics can easily be represented as microcodes with corresponding circuit implementations, achieving hardware acceleration (as shown with yellow in Figure 2 (c), details in Section 3).

To support our new taint calculation logics, we require wide-bit vectors of taint tags and thus design a tag manage-

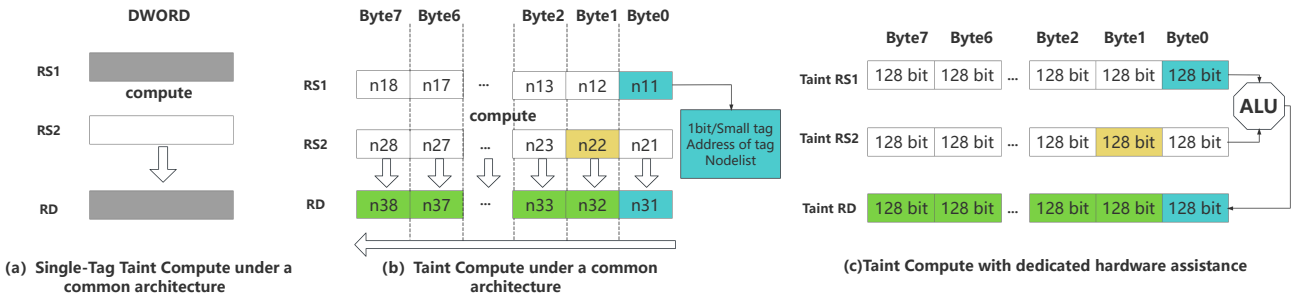

Figure 2: Sketch of taint calculation logic. The single-tag scheme simply performs an XOR operation on 0 or 1. Conventional multi-tag schemes require obtaining the taint tags of two operands byte by byte for calculation, and the tag of the lower byte of complex instructions also may affect the tag of the upper byte. MULCOTAINT uses a dedicated hardware unit to implement parallel processing of tags in a single step without losing semantic information. Blue + Yellow = Green.

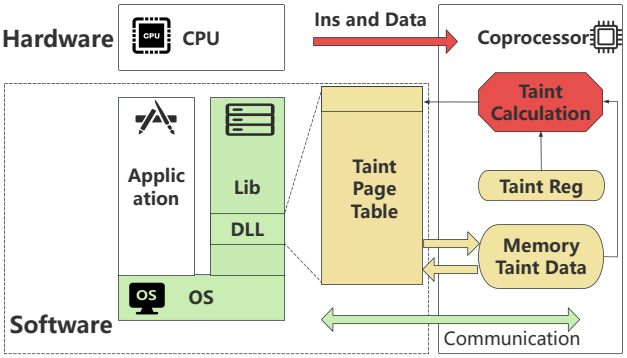

Figure 3: Overview of MULCOTAINT.

ment system based on a taint page table for storage and access (as shown with yellow in Figure 3, details are in Section 4). We also provide configurable analysis support through a dynamic link library (DLL), and rely on OS-level support to coordinate communication between the CPU and coprocessor (as shown with green in Figure 3, details are in Section 5).

3 DESIGN of Taint Calculation

We first introduce how to design taint tags as wide-bit vectors, and how to represent taint rules through dedicated microcodes.

3.1 Taint Tag as Vector

Our goal is to conduct multi-tag taint analysis, i.e., we aim to distinguish each byte of registers or memory in the execution with different bytes of the taint source. Previous works utilize a list to save multiple tags for registers or memory, as shown in Figure 2 (b). To reduce space overhead, FineDIFT [6] adopts a CAM-like structure to store the ranges of tagged objects; however, it is coarse-grained, and the number of objects it can mark is limited by the size of the CAM-like structure and the number of regions. It's unsuitable for managing byte-level, large-scale taint data. To support vectorized taint calculation, we design tags as vectors, as shown in Figure 2 (c).

Currently, we set the granularity as 128, i.e., we can clarify if a byte is affected by 128 different taint sources. Additionally, as we support 64-bit program analysis, the maximum data unit we process is 64-bit data (8 bytes). Thus, in our prototype of MULCOTAINT, we perform a byte-grained taint analysis and set a taint tag as a 1024-bit-width vector (i.e., calculated as $128 * 8$) width for registers and memory.

3.2 Taint Rules Based on Microcodes

We represent the taint rules for instructions by microcodes based on the instruction semantics. During the taint analysis of each instruction, it is necessary to fetch the memory and register operands accessed by the instruction, retrieve the corresponding taint data of the operands, and then perform the computation. Thus, we design 11 custom microcodes besides the general microcode provided by the existing hardware framework (i.e., PHMon [15], which we design MULCOTAINT based on). Details are shown in Appendix A.3.

Specifically, there are four types of custom microcodes: tag fetch/save, tag calculation, data fetch/save, and memory filter. The tag fetch/save type of microcode is used for reading and writing tag vectors between memory and taint register, as well as for reading and writing the corresponding tag according to the data granularity of the instruction (e.g., LW instruction reads a word, LB instruction reads a byte). The type of tag calculation is used for the actual tag calculation process, which takes one or two taint registers as inputs and one taint register as output, and completes the operation according to the instruction. The data fetch/save type is used in conjunction with the general microcode to implement indexing and access to the taint page table. The memory filter type is used

We chose 128 for engineering reasons, as our FPGA resources cannot accommodate a larger number of tags.

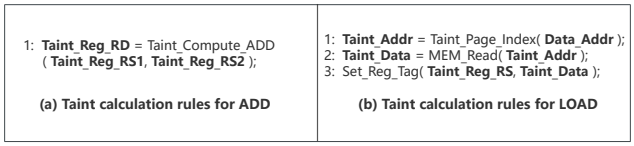

Figure 4: Microcode representation of taint rules.

to optimize the taint page table mechanism, which is further introduced in Section 5.2.3.

Each type of microcode has the operands of inputs and outputs, and we explain them in the column of More Details in Appendix A.3. Currently, we support forty-two instructions of RV64I Base Instruction Set [24] in our prototype (as shown in Table 4 in the Appendix). We illustrate the taint rules based on the following two examples.

The Calculation Rule of ADD instruction. We take ADD as an example (e.g., *ADD Reg_A*, *Reg_B*) to introduce the rules for calculation instructions. As shown in Figure 4 (a), the calculation rule is only necessary to consider how to obtain the taint data of the source registers, how to perform taint calculation, and how to give the taint result to the destination register. Since the two source operands and one destination operand of ADD are registers, the coprocessor hardware can directly obtain these register numbers when parsing the instruction, and take the value of the corresponding taint register as an input to participate in the computation. Therefore, the calculation rule of this instruction is only described by the pseudocode Taint_Compute_ADD, and the corresponding function code of microcode is FN_ALU_TAINT_REG.

The Calculation rule of LOAD instruction. We take LOAD as an example to introduce the rules of the memory access class instructions (shown in Figure 4 (b)). The rules for LOAD are supposed to give the byte-by-byte taint data corresponding to the memory address to each byte of the taint register corresponding to the destination register RD. Firstly, the 64-bit general-purpose microcode, including FN_WRITE_DATA, FN_READ_DATA, FN_ADD, FN_OR, etc., is used to implement the pseudocode of indexing the storage location of the taint data of the memory address in the taint page table (i.e., Taint_Page_Index in Figure 4 (b)). Then, the complete taint data is obtained by the coprocessor's memory access by the pseudocode MEM_Read based on the microcode FN_READ_TAG). Finally, the taint data is assigned to the taint register corresponding to the destination register with the pseudocode Set_Reg_Tag, and the corresponding microcode is FN_ALU_TAINT_REG.

4 DESIGN of Tag Management

Software schemes of taint analysis propose shadow memory to handle tagged memory, which takes additional memory at a certain corresponding compression ratio of the process

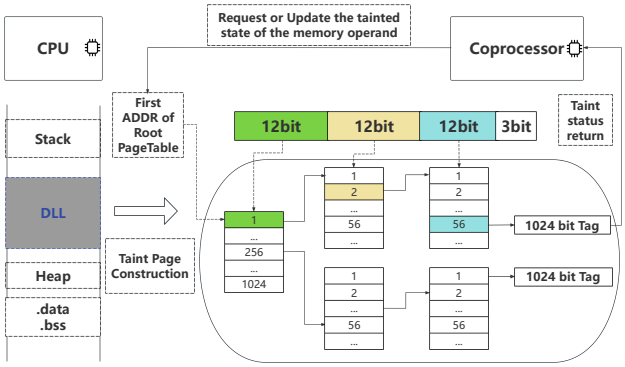

Figure 5: Taint Page Table.

memory. It is impossible for the limited resources of hardware schemes, let alone supporting M-DTA. To achieve the same taint granularity as MULCOTAINT, shadow memory schemes would require 128 bits of storage for each byte in the virtual address space. It motivates us to design new tag management.

4.1 Tag Storage

As introduced in Section 3.1, the taint tag is a 1024-bit-width vector. We support all 32 general-purpose registers of 64-bit width by constructing the corresponding large-bit taint registers of 1024 bits in the coprocessor.

The tag for memory is designed the same as the register tags, but it could be more complex as it is difficult to store them with hardware resources alone. Using the software memory interface and DDR memory is the inevitable choice, and we construct a taint page table mechanism to allocate a fixed-size space for each memory address accessed by the application. Specifically, in our prototype of MULCOTAINT, as the virtual address of RISC-V64 architecture is 39-bit in virtual memory layout SV39, we divide the virtual address into three parts and build a three-level index, as shown in Figure 5. For 64-bit data (8 bytes), we need 3 bits as an index to find its taint tag for each byte (i.e., $8 = 2^3$). Thus, the lowest 3 bits of the virtual address are kept, and the remaining 36 bits are used as indices for the taint page table.

4.2 Tag Access

The object of our tag calculation is a 1024-bit-width tag vector, so passing tag vector between taint registers and memory requires the ability to read and write large bit-width tags from memory. But existing architecture is not suitable for the 1024-bit-width taint data access, as the bus is 64-bit wide. Thus, we design a new memory access logic in the coprocessor, which utilizes a 64-bit-width bus to achieve reading and writing of 1024-bit-width taint tags.

For the Load instruction, our access module reads 128 consecutive bytes starting from the address in memory, i.e., 16

64-bit read operations are issued to the bus. The complete 1024-bit taint tag is then spliced into the corresponding taint register. For Store instruction, our access module splits the 1024-bit taint register into 16 parts and sends consecutive memory write operations to the bus.

At the same time, to be able to accurately describe the normal operations in taint calculation (e.g., looking up the taint page table) and reduce the overhead, it is not necessary to replace all memory access with the new 1024-bit memory access channel, and we retain the 64-bit width memory access channel. Therefore, the memory access of the coprocessor is implemented in two forms: 64-bit regular data access and 1024-bit taint tag access. We provide two software interfaces, MEM_WoR_1024 and MEM_WoR_64, which are invoked directly via microcodes including FN_WRITE_TAG, FN_READ_TAG, FN_WRITE_DATA, FN_READ_DATA when describing calculation rules for specific CPU instructions.

In addition, since MULCOTAINT is byte-grained taint analysis, we implement fine-grained tag fetching and saving based on large tag access. In our prototype for RISC-V64, the specific instructions are the Load and Store instructions, which have different sizes. We access the tag at the corresponding location based on the function code of the instruction and the lower 3 bits of the address. For example that $LBRD, addr$ is to read one byte from memory, it is processed by first using the high bit of $addr$ for indexing on the tainted page table to find the corresponding 1024-bit tag. And then, according to the function code of the load instruction and the low 3 bits of $addr$, we get the actual tag corresponding to a byte in the 1024-bit tag, and put it into RD corresponding taint register.

5 IMPLEMENTATION of MULCOTAINT

Generally, a taint analysis solution includes four components: taint source labeling, rule-based calculation, taint tag management, and sink point detection. As introduced in the previous two sections, we design the core taint engine of rule-based calculation and taint tag management. Next, we introduce how to implement the core taint engine via hardware/software co-design. And we introduce how to support taint source labeling and sink detection via interface implementation.

5.1 Taint Engine Implementation

5.1.1 Taint Analysis Datapath based on Hardware Units

The hardware architecture is shown in Figure 6, including Trace Unit, Monitoring Unit, Control Unit, and ALU Unit.

Trace Unit is to get instructions and related data. We make minimal modifications to the CPU pipeline's Write Back (WB) stage to implement the Trace Unit. When the instruction is transmitted to the WB stage, the instruction has been determined to be executed. So we can intercept the instruction

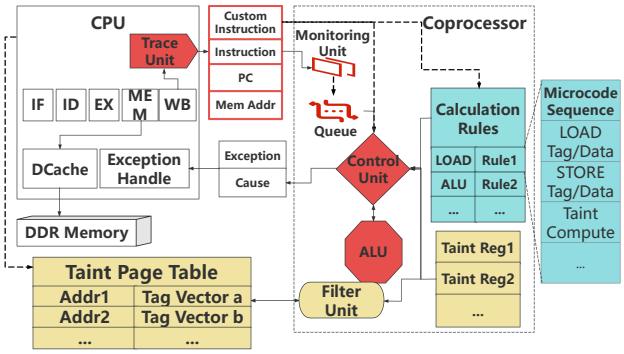

Figure 6: Coprocessor Architecture of MULCOTAINT. IF, ID, EX, MEM, and WB represent the five stages of the CPU's pipeline: Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write Back.

information at this time, including instructions, addresses of the access memory, etc.

Monitoring Unit collects instruction information sent by the Trace Unit and caches it in a queue (i.e., Queue in the Figure 6). With a preset range of instructions, Monitoring Unit filters the instructions passing to Queue. The actual taint calculation is done on instructions received from Queue in the coprocessor. In this way, MULCOTAINT decouples the normal execution and the taint calculation.

Control Unit is the core management unit of the coprocessor, which is first responsible for extracting the information from Queue, identifying the type of instruction, and then finding the corresponding calculation rules (i.e., expressed as microcode sequences) according to the type of instruction. It parses the microcode sequence and gets the taint tags of the registers and memory addresses associated with the instruction. After getting the taint tags, both taint rules and taint tags are sent to the ALU (Arithmetic Logical Unit) Unit to perform the actual taint calculation.

ALU Unit is responsible for vectorized taint calculation in the coprocessor part. As described in Section 3, the taint rules are presented as a microcode sequence, and we implement dedicated circuits for each microcode in ALU Unit for further acceleration instead of using a purely software generic instruction implementation.

The complete datapath also includes invoking large-bit-width memory-accessed microcode to read and write taint tags (yellow in Figure 6). The computation rules are configurable and set prior to taint analysis (blue in Figure 6).

5.1.2 Software/Hardware Collaboration based on Operating System

In order to correctly utilize the hardware's taint calculation capabilities, we also need to implement functional support at the operating system layer. As we set QUEUE as the bridge

to send instructions that requires taint analyst, the size of QUEUE determines the maximum number of instructions that can be carried (we set 9000 in our prototype). We suspend the CPU when QUEUE is full, and resume the execution after the coprocessor completes all operations. This mechanism is implemented by the Rocket Custom Coprocessor Interface (RoCC) coprocessor interrupt in PHMon [15], as shown in Figure 7 (a). Specifically, when the coprocessor detects that the Queue is full, it triggers an interrupt to notify the CPU and suspends its own computation. The CPU enters the interrupt processing flow to suspend the program execution and inform the coprocessor to continue after processing the interrupt.

Besides, as the processing speed of the CPU and coprocessor is not the same, we design a wait mechanism in the operating system (as shown in Figure 7 (b)), and implemented a dedicated system call SYSCALL_WAIT to let the CPU side wait for the coprocessor to finish the computation. When the CPU detects that the coprocessor has finished analyzing all the instructions in the Queue, it ends the work of the coprocessor and shows the results.

Thirdly, we implement a distribution mechanism for the first address of the taint page table (as shown in Figure 7 (c)). We add a new data structure in OS to store the first address of the taint page table and provide a system call SYSCALL_PT to pass this address to the OS. In addition, we pass this address to the coprocessor through the RoCC instruction, which is implemented by PHMon [15]. In this way, both the OS and the coprocessor can utilize the address for indexing taint tags.

Finally, we handle the case of process switching by modifying the OS (as shown in Figure 7 (d)). When switching from the target program to another program, OS stops instruction monitoring using RoCC instruction. When switching back to the target program, OS utilizes RoCC instruction to continue instruction monitoring. This prevents instructions of other programs from interfering with the taint propagation process. More details of these two instructions are introduced in Section 5.2.1.

5.2 Interface Implementation

5.2.1 Start/End Analysis

Taint analysis does not always need to start from the program entry, so we implement interfaces to start and stop it on demand. We design two RoCC-based control instructions, *Monitor_Start* and *Monitor_End*, to signal the coprocessor. The CPU interacts with the coprocessor through the standard RoCC interface. When taint analysis starts, Monitoring Unit begins fetching instructions and data into QUEUE, initiating the taint propagation workflow. By default, the taint analysis starts at the target program's main function. And the two control instructions can be instrumented in the target with-/without source code, allowing users to focus on the part of the target they are interested in.

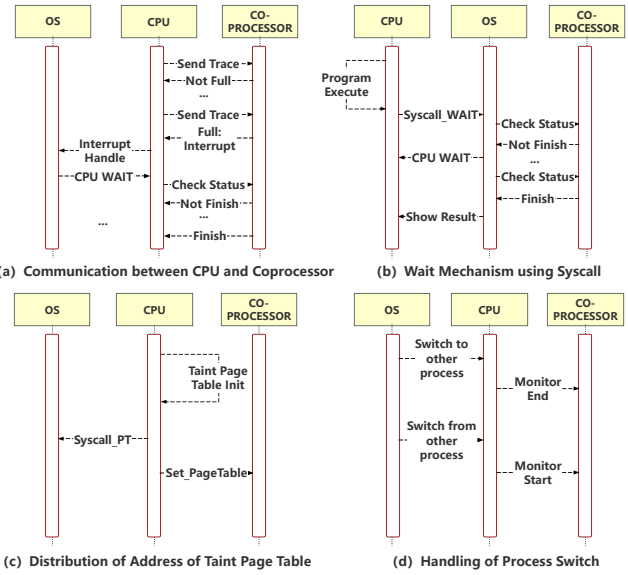

Figure 7: Software/Hardware Collaboration in MULCOTAINT.

5.2.2 Set Source/Sink Point

Defining source and sink points is essential in taint analysis, and we provide an intuitive interface for this purpose. Our interface allows users to directly label memory locations or registers with taint tags, providing more flexibility than using control instructions. This enables fine-grained configuration tailored to specific analysis goals, such as detecting different vulnerability types.

In vulnerability analysis, a common goal is to assess how external inputs affect program execution. By default, I/O APIs are defined as source points. We modify the kernel to monitor program-specific I/O APIs at the system level and generate taint labels for input data. For cases requiring manual tainting of memory, we use the same dynamic binary instrumentation process employed for start/end monitoring. Regarding sink points, which focus on program crashes, we extend the OS exception handling mechanism to capture and process crash events during analysis.

5.2.3 Configure Memory Filter

In practice, certain memory regions, such as program code and dynamic library segments, are typically read-only and executable. Therefore, taint calculation usually does not affect data in these regions. Taint propagation only impacts them if a vulnerability occurs. These events can be detected by the default sink points mentioned earlier. To optimize analysis, we provide an interface for memory filtering, implemented via Filter Unit shown in Figure 6. By configuring Filter Unit, the coprocessor can skip instructions accessing taint-independent memory during taint computation, thereby significantly reduc-

ing the analysis workload. Our prototype includes five filter units dedicated to filtering memory regions.

5.3 Implementation Details

MULCOTAINT is implemented on Zynq UltraScale+ board (with costs $2067), using the open-source RISC-V SoC RocketChip [2] as the baseline architecture. We build the taint computation engine based on the PHMon coprocessor framework [15]. PHMon provides an efficient and programmable hardware monitor capable of detecting user-defined events and performing simple actions on user programs, but it does not natively support taint tracking. We design and implement the entire taint analysis logic ourselves, including all necessary components for the taint propagation workflow and a custom tag management mechanism.

Specifically, we redesigned and implemented the Trace Unit, Control Unit, ALU, Microcode, and Memory Access modules of PHMon, as the blocks colored in Figure 6. We also added a list of Taint Reg, Filter Unit and their access and calculation functions.

On the software side, we add our functional logic to PHMon's OS code, e.g., control of monitoring during trap processing and sink point processing. In addition, we implemented the taint page table mechanism and dynamic binary instrumentation with a DLL.

6 Usage of MULCOTAINT

We provide the necessary setup steps required prior to execution. Typically, this involves two stages: (1) instrumenting the target program, and (2) initializing the taint engine with appropriate configuration parameters.

6.1 Target Program Instrumentation

As mentioned in Section 5.2, we may instrument the target program with two kinds of instructions, i.e., starting/ending analysis and setting source/sink points. For the target with source code, it is easy to instrument these instructions.

For binaries, we propose a dynamic binary instrumentation method. We leverage the mechanism of the runtime dynamic linker (i.e., LD_PRELOAD) that allows the dynamic library to be loaded before the target program, and the GCC keyword __attribute__((constructor)) to execute codes before the main function. Thus, we implement a dynamic library to modify the program binary after the target program is loaded into memory. Specifically, we save the critical instructions (e.g., the location of starting/ending analysis), and replace them with jump instructions. The jump target is a function implemented in the dynamic library that responds to the monitor enable/disable logic. At the end of the function, the saved instructions are copied back into the code segment of the loaded program binary, and the execution jumps back to the normal logic of the target program to continue.

6.2 Taint Engine Initialization

Before conducting taint analysis with MULCOTAINT, we should initialize the taint engine, including assigning initial values to registers and allocating space for taint page tables. We implement the initialization as a preprocess, which only needs to be run once for multiple analysis tasks.

As introduced in Section 5.2, we also provide configuration of the memory filter. We keep the taint propagation rules configurable, allowing users to initialize custom rules into the taint engine during setup. A set of default rules is provided, and users can further customize behavior by modifying the corresponding microcode sequences.

Supporting new instructions or propagation rules is done by configuring Monitoring Unit and microcode, without hardware changes. Existing rules can be reused for instructions with similar dataflows. New microcode can be supported by ALU extension. Additional discussion is in Section 8.

7 EVALUATION

We evaluate MULCOTAINT on real-world programs by answering the following research questions (RQ)s:

RQ1: What is the performance overhead of MULCOTAINT, compared to existing tools?

RQ2: How effective is MULCOTAINT for vulnerability analysis, compared to existing tools?

RQ3: What is the resource cost of MULCOTAINT?

7.1 Dataset Setup

To comprehensively evaluate MULCOTAINT, we constructed a diverse dataset consisting of 32 program binaries from five types as summarized in Figure 8:

SPEC CPU Benchmark: We use SPEC 2006 [21] as it is a benchmark with standardized CPU-intensive programs to evaluate general performance overhead, which is also used in related works (e.g., HardTaint [59] and Raft [57]). We also test SPEC 2017 as supplementary experiments.

Complex Applications: We follow HardTaint [59] to include large-scale real-world software Nginx and PHP, for a further test of performance evaluation.

Juliet Test Suite [44]: This test suite includes a wide range of Common Weakness Enumerations (CWEs) and is commonly used in vulnerability analysis works. We excluded programs that could not be built on a RISC-V Linux system and selected 10 binaries representing different CWE types.

CTF Challenges: We evaluated MULCOTAINT on five capture-the-flag (CTF) programs to test its performance on typical security challenges.

Table 1: Compared related works.

	Type[1]	Implementation	Available
PANDA [16]	M	Software (QEMU)	Yes
TaintRabbit [18]	M	Software (DynamoRIO)	Yes
FindDIFT [6]	S	Hardware (coprocessor)	No
RAFT [57]	S	Hardware (coprocessor)	No[2]
HardTaint [59]	S	Hardware (Intel PT)	No
Our	M	Hardware (coprocessor)	Yes

[1] M indicates Multi-tag and S indicates Single-tag.
[2] We cannot run Raft because of no specific hardware board.

Real-world CVEs: We selected 10 binaries containing known Common Vulnerabilities and Exposures (CVEs) to assess the applicability of analyzing real-world vulnerabilities.

In our experiments, SPEC CPU 2006 CINT and complex applications (i.e., PHP and Nginx) are used in performance evaluation. The other three types are used for both performance evaluation and vulnerability analysis. To conduct the experiments, we need inputs for all test programs. For the Juliet suite, we used the built-in test inputs that trigger known weaknesses. For CTF challenges and CVEs, we utilized PoC files obtained from official sources and online repositories. SPEC CPU programs were executed with standard benchmark inputs. For PHP and Nginx, we employed PHPBench [43] workloads and HTTP requests via curl, respectively. For example, HR/reporter render in Figure 8 represents that the input to PHPBench's report render test item is HtmlRendererBench, and VS/progress represents VariantSummaryFormatterBench. We access a text nginx webpage with HTTP GET requests and repeated five times to get average values.

7.2 RQ1: Performance Overhead

7.2.1 Experiment Setup

We selected several state-of-the-art dynamic taint analysis tools, including both software and hardware-based implementations, for performance comparison. On the software side, we considered representative works that support multi-tag dynamic taint analysis and finally selected TaintRabbit (TB for short) [18] and PANDA (PD for short) [16] (unfortunately, the most recent work, AirTaint [45], is not publicly available). For hardware-based DTA works, we included HardTaint [59], RAFT [57], and FineDIFT [6]. These five systems serve as the baselines to evaluate the performance overhead of MULCOTAINT. More details are presented in Table 1.

In comparison with software solutions, we configured TaintRabbit in multi-tag mode with the specific attribute RAW_BITVEC_FP. It should be noted that TaintRabbit operates on a 32-bit architecture, which supports 32-bit labels per byte, i.e., it can distinguish 32 different sources, whereas MULCOTAINT provides 1024-bit tags to distinguish 128 different sources. PANDA relies on QEMU [4] and sets the same architecture configuration (i.e., X86) as TaintRabbit. In contrast, they run on a high-performance PC with a significantly higher clock frequency than our FPGA-based prototype. Therefore, to ensure a fair performance comparison, we evaluate the relative performance overhead by computing the slowdown multiples within each respective platform, rather than relying on absolute runtime measurements.

In comparison with hardware-based solutions, due to their closed-source nature and lack of reproducibility, we use the performance data reported in their original papers.

For MULCOTAINT, we generated a second-level taint page table for the program during the initialization phase and used three filter units to mask taint computations from other memory locations. We initialized the taint engine with its default configuration, which required a one-time setup of approximately 2 seconds. To ensure consistent results across runs, we disabled the application address space randomization feature in the system kernel during all experiments.

7.2.2 Results to Software Solutions

A more reasonable way to compare the performance results is to consider the architecture differences, thus use the relative performance value (i.e., increased overhead) based on the baseline time overhead on the architecture. We present the detailed data in Figure 8 as the column of Cmp of REL. MULCOTAINT is 0.7% (1/136.65) of TaintRabbit, and 0.09% (1/1,117.51) of PANDA on average.

Specifically, according to Figure 8, our system can complete the taint analysis process efficiently from 18,162 us to 2,065,954,000 us (2,065s), for binaries with sizes from 12.2 to 7,128.0KB. It brings 1.07 to 15.81 times overhead to the normal execution on PHMon (values of Column Board). As a comparison, TaintRabbit takes 11,265 us to 1,514,380,000 us (1,514.38 s) for the same targets, which is 377.6 times the normal execution on average (8.08-1037.03x). PANDA takes 84,000,000 to 34,790,000,000 us (34,790 s), which is 3283.39 times the normal execution on average (401.26-14018.11x). The overhead increase is also illustrated visually in Figure 9.

We also measured the average overhead across each test set. Compared with TaintRabbit, MULCOTAINT achieves the smallest improvement on the SPEC CPU test set (15.79×) and the largest improvement on the Juliet test set (301.04×). Compared with PANDA, our work achieves the smallest improvement on the SPEC CPU test set (364.72×) and the largest improvement on the CVE test set (1871.86×).

Although it is unfair to compare the absolute time of MULCOTAINT and the other two works, we calculate the value of 'comparison of absolute time' as the Column Cmp of ABS(TR) and Cmp of ABS(PD) in Figure 8. We obtained the absolute time through the APIs provided by sys/time.h. On average, MULCOTAINT has the advantage that MULCOTAINT only takes 13.3% (1/7.49) of the time of TaintRabbit, and takes 0.02% (1/4,968.67) of the time of PANDA.

We measured the average absolute runtime overhead across each test set. Compared with TaintRabbit, our work achieves

Program		Bin Size (KB)	Input Size (Byte)	TaintRabbit (us)	Panda (us)	MulcoTaint (us)	Frequency-converted MulcoTaint(us)	Frequency-converted Cmp of ABS(TB)	Frequency-converted Cmp of ABS(PD)	Cmp of ABS(TB)	Cmp of ABS(PD)	PC (us)	PandaQEMU (us)	Board (us)	Cmp of REL(TB)	Cmp of REL(PD)
Juliet Test Suite	Juliet-CWE121	17.60	32	28,310	84,000,000	49,853	498.53	56.79	168,495.38	0.57	1,684.95	31	21,000	17,616	322.70	1,413.43
	Juliet-CWE122	17.70	31	60,183	89,000,000	49,552	495.52	121.45	179,609.30	1.21	1,796.09	105	26,000	17,256	199.71	1,192.71
	Juliet-CWE124	17.60	100	32,162	864,000,000	31,301	313.01	102.75	2,760,295.20	1.03	27,602.95	44	2,071,000	6,033	140.84	80.38
	Juliet-CWE127	17.60	100	48,080	104,000,000	36,890	368.90	130.33	281,919.22	1.30	2,819.19	61	23,000	21,234	452.99	2,598.70
	Juliet-CWE188	17.50	4	37,080	88,000,000	47,012	470.12	78.87	187,186.25	0.79	1,871.86	58	22,000	17,209	234.18	1,465.20
	Juliet-CWE391	17.60	4	54,549	104,000,000	33,946	339.46	160.69	306,368.94	1.61	3,063.69	61	26,000	14,093	371.06	1,659.75
	Juliet-CWE398	17.50	8	44,363	87,000,000	48,214	482.14	92.01	180,445.51	0.92	1,804.46	232	29,000	16,429	65.26	1,023.89
	Juliet-CWE469	17.60	24	50,816	91,000,000	39,302	393.02	129.30	231,540.38	1.29	2,315.40	182	24,000	14,328	101.90	1,383.82
	Juliet-CWE843	17.50	8	36,161	93,000,000	51,248	512.48	70.56	181,470.50	0.71	1,814.70	55	25,000	16,529	212.09	1,200.00
	Juliet-CWE126	17.60	100	40,444	105,000,000	23,403	234.03	172.82	448,660.43	1.73	4,486.60	39	24,000	20,606	909.68	3,837.72
	Average			43,215	170,900,000	41,072	410.72	111.56	492,599.11	1.116	4,925.99	86.8	229,100	16,133	301.04	1,585.56
CTF Subjects	CTF-stack_overflow	12.20	105	762,000	890,000,000	22,968	229.68	3,317.66	3,874,956.46	33.18	38,749.56	93,000	2,218,000	10,834	3.86	189.27
	CTF-heap_overflow	12.30	305	773,000	919,000,000	36,570	365.70	2,113.75	2,512,988.79	21.14	25,129.89	83,000	2,243,000	11,674	2.97	130.90
	CTF-heap_uaf	12.30	145	744,000	1,015,000,000	21,599	215.99	3,444.60	4,699,291.63	34.45	46,992.92	87,000	2,067,000	11,413	4.52	259.81
	CTF-stack_overflow_intover	12.30	39	11,265	104,000,000	31,695	316.95	35.54	328,127.46	0.36	3,281.27	25	22,000	11,648	165.66	1,737.97
	CTF-stack_used_un_init	12.20	8	18,722	102,000,000	21,732	217.32	86.15	469,353.95	0.86	4,693.54	22	23,000	9,498	371.62	1,936.59
	Average			461,797	606,000,000	26,913	269.13	1,799.54	2,376,943.66	18.00	23,769.44	52,609	1,314,600	11,013	109.73	850.91
Programs with CVEs	CVE-2023-50471-cjson	780.10	8	693,000		18,939	189.39	3,659.12		36.59		67,000		17,754	9.66	
	CVE-2023-50472-cjson	780.10	8	695,000		26,839	268.39	2,589.52		25.90		86,000		19,601	5.90	
	CVE-2024-31755-cjson	780.10	8	655,000		18,162	181.62	3,606.43		36.06		71,000		15,274	7.76	
	CVE-2023-39113-gif2tga	46.50	190	625,000	127,000,000	276,016	2,760.16	226.44	46,011.83	2.26	460.12	1,000	34,000	144,020	325.52	1,945.46
	CVE-2021-36530-gif2tga	43.10	56	714,000	103,000,000	66,307	663.07	1,076.81	155,338.05	10.77	1,553.38	1,000	19,000	36,524	392.31	2,978.60
	CVE-2021-36531-gif2tga	46.10	56	626,000	97,000,000	4,923,769	49,237.69	12.71	1,970.04	0.13	19.70	1,000	22,000	4,306,715	549.12	3,867.62
	CVE-2021-34055-jhead	209.80	3,310	843,000	91,000,000	336,943	3,369.43	250.19	27,007.54	2.50	270.08	79,000	34,000	44,961	1.42	357.34
	CVE-2021-3496-jhead	199.60	1,584	776,000	113,000,000	24,453,340	244,533.40	3.17	462.10	0.03	4.62	3,000	43,000	17,142,835	180.89	1,837.70
	CVE-2021-28277-jhead	199.20	362	878,000	162,000,000	460,931	4,609.31	190.48	35,146.26	1.90	351.46	1,000	29,000	66,197	126.15	802.62
	CVE-2021-28278-jhead	199.10	25,898	839,000	156,000,000	274,769	2,747.69	305.35	56,774.96	3.05	567.75	1,000	25,000	57,897	176.63	1,313.68
	Average			734,400	121285714.3	3,085,602	30,856.02	1,192.02	46101.54	11.92	461.02	31,100	29,428.57	2,185,178	177.54	1,871.86
SPEC_CPU	400	1,948.00	3,469	2,120,000	215,900,000	2,808,300	28,083.00	75.49	7,687.93	0.75	76.88	4,000	287,000	567,000	107.07	151.97
	403	3,478.00	162,091	13,400,000	11,528,000,000	2,065,954,000	20,659,540.00	0.65	558.00	0.01	5.58	818,000	12,947,000	137,534,000	1.09	59.28
	429	21.40	1,163,146		34,790,000,000	657,406,000	6,574,060.00		5,292.01		52.92	1,282,000	20,062,000	258,331,000		682.72
	445	4,526.50	1,737	1,789,000	2,252,000,000	233,303,000	2,333,030.00	0.77	965.27	0.01	9.65	90,000	1,795,000	14,757,000	1.26	79.35
	456	1,101.00	4,462	4,987,000	9,687,000,000	942,500,000	9,425,000.00	0.53	1,027.80	0.01	10.28	197,000	1,962,000	127,881,000	3.43	669.92
	458	903.10	176	128,064,000	30,365,000,000	32,343,000	323,430.00	395.96	93,884.30	3.96	938.84	2,151,000	30,325,000	12,801,000	23.53	395.78
	462	49.10	8	999,000	1,699,000,000	159,219,000	1,592,190.00	0.63	1,067.08	0.01	10.67	35,000	715,000	12,210,000	2.19	182.23
	464	1,355.00	14,273	1,514,380,000	6,967,000,000	15,109,000	151,090.00	10,023.03	46,111.59	100.23	461.12	8,668,000	497,000	1,903,000	22.00	1,765.51
	471	2,370.00	1,100	21,523,000	9,287,000,000	1,348,250,715	13,482,507.15	1.60	688.82	0.02	6.89	289,000	7,126,000	106,806,000	5.90	103.27
	473	861.80	65,544	73,782,000	15,838,000,000	1,001,804,430	10,018,044.30	7.36	1,580.95	0.07	15.81	5,273,000	10,515,000	84,101,000	1.17	126.47
	483	7,128.00	28,074	6,118,000	4,305,000,000	164,294,000	1,642,940.00	3.72	2,620.30	0.04	26.20	86,000	2,936,000	12,623,000	5.46	112.62
	401	797.70	278,475	21,897,000	29,826,000,000	731,092,000	7,310,920.00	3.00	4,079.65	0.03	40.80	2,167,000	39,739,000	46,245,000	0.64	47.47
	Average			162,641,727	13,063,325,000	612,840,287	6,128,402.87	955.70	13,796.98	9.56	137.97	1,755,000	10,742,167	67,979,917	15.79	364.72
PHP	Component/report generate	4,348.00	6,540	2,165,000		5,879,992	58,799.92	36.82		0.37		3,000		694,237	85.20	
	HR/reporter render	4,348.00	3,203	2,341,000		6,777,806	67,778.06	34.54		0.35		23,000		843,029	12.66	
	ParserFoo/expression	4,348.00	2,813	2,170,000		6,246,133	62,461.33	34.74		0.35		3,000		714,889	82.76	
	test/data	4,348.00	3,559	2,074,000		11,204,870	112,048.70	18.51		0.19		5,000		1,288,690	47.73	
	testa/assertion	4,348.00	4,554	2,198,000		66,125,424	661,254.24	3.32		0.03		33,000		4,605,868	4.64	
	testr/console	4,348.00	2,699	2,062,000		5,606,578	56,065.78	36.78		0.37		6,000		659,743	40.43	
	VS/progress	4,348.00	5,303	2,188,000		8,575,661	85,756.61	25.51		0.26		5,000		1,004,342	51.24	
	Average			2,171,143		15,773,781	157,737.81	27.17		0.27		11,143		1,401,543	46.38	
Nginx	CURL	1,052.00	294	904,990	361,000,000	395,176	3,951.76	229.01	91,351.70	2.29	913.52	1,175	55,376	92,280	179.95	1,523.15
	Average			904,990	361,000,000	395,176	3,951.76	229.01	91,351.70	2.29	913.52	1,175	55,376	92,280	179.95	1,523.15
Total Average				41,255,616	4,648,825,714	166,584,364	1,665,843.64	749.22	496,866.73	7.49	4,968.67	482,535	3,943,754	18,838,450	136.65	1,117.51

Figure 8: Details of performance overhead to software.

the largest improvement on the CTF test set (18.0×), while the PHP test set shows the weakest performance and is the only case where we perform worse than TaintRabbit (0.27×), but we support larger tag size. For PANDA, our work achieves the smallest improvement on the SPEC CPU test set (137.97×) and the largest on the CTF test set (23,769.44×).

Our current implementation is constrained by the hardware and operates at 50 MHz today, which leads to lower absolute wall-clock performance compared to approaches evaluated on high-performance CPUs (e.g., 5GHz). We think comparing absolute performance is not fair, thus we approximately convert our results to the same frequency of the general-purpose CPU used in the TaintRabbit and PANDA experiments (50 MHz vs. 5 GHz). We provide frequency-converted absolute time in the Column Frequency-converted MulcoTaint(us). The Columns Frequency-converted Cmp of ABS(TB) and Frequency-converted Cmp of ABS(PD) show the overhead multiplier of TaintRabbit and PANDA over MULCOTAINT under the frequency-converted absolute time. The average ratio of TaintRabbit's absolute time to Frequency-converted MulcoTaint's is 749.22×, and PANDA's average ratio is 496,866.73×. Then we measured the average comparison of absolute time across each test set. For TaintRabbit, our work achieves the largest improvement on the CTF test set (1,799.54×), while the PHP test set shows the smallest performance improvement (27.17×). For PANDA, our work achieves the largest improvement on the CTF test set (2,376,943.66×) and the smallest on the SPEC CPU test set (13,796.98×).

It noticed that both TaintRabbit and PANDA miss some data in Figure 8. PANDA's file_taint plugin was unable to correctly parse PHP input files, resulting in the inability to mark tags. And, PANDA was also unable to analyze CJSON's three CVEs, as their vulnerabilities did not originate from file input. TaintRabbit was stuck during the SPEC CPU-429 testing and was unable to complete the analysis correctly. We also provided data of SPEC CPU 2017 in Section A.4.

Further studies show that the performance advantage of MULCOTAINT compared to other systems varies due to the program types. Specifically, our tag computation unit efficiently accelerates compute-intensive workloads. However, for I/O-intensive programs, the overhead increases because of the cost of reading and writing wide taint tags associated with memory operations. Taking CVE-2021-34055-jhead and CVE-2021-28277-jhead as examples, the jhead program frequently reads, writes, and copies data from memory, resulting in tag passes of in-memory data taking up a lot of time, with arithmetic-type instructions taking up less time. The slowdown of SPEC-401 (bzip2), SPEC-403 (gcc), and SPEC-445 (gobmk) is also due to the same reasons. As a result, the multiplier of performance improvement is reduced. In addition to this, programs with a large number of arithmetic-type instructions received a significant increase in acceleration multiplier. Nevertheless, considering that our tag size is four times that of TaintRabbit, our performance is still superior to

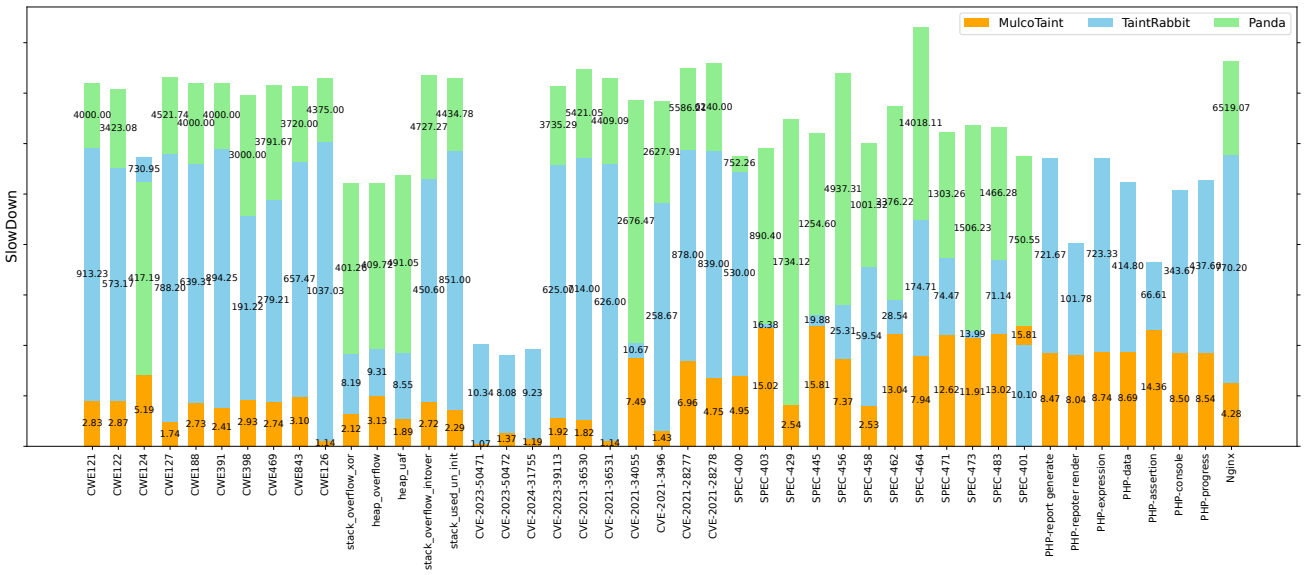

Figure 9: Comparison of relative performance overhead.

theirs, including the overhead of SPEC-401.

7.2.3 Results to Hardware Solutions

As HardTaint, FineDIFT, and RAFT only support single-tag taint tracking, they must run multiple times to get equivalent results to multi-tag analysis. Specifically, as MULCOTAINT's tag is 1024-bit width, they need to run up to 128 times to achieve what MULCOTAINT does in one run. This greatly increases their overhead compared to MULCOTAINT.

Compared with HardTaint, we select most of the test programs used in the paper, such as SPEC 2006, PHP, and Nginx. We run MULCOTAINT for the same targets and compare to the data in HardTaint's paper. HardTaint consists of three steps. The first step is time-consuming static analysis. The second part is the runtime overhead in the Column of HardTaint (TO) of Table 2. The third part is the taint calculation process, shown in the Column of HardTaint (LC). Thus, to achieve the effect of a 128-bit tag, HardTaint needs to perform 128 taint calculations, and the complete performance overhead LC_Complete is estimated using (LC + 1) * 128. The results show that HardTaint's average relative performance overhead is 19.57 times that of ours, which is average value of the Cmp of the REL (HT) column. If runtime overhead is taken into account, HardTaint will incur a greater overhead.

Since FineDIFT is not open source and has no detailed data, we rely on the average overhead of 5.03% reported in its original paper for our comparison. Estimated in the same way as HardTaint, FineDIFT's equivalent 128-bit taint propagation performance overhead is 134.44 times. Using the relative overhead multiples in Figure 9, MULCOTAINT's average overhead is 5.8x. Therefore, FineDift's overhead is 23.17 times that of ours.

Table 2: Details of performance overhead to hardware.

		MulCoTaint	HardTaint (TO)	HardTaint (LC)	HardTaint (LC_Complete)	Cmp of REL (HT)
SPEC_CPU	400	4.95	8.00%	1.73%	130.21	26.31
	401	15.81	5.91%	1.83%	130.34	8.24
	403	15.02	12.04%	1.01%	129.29	8.61
	429	2.54	7.09%	0.90%	129.15	50.85
	445	15.81	20.56%	0.83%	129.06	8.16
	456	7.37	2.40%	0.91%	129.16	17.53
	458	2.53	19.90%	0.60%	128.77	50.9
	462	13.04	2.19%	2.94%	131.76	10.1
	464	7.94	16.62%	0.80%	129.02	16.25
	471	12.62	12.09%	1.21%	129.55	10.27
	473	11.91	11.09%	0.62%	128.79	10.81
	483	13.02	9.56%	1.42%	129.82	9.97
PHP	PHPBench	9.33	11.54%	3.98%	133.09	14.26
Nginx	CURL	4.28	9.27%	6.35%	136.13	31.81
Average		9.73	10.59%	1.80%	130.3	19.57

RAFT was also tested on the SPEC 2006 benchmark suite, with an average runtime overhead of 0.13%. Similarly, the estimated overhead is 128.1664, because it is a single-tag solution. RAFT's overhead is 12.55 times that of MULCOTAINT, when the average overhead of MULCOTAINT is 10.21, calculated using the data in Figure 9.

7.3 RQ2: Vulnerability Analysis

7.3.1 Experiment Setup

We evaluated the vulnerability analysis capability using a subset of the performance evaluation dataset, including the Juliet test suite, CTF challenges, and real-world programs with reported CVEs, by selecting the targets including known vulnerabilities. The ground truth was constructed based on official vulnerability reports and manual verification.

To test whether MULCOTAINT can detect the vulnerabilities, we set the input or memory address of the target program

as taint source by the interfaces of MULCOTAINT. Then, following the methodology described in prior work [18], we marked memory addresses that were read from files or influenced by external inputs as taint sources, and identified taint sinks based on exception-related locations, such as the return address (RA) registers of crashed programs, or jump targets overwritten due to stack and heap overflows. Besides, to demonstrate the capability of multi-tag taint analysis, we backtrack taint sources by retrieving corresponding taint tags through the taint page table and taint registers. To validate correctness, we modified specific input bytes based on the analysis results and successfully observed corresponding changes at the sink, thereby confirming accurate multi-tag taint calculation. Here, MULCOTAINT uses the same taint page table and filter unit configurations as in the performance evaluation.

7.3.2 Results

The results are shown in Figure 10 that MULCOTAINT can detect all the vulnerabilities. We also clarify the source points and the sink points for each case. In our experiments, there are three types of source points, i.e., variable, buffer, and address on stack or heap. And there are four types of sink points, i.e., pointer, function pointer, variable on stack or heap, and register. We also clarify if the sink point is a vulnerable point, which means the root cause location of vulnerability, or is a crash point where the crash happens.

In particular, we present the results of multi-tag analysis in the last column of Figure 10. As we introduced in Section 3.1, each double-word data is assigned 1024-bit tags, i.e., each byte is assigned with 128-bit tags. Thus, the results are two 64-bit values, representing the high and low 64-bit tags, respectively. We present the results with a size value if the bytes are contiguous. For a better understanding, we further transfer the original results to input offsets, which are shown with square brackets in the Column of Results.

We conduct a technical comparison of RAFT and FineDIFT based on their paper, and put the results in Columns RAFT and FineDIFT of Figure 10. According to the paper of RAFT, it has limited capabilities to prevent attacks exploiting vulnerabilities on the heap (e.g., heap overflow) due to a lack of information about the allocation size. Therefore, the test items marked with red crosses all involve heap operations. FineDIFT has issues when library functions allocate temporary buffers on the heap. When the buffer is deallocated, the function has no way of identifying and removing metadata from the coprocessor. Meanwhile, the metadata of FineDIFT is stored in a dedicated hardware structure. Due to limitations in FPGA resources, the number of lines in the storage structure is limited, making it difficult for FineDIFT to support large programs with extensive heap operations. Therefore, we use a half-check to mark these programs.

Table 3: Hardware Resource.

	RAFT	FineDIFT	MULCOTAINT
Tag Type	Single-tag	Single-tag	Multi-tag
Luts	78,355	46,529	240,806
FF	56,879	10,846	105,361
Power	3.54W	-	4.395 W

7.3.3 Case Studies

Taking the results of CTF-stack_overflow_xor on the eleventh row in Figure 10 as an example, we mark the heap buffer variable that reads external input as the source point. Since the vulnerability is a stack overflow that overwrites the return address, we mark the RA register as the sink point. The RA register will be tainted if external input overwrites the return address, potentially hijacking the control flow. After our analysis with MULCOTAINT, we get the results of 0x0, 0x2000000 (size:8). Thus, the first byte that affects the return address is at the input offset of 26 (i.e., $0x0 * 2^{64} + 0x2000000$), and the last byte is 33 (i.e., $26 + 8 - 1$).

Juliet CWE-188 represents an integer boundary vulnerability that relies on specific memory layout assumptions. This program defines a structure containing two adjacent variables: a char variable *charFirst* and an int variable *intSecond*. The weakness is caused by memory boundary alignment, which is automatically performed by some compilers, i.e., three bytes are left free after *charFirst* before placing *intSecond*. Assigning a value to $ADDR\_charFirst + sizeof(int)$ memory is equivalent to assigning a value to intSecond. We label four consecutive bytes as source from $ADDR\_charFirst + sizeof(int)$. Finally, we get the label of the int type variable (size:4), confirming the weakness.

The CVE-2021-34055 case shows that the results are not always in contiguous memory bytes. The type of this CVE is UAF, so we marked the buffer (the pointer itself) on the heap as the source point. We set the sink point to check on that pointer. As shown in Figure 10, the pointer's label is overwritten by two 128-bit. Because the program calls the PUT16U function on the null pointer, it overwrites the two bytes of data. Therefore, the results are not contiguous bytes, which indicates a UAF vulnerability, and gives the specific address to be used. It better demonstrates the meaning of multi-tag taint analysis and the ability of MULCOTAINT.

7.4 RQ3: Resource Cost

7.4.1 Hardware Cost

In the experiments, MULCOTAINT was configured to run at 50MHz, with no timing violations reported. We configured the queue with 9,000 entries, deployed 15 monitoring units, and allocated space for 30 microcodes per rule. We record the data displayed by the engineering software during generation, including Luts, FFs, and Power. Specifically, Lookup Tables (LUTs) are the fundamental logic units in FPGAs that

Package	Vulnerability Type	Source Point	Sink Point	Result	MulcoTaint	RAFT	FineDIFT
Juliet-CWE121	Stack overflow due to overrun	variable on stack	function pointer on stack (Vulnerable Point)	[25,32] 0x0,0x1000000 (size: 8)	✓	✓	✓
Juliet-CWE122	Heap overflow due to overrun	variable on heap	function pointer (Vulnerable Point)	[17,24] 0x0,0x10000 (size: 8)	✓	✗	✓
Juliet-CWE124	Buffer underwrite on heap	variable on stack	variable on heap (Vulnerable Point)	[1,8] 0x0,0x1 (size: 8)	✓	✗	✓
Juliet-CWE127	Buffer underread on stack	variable on heap	variable on stack (Vulnerable Point)	[5,8] 0x0,0x10 (size: 4)	✓	✓	✓
Juliet-CWE188	Int-boundary and reliance on data memory layout	Address on stack	variable on stack (Vulnerable Point)	[1,8] 0x0,0x1 (size: 8)	✓	✓	✓
Juliet-CWE391	Unchecked error	variable on stack	variable on stack (Vulnerable Point)	[1,8] 0x0,0x1 (size: 8)	✓	✓	✓
Juliet-CWE398	Poor code quality and invalid addition operation	variable on stack	variable on stack (Vulnerable Point)	[1,8] 0x0,0x1 (size: 8)	✓	✓	✓
Juliet-CWE469	Subtracting pointer from a different string	variable on stack	variable on stack (Vulnerable Point)	[1,8] 0x0,0x1 (size: 8)	✓	✓	✓
Juliet-CWE843	Type confusion	variable on stack(char)	variable on stack(int) (Vulnerable Point)	[1,8] 0x0,0x1 (size: 8)	✓	✓	✓
Juliet-CWE126	Heap overflow due to overread	variable on heap	variable on heap (Vulnerable Point)	[121,128] 0x100000000000000,0x0 (size: 8)	✓	✗	✓
CTF-stack_overflow	Stack overflow overwrites return address	buffer on heap	Register (Crash Point)	[26,33] 0x0,0x2000000 (size: 8)	✓	✓	✓
CTF- heap_overflow	Heap overflow overwrites fuction pointer	buffer on stack	function pointer on heap (Crash Point)	[17,24] 0x10000,0x0 (size: 8)	✓	✗	✓
CTF-heap_uaf	Use after free	buffer on stack	function pointer on heap (Crash Point)	[1,8] 0x0,0x1 (size: 8)	✓	✗	✓
CTF-stack_overflow_intover	Integer overflow and out-of-bounds access	buffer on stack	function pointer on stack (Crash Point)	[1,8] 0x0,0x1 (size: 8)	✓	✓	✓
CTF-stack_used_un_init	Variables on stack not initialized	buffer on stack	variable on stack (Vulnerable Point)	[1,4] 0x0,0x1 (size: 4)	✓	✓	✓
CVE-2023-50471	Segmentation violation	variable on heap	pointer on heap (Crash Point)	[1,8] 0x0,0x1(size: 8)	✓	✗	✓
CVE-2023-50472	Segmentation violation	variable on heap	pointer on heap (Crash Point)	[1,8] 0x0,0x1(size: 8)	✓	✗	✓
CVE-2024-31755	Segmentation violation	variable on heap	pointer on heap (Crash Point)	[1,8] 0x0,0x1(size: 8)	✓	✗	✓
CVE-2023-39113	Segmentation violation	variable on heap	pointer on heap (Crash Point)	[17,24] 0x0,0x10000(size: 8)	✓	✗	✓
CVE-2021-36530	Heap overflow	variable on heap	pointer on stack (Vulnerable Point)	[41,48] 0x0,0x20000000 (size: 8)	✓	✗	✓
CVE-2021-36531	Heap overflow	variable on heap	variable on heap (Vulnerable Point)	[1,4] 0x0,0x1 (size: 4)	✓	✗	✓
CVE-2021-34055	Use after free	buffer on heap	variable on heap (Vulnerable Point)	[9,12-16] 0x0,0x100;0x0,0x800 (size: 6)	✓	✗	✓
CVE-2021-3496	Heap Out-of-Bounds Read Vulnerability	buffer on heap	variable on heap (Vulnerable Point)	[1,8] 0x0,0x1(size: 8)	✓	✗	✓
CVE-2021-28277	Heap Out-of-Bounds Read Vulnerability	buffer on heap	variable on heap (Vulnerable Point)	[9,16] 0x0,0x100(size: 8)	✓	✗	✓
CVE-2021-28278	Heap Out-of-Bounds Read Vulnerability	buffer on heap	variable on heap (Vulnerable Point)	[1,8] 0x0,0x1(size: 8)	✓	✗	✓

Figure 10: Details of vulnerability detection.

implement combinational logic functions. Flip-Flops (FF) are sequential logic elements that store state information. Power represents the total electrical energy consumed by our design.

We compare the hardware resources of MULCOTAINT to RAFT and FineDift with the data in their papers. The results are shown in Table 3. MULCOTAINT takes up more hardware resources. MULCOTAINT is 3.0 times larger than RAFT and 5.17 times larger than FineDIFT in terms of the occupancy of Luts, which is acceptable in practice. Also, in terms of FFs, MULCOTAINT is 1.8 times larger than RAFT and 9.7 times larger than FineDIFT. The higher hardware cost of our system is attributed to its support for efficient multi-tag taint analysis, in contrast to prior work that primarily targets single-tag or limited-tag tracking for violation detection.

7.4.2 Memory Cost

Section A.2 shows the memory usage of MULCOTAINT during the experiment, including: taint page tables and tags storage. The memory overhead is acceptable. We also provide data in Table 7 on the density of unused entries under the multi-level taint page table to demonstrate the effectiveness of our tag management mechanism.

8 Limitation and Future Work

8.1 Hardware Requirement

Large programs require a lot of taint space, especially those containing many recursive calls. Therefore, the size of programs that can be analyzed by MULCOTAINT is limited by the actual size of physical memory in the environment. If the memory cannot carry a complete tainted page table, our system can only temporarily swap part of the taint data into the disk space through the operating system's SWAP mechanism. However, frequent swapping in and out will significantly reduce the system's efficiency in taint analysis.

It is noticed that our design of MULCOTAINT does not depend on specific architectures. However, due to the need to add hardware processing units, we designed and implemented a prototype system based on RISC-V in this paper. If we can modify CPU chips in the future, we will obtain a more practical MULCOTAINT.

Although MULCOTAINT is based on a sequential execution CPU implementation, for Out-of-Order CPUs, our scheme can still be implemented with some modifications. The coprocessor needs to receive committed instructions from the processor. In Out-of-Order architecture, the instruction will enter

the rearrangement phase after execution before the instruction is committed, so we can get the instruction information and data access information in this phase.

8.2 Taint Rule Design

We have implemented the taint rule of the instructions in RV64I, and our experiments present that it is sufficient for the analysis of real-world programs. As mentioned in Section 6.2, we keep the rule configurable. On the one hand, our design anticipates future support for extended instruction sets. For instance, although certain floating-point operations can be emulated in RV64I, native floating-point execution requires dedicated instruction set extensions. Currently, MULCOTAINT does not support floating-point instructions. To add instruction support in the future, we can add configuration in Monitoring Unit and directly reuse existing microcode rules if the new instructions' dataflow relationships are similar to those in RV64I. We can also extend the ALU to express new dataflow relationships. On the other hand, the extensible design also enables support for more complex scenarios, such as implicit taint propagation in control flow [10]. Currently, we do not support implicit taint propagation as in tools like PANDA [16], but we plan to incorporate static analysis, similar to DepTaint [33], to account for control-flow dependencies.

8.3 Multi-thread Program Support

Our scheme is based on a single-core RISC-V SoC implementation, which currently does not support the parallel analysis of multi-threaded programs. An alternative is to disable the multi-core support of the operating system, allowing multi-threaded programs to switch to sequential execution and compute taint data instruction by instruction. To truly support multi-thread programs, we consider the establishment of multiple coprocessors, which requires communication between multiple CPUs and multiple coprocessors, as well as coordination and data transfer between coprocessors.

9 RELATED WORK

9.1 Software Schemes

MULCOTAINT is an online, multi-tag explicit taint analysis scheme. In fine-grained analysis scenarios (e.g., vulnerability fix and exploitation), the support for fine-grained multi-tag dataflow analysis allows MULCOTAINT to offer higher performance and stronger analysis capabilities than existing solutions. However, in coarse-grained scenarios (e.g., violation detection or cases where only the presence of taint matters), our performance does not surpass single-tag schemes such as RAFT [57], FineDIFT [6], and AirTaint [45]. MULCOTAINT's hardware resource usage is also higher than that of embedded-oriented designs like RAFT [57] and FineDIFT [6]. In addition, because MULCOTAINT performs explicit taint analysis, it currently does not support control-flow–dependent implicit taint propagation.

There are many software schemes for dynamic taint analysis (DTA) [12, 14, 16, 18, 23, 29, 37], and researchers propose several attempts to improve the efficiency [8, 45]. For example, SelectiveTaint [8] removes taint-independent CPU instructions through pre-analysis to reduce the time overhead of the analysis process. AirTaint [45] further attributes the taint propagation behavior at the level of basic blocks of the program to reduce the redundant computation that occurs in the instruction-by-instruction analysis.

However, the above approaches are limited by the analytical capability of single-tag designs, which is critical in fine-grained analysis scenarios, such as vulnerability analysis. The tag size directly influences the granularity of taint source tracking—larger tags enable more precise identification of the input origin. In the context of attack pattern detection and defense, tag size also constrains the number of enforceable security policies. But multi-tag schemes come at a higher cost of increased storage requirements and performance overhead.

To conduct multi-tag dynamic taint analysis (M-DTA), Dytan [10] implements a generic engine for taint propagation that supports multi-tag, but the performance loss is particularly severe. In order to improve the efficiency of taint analysis, TaintRabbit [18] implements basic block instrumentation based on DynamoRIO [5], which monitors and generates fast paths during program execution. This reduces unnecessary instruction processing and accelerates DTA. However, for software-based solutions, the overhead is almost inevitable.

9.2 Hardware Schemes

Researchers explore hardware-based implementations to improve the efficiency of DTA. One class of hardware solutions leverages existing hardware features [25, 32, 53, 54, 59]. For example, HardTaint [59] combines static analysis, instrumentation, and runtime analysis together to perform offline single-tag taint analysis based on Intel PT. However, its static analysis and taint calculation phase remains time-consuming. GPU-based solutions are promising. FlowMatrix [25] uses GPU to conduct the taint calculation, which can be effectively used for customized program fragments. GPUs are well-suited for offline taint analysis, because they excel at matrix computations, instruction propagation rules can be represented in matrix form, but the matrices need to be pre-generated.For ARM, several tools [32, 53, 54] use CDI (Core Debug Interface) to extract information about ARM CPU execution. Lee et al. [32] use ARM CoreSight Event Tracking Macrocells (ETMs) to track executed instructions. To improve the performance, Wahab et al. [53, 54] used CoreSight Program Tracking Macro Units (PTM) as a tracking component and only tracked the instructions related to the control flow.

The second class of hardware solutions relies on specialized

hardware. For example, RAFT [57] and FineDIFT [6] implement dedicated co-processors to complete the taint analysis process. FineDIFT designs CAM-like structures to reduce the number of tags stored and improve performance. RAFT reduces the need for Tag storage by applying coarse granularity to the heap, providing fine-grained protection of the stack and global variables. Both approaches implement only coarse-grained dynamic data flow tracking and cannot support multiple taint sources. Meanwhile, they have a very limited range of markers. FlexiTaint [51] and Raksha [13] modify the CPU to include the process of data flow tracing along with instruction execution. Raksha duplicates every stage of the instruction pipeline, and FlexiTaint adds two stages in the pipeline to complete the taint calculation. In Raksha, all storage elements, including caches, registers, and memory are extended with tags. In FlexiTaint, the taints for memory locations are stored as a packed array in regular memory. But large modifications to the CPU is costly in reality.

10 CONCLUSION

In this paper, we propose a vectorized redesign of taint calculation logic and design MULCOTAINT, a hardware/software co-designed multi-tag taint analysis framework. We decouple taint analysis from normal program execution based on a coprocessor and address key challenges in taint calculation, tag management, and interface design. Using a diverse dataset containing 32 program binaries of five types, including 18 real-world vulnerable programs, we demonstrate that MULCOTAINT achieves both efficient taint analysis and effective vulnerability detection. Compared to the state-of-the-art multi-tag software-based approach TaintRabbit and PANDA, and single-tag hardware solutions HardTaint, Raft, and FineDIFT, MULCOTAINT delivers significantly better performance and stronger analysis capabilities.

11 Acknowledgments

We thank the anonymous reviewers for their insightful comments and feedback. This research was supported, in part, by National Natural Science Foundation of China (Grant No.62232016, 62472414, 62372437).

Ethical Considerations

Stakeholders

The research focuses on accelerating multi-label taint analysis, and improving analysis capabilities for known vulnerabilities. The stakeholders potentially affected include:

1. **End users** — Users of software containing known vulnerabilities. Although these vulnerabilities have been disclosed, some users may not have applied patches.
2. **Software developers / vendors** — Their products contain the known vulnerabilities used as analysis subjects.
3. **The security research community** — Researchers can build upon our framework to improve taint analysis tools and better understand vulnerabilities.
4. **The general public** — Society at large may benefit from improved software security and reduced attack costs in the long term.
5. **The research team** — The authors, who are responsible for ensuring the research process adheres to ethical and academic standards.

Ethical Principles Considered

We considered the four principles from *The Menlo Report*:

- **Beneficence** — The purpose of this study is to accelerate taint analysis methods to better understand vulnerability causes, thereby improving security tools and defenses. The study does not introduce new risks.
- **Respect for Persons** — This study does not involve human subjects or the collection of user data; therefore, no privacy violations or issues of informed consent arise.
- **Justice** — We analyze only *publicly disclosed vulnerability cases*, without disproportionately targeting any particular vendor or group.
- **Respect for Law and Public Interest** — We comply with applicable laws and ethical research norms. All experiments were conducted in controlled environments, without affecting real systems or users.

Potential Harms and Mitigations

- **Potential harm:** This work is intended for defensive analysis. Our method clarifies the root causes of vulnerabilities, which is useful for fixing, but it may also be leveraged for exploitation.
  Mitigation: We do not release exploit code or attack-specific details. The released artifacts are limited to analysis results and performance evaluations conducted on a controlled development platform, and all of them pertain to known vulnerabilities. We consider the primary contribution of this work to be defensive, such as fixing and detection. We believe that the benefits outweigh the limited risk of misuse.
- **Privacy risks:** Since the study focuses only on public vulnerability cases and controlled programs, no real user data was involved, and there are no privacy concerns.

- **Reputational risks:** Vendors or developers whose products contained vulnerabilities might be concerned about reputational impacts from being mentioned again.
  Mitigation: We selected only patched vulnerabilities as case studies and avoided unnecessarily emphasizing vendor or developer responsibility. Our work can help them locate issues and assist in vulnerability remediation.

Decision to Proceed

We decided to proceed with this research for the following reasons:

1. **Low risk** — The research is limited to known vulnerabilities; no new attack vectors are introduced.

2. **High potential benefit** — The research improves understanding of vulnerability mechanisms, and supports the development of future defensive technologies.

3. **Effective mitigations** — Experiments are conducted in controlled environments, without exposing real users or systems to harm. Exploit code is omitted, and only propagation paths and tag analysis results are shared.

Considering both **beneficence-based reasoning** (weighing risks and benefits) and **deontological reasoning** (avoiding rights violations), we conclude that conducting this research is ethically justified.

Decision to Publish

We decided to publish the research for the following reasons:

1. **Educational and community value** — Sharing the framework to improve taint analysis tools and better understand known vulnerabilities.

2. **Minimal additional risk** — By omitting exploit details, focusing on publicly patched vulnerabilities, and avoiding sensitive data, potential harms from publication are minimized.

3. **Societal benefit** — Disseminating this research can help software developers quickly locate issues and assist in vulnerability remediation.

Based on both **beneficence-based reasoning** and **deontological reasoning**, we conclude that publishing this research is ethically sound, with appropriate mitigations in place. While we have minimized potential harms, we acknowledge that some residual risk of misinterpretation or misuse remains.

Open Science

This appendix lists all artifacts necessary to evaluate the contributions of this paper. Since our work is part of a commercial product, we cannot fully open-source it. We provide the complete development board image and the full test suite, which are sufficient for evaluating this work. The artifacts are available at: https://doi.org/10.5281/zenodo.17939551

- **Fpga Image:** All images and files are in Image_for_evaluation directory (test program is located in rootfs.ext2. For usage instructions, please refer to the readme file in the MulcoTaint_test/experiment_for_Mulco directory.)

- **Experimental Datasets:** Source code and binary files for the evaluation experiments have been organized in MulcoTaint_test directory. Both the original version of the test program and the version integrated with our analysis system are available in experiment_for_Mulco directory. SPEC CPU is not permitted to be open-sourced, and ngiflib does not allow unauthorized distribution; therefore, we only provide patches. The patches for the additional SPEC 2017 experiments included in the appendix is provided in additional test directory.

References

[1] Mark W Aldrich, Alexi Turcotte, Matthew Blanco, and Frank Tip. Augur: dynamic taint analysis for asynchronous javascript. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–4, 2022.

[2] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 4:6–2, 2016.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. California, USA, 2005.

[5] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 133–144, 2012.

[6] Kejun Chen, Orlando Arias, Qingxu Deng, Daniela Oliveira, Xiaolong Guo, and Yier Jin. Finedift: Fine-grained dynamic information flow tracking for data-flow integrity using coprocessor. *IEEE Transactions on Information Forensics and Security*, 17:559–573, 2022.

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

[8] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. SelectiveTaint: Efficient data flow tracking with static binary rewriting. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1665–1682. USENIX Association, August 2021.

[9] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *11th IEEE Symposium on Computers and Communications (ISCC'06)*, pages 749–754. IEEE, 2006.

[10] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, 2007.

[11] James Clause and Alessandro Orso. Penumbra: automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, page 249–260, New York, NY, USA, 2009. Association for Computing Machinery.

[12] Lei Cui, Youquan Xian, Peng Liu, and Longjin Lu. Taintemu: Decoupling tracking from functional domains for architecture-agnostic and efficient whole-system taint tracking. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1235–1250, 2025.

[13] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. *ACM SIGARCH Computer Architecture News*, 35(2):482–493, 2007.

[14] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. {DECAF++}: Elastic {Whole-System} dynamic taint analysis. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 31–45, 2019.

[15] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. {PHMon}: A programmable hardware monitor and its security use cases. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 807–824, 2020.

[16] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 839–850, 2013.

[17] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[18] John Galea and Daniel Kroening. The taint rabbit: Optimizing generic taint analysis with dynamic fast path generation. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 622–636, 2020.

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

[20] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484. IEEE, 2009.

[21] SPEC Groups. The standard performance evaluation corporation (spec) cpu 2006 benchmark (spec2006). https://www.spec.org/spec/, 2006.

[22] Liang He, Hong Hu, Purui Su, Yan Cai, and Zhenkai Liang. {FreeWill}: Automatically diagnosing use-after-free bugs via reference miscounting detection on binaries. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2497–2512, 2022.

[23] Andrew Henderson, Lok Kwong Yan, Xunchao Hu, Aravind Prakash, Heng Yin, and Stephen McCamant. Decaf: A platform-neutral whole-system dynamic binary analysis platform. *IEEE Transactions on Software Engineering*, 43(2):164–184, 2016.

[24] RISC-V International. The risc-v instruction set manual volume i: Unprivileged isa. https://riscv.org/specifications/ratified/, May 2024. Online; accessed Apr 15, 2024.

[25] Kaihang Ji, Jun Zeng, Yuancheng Jiang, Zhenkai Liang, Zheng Leong Chua, Prateek Saxena, and Abhik Roychoudhury. {FlowMatrix}:{GPU-Assisted}{Information-Flow} analysis through {Matrix-Based} representation. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2567–2584, 2022.

[26] Xiangkun Jia, Chao Zhang, Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng. Towards efficient heap overflow discovery. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 989–1006, 2017.

[27] Xiao Kan, Cong Sun, Shen Liu, Yongzhe Huang, Gang Tan, Siqi Ma, and Yumei Zhang. Sdft: A pdg-based summarization for efficient dynamic data flow tracking. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 702–713. IEEE, 2021.

[28] Hari Kannan, Michael Dalton, and Christos Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 105–114. IEEE, 2009.

[29] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 121–132, 2012.

[30] David Korczynski and Heng Yin. Capturing malware propagations with code injections and code-reuse attacks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1691–1708, 2017.

[31] Lap Chung Lam and Tzi-cker Chiueh. A general dynamic information flow tracking framework for security applications. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 463–472. IEEE, 2006.

[32] Jinyong Lee, Ingoo Heo, Yongje Lee, and Yunheung Paek. Efficient security monitoring with the core debug interface in an embedded processor. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(1):1–29, 2016.

[33] Binbin Li, Rui Ma, Xuefei Wang, Xiajing Wang, and Jinyuan He. Deptaint: A static taint analysis method based on program dependence. In *Proceedings of the 2020 4th International Conference on Management Engineering, Software Engineering and Service Sciences*, ICMSS 2020, page 34–41, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. Pata: Fuzzing with path aware taint analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2022.

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

[36] Changhua Luo, Penghui Li, and Wei Meng. Tchecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in php applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2175–2188, 2022.

[37] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. {TaintPipe}: Pipelined symbolic taint analysis. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 65–80, 2015.

[38] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[39] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[40] Edmund B Nightingale, Daniel Peek, Peter M Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. *ACM SIGARCH Computer Architecture News*, 36(1):308–318, 2008.

[41] Yicheng Ouyang, Kailai Shao, Kunqiu Chen, Ruobing Shen, Chao Chen, Mingze Xu, Yuqun Zhang, and Lingming Zhang. Mirrortaint: Practical non-intrusive dynamic taint tracking for jvm-based microservice systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2514–2526. IEEE, 2023.

[42] Christian Palmiero, Giuseppe Di Guglielmo, Luciano Lavagno, and Luca P Carloni. Design and implementation of a dynamic information flow tracking architecture to secure a risc-v core for iot applications. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[43] PHPBench. Phpbench. https://github.com/phpbench/phpbench, 2021. Online; accessed Aug 01, 2025.

[44] Alexander Richardson. Juliet test suite for c/c++. https://github.com/arichardson/juliet-test-suite-c, Apr 2019. Online; accessed Nov 29, 2024.

[45] Qian Sang, Yanhao Wang, Yuwei Liu, Xiangkun Jia, Tiffany Bao, and Purui Su. Airtaint: Making dynamic taint analysis faster and easier. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3998–4014. IEEE, 2024.

[46] Florent Saudel and Jonathan Salwan. Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes*, pages 31–54, 2015.

[47] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.

[48] Ali Shuja Siddiqui, Geraldine Shirley, Shreya Bendre, Girija Bhagwat, Jim Plusquellic, and Fareena Saqib. Secure design flow of fpga based risc-v implementation. In *2019 IEEE 4th International Verification and Security Workshop (IVSW)*, pages 37–42. IEEE, 2019.

[49] Standard Performance Evaluation Corporation. Spec cpu2017 benchmark suite, 2017.

[50] Zhiyou Tian, Cong Sun, Dongrui Zeng, and Gang Tan. Podft: On accelerating dynamic taint analysis with precise path optimization. In *Proc. Binary Anal. Res. Workshop*, 2023.

[51] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 173–184. IEEE, 2008.

[52] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, volume 2007, page 12, 2007.

[53] Muhammad A Wahab, Pascal Cotret, Mounir N Allah, Guillaume Hiet, Vianney Lapotre, and Guy Gogniat. Armhex: A hardware extension for dift on arm-based socs. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7. IEEE, 2017.

[54] Muhammad Abdul Wahab, Pascal Cotret, Mounir Nasr Allah, Guillaume Hiet, Arnab Kumar Biswas, Vianney Lapotre, and Guy Gogniat. A small and adaptive co-processor for information flow tracking in arm socs. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2018.

[55] Minghua Wang, Purui Su, Qi Li, Lingyun Ying, Yi Yang, and Dengguo Feng. Automatic polymorphic exploit generation for software vulnerabilities. In *International Conference on Security and Privacy in Communication Systems*, pages 216–233. Springer, 2013.

[56] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.

[57] Yu Wang, Jinting Wu, Haodong Zheng, Zhenyu Ning, Boyuan He, and Fengwei Zhang. Raft: Hardware-assisted dynamic information flow tracking for runtime protection on risc-v. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. Association for Computing Machinery, 2023.

[58] Wei Xu, Sandeep Bhatkar, and Ramachandran Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, pages 121–136, 2006.

[59] Yiyu Zhang, Tianyi Liu, Yueyang Wang, Yun Qi, Kai Ji, Jian Tang, Xiaoliang Wang, Xuandong Li, and Zhiqiang Zuo. Hardtaint: Production-run dynamic taint analysis via selective hardware tracing. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):1615–1640, 2024.

A Appendix

A.1 Supported Commands

A RISC-V ISA is defined as a base integer ISA, which must be present in any implementation, plus optional extensions to the base ISA. Base integer ISA (RV64I) can be extended with one or more optional instruction-set extensions for Special needs. Our prototype system analyzes all 42 instructions of RV64I, and all test programs and filesystems are compiled using a compiler that supports RV64I.

Table 4: Instructions of 64-bit RISC-V architecture.

LB	LH	LW	LBU	LHU	SB
SH	SW	ADDI	SLTI	SLTIU	XORI
ORI	ANDI	SLLI	SRLI	SRAI	ADD
SUB	SLL	SLT	SLTU	XOR	SRL
SRA	OR	AND	LWU	LD	SD
SLLI	SRLI	SRAI	ADDIW	SLLIW	SRLIW
SRAIW	ADDW	SUBW	SLLW	SRLW	SRAW

A.2 Memory usage of taint space

Table 7 shows the memory usage of MULCOTAINT during the experiment, including: taint page tables and tags. For JULIET, CTF, and CVE, we allocate more space than necessary, so their values are consistent. For other test items, we allocate space more precisely. For the density of page tables, the unused average density of the 2-level page table is 44.07%, and that of the 3-level page table is 5.2%.

Table 5: Microcodes used to represent taint calculation rules.

	MicroCode	Type	In1	In2	Out	More Details
Custom	FN_ALU_TAINT_REG	tag calculation	TAINT_VEC	TAINT_VEC	TAINT_VEC	TAINT_VEC denotes that get the register number of the operand, then use the got register
			Taint_Reg0/1	Taint_Reg0/1	Taint_Reg0/1	Taint_Reg0/1 denotes a fixed taint register
			TAINT_VEC	Taint_Reg0/1	TAINT_VEC	
			TAINT_VEC	Taint_Reg0/1	TAINT_VEC1	
	FN_ALU_TAINT_IMM	tag calculation	TAINT_VEC	IMM	TAINT_VEC	IMM denotes the immediate value in the instruction
			TAINT_VEC	IMM	TAINT_VEC1	
	FN_ALU_TAINT_SHIFT	tag calculation	TAINT_VEC	TAINT_VEC	TAINT_VEC	
			TAINT_VEC	TAINT_VEC	TAINT_VEC1	TAINT_VEC1 denotes the operand is calculated as both source and destination
	FN_PART_TAG_LOAD	tag fetch and save	Taint_Reg0/1	PART_TAG	Taint_Reg0/1	PART_TAG denotes that get partial tag in tag vector
	FN_PART_TAG_STORE	tag fetch and save	TAINT_VEC	PART_TAG	Taint_Reg0/1	The first input from taint register in instruction
			Taint_Reg0/1	PART_TAG	Taint_Reg0/1	The first input from a fixed taint register
	FN_MASK_TAG	tag fetch and save	Taint_Reg0/1	PART_TAG	Taint_Reg0/1	PART_TAG denotes that clear partial tag in tag vector
	FN_WRITE_TAG	tag fetch and save	TAINT_VEC	Local_Reg0-5(S/L)	-	Local_Reg0-5(S/L) denotes the first address of the tag in memory
			Taint_Reg0/1	Local_Reg0-5(S/L)	-	
	FN_READ_TAG	tag fetch and save	-	Local_Reg0-5(S/L)	Resp_TAG	Resp_TAG indicates where the read tag is stored
	FN_WRITE_DATA	data fetch and save	Local_Reg0-5(D)	Local_Reg0-5(S/L)	-	Local_Reg0-5(D) denotes the first address of the data in memory
	FN_READ_DATA	data fetch and save	-	Local_Reg0-5(S/L)	Resp	Resp indicates where the read data is stored
	FN_FILTER	memory filter	Local_Reg0-5(S/L)	-	DONE	Local_Reg0-5(S/L) denotes filtered memory range
General	FN_ADD	data calculation	Local_Reg0-5(D)	Local_Reg0-5(D)	Local_Reg0-5(D)	Addition operations
	FN_SUB	data calculation	Local_Reg0-5(D)	Local_Reg0-5(D)	Local_Reg0-5(D)	Subtraction operations
	FN_SL	data calculation	Local_Reg0-5(D)	Local_Reg0-5(D)	Local_Reg0-5(D)	Left(S)hift operations
	FN_SR	data calculation	Local_Reg0-5(D)	Local_Reg0-5(D)	Local_Reg0-5(D)	Right(S)hift operations
	FN_SLT	data calculation	Local_Reg0-5(D)	Local_Reg0-5(D)	Local_Reg0-5(D)	Comparison judgment
	FN_SEQ	data calculation	Local_Reg0-5(D)	Local_Reg0-5(D)	Local_Reg0-5(D)	Equalization judgment
	FN_AND	data calculation	Local_Reg0-5(D)	Local_Reg0-5(D)	Local_Reg0-5(D)	AND operations
	FN_OR	data calculation	Local_Reg0-5(D)	Local_Reg0-5(D)	Local_Reg0-5(D)	OR operations
	FN_XOR	data calculation	Local_Reg0-5(D)	Local_Reg0-5(D)	Local_Reg0-5(D)	XOR operations

Table 6: Details of SPEC CPU 2017 performance overhead to software.

Program	Bin Size (MB)	Input Size (Byte)	Panda (us)	MulcoTaint (us)	Frequency-converted MulcoTaint(us)	Frequency-converted Cmp of ABS(PD)	Cmp of ABS(PD)	PandaQEMU (us)	Board (us)	Cmp of REL(PD)	SlowDown of MulcoTaint	SlowDown of PANDA
600 perlbench	7.4	146	8,628,000,000	826,852,367	8,268,523.67	1,043.48	10.43	7,689,000	77,519,484	105.20	10.67	1,122.12
602 gcc	15.9	69	410,000,000	8,818,134	88,181.34	4,649.51	46.50	432,000	1,454,996	156.60	6.06	949.07
605 mcf	5.2	1,163,146	X	2,951,289,932	29,512,899.32	X	X	55,468,000	575,892,559	X	5.12	X
620 omnetpp	11.5	2,219	X	4,257,557,470	42,575,574.70	X	X	49,175,000	450,930,674	X	9.44	X
623 xalan	15	28,074	2,143,000,000	228,389,667	2,283,896.67	938.31	9.38	1,842,000	15,429,993	78.60	14.80	1,163.41
625 x264	5.5	2,496,219	1,183,000,000	188,299,362	1,882,993.62	628.25	6.28	330,000	34,424,879	655.38	5.47	3,584.85
631 deepsjeng	4.4	73	948,000,000	401,651,638	4,016,516.38	236.03	2.36	3,157,000	29,918,912	22.37	13.42	300.29
641 leela	6.8	1,562	X	9,494,528,570	94,945,285.70	X	X	82,347,000	632,300,611	X	15.02	X
648 exchange2	4.9	3,001	X	50,290,820,000	502,908,200.00	X	X	654,392,000	3,210,190,000	X	15.67	X
657 xz	4.8	1,287,176	X	1,634,218,769	16,342,187.69	X	X	11,185,000	172,183,678	X	9.49	X
Average			2,662,400,000	7,028,242,591	70,282,425.91	1,499.11	14.99	86,601,700	520,024,578.60	203.63	10.52	1,423.95

Table 7: Memory usage and Unused dense of taint space.

Program	Memory usage of taint space (MB)	2-level dense	3-level dense
JULIET	27	46.49	5.88
CTF	27	46.49	5.88
CVE	27	46.49	5.88
SPEC-400	25	74.67	8.33
SPEC-403	97	14.19	1.219
SPEC-429	15	85.02	19.427
SPEC-445	62	28.96	2.5
SPEC-456	19	71.4	11.106
SPEC-458	83	45.16	2.84
SPEC-462	15	85.6	14.048
SPEC-464	74	27.97	2.173
SPEC-471	122	19.29	1.18
SPEC-473	37	20.83	0.896
SPEC-483	112	26.09	1.058
SPEC-401	61	31.73	1.999
php	107	18.562	0.96
nginx	69	60.26	3.04
Average	57.58	44.07	5.2

A.3 Microcodes

Table 5 is Microcodes used to represent taint calculation rules.

A.4 Performance

Table 6 shows performance overhead to all 10 INT test programs in SPEC CPU 2017. Among the comparison tools, TaintRabbit only supports 32-bit, whereas the SPEC 2017 INT Speed benchmark supports only 64-bit. The only other tool we can compare against is PANDA. In our tests, we found that 5 SPEC 2017 test programs crashed (tagged with X) during PANDA's analysis due to memory issues. The PANDA test environment is equipped with 32 GB of memory.

MULCOTAINT's average performance overhead over native execution is 10.52x (ranging from 5.12x to 15.67x), whereas PANDA's is 1423.95x (300.29x to 3584.85x). In absolute time, comparison of PANDA and MULCOTAINT is 2.36-46.50x (with an average of 14.99x). In relative time, comparison of PANDA and MULCOTAINT is 22.37-655.38x (with an average of 203.63x). After applying the same frequency conversion used in the evaluation section, the estimated absolute-time performance overhead comparison is 236.03–4649.51× (with an average of 1499.11×). Overall, PANDA consistently underperforms compared to MULCOTAINT.