Raft: Hardware-assisted Dynamic Information Flow Tracking for Runtime Protection on RISC-V

Yu Wang
Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China
Department of Computer Science and Engineering, Southern University of Science and Technology, China
12032879@mail.sustech.edu.cn

Jinting Wu
Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China
Department of Computer Science and Engineering, Southern University of Science and Technology, China
wujt@mail.sustech.edu.cn

Haodong Zheng
Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China
Department of Computer Science and Engineering, Southern University of Science and Technology, China
zhenghd@mail.sustech.edu.cn

Zhenyu Ning
Hunan University, China
Southern University of Science and Technology, China
zning@hnu.edu.cn

Boyuan He
Huawei Technologies Co., Ltd., China
heboyuan@huawei.com

Fengwei Zhang*
Shenzhen Key Laboratory of Safety and Security for Next Generation of Industrial Internet, Southern University of Science and Technology, China
Department of Computer Science and Engineering, Southern University of Science and Technology, China
zhangfw@sustech.edu.cn

ABSTRACT

Dynamic Information Flow Tracking (DIFT) is a fundamental computer security technique that tracks the data flow of interest at runtime, overcoming the limitations of discovering data dependencies statically at compilation time. However, software-based DIFT tools often suffer from unbearably high runtime overhead due to dynamic binary instrumentation or virtual machine, limiting the usefulness of DIFT. Even though hardware-assisted DIFT frameworks cut down the performance overhead effectively, it is still unacceptable for applications under rigorous time constraints.

This paper presents Raft, a flexible hardware-assisted DIFT framework that provides runtime protection for embedded applications without delay to the programs. Our framework is designed as a coprocessor for a RISC-V Rocket Core, introducing minimally-invasive changes to the main processor. In Raft, we apply a novel storage mechanism with hybrid byte/variable granularity to reduce the size of tag storage and provide fine-grained protection. We deploy Raft on the Rocket emulator and FPGA development board to evaluate its effectiveness and efficiency. The experiment results show that, compared to previous approaches, Raft cuts down the performance overhead from more than 20% to less than 0.1% on NBench and CoreMark microbenchmarks. The performance overhead of Raft on SPEC CINT 2006 benchmarks is negligible (0.13%). We also utilize a customized program to demonstrate its functionality and conduct a detailed evaluation with a real-world embedded medical application and known CVEs.

*The corresponding author.

RAID '23, October 16–18, 2023, Hong Kong, China
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0765-0/23/10.
https://doi.org/10.1145/3607199.3607246

CCS CONCEPTS

• **Security and privacy → Information flow control**.

KEYWORDS

Dynamic Information Flow Tracking, Hardware-software Codesign, RISC-V

ACM Reference Format:
Yu Wang, Jinting Wu, Haodong Zheng, Zhenyu Ning, Boyuan He, and Fengwei Zhang. 2023. Raft: Hardware-assisted Dynamic Information Flow Tracking for Runtime Protection on RISC-V. In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3607199.3607246

1 INTRODUCTION

Dynamic Information Flow Tracking (DIFT) [44], which tracks the data flow of interest during program execution, is a fundamental computer security technique. It has a wide range of applications in software and system security, such as exploit detection [14, 16, 25, 29, 38, 49], privacy leak detection [5, 11, 28, 61], malware analysis [23, 34], and protocol reverse engineering [8, 35]. Static Information Flow Tracking (SIFT) [36, 41], meanwhile, detects potential security issues based on an overestimation of all possible program paths through lexical analysis, syntax analysis, and semantic analysis. However, though efficient, SIFT is labor-intensive and leads to false positives due to the lack of the program's runtime behavior [60]. Moreover, some works [37, 60] have shown that SIFT also induces some false negatives owing to the complexity of programs in real-world scenarios. Unlike SIFT, DIFT monitors the data flow of the program at runtime. DIFT only tracks the execution path taken by the program and captures its runtime behavior, which results in an approach more precise than SIFT.

At the same time, the implementation of DIFT often suffers from high performance overhead. Software-based DIFT solutions build their analysis frameworks and optimizations atop Dynamic Binary

Instrumentation (DBI) (e.g., Intel PIN [39] and DynamoRIO [6]) or simulators. They dynamically disassemble the binary and instrument analysis logic, then compile and reassemble the executed code at runtime, which causes unbearably high runtime overhead. For instance, a state-of-the-art dynamic information flow analysis tool libdft [32] imposes a slowdown that ranges between 1.20x and 6.03x on commonly-used Unix utilities. There has been much research that seeks to improve the performance of DIFT. For instance, TaintPipe [42] explored a parallel and pipeline scheme on multicore platforms, achieving a performance 2.38x faster than libdft. SelectiveTaint [13] applied static binary rewriting instead of dynamic binary instrumentation to selectively instrument the instructions involving DIFT, which is 1.7x faster than that of libdft. Nonetheless, pure software implementations still incur huge performance overhead. Therefore, a research interest emerged in developing hardware accelerators to improve the performance of DIFT at the expense of flexibility.

Hardware-assisted DIFT solutions extend each register and memory location with tags. During the program execution, the hardware accelerator transparently propagates or checks tags for each executed instruction without additional instrumentation. Earlier hardware-assisted DIFT systems [17, 18] integrated analysis logic directly into the main processor's pipeline. Each stage of the pipeline is duplicated with a specific hardware module to complete tag-related operations and regular computation in parallel. Although this approach has low performance overhead, the invasive design makes it difficult to port on hardcore CPUs built with non-reconfigurable silicon [55]. Later studies [21, 31] utilized a dedicated coprocessor to perform DIFT. The main core is responsible for initializing tags and committing instructions to the coprocessor in charge of propagation and verification. Even though this approach applies minor modifications to the main core and is easier to deploy, it also incurs non-negligible performance overhead (e.g., 26% on SPEC 2006 benchmarks in SIFT [46]) for the main processor due to the communication between the main core and coprocessor. Such performance overhead seems reasonable for program analysis but is still unacceptable when protecting time-critical applications at runtime. Embedded applications in delicate domains, such as medical applications and vehicle control units that serve highly critical tasks in embedded systems, have strict time constraints. Unfortunately, such time-critical embedded applications often suffer from programming errors and exploitable vulnerabilities that are already known from classical computing [54]. The focus of this paper is to significantly improve the performance of DIFT so that it can be exploited for runtime protection on time-critical embedded applications.

To further reduce the performance overhead of DIFT, we first analyze performance factors in traditional coprocessor-based DIFT solutions. We find that major runtime overhead comes from the communication between the coprocessor and the main core. The traditional tag-storage mechanism [31, 33] uses a separate memory region (called shadow memory) for tag storage and directly maps the tags of every memory address into shadow memory. However, the coprocessor generally cannot access memory directly. Thus, it needs to send memory requests to the main core when obtaining and updating tags. The main processor is stalled when handling tag read/write operations, which is the main factor incurring performance overhead.

To address the above challenges, we present RAFT (Runtime protection using dynamic information Flow Tracking), a flexible hardware-assisted DIFT framework that provides runtime protection for embedded applications without delay to the programs. We utilize a coprocessor to perform analysis logic in RAFT, avoiding invasive modification to the existing architecture. Specifically, we implement RAFT as a coprocessor for a RISC-V Rocket Core. RISC-V [58] is an emerging open and free Instruction Set Architecture (ISA). It is widely used in embedded devices and has the advantages of RISC enhanced by its open-source nature. In RAFT, we introduce an innovative tag-storage mechanism with hybrid byte/variable granularity, which reduces the storage space while ensuring fine-grained protection. By reason of the limited program size of embedded applications and our optimized storage mechanism, it is feasible to store tags in the coprocessor for the embedded systems. Thus, we deposit tag storage in the coprocessor rather than in the memory of the main processor, which avoids frequent memory access requests to the main core when updating tags. We build our prototype of RAFT based on PHMon [20], a programmable hardware monitor, which is implemented as a RISC-V coprocessor as well. We modify PHMon to perform actions in a non-blocking manner and implement the pipelining of DIFT tasks, increasing the efficiency of processing instructions in the coprocessor so as to decrease the possibility of stalling the main core. By virtue of the programmability of PHMon, RAFT is capable of enforcing various security policies flexibly.

RAFT is deployed on the Rocket emulator and FPGA development board to evaluate its effectiveness and efficiency. We implement the same tag propagation framework with the traditional tag-storage mechanism (directly mapped to shadow memory) and our new tag-storage mechanism, and the experiment results show that our mechanism effectively cuts down the performance overhead from more than 20% to less than 0.1% on NBench [40] and CoreMark [1] microbenchmarks (detailed in Section 7.3). The runtime overhead introduced by RAFT on SPEC CINT 2006 [26] is negligible (0.13%), implying that it is suitable for security-critical applications under strict delay constraints (detailed in Section 7.4). RAFT incurs 34% LUTs and 93% Flip-Flops hardware overhead, and 2.32% power overhead compared to an unmodified RISC-V core. The hardware cost introduced by RAFT is reasonable, which will be discussed in detail in Section 7.5. Furthermore, we utilize a customized program to demonstrate its flexible functionality in detecting various security use cases. We also conduct a detailed evaluation with known CVEs and a medical application OpenSyringePump [59], which is often used as a sample real-world application in security applications for embedded systems (detailed in Section 7.6).

Goals and Contributions. This paper proposes RAFT, a flexible hardware-assisted coprocessor-based DIFT framework that provides runtime protection for embedded applications. Our goal is to track live data flow and detect security violations at runtime without delay to the protected programs, meeting the requirements of time-critical applications.

In summary, this paper presents the following contributions:

- **DIFT Framework**. We propose Raft, a flexible hardware-assisted DIFT framework that provides runtime protection for embedded applications without delay to the programs. We filter out DIFT-unrelated instructions by instrumentation and perform DIFT operations in a RISC-V coprocessor. The framework also provides programmers with abundant interfaces to enforce various security policies flexibly.
- **Storage Mechanism**. We introduce a novel storage mechanism with hybrid byte/variable granularity, reducing the size of tag storage while ensuring fine-grained protection. We retain tag storage in the coprocessor instead of memory, significantly reducing the communication between the coprocessor and the main core.
- **Prototype**. We implement the prototype of Raft on the RISC-V Rocket emulator and FPGA development board. We utilize known CVEs and a real-world embedded application to demonstrate its functionality and security provisions. The macrobenchmark and microbenchmarks evaluation result shows that the performance overhead introduced by Raft is negligible. The source code of Raft can be found via `https://github.com/Compass-All/Raft`.

The remainder of the paper is organized as follows. Section 2 provides the background information about DIFT and PHMon. Section 3 reviews related hardware-assisted DIFT works and previous tag-storage mechanisms. Then, we introduce the scope and assumptions of our work in Section 4. Section 5 presents the design of our proposed Raft and demonstrates our new tag-storage mechanism. We give the implementation of Raft in Section 6. Section 7 shows the security analysis, hardware resource cost, and performance evaluation of Raft. Section 8 discusses the limitations of our work. Section 9 concludes the paper.

2 BACKGROUND

2.1 Dynamic Information Flow Tracking

DIFT [44] is a widely-applied computer security technique that tracks live data flow during program execution, which is also called Dynamic Data Flow Tracking (DDFT) or Dynamic Taint Analysis (DTA). The processing of DIFT typically consists of three stages: tag initialization, tag propagation, and tag checking. It utilizes taint tags to mark untrusted data or critical data and propagate tags as the program executes. DIFT checks whether the used data is tainted when security-sensitive operations occur. The untrusted data typically comes from local input or remote input (called sources). Taint tags are propagated during program execution in accordance with instruction type, instruction operand, and predefined tag propagation rules. Note that most DIFT works only propagate tags based on data dependencies [13]. Program points where DIFT performs tag checking are called sinks. For instance, we can specify control-flow transfer instructions as sinks to detect control-flow hijacking attacks, output functions to detect sensitive information leakage attacks, or locations where critical data is used to detect data corruption attacks.

There are different tag granularities to mark a bit, a byte, a word, or a block of data. Fine-grained protection offers enhanced security but requires more storage space to store tags. As data operations usually take bytes as the smallest unit, byte-level tag granularity is commonly considered sufficient for fine-grained protection. Moreover, some works [21, 48] support multiple-bit tags to enforce different security policies at the same time. On the other hand, longer tag sizes will lead to more wasted storage space.

2.2 PHMon

Programmable Hardware Monitor (PHMon) [20] is an efficient programmable hardware monitor to enforce an event–action monitoring model. Hardware-assisted DIFT frameworks are tag-based monitors that monitor the program and take actions based on the tag propagation. Unlike hardware-assisted DIFT works, PHMon is a trace-based monitor, which monitors the user-defined events and performs actions based on the trace of program execution. They define a set of events, and each event is defined by a set of monitoring rules. Once an event is detected, PHMon performs a sequence of follow-up actions. PHMon consists of three main architectural units: a Trace Unit, Match Units, and an Action Unit. The Trace Unit collects the instruction execution trace of a processor, Match Units examine the execution trace to find matches with programmed events, and the Action Unit takes follow-up actions. PHMon is implemented as a coprocessor of the RISC-V Core as well and provides a list of functions to communicate with the coprocessor. Moreover, PHMon performs actions in a blocking manner, i.e., it only performs one action at a time. When a memory request to the main processor is sent, PHMon blocks the rest of the actions until a memory response is received.

3 RELATED WORK

3.1 Hardware-assisted DIFT Techniques

Software-based DIFT solutions [4, 19, 24, 42, 44] often incur huge performance overhead due to DBI. In order to overcome it, various hardware-assisted mechanisms have been proposed. Hardware-assisted DIFT frameworks logically extend each register and each memory location with a tag (single bit or multiple bits). During program execution, the hardware transparently propagates and checks tags to track untrusted data. Table 1 summarizes and compares existing hardware-assisted DIFT works. Tag Size refers to the maximum width of the tag supported in work, and Tag Granularity is the size of the memory location with a tag. There are three design alternatives: integrated in-core designs, multicore-based offloading designs, and coprocessor-based off-core designs.

In-core. This approach directly integrates DIFT operations into the pipeline of the main core. Suh et al. [51] identify a set of input channels as spurious and track information flows in parallel with data processing. Raksha [18] duplicates each stage of the pipeline to perform tag propagation and checks in parallel with regular instruction execution. All storage elements, including registers, caches, and DRAM, are extended with tags. This work first provides a set of policy configuration registers to describe the propagation and check rules. FlexiTaint [53] adds two pipeline stages ahead of the final commit stage, which updates a separate register file and cache for tags. This simplified design minimizes the impact on the complex out-of-order core. Similar to Raksha, Palmiero et al. [47] apply DIFT to RISC-V architecture. Although the performance impact of integrated in-core designs is minimal, it requires invasive

Table 1: Comparison on hardware-assisted DIFT techniques.

Type	Works	Tag Size	Tag Granularity	Experimental Target	Communication Interface	Performance Overhead	Hardware Overhead
In-core	[51]	1-bit	1-byte	Alpha, SimpleScalar simulator	Signals	0.8%	4.5%(Storage)
	[18]	4-bit	Per word	SPARC, XC2VP6000 FPGA	Signals	Negligible	12.5%(Storage)
	[53]	2-bit	Per word	MIPS, SESC simulator	Signals	1%-3.7%	Unspecified
	[47]	4-bit	Per word	RISC-V, XC7Z020 FPGA	Signals	Negligible	12.5%(Storage), 0.82%(LUTs)
Offloading	[43]	32-bit	Per word	x86, Simics simulator	Hardware queue	48%	Unspecified
	[50]	8-bit	Per word	x86, Simics simulator	Log buffer in cache	120%	Unspecified
	[12]	8-bit	Per word	x86, Simics simulator	Log buffer in cache	2%-51%	Unspecified
Off-core	[31]	4-bit	Per word	SPARC, XC2VP30 FPGA	Signals	0.79%	16%(BRAMs), 7.64%(LUTs)
	[21]	1-bit	Per word	SPARC, Virtex-5 FPGA	Signals	17%	14.8%(Area)
	[22]	1-bit	Per word	SPARC, Leon3	Signals	<20%	55%(Area)
	[27]	1-bit	Per word	SPARC, Virtex-5 FPGA	System bus	45.7%	14.47%(Area)
	[33]	1-bit	Per word	SPARC, XC5VLX330 FPGA	Core debug interface	1.6%	60%(BRAMs), 28.36%(LUTs)
	[57]	1-bit	1-byte	ARM, Zedboard	EMIO and system bus	5.37%-24.6%	0.47%(Area)
	[56]	32-bit	1-byte	ARM, Zedboard	EMIO and system bus	335%	0.95%(Area)
	[10]	1-bit	Per variable	RISC-V, Artix-7 FPGA	Signals	5.03%	24%(BRAMs), 214%(LUTs), 33%(FFs)
	This Work	1-bit	1-byte	RISC-V, KC705 FPGA	Signals	Negligible	2.31%(Power), 34%(LUTs), 93%(FFs)

modification to the processor architecture, which introduces huge design and verification costs. The extra effort required to redesign and revalidate a complex superscalar processor deters hardware vendors from adopting this approach [31]. Even on open-source RISC-V, performing invasive hardware modifications to existing processor designs presents a major obstacle in adopting DIFT in practice.

Offloading. An alternative approach is to involve a separate processor core to perform DIFT analysis for another core in a multicore chip. Nagarajan et al. [43] spawn a helper thread scheduled on a separate core, which is only responsible for DIFT. Aside from program execution, the first core compresses and stores the information required for DIFT inside a hardware queue. The second core decompresses this information and enforces DIFT operations. Similarly, Ruwase et al. in [50] and Chen et al. in [12] utilize a log buffer in the cache to enable communication between the main thread and the DIFT thread. The modification of multicore-based offloading designs to the existing hardware platforms is negligible. On the other hand, it introduces high performance overhead for inter-core coordination. Moreover, these designs are suitable for multicore systems and require a full general-purpose core for DIFT analysis, which reduces the number of available cores and increases energy consumption.

Off-core. This approach performs DIFT operations on a dedicated coprocessor instead of a general-purpose processor. Coprocessor-based off-core designs are a compromise approach. This approach only makes minor modifications to the main processor and requires a simple hardware accelerator instead of a general-purpose processor core. Since DIFT operations are delegated to a coprocessor, which can be easily packaged, it incurs a relatively low development and deployment cost. That is the main reason why we adopt the coprocessor-based approach in this paper. Off-core designs also require communication with the main core. Such communication, which includes memory access requests and synchronisation checks, is mainly responsible for the performance overhead of this approach. The key to reducing runtime overhead is to diminish the required information and increase the speed at which the coprocessor processes instructions.

Raksha v2 [31] first decouples DIFT functionality onto a coprocessor and provides the same security guarantees as previous in-core designs. The pipeline is slightly modified to export the required information to the coprocessor. Similarly, Deng et al. [21, 22] propose a generic hardware accelerator that works up to 1 GHz. The accelerator can be configured to enforce different security policies. Heo et al. [27] instrument memory addresses and branch instructions to reconstruct the CFG and recover memory addresses. Although the way that uses instrumentation to recover DIFT information is more flexible and portable on hardcore CPUs, it incurs more performance overhead. Unlike previous frameworks, some works [33, 56, 57] utilize Core Debug Interface (CDI) to extract the required information for DIFT. Lee et al. [33] exploit ARM CoreSight Event Trace Macrocell (ETM) to track all instructions. To improve performance, Wahab et al. [56, 57] use CoreSight Program Trace Macrocell (PTM) as a trace component. They only trace instructions related to control flow and apply static analysis and instrumentation to retrieve the missing information. Additionally, FineDIFT [10] uses a Content Addressable Memory (CAM)-like structure as tag storage to reduce storage demands. However, recursive function calls and nested function calls may not be protected effectively due to the limited number of CAM-like entries. Some off-core designs [31, 33] utilize a tag cache to mitigate memory traffic from the main core and report fairly low runtime overhead in Table 1. However, when clocking the coprocessor at a lower frequency than the main core, the performance overhead becomes non-negligible (11.7% at the 2x ratio in [31]).

3.2 Tag-Storage Mechanism

For hardware-assisted DIFT methods, all registers and memory are extended to support tags. There are two basic types of tag storage: extended memory and shadow memory. The former directly expands memory word width. This approach obtains data and tags simultaneously when accessing memory. However, it is difficult to deploy and incompatible with existing techniques. The latter stores tags in a separate memory region referred to as shadow memory. It obtains tags by mapping the address of data into shadow memory. This approach only requires reserving a separate memory in advance and no modification to the memory mechanism, which is commonly adopted in existing hardware-assisted DIFT frameworks [18, 31, 53]. Nonetheless, the method that directly maps into shadow memory requires massive storage space. For instance, it will take 1 GiB shadow memory to cover the entire 8 GiB memory space for byte-level granularity protection, which incurs waste on storage space. Recently, FineDIFT [10] applied a CAM-like structure as tag storage, which records the base address of the allocated memory and its size. This method reduces the size of tag storage while increasing the complexity of accessing tags (an increase of 214% in the use of lookup tables with a CAM-like structure).

Among coprocessor-based DIFT architectures, FineDIFT [10] is the closest to Raft in terms of targeting embedded applications, depositing tag storage in the coprocessor, and being implemented on RISC-V. FineDIFT focuses on utilizing a series of custom instructions to dictate propagation. Thus, it introduces higher performance overhead and an extra increase in binary size. In addition, FineDIFT relies on programmers to configure which variables should be tracked and how to track them, which is labor-intensive work. Programmers need to be familiar with the management of registers and memory used in the program and manually set appropriate flags for the corresponding variables. Compared with it, Raft automatically performs taint analysis for each instruction according to the security policy and is easier to use. Since FineDIFT requires allocation merge and split operations in the CAM-like entries, it has issues with variadic functions, aggregates, and library functions deallocating memory blocks, etc. Moreover, FineDIFT provides a coarser variable-level granularity trace (i.e., tag a block of data) instead of byte-level protection.

4 SCOPE AND ASSUMPTIONS

We aim to protect applications running on embedded devices, especially those time-critical applications. Our current work provides fine-grained runtime protection for a single program on a single core. Recent studies [10, 21, 31] on coprocessor-based DIFT also focus on a single processor without multi-threading as the first step. We discuss this in Section 8.2. We assume that an attacker can remotely or locally utilize applications' vulnerabilities to corrupt the internal data and state of the program or leak sensitive data. We also assume that our protected applications running on embedded devices are benign but might be vulnerable because it makes no sense to protect inherently malicious programs. Furthermore, we assume all hardware components are trustworthy. Thus, we do not consider the corruption of executed instructions committed to the coprocessor and the security of tags in the coprocessor. Finally, physical attacks such as side-channel and JTAG attacks are out of scope.

5 RAFT DESIGN

We propose a flexible hardware-assisted architecture to enforce DIFT and improve its performance. Our framework Raft is designed as a coprocessor, thus introducing minimally-invasive changes to the main processor and allowing it to be integrated into different architectures. This framework provides programmers with various interfaces to specify which instructions and memory regions to track and what policy to utilize for each instruction. In this section, we first present the architecture of Raft. Then, we introduce our novel tag-storage mechanism in detail. Lastly, we further demonstrate the thorough process of DIFT in Raft under the new storage mechanism.

5.1 Architecture

Figure 1 shows an overview of Raft's architecture. Raft is designed as an extension of the RISC-V Rocket [3] Core and communicates with the main core by Rocket Custom Coprocessor (RoCC) Interface. The gray area represents the modified and added components in our architecture design. To perform coprocessor-based DIFT, Raft consists of four main components: a Trace Unit (TU), a Filter Unit, a Control Unit, and an Interrupt Manager. The Trace Unit is used to collect the runtime execution information of the main processor and commit it to Raft by the RoCC Interface. After that, the Filter Unit filters out those instructions unrelated to DIFT and enqueue the remaining instructions into the Instruction Queue (Ins Queue). The Control Unit is responsible for controlling analysis logic to perform DIFT operations, which will be presented in detail in Section 5.4. Once the Instruction Queue is full, a full signal is sent to the Interrupt Manager to stall the main processor for avoiding executed instructions miss. Security violations will also trigger the coprocessor interrupt and then be handled by the Linux interrupt handler in the main processor.

Trace Unit. The Trace Unit is responsible for collecting the runtime information about the main processor and committing it to our DIFT framework. The collected information (called commit log) contains four separate entries, i.e., the undecoded instruction (`inst`), the current program counter (`pc`), the memory/register address used in the current instruction (`addr`), and the data accessed by the current instruction (`data`). This component is built upon PHMon's Trace Unit. For each executed instruction, the Trace Unit generates a commit log in the write-back stage of the Rocket processor's pipeline and subsequently transfers it to the coprocessor by the RoCC interface.

Filter Unit. The Filter Unit is responsible for filtering out DIFT-unrelated instructions that do not involve tag propagation in DIFT. Specifically, we use compiler instrumentation to insert two custom instructions before and after the block of DIFT-unrelated instructions, implying that these instructions do not involve tag propagation. The Filter Unit is set to neglect the block of instructions wrapped by these custom instructions. Because filtered-out instructions do not affect the result of tag propagation, the Filter Unit does not cause additional false negatives. Note that it is necessary to set a hardware component in the coprocessor to filter out DIFT-unrelated information and further improve performance, but the strategy that decides which instructions should be filtered is not our contribution.

Control Unit. The Control Unit is responsible for controlling Raft analysis logic. It performs the following tasks: 1) Fetch. Dequeue an instruction from the Instruction Queue. 2) Decode. Identify the opcode of the instruction and the index of registers. 3) Inquiry.

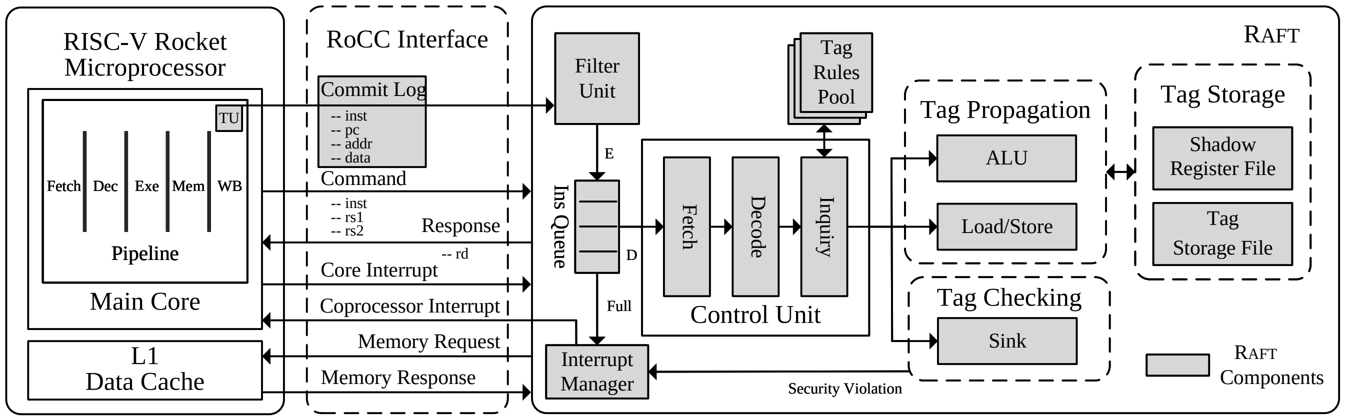


Figure 1: Architecture overview of RAFT.

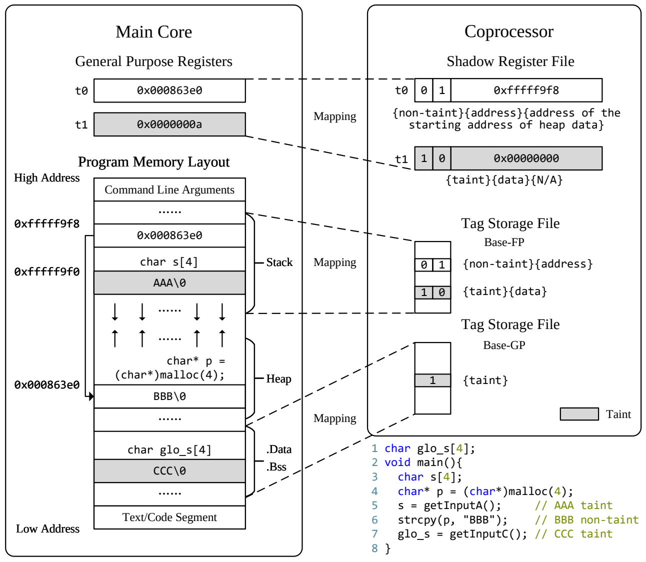
Query Tag Rules Pool to obtain tag propagation or check rules according to the current instruction' opcode. Subsequently, RAFT performs the DIFT operation according to the rule. When the DIFT operation is done, the Control Unit dequeues the next instruction and repeats these tasks until the Instruction Queue is empty.

Interrupt Manager. The Interrupt Manager is responsible for managing the coprocessor interrupt. A full signal indicates that the Instruction Queue is full. It will trigger a coprocessor interrupt to stall the main processor until the queue is empty. A security violation signal indicates that tainted data is used to perform an insecure operation. The Interrupt Manager will record the current pc value in a pre-reserved memory and trigger a coprocessor interrupt to notify the main processor.

5.2 Tag Storage

As mentioned in Section 3.2, the previous scheme allocates a tag for every memory address and utilizes shadow memory for tag storage. Such a scheme takes 1 GiB shadow memory to tag 8 GiB memory with byte-level granularity, requiring massive memory for tag storage. In RAFT, we propose a new structure of tag to reduce the size of tag storage effectively. Our new tag structure is based on an observation that the way to access an allocated heap memory is usually through the start address of heap data stored on the stack or registers. We extend the tag with one bit to indicate whether the content stored on the stack is an address pointing to heap data or not. Consequently, we can utilize the information on the stack to represent the tags of heap data. Figure 2 sketches the tag-storage mechanism in RAFT, which consists of Tag Storage File (TSF) and Shadow Register File (SRF). TSF is used to tag program memory, consisting of TSF (Base-FP) and TSF (Base-GP). FP and GP are the abbreviations of Frame Pointer and Global Pointer, respectively. TSF (Base-FP) stores the tags of the stack, and TSF (Base-GP) stores the tags of the uninitialized and initialized data segment. SRF is used to tag general-purpose registers in the main processor.

Tag Storage File. In TSF (Base-FP), we use two bits to tag 1-byte memory on the stack. The first bit indicates whether tainted or not, and the second bit indicates data or address. In TSF (Base-GP), we use one bit to tag 1-byte memory on the .bss segment or .data



```
1 char glo_s[4];
2 void main(){
3   char s[4];
4   char* p = (char*)malloc(4);
5   s = getInputA();      // AAA taint
6   strcpy(p, "BBB");     // BBB non-taint
7   glo_s = getInputC(); // CCC taint
8 }
```

Figure 2: Tag storage in RAFT.

segment. To see an example, consider the code snippet shown in Figure 2. The local variable s (line 5) and global variable glo_s (line 7) accept unsafe external input, which should be tagged as taint. The constant 'BBB' is copied to the heap memory pointed by variable p (line 6), which is non-taint. As Figure 2 depicts, the program layout is shown in the left half, and the gray area represents tainted memory. The right half of Figure 2 shows the corresponding tags in the coprocessor. For instance, the value of variable p is stored on the stack 0xfffff9f8. In TSF (Base-FP), the tag of the memory 0xfffff9f8 is 0-1, signifying that there stores an address and the memory region 0x000863e8 pointed by variable p is not tainted. The local variable s is stored on the stack 0xfffff9f0, and the tag is 1-0 in TSF (Base-FP), indicating that the variable s is tainted. In such a way, we can denote both heap and stack tags in TSF (Base-FP). Therefore, we adopt a storage mechanism with hybrid byte/variable

Table 2: Example code of tag propagation under the new tag-storage mechanism. The third column is the corresponding tag storage in the coprocessor.

	Main Core	**Coprocesoor**
C Code	strcpy(p, s);	
ASM Code	1 lw a1, 48(sp) // load address p to a1 register	SRF(a1): 0-1-48(sp)
	2 lw a4, (-1)a0 // load tainted data from buffer s to a4 register	SRF(a4): 1-0-N/A
	3 sw a4, (1)a1 // store tainted data to address p	TSF(sp+48): 1-1

granularity (i.e., tag stack/data segment with byte-level granularity and tag heap with variable-level granularity).

Shadow Register File. In SRF, we use 64 bits to tag a general-purpose register. Each coprocessor register holds the tag of the corresponding register in the main core. Similar to TSF, the tag in SRF is extended to 64 bits containing three parts: the first bit indicates whether tainted or not, the second bit indicates data or address, and the remaining bits indicate the location of the address stored on the stack. The third part is used to guarantee updating tags correctly in TSF when the content is stored from the register to memory. For instance, when the variable `p` is loaded to the register `t0`, the tag of register `t0` in SRF is `0-1-0xfffff9f8`, signifying that the memory pointed to by variable `p` is not tainted. The start address of the allocation is stored on the stack `0-1-0xfffff9f8`.

5.3 Tag Initialization

In RAFT, we utilize custom instructions to initialize tags in the coprocessor. When the program is executed, the main processor will skip these custom instructions and commit to the coprocessor. We instrument external input functions to initialize taint (the first bit of the tag). For instance, we insert an instruction behind the file input function `fgetc()`, implying that the function return value register is tainted. Similarly, we insert an instruction ahead of the file output function `fputc()` to check whether the function argument register is tainted or not. Furthermore, to distinguish data and address, we instrument memory allocation functions (e.g., `malloc()`) to mark the return value register as an address (the second bit of the tag).

5.4 Tag Propagation and Tag Checking

Tags are propagated in accordance with instruction type, instruction opcode, and configured rules. As described in Algorithm 1, we divide instructions into four types: ALU, Load, Store, and Sink operations. For ALU operations, tags are propagated in SRF as values are delivered among general-purpose registers. For Load operations loading a value from memory M to a register, tags are propagated from TSF to SRF. If the loaded value is an address, we record the memory address M in SRF as well. For Store operations storing a value in the register to memory, tags are propagated from SRF to TSF. If the stored value is an address, we update TSF according to the recorded address M. RAFT decides to update TSF (Base-FP) or TSF (Base-GP) based on the range of the address. For Sink operations, we perform security checks. Once the register used by the current instruction is tainted, a security violation signal is sent to the Interrupt Manager. Note that we can specify which instruction belongs to which instruction type by a list of functions. This software interface is built upon functionality provided by PHMon. For instance, to detect control-flow attacks, we can specify jump instructions as a Sink operation. Thus, the behavior of jumping to the address controlled by an attacker will be detected by RAFT.

Table 2 demonstrates an example of the thorough process of tag propagation under the new storage mechanism. The second row is a simplified ASM code of `strcpy()` that copies buffer `s` to the memory pointed by `p`. The third column is the corresponding tag storage in the coprocessor. Line 1 loads the address `p` to register `a1`. The address of buffer `s` is stored in register `a0`, and line 2 loads tainted data to register `a4`. When the program stores tainted data to address `p` (line 3), RAFT can update TSF according to the recorded address M `sp+48`. Taints are successfully propagated when data is copied between stack and heap regions. Listing 1 shows another example to demonstrate the validity of tag propagation via heap in RAFT. Line 1 accepts an external input and tags variable `ch` as tainted. Line 3 accesses a heap memory region by an offset and copying the tainted data `ch` to `p[2]`. The taint is successfully propagated to variable `y` via the heap memory region pointed by `p` (line 3). These two basic examples indicate there are no taint loss issues due to the inclusion of heap objects on the taint-propagation path. Note that Listing 1 also indicates RAFT may have overtaint issues for heap variables, a concern we discuss in detail in Section 8.1.

Algorithm 1: Tag Propagation and Tag Checking

```
   Input: instruction type $instType$
1  if $instType == ALU$ then
       /* update SRF                                   */
2      if $rs2$ is N/A then
3          $Tag(rd) \leftarrow Tag(rs1)$;
4      else
5          $Tag(rd) \leftarrow Tag(rs1) \vee Tag(rs2)$;
6      end
7  else if $instType == Load$ then
       /* update SRF                                   */
8      $Tag(rd) \leftarrow Tag(Mem[rs1 + offset])$;
9  else if $instType == Store$ then
       /* update TSF                                   */
10     $Tag(Mem[rd + offset]) \leftarrow Tag(rs1)$;
11 else if $instType == Sink$ then
       /* taint checking                               */
12     if $rs1$ is tainted then
13         $triggerCoprocessorInterrupt()$;
14     end
15 end
```

As Algorithm 1 shows, we apply direct data tainting that tracks external input. The propagation rules are similar to the rules used for input tainting by FlexiTaint [53]. Although we do not apply pointer tainting that tracks valid heap pointers considering false positives and taint explosion, our tag-storage mechanism is inherently compatible with propagating both taints according to data pointing and pointer tainting. The reason is that the tag of Raft contains two parts where the first bit is tainted external input data, and the second bit is a heap pointer, which meets the requirements and has no conflicts with pointer tainting.

```
    char ch = getchar(); // source
    int* p = (int*)malloc(sizeof(int));
    p[2] = ch;
    int y = *p; // taint is propagated to y
```

Listing 1: Example code of tag propagation via heap.

6 IMPLEMENTATION

We implemented our prototype based on PHMon [20]. In this section, we first introduce Raft instructions. After that, we detail the implementation of the coprocessor, especially the improvements for PHMon. Then, we present the necessary modifications to the toolchain and Linux kernel.

6.1 Custom Instructions

Table 3 shows the functionality of custom instructions we implemented to assist the coprocessor in performing DIFT operations. The main processor will skip these instructions and commit them to the coprocessor, thus not stalling the processor's pipeline.

Tag Initialization. When a program obtains external input data, we use the `taint` instruction to pass the index of the register that stores tainted data according to the calling convention. In addition, when the program calls memory allocation functions, we use the `src` instruction to pass the index of the register that stores the heap pointer to the coprocessor.

Taint Propagation. We utilize custom instructions to simplify the process of tag propagation in some library functions. For instance, for the function `atoi()` that converts a string to an integer, we instrument the `arg` instruction behind the function, explicitly indicating that the tag of the argument register is propagated to the return value register. The tag is copied from the source register of `arg` to the destination register. Moreover, the `open` and `close` instructions are used to filter out DIFT-unrelated instructions. The block of instructions they enclose will be neglected by the coprocessor. For instance, for the function `islower()` that checks if the given character is lowercase, we instrument the `open/close` instructions behind/after the function. The instructions that implement function `islower()` are DIFT-unrelated instructions, which will be disregarded by Raft.

Taint Checking. When a program executes security-sensitive functions, we utilize the `sink` instruction to pass the potentially tainted parameters of these functions to the coprocessor. When detecting tainted, the coprocessor triggers a coprocessor interrupt and reports which instruction causes the security-violation operation by writing reserved memory.

Program Addressing. The `base` instruction is used to pass the frame pointer and global pointer to the coprocessor. These two base addresses are used to assist the coprocessor in recording tags in TSF (Base-FP) and TSF (Base-GP), respectively.

6.2 Coprocessor Implementation

In order to support DIFT, we store the taint information of the Action Unit's execution result in SRF and TSF. Furthermore, we add four new types of actions: taint alu operation, taint load operation, taint store operation, and sink operation.

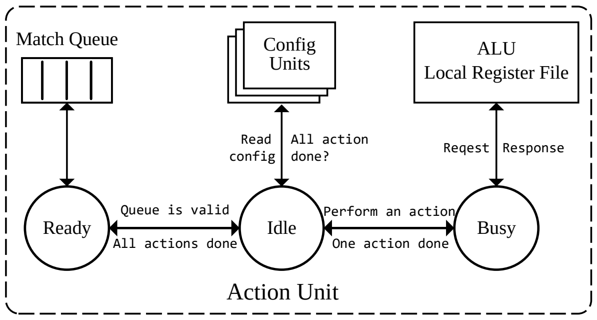


Figure 3: The finite-state machine of Control unit in PHMon.

The Action Unit in PHMon comprises Config Units, a Control Unit, an ALU, and a Local Register File. As shown in Figure 3, the Finite-State Machine (FSM) of the Control Unit includes three states: ready, idle, and busy. After receiving data from the queue, the state is converted from ready to idle. In the idle state, the coprocessor parses data, obtains the corresponding rules from the Config Unit, and transfers to busy. In the busy state, the coprocessor executes the corresponding action and returns to ready. It takes at least three clock cycles for the coprocessor to process an instruction, while the main processor only needs one clock cycle to enqueue the instruction. Because these three states in DIFT are decoupled, we are capable of pipelining tasks. Specifically, we optimize the Control Unit to perform non-blocking by adding registers after each state to buffer the information of the previous state. As a result, the coprocessor can dequeue data, match rules, and take actions simultaneously. In this way, we improve the performance of the Control Unit and achieve greater throughput to meet the requirements of DIFT.

6.3 Toolchain and OS Support

We add new passes to the LLVM compiler to recognize and emit the aforementioned custom instructions. Moreover, we modify the Linux kernel to handle the incoming interrupts from the RoCC interface. We modify the OS to reserve a separate memory region for distinguishing the coprocessor interrupt triggered by a queue full or security violation signal and handle the interrupt in different ways. For a queue full signal, the OS executes `nop` instructions until the queue is empty. For a security violation signal, the OS reports the current `pc` value and terminates the process. Raft is turned on when switching to the protected process and turned off when a context switch into another process occurs. We also modify the OS to save or restore per-process DIFT-related information during context switches.

7 EVALUATION

In this section, our evaluation aims to answer the following research questions.

Table 3: Custom instructions functionality. Among them, base, open, and close instructions are directly executed by the coprocessor, and taint, arg, and src instructions are committed to the coprocessor and processed as a commit log.

Instruction	Usage
taint rs1, rs2	Mark the taintedness
src rs1, rs2	Mark heap pointer variables
arg rd, rs1	Assist tag propagation
open/close zero, zero	Filter out DIFT-unrelated instructions
sink rs1, rs2	Perform security checks
base fp/gp, rs2	Pass the frame pointer and global pointer to the coprocessor

RQ1: How effective is the Filter Unit?

RQ2: How does our new storage mechanism compare against traditional storage using shadow memory?

RQ3: What is the performance overhead of Raft? What is the performance of the coprocessor?

RQ4: What is the hardware resource cost of Raft?

RQ5: How does Raft perform when detecting malicious operations?

7.1 Experimental Setup

We prototype Raft as a coprocessor of RISC-V Rocket Core on a Xilinx Kintex-7 FPGA KC705 evaluation board. Due to the evaluation board limitation, the maximum frequency of the Rocket Core is 60 MHz in our experiments. The system is configured with a 16 KiB 4-way L1 instruction cache, a 16 KiB 4-way L1 data cache, a 32-entry instruction TLB, and a 32-entry data TLB. Moreover, it has 128 MiB boot ROM and 1 GiB DDR3 memory. We perform experiments with a modified RISC-V Linux (v5.4) kernel. All programs are compiled by LLVM 12.0.1 for RV64GX (G for the general combination of standard instructions and X for customized instructions) architecture.

7.2 RQ1: Effectiveness of Filter Unit

Table 4: Statistics of reduced instructions by the Filter Unit.

Program	Num of Instructions		%
	Before	After	
cat file	9,710	7,722	- 20.47%
comm -3 file1 file2	9,186	7,181	- 21.83%
cut -c num file	12,329	9,866	- 19.98%
head file	9,425	7,428	- 21.19%
nl file	14,971	10,289	- 31.27%
od file	24,372	21,903	- 10.13%
ptx file	38,986	34,745	- 10.88%
tail file	10,240	8,238	- 19.55%
truncate –size num file	10,169	8,175	- 19.61%
uniq file	12,678	10,675	- 15.80%

The Filter Unit is used to filter out DIFT-unrelated instructions. We use ten file content processing utility cat, comm, cut, head, nl, od, ptx, tail, truncate, and uniq from Coreutils (version 9.1) to evaluate the effectiveness of the Filter Unit. In detail, we count the number of instructions processed by Raft with and without instrumentation (i.e., with and without `open`/`close` instructions). Table 4 shows the statistics of the reduced instructions by the Filter Unit. The first column is the ten C/C++ programs we used in our evaluation. The second and third columns show the total number of instructions processed by the coprocessor before and after instrumentation, followed by the last column of the ratio of reduced instructions. The result shows that we have reduced 10.13%-31.27% of the possible tainted instructions compared to the original program. We only filter out instructions that must not involve tag propagation in the experiment. If adopting a more radical strategy, more DIFT-unrelated instructions will be filtered out. The filtering strategy itself is not our contribution. The Filter Unit is set in Raft to provide hardware support for diminishing the amount of information required for DIFT. The result shows that the Filter Unit effectively cut down the number of instructions required for DIFT. Note that for the sake of fairness, the Filter Unit is turned off to ensure that the coprocessor processes the same number of instructions for DIFT in the following evaluation.

7.3 RQ2: Comparison against Traditional Storage

In order to see the performance improvement of Raft with our new tag-storage mechanism, we also implement a hardware-assisted DIFT framework with traditional storage using shadow memory. The implementation of this DIFT framework is the same as Raft, apart from using different tag-storage mechanisms. We utilize NBench [40] and CoreMark [1] microbenchmarks to evaluate the performance of Raft with our tag-storage mechanism (denoted as Our Work) and traditional storage using shadow memory (denoted as Shadow Memory) and make a comparison. Specifically, NBench is a synthetic computing benchmark used to expose the capabilities of a CPU, FPU, and memory system, including ten different tests. CoreMark is a sophisticated benchmark designed to test CPU bound by producing a single-number score. In performance experiments, we annotated I/O input and output functions in glibc as the locations of sources and sinks. We ran each benchmark ten times and calculated the arithmetic mean.

Figure 4 depicts the performance overhead comparison with Shadow Memory and Our Work on the NBench benchmark. Note that we turn off the Filter Unit, and both frameworks are performed to process the same number of instructions. In Figure 4, the baseline (i.e., original programs without DIFT) is standardized as 1, and we can find that the performance of Raft is approximately

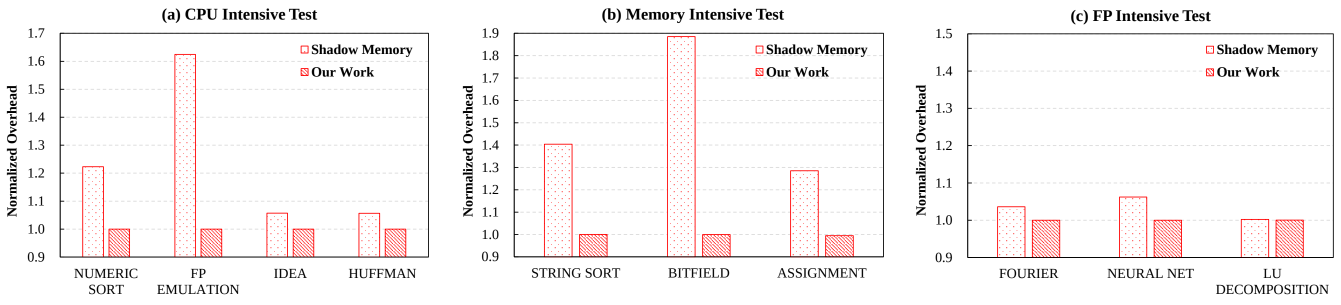


Figure 4: Performance overhead comparison on NBench.

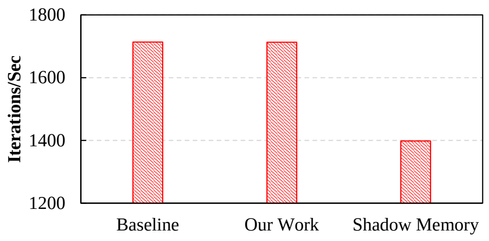


Figure 5: Performance comparison on CoreMark.

the same as the baseline. Since there are fluctuations in the running environment, the normalized performance overhead of some programs is slightly less than 1. Moreover, the framework using shadow memory incurs 26.37% overhead on average. It introduces 88.55% on BITFIELD at most, and memory-intensive tests lead to higher overhead.

Figure 5 portrays the performance comparison on the CoreMark benchmark. When employing the framework using shadow memory, the iterations per second are dropped from 1,713.60 to 1,398.38, introducing 22.54% overhead on average. However, we can observe from Figure 5 that the iterations per second in Raft are almost the same as the baseline. The impact on the program is negligible for Raft on Cormark, while the approach employing shadow memory as tag storage significantly affects the program. In other words, compared to the approach using shadow memory, Raft effectively cuts down the performance overhead from >20% to <0.1% with our new tag-storage mechanism.

Our Work vs. Shadow Memory. To further understand the reason for the performance improvement, we discuss it in detail. The framework using shadow memory exploits a separate memory to store tags. When tracking load/store instructions, the coprocessor needs to send a memory request to update tags in shadow memory. Subsequently, the main processor is stalled and turns to process the memory request and then sends back a memory response. Therefore, the performance overhead introduced by this approach mainly comes from frequent memory operations. That is the reason why it incurs higher performance overhead on memory-intensive tests.

It is unsuitable to directly deposit tags in the coprocessor due to the limited storage space of the coprocessor. Our work adopts a new tag-storage mechanism to reduce the size of tag storage at the expense of granularity. The reduced tag-storage space and the limited size of embedded applications help Raft to store tags in the coprocessor instead of the main processor, thus avoiding frequent memory requests. When tracking load/store instructions, Raft directly updates tags in the coprocessor without waiting for memory responses. It increases the speed of tracking instructions in the coprocessor, and further decreases the possibility of Instruction Queue full blocking the main core. Additionally, it also eliminates the performance impact on the main processor due to processing memory requests. Therefore, Raft incurs negligible performance overhead for the main core and has little impact on protected programs.

7.4 RQ3: Performance Overhead

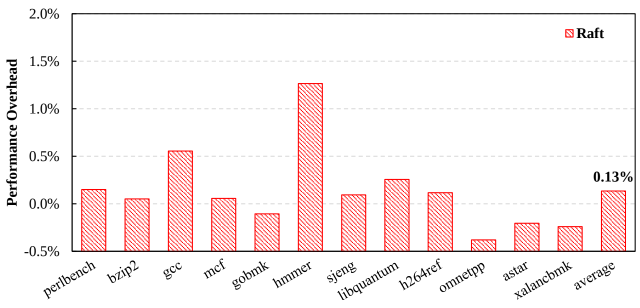


Figure 6: Performance overhead on SPEC CINT 2006.

As Figure 4 and Figure 5 show, the performance overhead introduced by Raft is negligible. To better measure the performance overhead of Raft, we also run SPEC CPU benchmark suites. Due to its high memory requirements, we could not run SPEC CPU2017 on our RISC-V platform. We test SPEC CINT 2006 [26] benchmarks, and each benchmark is run ten times. As Figure 6 sketches, the performance overhead introduced by Raft on SPEC CINT 2006 is 0.13% on average. In other words, applying Raft to provide a program with runtime protection has little impact on the program running on the main processor.

The performance overhead of Raft is contingent on the speed of the main processor executing instructions and the speed of the coprocessor processing instructions. Due to the simplicity of DIFT operations, the coprocessor has a very shallow pipeline. Raft processes instructions with a 3-stage pipeline consisting of fetch/decode, inquiry, and execute. It takes three clock cycles to process an ALU operation, and the Load and Store operations are the same because there is no need to access memory. Moreover, it only handles committed instructions from the main core. By contrast, the main core has to deal with mispredicted instructions, instruction cache misses, and pipeline dependencies. Therefore, even for a superscalar processor, a DIFT-dedicated coprocessor can achieve close Instruction Per Clock (IPC).

Table 5: Performance overhead comparison when the coprocessor is paired with higher-frequency main cores.

	Processor/Coprocessor Frequency Ratio			
	1x	1.5x	2x	3x
Raksha v2 [31]	0.79%	3.8%	11.7%	\
RAFT	<0.1%	\	<0.1%	<0.1%

To further explore the performance of the coprocessor, we construct an experiment in which we clock the coprocessor at a lower frequency than the main core, similar to Raksha v2 [31]. We set up a main core with a peak instruction processing rate 2x and 3x that of the coprocessor and rerun the CoreMark benchmark. The size of the Instruction Queue is set to 16 to keep consistent with Raksha v2. As Table 5 demonstrates, RAFT introduces negligible performance overhead (<0.1%) at the 3x ratio, while Raksha v2 reports 11.7% [1] at the 2x ratio in the paper. Raksha v2 utilizes a tag cache to mitigate memory traffic from the main core. Consequently, as the frequency gap increases, the overhead becomes higher owing to the tag cache miss and the full queue that blocks the main core. Compared with Raksha v2, RAFT deposits tag storage in the coprocessor and updates tags via register access. The case where the queue is full does not happen in the experiment, so the performance overhead remains negligible. This indicates that the coprocessor performs well and can be used with a superscalar core.

7.5 RQ4: Hardware Resource Cost

To assess the hardware resource cost of RAFT, we instantiate the original and the modified RISC-V Rocket Cores and synthesize them on the FPGA development board using Vivado 2018.3. In our evaluation, we configure RAFT with 9 MUs, 32 64-bit registers for SRF, 1 KiB registers for TSF (Base-FP), and 9 KiB registers for TSF (Base-GP), which is sufficient to accommodate all experiments in our evaluation. The Instruction Queue is set to hold up to 1,024 elements.

Table 6 shows the power and area overheads of RAFT. Whole System refers to the hardware resource cost of the whole system with a Rocket core and peripherals. We observe an extra usage of 34.04% slice LUTs and 93.17% slice Registers in the whole system. In addition, it incurs 2.31% power consumption. The hardware overhead of RAFT is reasonable. As mentioned before, our new storage mechanism optimizes the structure of tag storage and moves tags from shadow memory to the coprocessor. Although the hardware resource is limited, it is sufficient to track the information flow of embedded applications whose size is small. For instance, 1 KiB TSF (Base-FP) and 1 KiB TSF (Base-GP) support tracking programs with 8 KiB data segments and 4 KiB stack maximum size. In fact, when tracking the same program, the scheme using shadow memory requires a much larger storage space compared to our design. Similar work [10] that places tags in the coprocessor introduces higher hardware resource overhead (≈ 2× slice LUTs and ≈ 0.34× slice Registers).

[1] The experiment result of Raksha v2 is from their paper.

Table 6: Hardware resource cost of RAFT.

	Whole System		Power
	Slice LUTs	Slice Registers	
Without RAFT	58,442	29,445	3.46 W
With RAFT	78,355	56,879	3.54 W
%	+ 34.07%	+ 93.17%	+ 2.31%

7.6 RQ5: Security Features of RAFT

7.6.1 Vulnerable Sample Program. We use a vulnerable sample program to demonstrate how our framework works to protect programs at runtime and discuss the security provided by RAFT. Listing 2 shows the code of the simplified program. The variable `private_data` is sensitive data (line 8) that should not be tampered with or leaked. The program accepts a file input and has a buffer overflow vulnerability (lines 9-15), failing to check the bounds of the fixed-size array `buffer`. The sensitive data is propagated to the variable `temp` (line 16). We construct three exploits to illustrate the functionality of RAFT, and all attacks are successfully detected by our framework.

```
1  struct SAMPLE{
2    char buffer[20];
3    int private_data;
4  };
5  int test()
6  {
7    struct SAMPLE s;
8    s.private_data = 5; // source ①
9    signed char input_char;
10   int i = 0;
11   while ((input_char = fgetc(fp)) != EOF) // source ②
12   {
13     s.buffer[i] = input_char;
14     i++;
15   }
16   int temp = s.private_data;
17   process(temp); // sink ③
18   return 0; // sink ④
19 }
```

Listing 2: Example of a vulnerable sample program.

Our first exploit, a non-control-data attack, corrupts critical data `private_data` by using a buffer overflow vulnerability. We tag the input `input_char` as a source at ② and perform checks when the critical data is used at ③. Since the tag is propagated to the variable `temp`, RAFT triggers a coprocessor interrupt to notice the main core when tainted data is used in the program. Our second attack overflows the array `buffer` and overwrites the return address stored on the stack. Subsequently, it redirects the program's control flow to the attacker-chosen location. Similarly, we tag the input `input_char` as a source at ② and check the tag of the return address register `ra` at ④. As the return address is tainted, we defend this attack when function `test()` returns. Our third exploit is pure information leakage, leading to leakage of the private data `private_data`. The function `process()` is undermined to output the private data. We tag the variable `private_data` as a source at ① and check the output function. As a result, we detect information leakage.

Control-flow Attacks. Control-flow hijacking is an attack technique that compromises the program's control flow integrity. Conventional control-flow attacks depend on injecting shellcode into writable memory and redirecting the control flow to execute it. Such code injection attacks have been well prevented by the widespread deployment of measures like Data Execution Prevention (DEP). However, code reuse attacks, such as Return-oriented Programming (ROP) [7] and Jump-oriented Programming (JOP) [9], are still quite prevalent. Attackers construct gadgets ending with `return` or `jump` instructions from existing code without code injection and redirect the control flow to execute in an attacker-controlled order. Raft defends against control-flow attacks by tagging external input as sources and checking all control-flow transfer instructions. Once the jump target address is tainted data, the attacks will be detected by Raft.

Non-control Data Attacks. Non-control-data attacks [15] influence program behavior without breaking the program's control-flow integrity. Attackers corrupt critical data (e.g., variables used for decision-making), which can lead to escalating privileges. Alternatively, pure non-control-data attacks manipulate the data pointer to output a private key, leading to sensitive data leakage. Such attacks are beyond the scope of defense techniques that ensure control-flow integrity and are difficult to defend against. Raft can be configured to tag private data as sources. Thus, information leakage will be detected when tagged private data are used by output functions. Similarly, Raft detects data corruption by performing checks when critical data is used.

7.6.2 Real-World Application. We apply Raft to protect an open-source real-world embedded application OpenSyringePump [59]. A syringe pump is a medical device that controls the quantity of fluid to dispense or withdraw at regular time intervals. It is widely used to inject medicines into patients. Hence, a syringe pump must highly assure correct operations and has strict time requirements. The device consists of a stepper motor, a fluid-filled syringe, and a microcontroller. The control system accepts commands from a keypad and a serial terminal and moves the stepper motor. OpenSyringePump is an open-source implementation of a syringe pump, which has already been used in previous works [2, 45, 52] to evaluate embedded system security. Since the original application is written in Arduino Script, we use a C version [2] adopted by [2] and port it on RISC-V.

We slightly modify the code and construct a non-control data attack by exploiting a buffer overflow vulnerability. The program receives user input from the serial terminal and stores it in a buffer `serialStr` without checking the bounds. When a key is pressed, the program receives an analog value and iterates through the predefined static key-map array to recognize the pressed key. We overflow the buffer and corrupt the static key-map array `adc_key_val` used for processing input from the keypad. This attack causes the program to perform actions when no physical key is pressed (e.g., the right key triggers the syringe to inject liquid). We instrument the program and configure the critical data `adc_key_val` as sources. Raft successfully detects this attack by performing checks when the static key-map array is used in the function that converts ADC value to key number.

[2]https://github.com/control-flow-attestation/c-flat/tree/master/samples/syringe

Table 7: Summary of tested software vulnerabilities.

ID	CVE ID	Program	Vulnerability	Detection
1	CVE-2009-4496	Boa	Information Leakage	✓
2	CVE-2014-8503	Size	Buffer Overflow	✓
3	CVE-2016-3186	Gif2tiff	Buffer Overflow	✓
4	CVE-2018-17100	Ppm2tiff	Integer Overflow	✓
5	CVE-2010-0001	Gzip	Integer Underflow	✓

7.6.3 CVEs. We further test Raft with 5 programs whose vulnerabilities are listed in Table 7, which covers common software exploits concerning spatial safety. Among them, the buffer overflow vulnerabilities in `size` and `gif2tiff` use file input data as parameters of potentially unsafe functions, which causes a denial of service. `ppm2tiff` uses integer overflowed value in memory allocation functions, leading to insufficient memory allocation. Integer underflow in `gzip` leads to an array index error. `boa` writes data to a log file without sanitizing non-printable characters. We manually mark the locations of sources and sinks according to publicly available vulnerability reports and successfully detect these CVEs.

8 DISCUSSIONS AND LIMITATIONS

8.1 Limitations

Raft reduces the demand for tag storage by applying a coarser granularity for the heap. We record the start address of the allocated heap memory without the size of the allocation. In such a manner, we effectively reduce the tag storage size and further substantially improve performance. Accordingly, it is difficult to determine whether the whole heap variable is fully or partially tainted. Thus, Raft's design may have overtaint issues for heap variables. However, there are no undertaint issues on the taint-propagation path containing heap objects. The usage of the heap will not cause the system to not work. Note that undertaint has a greater impact than overtaint, especially for security-critical applications. Additionally, Raft has limited capabilities to prevent attacks exploiting vulnerabilities on the heap (e.g., heap overflow) due to lacking information about the allocation size. Our work provides a fine-grained, robust guarantee for stack and global variables and coarse-grained, weak protection for heap variables. Therefore, Raft is more applicable for protecting security-critical applications without complicated heap usage. Moreover, Raft requires marking whether the value stored in a register is a heap address. We instrument standard memory allocation functions to specify heap pointers, but programmers are still required to deal with custom memory allocators.

In addition, the size of programs that Raft is capable of tracking is contingent on the available storage resource in the coprocessor. Large programs may not be effectively protected due to the limited size of TSF. Recursive and nested function calls may also cause tag storage exhaustion due to stack explosion. As mentioned in

Section 7.3, we also implement a hardware-assisted DIFT framework on RISC-V using shadow memory, which can handle heap issues and protect larger programs but be relatively slow. One solution is to apply different tag-storage mechanisms in different scenarios and switch in accordance with tracked program size. We plan to integrate alternative tag-storage mechanisms into Raft and expose functions for the automatic switch to programmers in the future.

Furthermore, a limitation of standard DIFT approaches is implicit flows because they do not propagate taints along control dependencies. Tainted data values influence control flow so that the control flow difference influences other data in turn [30]. Like other existing hardware-assisted DIFT frameworks [10, 31], we do not handle implicit flows due to the complexity. Ignoring implicit flows leads to under-tainting, while indiscriminately counting all implicit flows leads to over-tainting [30]. Raft has the feature to disable taint tracking for specific blocks and use manual taints. In the future, we plan to identify implicit flows in common cases (e.g., if and switch statements) by additional instrumentation to obtain instruction context.

8.2 Towards Multicore and Rich OS Systems

Similar to [10, 21, 31], our current work focuses on a single processor without multithreading. For multithreading, thread ID and multiple TSF structures are needed, with a separate TSF structure corresponding to each thread. Every time a thread switch occurs and the coprocessor is activated, the main core writes the current thread ID to a dedicated register in the coprocessor. For multi-core processors, each core needs a dedicated DIFT coprocessor. There are consistency issues among multiple DIFT-supported coprocessors. Extra coherency protocols are required for the control unit and tag storage to ensure the coherency of tags among multiple processors, which is quite complex to address. For Out-of-Order (OOO) cores, instructions in program order can be obtained from the reorder buffer of OOO cores [31] and then committed to the coprocessor. Therefore, it will only require slight modifications for the current implementation to support OOO execution.

9 CONCLUSION

In this paper, we present Raft, a hardware-assisted coprocessor-based DIFT framework that provides runtime protection for embedded applications without delay to the programs. We diminish the amount of information required for DIFT and apply a novel storage mechanism to reduce the size of tag storage while providing fine-grained protection. We deploy Raft on the RISC-V Rocket emulator and FPGA development board to evaluate its effectiveness and efficiency. Compared to previous approaches (over 20% runtime overhead), Raft has significantly improved performance, and its overhead is 0.13% on SPEC CPU benchmarks.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments. This work is partly supported by the National Natural Science Foundation of China under Grant No.62002151 and No. 62102175, Shenzhen Science and Technology Program under Grant No. SGDX20201103095408029 and No. ZDSYS20210623092007023.

REFERENCES

[1] Accessed: 2021. Riscv-coremark. Available: https://github.com/riscv-boom/riscv-coremark.

[2] Tigist Abera, N Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*. 743–754.

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

[4] Erik Bosman, Asia Slowinska, and Herbert Bos. 2011. Minemu: The world's fastest taint tracker. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 1–20.

[5] Amiangshu Bosu, Fang Liu, Danfeng Yao, and Gang Wang. 2017. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIACCS'17)*. 71–85.

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

[7] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*. 27–38.

[8] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*. 317–329.

[9] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. 559–572.

[10] Kejun Chen, Orlando Arias, Qingxu Deng, Daniela Oliveira, Xiaolong Guo, and Yier Jin. 2022. FineDIFT: Fine-Grained Dynamic Information Flow Tracking for Data-Flow Integrity Using Coprocessor. *IEEE Transactions on Information Forensics and Security (TIFS'22)* 17 (2022), 559–573.

[11] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. 1687–1700.

[12] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B Gibbons, Todd C Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. 2008. Flexible hardware acceleration for instruction-grain program monitoring. *ACM SIGARCH Computer Architecture News* 36, 3 (2008), 377–388.

[13] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. 2021. SelectiveTaint: Efficient Data Flow Tracking With Static Binary Rewriting. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security'21)*. 1665–1682.

[14] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K Iyer. 2005. Defeating memory corruption attacks via pointer taintedness detection. In *International Conference on Dependable Systems and Networks (DSN'05)*. IEEE, 378–387.

[15] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats.. In *USENIX Security Symposium (USENIX Security'05)*, Vol. 5. 146.

[16] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. 2018. DTaint: detecting the taint-style vulnerability in embedded device firmware. In *International Conference on Dependable Systems and Networks (DSN'18)*. IEEE, 430–441.

[17] Jedidiah R Crandall and Frederic T Chong. 2004. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO'04)*. IEEE, 221–232.

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

[19] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. 2019. DECAF++: Elastic whole-system dynamic taint analysis. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID'19)*. 31–45.

[20] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. 2020. PHMon: A Programmable Hardware Monitor and Its Security Use Cases. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*. 807–824.

[21] Daniel Y Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G Edward Suh. 2010. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. IEEE, 137–148.

[22] Daniel Y Deng and G Edward Suh. 2012. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *International Conference on Dependable Systems and Networks (DSN'12)*. IEEE, 1–12.

[23] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-Based Detection of Android Malware through Static Analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. 576–587.

[24] John Galea and Daniel Kroening. 2020. 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 (ASIACCS'20)*. 622–636.

[25] William GJ Halfond, Alessandro Orso, and Panagiotis Manolios. 2006. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'06)*. 175–185.

[26] JL Henning. 2006. SPEC CPU2006 Benchmark descriptions. ACM SIGARCH Comput Archit News 34 (4): 1–17.

[27] Ingoo Heo, Minsu Kim, Yongje Lee, Changho Choi, Jinyong Lee, Brent Byunghoon Kang, and Yunheung Paek. 2015. Implementing an application-specific instruction-set processor for system-level dynamic program analysis engines. *ACM Transactions on Design Automation of Electronic Systems (TODAES'15)* 20, 4 (2015), 1–32.

[28] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*. 977–992.

[29] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 6–pp.

[30] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. Dta++: dynamic taint analysis with targeted control-flow propagation.. In *Network and Distributed System Security (NDSS'11)*.

[31] Hari Kannan, Michael Dalton, and Christos Kozyrakis. 2009. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *2009 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'09)*. IEEE, 105–114.

[32] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. 2012. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'12)*. 121–132.

[33] Jinyong Lee, Ingoo Heo, Yongje Lee, and Yunheung Paek. 2016. Efficient security monitoring with the core debug interface in an embedded processor. *ACM Transactions on Design Automation of Electronic Systems (TODAES'16)* 22, 1 (2016), 1–29.

[34] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition.. In *Network and Distributed System Security (NDSS'13)*, Vol. 2. 4.

[35] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. 2008. Automatic protocol format reverse engineering through context-aware monitored execution.. In *Network and Distributed System Security (NDSS'08)*, Vol. 8. Citeseer, 1–15.

[36] Yin Liu and Ana Milanova. 2010. Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE, 146–155.

[37] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: A manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

[38] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*. 229–240.

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

[40] Uwe F Mayer. 2003. Linux/unix nbench.

[41] Ibéria Medeiros, Nuno Neves, and Miguel Correia. 2015. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability* 65, 1 (2015), 54–69.

[42] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. 2015. Taintpipe: Pipelined symbolic taint analysis. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*. 65–80.

[43] Vijay Nagarajan, Ho-Seop Kim, Youfeng Wu, and Rajiv Gupta. 2008. Dynamic information flow tracking on multicores. In *Proceedings of the Workshop on Interaction Between Compilers and Computer Architectures*.

[44] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software.. In *Network and Distributed System Security (NDSS'05)*, Vol. 5. Citeseer, 3–4.

[45] Christian Niesler, Sebastian Surminski, and Lucas Davi. 2021. Hera: Hotpatching of embedded real-time applications. In *Proceedings of the 28th Network and Distributed System Security Symposium (NDSS'21)*.

[46] Meltem Ozsoy, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Tameesh Suri. 2011. SIFT: A low-overhead dynamic information flow tracking architecture for smt processors. In *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF'11)*. 1–11.

[47] Christian Palmiero, Giuseppe Di Guglielmo, Luciano Lavagno, and Luca P Carloni. 2018. 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'18)*. IEEE, 1–7.

[48] Joël Porquet and Simha Sethumadhavan. 2013. WHISK: An uncore architecture for dynamic information flow tracking in heterogeneous embedded SoCs. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'13)*. IEEE, 1–9.

[49] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. 2021. SpecTaint: Speculative taint analysis for discovering spectre gadgets. In *Network and Distributed System Security (NDSS'21)*.

[50] Olatunji Ruwase, Phillip B Gibbons, Todd C Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. 2008. Parallelizing dynamic information flow tracking. In *Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures (SPAA'08)*. 35–45.

[51] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. 2004. Secure program execution via dynamic information flow tracking. *ACM Sigplan Notices* 39, 11 (2004), 85–96.

[52] Sebastian Surminski, Christian Niesler, Ferdinand Brasser, Lucas Davi, and Ahmad-Reza Sadeghi. 2021. RealSWATT: Remote Software-based Attestation for Embedded Devices under Realtime Constraints. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*. 2890–2905.

[53] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. 2008. Flexitaint: A programmable accelerator for dynamic taint propagation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture (HPCA'08)*. IEEE, 173–184.

[54] John Viega and Hugh Thompson. 2012. The state of embedded-device security (spoiler alert: It's bad). *IEEE Security & Privacy* 10, 5 (2012), 68–70.

[55] Muhammad Abdul Wahab. 2018. *Hardware support for the security analysis of embedded softwares: applications on information flow control and malware analysis*. Ph. D. Dissertation. CentraleSupélec.

[56] Muhammad Abdul Wahab, Pascal Cotret, Mounir Nasr Allah, Guillaume Hiet, Arnab Kumar Biswas, Vianney Lapotre, and Guy Gogniat. 2018. A small and adaptive coprocessor for information flow tracking in ARM SoCs. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig'18)*. IEEE, 1–8.

[57] Muhammad A Wahab, Pascal Cotret, Mounir N Allah, Guillaume Hiet, Vianney Lapotre, and Guy Gogniat. 2017. ARMHEx: A hardware extension for DIFT on ARM-based SoCs. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL'17)*. IEEE, 1–7.

[58] Editors Andrew Waterman and Krste Asanović. December 2019. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation.

[59] Bas Wijnen, Emily J Hunt, Gerald C Anzalone, and Joshua M Pearce. 2014. Open-source syringe pump library. *PloS one* 9, 9 (2014), e107216.

[60] Xueling Zhang, Xiaoyin Wang, Rocky Slavin, and Jianwei Niu. 2021. ConDySTA: Context-Aware Dynamic Supplement to Static Taint Analysis. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (S&P'21)*. IEEE, 796–812.

[61] David Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. 2011. TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking. *ACM SIGOPS Operating Systems Review* 45, 1 (2011), 142–154.