FineDIFT: Fine-Grained Dynamic Information Flow Tracking for Data-Flow Integrity Using Coprocessor

Kejun Chen, Orlando Arias, Qingxu Deng, *Member, IEEE*, Daniela Oliveira, Xiaolong Guo, *Member, IEEE*, and Yier Jin, *Senior Member, IEEE*

***Abstract*— Dynamic Information Flow Tracking (DIFT) is a technique that facilitates run-time data-flow analysis on a running process, allowing a system to overcome the limitations of finding data dependencies statically at compilation time. DIFT serves as the backbone for applications including data-flow integrity (DFI). However, previous uses of DIFT towards DFI often have large overhead in terms of hardware, software or both, and often cannot provide fine-granularity tracking for software object, such as variables. To address these limitations, we present FineDIFT as a DFI framework which utilizes DIFT to generate a live data-flow graph of a running process and perform hardware-based assisted analysis at fine-granularity, thus being able to enforce the application's Data-Flow Graph (DFG). We provide a sample implementation on a RISC-V core with a performance overhead of 5.03% for BEEBS benchmarks and hardware overhead of 6% LUTs and 8% Flip-Flops in the FPGA implementation, if excluding the Content-Addressable Memory (CAM) like structure used for metadata storage. With CAM-like structure being synthesized using FPGA logic, the total hardware overhead is ≈2× LUTs and 33% Flip-Flops compared to the original RISC-V core. We also use the real-world application and customized vulnerable application to demonstrate the effectiveness of the proposed framework in protecting computing systems.**

***Index Terms*— RISC-V, information flow tracking, data-flow integrity.**

I. Introduction

DATA-FLOW integrity (DFI) is a general purpose defense to enforce that data usage in an application follows the intended data-flow graph (DFG) [1]. This ensures that data is not misused as the program executes. For example, a memory vulnerability can be exploited to alter a code pointer or a critical variable which would affect the execution of a process. By enforcing the DFG, DFI presents such data corruption.

A DFI policy can be static, where the DFG of the application is obtained by performing analysis on the application [2]–[4]. These approaches instrument software to aid with the tracking of data-flow by inserting static checks as the code executes. Unfortunately, static analysis has limitations that result from the approach ultimately having to solve the decision problem resulting from pointer analysis [5]. As a result, dynamic DFI policies aim to construct an application's DFG dynamically. For this purpose, Dynamic Information Flow Tracking (DIFT) provides a promising avenue for building a DFG at runtime.

The software instrumentation required by software-based DIFT approaches introduce large performance overheads [2], [3]. Meanwhile, hardware-assisted designs have been introduced to alleviate the performance penalty. However, previous hardware-based DIFT designs suffer from some limitations such as expensive modifications to the processor's pipeline [6]–[10], utilization of inefficient memory systems [6], [8], [10], or implementing inaccurate DFI policies [7], [10], [11]. For example, the single bit tag used in [4], [6], [7], [12] results in ambiguity at runtime. Different regions of memory cannot be differentiated from each other creating, in effect, a taint map of the application. An adversary is then capable of using a memory vulnerability to change data in an already tainted area of memory while the policy is unable to detect it.

To address these shortcomings, we present FineDIFT, a hardware-based DFI framework which DIFT to generate a live data-flow graph. We implement FineDIFT in a coprocessor, avoiding expensive modifications to existing architectures. FineDIFT is intrinsically designed to individually identify memory regions and enforce per-region rules on how they are utilized. To handle the amounts of metadata required for identification and tracking of memory regions, FineDIFT uses a novel storage mechanism which reduces the size of the storage element required to keep live data-flow information by keeping range information instead of tracking each individual address. Further, our storage element is also capable of fast lookups and checks of metadata at runtime, eliminating the need to stall the main processor while checks take place. We implement and evaluate FineDIFT as a coprocessor for a RISC-V core, demonstrating its functionality, performance, and security provisions in a Digilent Arty-7 FPGA board.

Manuscript received July 29, 2021; revised November 7, 2021 and December 26, 2021; accepted January 7, 2022. Date of publication January 19, 2022; date of current version February 8, 2022. This work was supported in part by the National Science Foundation under Grant CNS-1801599 and Grant CCF-2019310, in part by the National Natural Science Foundation of China under Grant 62072085 and Grant U1908212, and in part by the Liaoning Revitalization Talent Program under Grant XLYC1902017. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Georg Sigl. *(Corresponding author: Yier Jin.)*

Kejun Chen and Qingxu Deng are with the Department of Computer Science and the Engineering School, Northeastern University, Shenyang 110169, China (e-mail: kejunchen@stumail.neu.edu.cn; dengqx@mail.neu.edu.cn).

Orlando Arias, Daniela Oliveira, and Yier Jin are with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: orlandoa@ufl.edu; daniela@ece.ufl.edu; yier.jin@ece.ufl.edu).

Xiaolong Guo is with the Department of Electrical and Computer Engineering, Kansas State University, Manhattan, KS 66506 USA (e-mail: guoxiaolong@ksu.edu).

Digital Object Identifier 10.1109/TIFS.2022.3144868

The main contributions of the paper are listed as follows:

- A thorough examination of previous DFI approaches and a discussion of their limitations.
- The introduction of FineDIFT, a fine-grained DFI framework to detect security violation at runtime. This framework incorporates a coprocessor to support all proposed DIFT operations. The coprocessor also provides flexible tag management for programmers to protect data usage.
- We propose a novel range-based metadata allocation technique to reduce storage demands, and per-region identifiers as well as rules to facilitate dynamic analysis.
- Demonstration and evaluation of FineDIFT on a RISC-V based system on chip (SoC) with the hardware implementation on an FPGA board. We also test our approach on a real-world application showing the effectiveness and the low performance overhead of the proposed scheme.

The rest of the paper is organized as follows. We provide background information in Section II. Section III presents threat models and assumption of our work. Section IV discusses related work. Section V discusses the design of the coprocessor. Section VI gives our implementation of the framework. Section VII and Section VIII provide an evaluation of the security features, performance, and cost. Finally, Section IX gives the limitation of our work. Section X concludes the paper.

II. Background

A. Dynamic Flow Information Tracking

Dynamic Flow Information Tracking (DIFT) [6] is a technique for tracking information flow of a process or system, allowing for state information to be restored and runtime analysis to take place. DIFT can be performed online (as the program executes) or offline, where information flow metadata is recorded to be later analyzed.

DIFT serves as the backbone for various mechanisms such as fuzzing, dynamic taint analysis, and data-flow integrity. In fuzzing, randomly crafted inputs are given to a program and execution is tracked to build a coverage map. Inputs are then mutated and changes in execution are then observed to allow bug discovery. Dynamic taint analysis tracks changes in memory due to store operations constructing a map of accessed addresses as well as their relations. Lastly, data-flow integrity is a general purpose defense which can be employed to prevent the effects of memory vulnerabilities in software. Those memory vulnerabilities can be exploited towards control-data and non-control-data style attacks.

B. Data-Flow Integrity

Data-flow integrity (DFI) is a defense mechanism to enforce that data usage in an application follows the intended data-flow graph (DFG) [1]. A DFG can be defined through reaching definitions analysis [13], where an instruction that writes to a memory location *defines* the value and an instruction that reads the value *uses* it. When performing reaching definition analysis, we see that there is an intrinsic relationship between the control-flow graph of an application as well as its DFG. For example, the value of a variable may indicate which path on a conditional jump is taken, a use, for the taken path to determine the initial value, or definition, of a different variable.

A DFG can be computed statically or dynamically. However, approaches for static computation of the DFG are unable to solve the decision problem which arises from pointer aliasing [5]. This often results in an overestimated DFG. Dynamic techniques applied to the computation of the DFG can result in improving its accuracy. However, sourcing multiple inputs to perform analysis is limited by the undecidable nature of the halting problem.

Incompleteness of the DFG of an application, as well as deficiencies in the enforcing policy result in weaknesses in defense solutions leveraging DFI. We purpose to define a framework that utilizes DIFT to dynamically construct the data-flow graph of a running program.

C. Application Binary Interface

The Application Binary Interface (ABI) is a set of agreed-upon conventions on how software allocates CPU and memory resources for program usage. Among the rules, we find the *calling conventions* of a particular platform, how stack frames are used, and how registers are used between functions.

In an ABI, registers can be defined as *caller saved*, *callee saved*, or *preserved*. The former category are saved by functions performing a call if they are later to be used by the function. Callee saved registers are saved by the function being called if they are to be used. Lastly registers that are preserved across function are saved by a function and their value is restored before the function exits, either through a return or calling another function, if it is used.

Saved registers are saved in a function's *stack frame*. The minimum size and alignment constraints of the stack frame are defined by ABI rules. A function creates its stack frame during the execution of its prologue. Stack frames are deallocated by the function's epilogue, as it is about to return.

III. Threat Model and Assumptions

We aim to protect embedded devices running freestanding programs. Application software in these devices manage all on-device resources without assistance of operating system (OS). Although our system is currently designed and tested with freestanding programs in mind, it can be extended to support rich OS-based systems. We consider an adversary that has two goals. First, the attacker may wish to acquire sensitive data from target program by launching a series of software-based attacks. Second, the attacker may wish to corrupt the internal data and state of the program by changing the program's normal execution. The attackers may achieve their goals by utilizing memory vulnerabilities that allow for arbitrary reads and writes to the memory map of the program. These vulnerabilities may be exposed through the input/output system of the victim process which the attacker has direct access to, either locally or remotely.

We further assume that software running on the device is immutable, and that it is not inherently malicious. Also, we assume that all hardware components are trustworthy and bug-free. We do not protect against invasive attacks, such as those using a JTAG probe to extract memory contents, nor do we protect against physical side-channel attacks.

IV. Related Works

Various software-based DIFT applications have been proposed [3], [4], [14]–[16]. Software-based methods often introduce large performance overhead through statically instrumenting application's source code and compiled binary or transforming program code to identify potential data source and definitions. Further, static analysis cannot capture the runtime behavior effectively. Hardware-based methods [9], [10], [17], [18] improve not only the performance, but also the reliability. These mechanisms extend CPU, Caches, and memory system providing an avenue for metadata to flow alongside data being used, and for rule checks to be performed in parallel to data usage. Table I summarizes and compares different DIFT designs.

Xu *et al.* follow on the same premise by providing a taint-enhancement policy enforcement mechanism [19]. Program code is transformed to identify potential data sources or definitions which are then tracked under different security constructs. Policies are checked at the time of usage of these values. The authors implement a *tagmap* by dynamically allocating regions of memory and initializing metadata as it is used. Security policies are defined in a similar fashion to that of TaintCheck [20], but applications can run natively.

A. Tag Design

Some approaches adopt single bit tag [4], [6], [7], [12]. Single bit tag based design will not introduce high performance overhead on tag storage. But it may not be helpful for complex attack scenarios because it cannot differentiate multiple security property at the same time. In order to support flexible tag based security policy, the authors in [23], [24] support variable tag sizes. The user and programmer can define different security policy to detect and prevent low-level memory corruption attacks and high-level semantic attacks. The design of tag needs to strike a balance between flexibility and scalability. The fine-granularity tag will incur high storage overhead, such as byte-level tags. On the other hand, the coarse-granularity tag cannot provide enough information for security analysis. Also, the design needs to consider the waste of tag storage. Longer tag sizes will incur more waste on storage space when every memory word or byte has a tag. Offload based design needs a communication channel, e.g., shared memory and hardware based buffer, to store and transmit the execution information. At the same time, the synchronization between processor cores and dedicated core may introduce extra performance overhead.

B. Previous DIFT Design

We examine previous DIFT design and analyze the existing works according their implementation type (see Table I). We compare different DIFT designs from the tag granularity, design types, performance overhead and design goals. We also analyze the security policies used in these works. The security policies include control-flow integrity (CFI), data-flow integrity (DFI) and pointer integrity (PI). The CFI mainly focuses on ensuring the program's control flow is not redirected by malware attacks. The DFI is used to prevent unauthorized access to data and malicious corruption, modification and disclosure on data. PI guarantees all code pointers which are not maliciously modified by attackers.

Figure 1 shows an overview of common DIFT methods. For hardware based methods, the information flow tracking logic is implemented in processor core, coprocessor and multiple processors. In processor core, the general-purpose registers and pipeline stages need to be extended to support metadata information processing logic. Also, the memory system needs to be modified to support metadata tracking. In addition, the information tracking logic and metadata storage can be implemented as dedicated hardware module such as a coprocessor attached to processor to extract runtime information including instructions related information and machine states. On multi-processors platform, extra processor can be used to track the information flow on target program. Extra information communication channels between two processors is needed, including share memory and special hardware information queues. For software based methods, target program is pre-processed to add extra information checking operations. The original source code is adjusted to support information flow tracking using compiler-aided methods, e.g., inserting extra instructions or adding metadata information. Differently, in binary instrumentation based methods, executable binary will be modified using specific instrumentation tools.

1) Software Based DIFT Framework: Various software-based DIFT applications have been proposed [2]–[4], [11], [12], [14], [15], [19], [20]. Newsome and Song in TaintCheck [20] ensure that data in a trusted source is not used if it was illegitimately overwritten by an untrusted source. For this purposes, the authors develop a framework on top of Valgrind [32] and DynamicRIO [33] providing a way to examine data-flow without having to statically instrument an application's source code or compiled binary. Based on TaintCheck [20], Cheng *et al.* in [11] use the dynamic binary instrumentation to reduce the performance overhead of the taint analysis. Xu *et al.* follow on the same premise by providing a taint-enhancement policy enforcement mechanism [19]. Program code is transformed to identify potential data sources or definitions which are then tracked under different security constructs. Policies are checked at the time of usage of these values. Chen *et al.* in [12] treat the tainted state as speculative state and use the existing architectural support for speculation execution to track the tainted state. Vachharajani *et al.* in [2] provide a software based DIFT framework from user's perspective. Furthermore, the user empowered the ability to define the security policy instead of relying on the programmer. Kemerlis *et al.* in [21] propose a practical dynamic data flow tracking tool which can be used for commodity software. This implementation is also suitable for shared library. Also, this user can use the provided API to monitor the data of interest. The authors in [3], [4], [14], [15] implement DIFT framework on smart phones and IoT system to support taint analysis on

TABLE I

CLASSIFICATION AND COMPARISON ON INFORMATION FLOW TRACKING TECHNIQUES. TAG IN OUR WORK REPRESENTS THE AUXILIARY INFORMATION AND SECURITY ATTRIBUTES OF CORRESPONDING DATA. TAG GRANULARITY MEANS THE SIZE OF TAG ATTACHED TO DATA. SW, SW AND H&W REPRESENT THE SOFTWARE BASED, HARDWARE BASED AND CO-DESIGN BASED DIFT FRAMEWORK RESPECTIVELY. ALSO, WE DISCUSS THE PERFORMANCE OVERHEAD AND HARDWARE OVERHEAD CAUSED BY THESE DIFT WORKS. CFI, DFI, PI, AND NI ARE THE ABBREVIATION OF CONTROL-FLOW INTEGRITY, DATA-FLOW INTEGRITY, POINTER INTEGRITY AND NEW INFRASTRUCTURE, RESPECTIVELY. THE NEW INFRASTRUCTURE MEANS THE NEW MICROARCHITECTURE OR IMPLEMENTATION DIFT MECHANISMS. WE REPRESENT SYSTEMS THAT OFFER NO PROTECTION IN A CATEGORY WITH ○, SYSTEMS THAT OFFER WEAK PROTECTION IN A CATEGORY WITH ◐, AND SYSTEMS THAT OFFER FULL PROTECTION ON A CATEGORY WITH ●

Works	Tag Granularity	Design Type	Performance Overhead	Hardware Overhead	CFI	DFI	PI	NI
[2]	Unspecified	SW	0%-100%	-	◐	●	○	✗
[20]	1/Byte	SW	250%-3700%	-	◐	●	◐	✗
[19]	1/Byte	SW	61%-106%	-	◐	●	●	✗
[11]	1/Byte	SW	650%	-	◐	●	○	✗
[12]	1/Word or Byte	SW	327%-381%	-	◐	●	○	✗
[21]	1,8/Variable	SW	224%-700%	-	●	●	○	✗
[3]	32/Variable	SW	25.9%	-	○	●	○	✗
[15]	32/Variable	SW	14%	-	○	●	○	✗
[4]	1/Word	SW	15%	-	○	●	○	✗
[14]	message based	SW	Unspecified	-	○	●	○	✗
[6]	1/Byte	HW	1.1%	1.4%(storage)	◐	●	○	✗
[10]	4/Word	HW	Negligible	12.5%(storage), 0.82%(LUT)	◐	●	○	✓
[22]	pointer-sized	HW	Unspecified	10%-40%	●	●	○	✓
[17]	pointer-sized	HW	110%(Storage)	10%	●	●	○	✓
[18]	4/Word	HW	0.79%	8%(BRAMs), 4.85%(LUTs)	◐	●	●	✓
[23]	Variable/Word	HW	7%	32.5%(Area)	○	○	○	✓
[24]	Variable/Word	HW	7.25%	3%(Storage)	○	○	○	✓
[25]	1/Word	HW	1.6%	60%(BRAMs), 28.36%(LUTs)	●	●	○	✗
[26]	Unspecified	HW	Unspecified	11.4%(Storage), 1.9%(LUTs)	◐	●	○	✓
[27]	1/Word	HW	Unspecified	Unspecified	◐	●	○	✓
[7]	1/Word	S&H	3.125%	Unspecified	◐	●	○	✗
[8]	4/Word	S&H	234%	12.5%(Storage)	◐	●	○	✓
[28]	2/Word	S&H	48%	Unspecified	●	●	○	✓
[29]	Unspecified	S&H	26%	4.5%(Area)	○	○	○	✓
[9]	2/Word	S&H	1%-3.7%	Unspecified	○	○	○	✓
[30]	1/Variable	S&H	Unspecified	4.62%(Softcore), 5.2%(IP)	○	●	○	✓
[31]	Variable/Word	S&H	480%	1%(Area), 2%(Power)	●	○	○	✓
This Work	Variable/Variable	S&H	5.03%	6%(LUTs)*, 8%(FFs)*, 42%(Area)**	◐	●	●	✓

* Hardware overhead in the absence of CAM-like structure.
** Area overdead when adopting unoptimized CAM-like structure in FPGA.

sensitive information usage. The sensitive data will be tracked to analyze whether the data is used in an illegal manner.

The main difference among existing DIFT software approaches lies in how tagged data is identified, and the rules that are followed during propagation. The software approaches provide flexibility and scalability for security policy definition. However, software based DIFT methods use the binary instrumentation and compiler aided tools to insert the tag related operation. As expected, performance overhead suffers in these approaches due to the required instrumentation or the wrappers that allow examination of the software to run.

2) Hardware Assisted DIFT Architecture: Hardware assisted DIFT solutions can be divided into three categories:

1) In-core based designs [6]–[10], [22], [27] where tagging and related operations are implemented within the processor core and memory system;
2) Off-core based designs [9], [18], [23], [25], [26], [30], [31] where DIFT operations are implemented in dedicated hardware unit outside the processor core;
3) Offload-based design [28], [29] where a dedicated processor core is responsible for all DIFT operations.

In-core based design needs modification or extension on processor pipeline to support tag tracking and propagating. However, this method is not feasible for modern commercial processor under industry-standard licensing norms, where licensees of a processor IP are unable to make changes to it. Off-core based design mainly focuses on attack detection. This type of system works well with the current IP licensing model, as long as the processor IP provides a mechanism to tap into the instruction pipeline or provide a way to extend the instruction set architecture (ISA). Offload based design is an alternative approach. The target application runs on one processor core, while DIFT related operations run on another processor core. One synchronization mechanism needed to support communication between target application and DIFT operation, e.g., shared memory, special hardware channel.

Suh *et al.* in [6] leverage architectural support to track the data from I/O inputs. Operating system will tag the data from inputs. Furthermore, the tag will be tracked in parallel with the data processing. Crandall and Chong in Minos [7] modify both operating system and out-of-order processor to prevent complex semantic attacks, e.g., format string attacks, buffer overflow and heap globbing. Dalton *et al.* in [8] extend the processor pipeline stages and modify the operating system to support full-system DIFT framework. This work first adopt global rule registers in security check enforcement. Similarly, Palmiero *et al.* in [10] adopt the DIFT design on RISC-V based

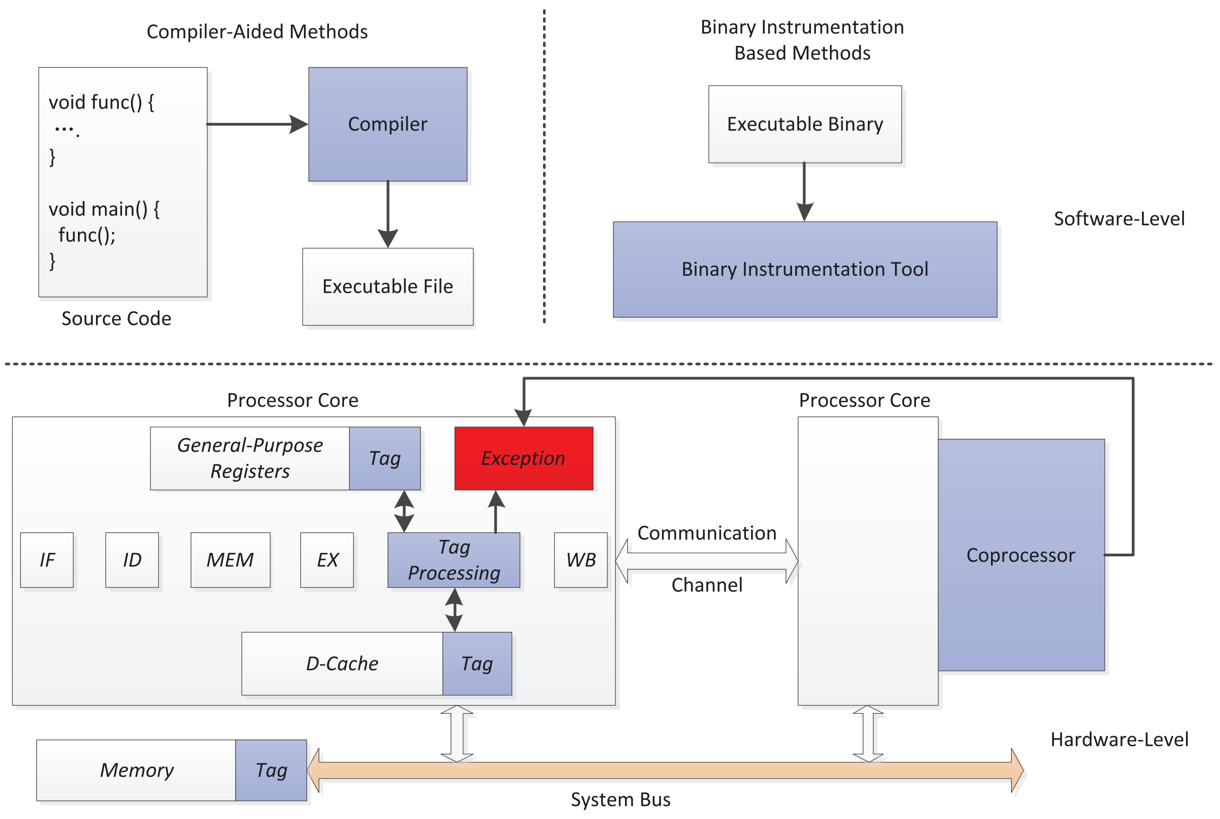


Fig. 1. Overview of different kinds of DIFT methods. Hardware based DIFT methods include in-core, off-core and offload based designs. Software based DIFT methods include compiler-aided designs and binary instrumentation based designs.

processor core. Different to in-core designs, Dhawan *et al.* in [17], [22] aim at providing general architecture for metadata processing. This architecture support unbounded metadata and fully software-defined policies. The authors in [28], [29] adopt offload based design which use one thread running on another processor core to enforce DIFT related check. Nagarajan *et al.* in [28] use a hardware FIFO to serve as communication channel between main thread and DIFT thread. Ozsoy *et al.* in [29] modify the processor to generate the DIFT operation related instructions after the corresponding instruction committed. Furthermore, the generated DIFT operation related instructions is running on another processor core. Siddiqui *et al.* in [27] propose a runtime protection framework which use Trusted Platform Module (TPM) to support secure boot and protect data from leakage. At runtime, the framework uses information flow tracking models to protect memory corruption and runtime attacks such as buffer overflows and format strings.

Venkataramani *et al.* in FlexiTaint [9] employ a packed array in a protected memory area within the process's address space. Regular data addresses in the process have a corresponding entry in the array. Taint analysis is done by a modified processor back-end which includes verification rules keeping the performance critical data-flow engine of the CPU largely unchanged. Similarly, works in [18], [23], [25], [26] extend the processor pipeline to extract the committed instructions and machine states. Coprocessor is introduced to serve as a runtime verifier to challenge the software state. Muhammad *et al.* attempt to further decoupling from the architecture [30], [31]. These works remove DIFT-related components from the processor's pipeline but use ARM CoreSight debug components to extract the essential signals and adopt static analysis to guide the DIFT operation.

In addition to the above three types of hardware based design, WHISK [24] provides a slightly different infrastructure for tag management on SoC. The tag mapping and fetching are processed through bus interconnect or bridge. This serves as an off-core design, but does not rely on a coprocessor to examine data-flow. Instead, the interconnect is responsible for differentiating between data accesses and instruction accesses.

In-core methods improve not only performance, but also reliability. These mechanisms extend CPU, caches, and memory system providing an avenue for metadata to flow alongside data being used, and for rule checks to be performed in parallel to data usage. However, efforts have been made to reduce the number of changes required in the architecture. The off-core designs receive information flow from the processor and maintains tag metadata information. Synchronization between the processor and co-processor occur at the time of system calls or custom instructions execution, reducing the latency requirements in the system as the processor no longer stalls while waiting for the coprocessor to finish multi-cycle operations such as accessing tag metadata from main memory.

C. Data-Flow Integrity

To provide a proper framework for DFI, it is necessary to uniquely identify data in memory, as well as associate a policy for the propagation and usage of that data. However, a static, compiler-based approach will lead to an overestimation. That is, the compiler is limited to the types of tracking it can perform during the static single assignment passes. As a result, it may lead to an explosion of states [34] or the incapability of solving the pointer aliasing problem for the general case [5].

Previous DFI approaches mainly utilize *dynamic taint analysis* to identify when data affects other data in memory [19], [20]. Employing this method results in data being classified as *tainted* or *untainted*, aliasing all different data types into two groups without regards of usage. Other approaches provided mechanisms to uniquely identify data groups through *tagging* and allow programmer-defined checks to take place at certain places during code execution [18]. We find these approaches to be limited. Dynamic taint analysis does not provide a reliable way to determine if tainted data is still valid. For example, a variable can be indirectly modified from a user-controlled input which results in it being flagged as tainted. Thereafter the same variable can be modified through a write-anywhere vulnerability, which would go undetected.

Storage of metadata is also of concern. Existing solutions often utilized a shadow memory or extended memory words to allocate memory metadata. We find this approach to be inefficient for storage. For example, to cover a 32 bit address space with a 1 bit tag, 512 MiB of RAM needs to be added into the system. Using wider tags requires larger amounts of memory, increasing the area overhead. Under this scheme, tagging and untagging large memory areas requires software to utilize constructs such as loops, or the additions of new instructions to the CPU to accelerate the process. This results in detrimental performance overhead, as workloads must be halted until the memory tagging operations are completed.

V. FineDIFT Design

We propose a DIFT-based hardware-based mechanism to accelerate the enforcement of DFI policies, i.e., a set of data-flow rules are enforced during the program execution. Our solution introduces minor changes to the processor's architecture while providing a flexible programming model. A software developer is able to utilize intrinsics and attributes defined by an instrumenting compiler to mark the variables or memory regions which will be tracked, and what policy to utilize for each of those regions. The compiler issues the necessary instructions to enable FineDIFT's hardware tracking mechanism. To make the approach generic to all architectures, we choose a coprocessor based solution which can be integrated into different architectures.

A. FineDIFT Overview

Figure 2 shows how the policy is enforced. We use unique tags to identify data and track its movement. An associated set of flags dictate the rules of the policy to be enforced. As the processor executes store operations, tag and flag information belonging to the source register as well as the store instructions related information including target address and store size are used by the coprocessor to create or update memory regions in the metadata memory. On loads, the coprocessor uses the address and load size to find the proper region in the metadata memory. The coprocessor uses flags and tag information from the region and the target register to determine if the load is allowed. Rules defined in the flags determine whether an exception should be triggered or not.

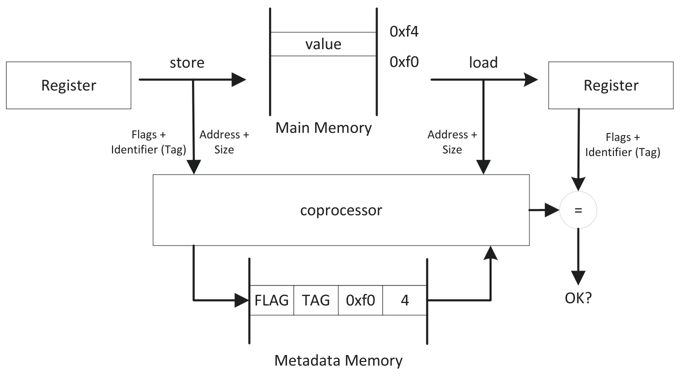


Fig. 2. Basic system operation.

Figure 3 shows the architecture of our design where a coprocessor is used to track the data flow at runtime. The coprocessor has its own instruction decoder to receive instructions which include a custom ISA extension, memory, ALU, and branch instructions. The custom instructions are added as part of the program.

For memory and branch operations, the coprocessor also obtains the targeted address. We process this information in the metadata controller, which performs the necessary lookups for register and address rules in their respective storage pools. Metadata information is then forwarded to the rule check engine. The latter will raise an exception if a violation in the set policies is detected, or instruct the metadata controller on how to modify the stored information.

Our system is capable of tracking multiple variables or data ranges independently. Furthermore, our system automates the checks performed on data without having to explicitly rely on the programmer. Every time a new memory region or allocation is created, it is assigned a new identifier. Our coprocessor module tracks data movements from and to these regions using the assigned identifier, creating a *taint map* of each initial data allocation. This allows us to disambiguate data contexts, disallowing for aliasing of distinct data sets to occur at runtime. We attach a series of flags to the tagged data dictating how it can be propagated and used. The coprocessor automatically performs the necessary checks when the data is copied or used in computations, raising an exception when a data region is used in a way conflicting with the set rules.

B. Metadata Storage

Figure 4 shows the basics of our storage mechanism. Instead of identifying memory addresses using a direct mapping between memories, we use a storage system similar to a ternary Content Addressable Memory (CAM). For each identified memory region, we keep the base address of the allocation and its size. Each defined region is accompanied with an identification, or *tag*, and usage information, or *flags*. On a memory access, we check the contents of the storage for

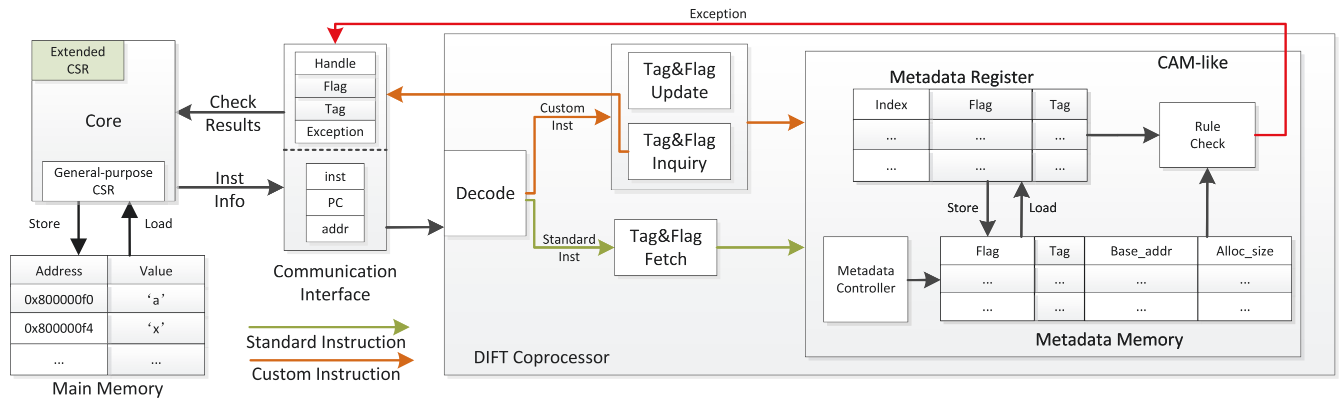


Fig. 3. Overview of the proposed coprocessor architecture. Instructions and register values are received through the coprocessor interface, and perform checks on the requested operations using internally stored metadata. If a policy violation is detected on an operation, an exception will be raised.

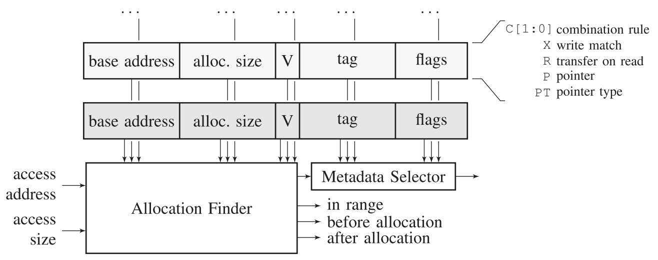


Fig. 4. Metadata storage system for addresses in memory.

potentially matching regions to the address being accessed and the size of the access. Matches are performed in parallel. This avoids stalling execution until a matching region is found at the cost of more overhead in hardware. We find this trade-off to be acceptable especially when considering realtime and timing-critical applications. The Allocation Finder returns whether a particular allocation exists that fully matches as well as regions that share a bound with the access being performed.

Metadata for registers, like identifier, tag, flags and usage rules, is stored using a direct-mapped memory with one entry on the memory corresponding to a register in the core. This strategy is chosen because the number of registers in the core is known, and the total number of registers small compared to the overhead of a ternary CAM-like structure. Register metadata can be automatically inherited from memory regions when the program performs a load instruction, and create new memory regions when the program executes a store instruction.

Both the storage elements for registers and the allocation finder are *not* directly accessible via standard load and store instructions. These instructions are only capable of indirectly modifying the contents of these areas based on the metadata associated to the registers or memory regions (direct manipulation of the contents of these areas are discussed in Section VI-A). In order to deal with data hazard caused by metadata update, we have specific logic to forward the updated metadata to subsequent memory instructions.

C. Metadata Generation

As shown in Figure 3, the decoded instructions are sent to CAM-like structure according to instruction type. Custom instructions are used to configure the tag and flag of CAM-like structure line and register. Meanwhile, the standard instructions will update the metadata at runtime. After tag and flag update, the security check will be performed.

Software developers utilize hints in code to indicate the compiler which data should be tracked. The developer does not need to know vulnerabilities in the code, only variables that carry sensitive information. The compiler generates code which signals the coprocessor of memory regions creation. When memory instructions related to this area are committed, the corresponding tag and flag check operation are executed by the coprocessor based on the instruction type.

D. Metadata Usage and Rule Checks

We utilize the flag field in our metadata to indicate the type of data, how it can be used, and how the metadata is propagated. We also provide a field to indicate whether a particular memory region or register contains a pointer, and if so the type of pointer (data or code). We provide fields to determine copy rules, allowing or disallowing copying data between regions with different tags or flags. Lastly, for ALU operations, we provide rules stating if the data in two regions can be *mixed* as well as specifying the resulting metadata of the operation. By combining the flag field and the tag field, the programmer can define policies to enforce data-flow integrity and prevent malicious data execution, data leakage and buffer overflow based attacks.

We include different subfields in the flags area, as shown in Figure 4. These are utilized to determine the policy that corresponds to the allocation. The P field represents whether the current data is a pointer value and the PT field is used to indicate whether we are in presence of a code or data pointer. The whole memory area consists of pointers if this area is tagged using the P field. This is useful to indicate the location of constructs like string tables, jump tables, or virtual function tables. The former are an array of pointers to data, the latter two are internally arrays of pointers to regions of code. According to the P field of one memory area, we can differentiate whether it can be executed. In addition, malicious read and write operations on specific memory area will be checked. When the memory area is used as source, the R field is used to decide whether it will be propagated to destination. The C field focuses on ALU instructions and it is used to indicate the mixing and propagation rule between register operations. The X field is used to force checks on matching metadata information in the source and destination. This allows us to constrain writes to memory regions from only properly tagged registers.

Algorithm 1 On Memory Accesses, the Coprocessor Entries That Match and/or Border the Address Being Accessed

```
Require: target address $addr$
Require: access size $size$
 1: for $line \in allocs$ do
 2:   if $addr \in line'range$ then
 3:     $line \rightarrow match$
 4:   end if
 5:   if $addr == line'end + 1$ then
 6:     $line \rightarrow after$
 7:   end if
 8:   if $addr + size == line'start$ then
 9:     $line \rightarrow before$
10:   end if
11: end for
12: return ($match, after, before$)
```

Whenever software makes a memory access, the coprocessor attempts to match the address being accessed with the contents of the metadata memory. Through a series of comparators, the coprocessor examines all allocations following the steps outlined in Algorithm 1. Note that the usage of these matches depend on whether the access was on a load or a store instruction. This operation is done in parallel for all storage entries. If the access is due to a load instruction, the coprocessor utilizes the matched line in the rule check engine to determine if the access can take place, and if any metadata associated with the memory region must be propagated to the registers. If the access is deemed to be illegal, the coprocessor raises an exception. If metadata is required to be transfered to registers, then the coprocessor updates the register's metadata in the register metadata storage. If the access was due to a store instruction, the coprocessor uses the match to determine whether or not the access can take place, raising an exception if it is deemed illegal. However, if the access lies in at the edge of an existing allocation and the allocation has the same metadata as the one in the access, the coprocessor will utilize the $before$ or $after$ to update the allocation start address or size as needed. In the event where both $before$ and $after$ are valid and have matching metadata to the store taking place the coprocessor will merge the two allocations into one, saving precious storage resources.

E. Handling Application Binary Interface

The Application Binary Interface is a set of agreed-upon conventions on how software allocates CPU and memory resources for program usage. Among the rules, we find the *calling conventions* of a particular platform, how stack frames are used, and how registers are used between functions. Table II shows the ABI for the RISC-V platform. Note that although the proposed FineDIFT framework can be applied to any ISAs, we implement it on the RISC-V core for demonstration in this paper. Registers are divided into caller saved and callee saved. The former group of registers have their value saved by a function before it calls another function. The latter group are registers that are saved by a function *before* it uses them. Furthermore, we see that arguments to functions are passed in registers. In RISC-V, the first eight arguments are passed in registers and the rest are put in the stack. This holds true for the ABI of other modern platforms, as it alleviates the need of writing to and reading from memory.

TABLE II
RISC-V CALLING CONVENTIONS AND REGISTER USAGE FOR THE INTEGER FILE. THE PRESENCE OF A FRAME POINTER IS OPTIONAL. IF A FRAME POINTER EXISTS IT MUST RESIDE IN x8 (s0), THE REGISTER REMAINS CALLEE-SAVED

Register	ABI Name	Description	Saver
x0	zero	Zero register	–
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	–
x4	tp	Thread pointer	–
x5–x7	t0–t2	Temporary registers	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–x11	a0–a1	Function arguments/return values	Caller
x12–x17	a2–a7	Function arguments	Caller
x18–x27	s2–s11	Saved registers	Callee
x28–x31	t3–t6	Temporary registers	Caller

We tie our solution to the platform's application binary interface while still trying to be architecture and platform agnostic by defining a configurable register mask which defines which registers are caller saved. On function calls and returns, the register mask is applied to the register metadata store, clearing metadata for caller saved registers. This ensures that the callee or return target can utilize the registers without any metadata conflicts.

When a function call occurs, the value of a caller saved registers are stored in the caller's stack frame. The store operation has the side effect of creating a new memory region in our coprocessor's memory metadata storage containing any tag and flag information associated with the register. Once the callee returns, the caller restores the value of the registers by performing load operations from its stack frame. As a side effect of this operation, any metadata associated with the memory region is automatically associated by the coprocessor to the register. This restores any metadata context held by the caller function.

The configurable mask can be overridden between function calls and returns, allowing parameter passing on registers to have associated metadata, as well as preserving register metadata during a function return. This allows the use of functions that take arguments with associated metadata, and for that metadata to be used internally. When a function is called, we issue instructions to temporarily override the register mask indicating which of the caller-saved and argument registers should keep metadata on the call. On function returns, we also set the mask to indicate whether a return value on a register should keep its associated metadata.

Lastly, a function may store temporary data into its stack frame which may have associated metadata. The stack frame is deallocated as part of the function epilogue. As part of our information flow tracking framework, we discard any metadata associated with the function's stack frame from the memory metadata store. This allows for clean reuse of the stack region by other functions without triggering an exception.

VI. FineDIFT Demonstration

A. FineDIFT on Rocket-Chip

The Rocket-Chip and associated SoC generator are baseline implementations of the RISC-V ISA [35]. Particular to the Rocket-Chip, a coprocessor interface called RoCC is a non-standard extension to the RISC-V architecture which allows for the addition of custom coprocessors. It transmits instructions and data from the processor to the coprocessor and vice versa. The RISC-V ISA specifies instructions dedicated for custom extensions which can be leveraged by an RoCC coprocessor. These instructions are treated as nop by the processor.

We developed a test implementation of our framework using the Rocket core. We utilize the Control and Status Register (CSR) area as means of configuring the coprocessor and exposing exception information to software. An extended RoCC provides the means of collecting instructions being executed, as well as addresses being targeted by load, store, or branch instructions. We decouple the logic of the metadata controller and rule check engine from the interfaces to the processor and the instruction decoder. In this way, our implementation becomes portable across different ISAs.

1) Extended RoCC Interface and RoCC Custom Instructions: We extended the RoCC interface to expose signals from the write-back interface to our coprocessor. These signals include the encoding of the committed instruction, as well as the address it targets. This allows to synchronize memory and ALU instructions with their addresses and their used registers, respectively.

We further use the RoCC to provide an extension to the RISC-V ISA. We show our new instructions and their usage in Table III. The new instructions can directly modify the metadata in both metadata storage units. Based on the microarchitecture of our design, these new custom instructions do not stall the processor's pipeline because the security checks are executed in parallel with the instructions executed in the main processor. Once the instruction is committed in the processor, the related information, such as instruction encoding, the program counter and the memory address to be accessed are transmitted to the coprocessor for further security check. An exception is raised once the security check fails in the coprocessor. In Section IX.B, we discuss the possible cases on more complex CPUs which adopt out-of-order execution and speculative execution. We added two main categories of instructions. Some instructions act directly on the memory metadata controller and its storage unit. These instructions can create, configure, enable, and disable memory regions. Others directly interact with the register metadata unit, allowing us to fine-tune register usage.

Any additional information needed by these instructions is gathered from the write-back stage of processor core since at this point the address is readily available and does not need to duplicate any existing logic. The program counter associated with every executed instruction is also extracted from the processor core. It is utilized to provide accurate exception information in case a data violation is detected by the rule check engine. Exception cause and location in program code is exposed through the CSR file, allowing a trap handler to perform further analysis.

For ISA extension, we use the reserved ISA space to define the custom coprocessor instructions. RISC-V is a new and open ISA which provide support for extensions and customisation. Also, we use the Rocket Chip Coprocessor (RoCC) interface to implement our coprocessor. The RoCC interface also reserves four custom opcodes for developer to define their coprocessor instructions. Besides, Rocket Chip also provide Simple Custom Instruction Extension (SCIE) interface for extending the custom instructions. It allows us to implement instructions with two source and one destination register in the custom-0/1 opcode spaces. Therefore, there is no conflict between existing ISA and custom instructions.

2) Extended Control and Status Register (CSR): We extend the CSR file to support the global coprocessor configuration and provide exception handling capabilities. The RISC-V ISA specifies a series of areas in the CSR space for custom extensions. We place the configuration and exception registers in the system mode access region, while placing the register masks in the user CSR file. This allows us easy migration to extend the system to a full OS stack. Only the OS is capable of controlling the status of the coprocessor while the userland processes can set register masks freely.

B. Toolchain Support

For experimentation, we modified the GNU Binary Utilities version 2.34 adding a custom extension to the RISC-V backend which defines all our new instructions and additions to the CSR file. We also modified GNU Compiler Collection version 10.1 to add our extension as part of the multilib environment, and allowing the compiler to emit the necessary changes to the epilogue to clear any temporary allocations from the metadata store pool. We added attributes to functions and parameters which can be used by a programmer to indicate whether or not ABI register metadata should be preserved alongside the function call. Lastly, we provide a small library with common functions which are used to enable the coprocessor and to create static memory regions at runtime.

We provide a set of extensions to GCC's variable and function attributes to allow for the instrumentation of function calls and returns. The former allows a programmer to define which variables should be instrumented and how to instrument the program. The latter allows for fine-grained control of the register mask, allowing for instrumented parameters to be passed to functions and for instrumented returns.

TABLE III

NEW COPROCESSOR INSTRUCTIONS ADDED BY FINEDIFT. THESE INSTRUCTIONS CAN BE USED TO MANIPULATE DIFT METADATA. INSTRUCTIONS THAT RETURN A HANDLE PROVIDE A HARDWARE-SPECIFIC DESCRIPTOR OF A REGION FOR SUBSEQUENT MODIFICATION BY OTHER INSTRUCTIONS. INSTRUCTIONS THAT READ OR WRITE DATA FROM SHADOW STORAGE ACT ON THE COPROCESSOR'S METADATA MEMORY

Mnemonic	**Function**
`dift.memcreate rZ, rX, rY`	create new memory range in storage and return handler in `rZ`
`dift.memdis rX, rY`	disable allocation starting at `rX` with size `rY`
`dift.memen rX, rY`	enable allocation starting at `rX` with size `rY`
`dift.memsettag rX, rY`	update tag of memory region with handler `rX`
`dift.memsetflag rX, rY`	update flags of memory region with handler `rX`
`dift.memgettag rZ, rX, rY`	return the tag of an allocation starting at `rX` with size `rY`
`dift.memgetflag rZ, rX, rY`	return the flags of an allocation starting at `rX` with size `rY`
`dift.memprep rZ, rX, rY`	return a handle to an allocation starting at `rX` with size `rY`
`dift.tagreg rX, rY`	create a tag for register `rX` and assign it flags in `rY`
`dift.untagreg rX`	invalidate metadata for register `rX`
`dift.settag rX, rY`	set tag of register `rX` to value in `rY`
`dift.setflags rX, rY`	set flags of register `rX` to value in `rY`
`dift.savereg rX`	save tag and flag of `rX` to shadow storage
`dift.restore rX`	restore tag and flag of `rX` from shadow storage
`dift.gettag rY, rX`	get the tag of `rX` and store it in `rY`
`dift.getflags rY, rX`	get the tag of `rX` and store it in `rY`

VII. HARDWARE AND SOFTWARE OVERHEAD

A. Area and Timing Overhead

We synthesized coprocessor alongside a baseline Rocket-chip E300 core targeting the Xilinx Artix-7 FPGA with a total of 16 CAM-like structure lines. The CAM-like structure size is sufficient to accommodate the largest datasets in BEEBS [36], a benchmark suite representative for embedded devices.

FineDIFT introduces hardware overhead with 6% LUTs and 8% Flip-Flops compared to the original RISC-V core (see Table IV). We note an increase of $\approx 2\times$ in the use of lookup tables and $\approx 0.34\times$ in flip-flops when using CAM-like structure as the metadata storage. The main overhead comes from CAM-like structure as it is implemented using FPGA logic. Note that the overhead can be reduced significantly with a dedicated CAM-like structure in ASIC designs.

The overhead is in fact much smaller compared to the scheme where a similar-size tag for every memory address is used with a shadow memory for tag storage. Such system would require a massive 2 GiB of storage, if covering the entire address map. Even when removing tag storage from unused memory addresses, hundreds of megabytes storage space is still needed. If we adopt the block memory as the metadata storage, it will introduce extra clock cycles to finish the inquiry and update of metadata. In addition, we need to add a FIFO queue to store the committed instructions and extra coprocessor stalls will be introduced.

Our design was tested against the target frequency of the Rocket-chip E300 core of 32.5 MHz and no timing violations were reported with standard synthesis settings. By itself, the maximum frequency of our FineDIFT is 33.5 MHz, which is above the requested frequency for the E300. We use comparators to ensure the quick inquiry and update for the metadata. Thus, the CAM-like structure is the major limitation in terms of clock frequency increasing.

B. Performance Overhead

For software performance overhead evaluation, we compiled benchmarks from BEEBS [36] which emphasizes on memory accesses. The programs which have frequent memory access are more sensitive to our design. In order to test the worst case in our design, we choose the benchmarks which have heavy memory I/O access. Other benchmarks in the suite focus on computation rather than memory access. We show the normalized results of the benchmarks in Figure 5. The average observed overhead is of 5.03%, with 10.8% average increase in binary sizes. The binary size increase is mostly due to library routines added to fully support our framework, with smaller binaries being the most noticeably affected.

The `crc` benchmark computes a cycling redundancy check of a dataset. The `duff` benchmark moves data from a memory location to another using a mechanism called Duff's Device. The `fir` benchmark filters a dataset based on a set of coefficients. The `lcdnum` benchmark drives a memory mapped general purpose I/O file which is connected to the lamps of a 7-segment display. The `matmult` benchmark performs multiplication of two square matrices. Lastly, the `nettle-sha256` computes the SHA256 checksum of a dataset. This subset of benchmarks perform a significant number of memory operations over datasets and are representative for embedded devices. Other benchmarks in the BEEBS suite gear more towards computational aspects, which FineDIFT has little or no impact. As such, this group of benchmarks was chosen to illustrate a worst case scenario.

Through our experiment, we note that the larger the original uninstrumented binary is the less noticeable the performance and binary size overheads incurred are. An increase on binary size does not linearly translate into a higher overhead. As such, the smallest binary, `lcdnum` suffers from over 40% binary size increase, with a performance overhead of $\approx$15%, whereas the largest benchmark, `nettle-sha256`, exhibits an overhead that is barely noticeable.

The added coprocessor instructions are the cause of the performance overhead. Instructions need to traverse the Rocket-chip's pipeline. This comes at the cost of 1 clock cycle which is needed to propagate the instruction from one stage to the next. For instructions which obtain data from the coprocessor, the RoCC halts the pipeline for 7-cycle until the requested

TABLE IV
HARDWARE OVERHEAD OF FINEDIFT ON THE RISC-V CORE

	SoC Core	SoC Core+FineDIFT				Overhead	
		CAM-like	Decode	Other	Total	Without CAM-like	With CAM-like
Lookup Tables	14705	30419	511	409	46259	+6%	+214%
Flip-Flops	8122	2040	512	188	10846	+8%	+33%
Block RAM	653	0	0	162	810+5	+24%	+24%

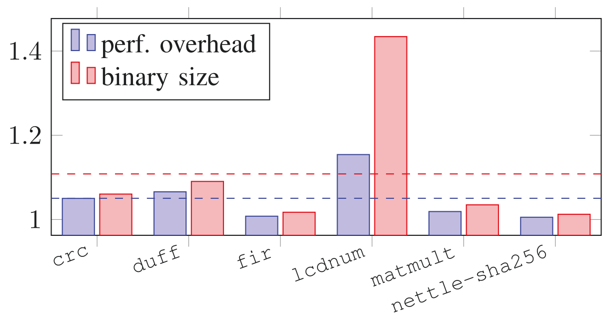


Fig. 5. Normalized performance and binary size overhead on memory intensive BEEBS benchmarks.

```
struct operation { int op_a; char desc[24];↩
     int op_b; int (*fn)(int, int); };
int add(int a, int b);

int test(void) {
  struct operation op1;
  op1.op_a = 2; op1.op_b = 4; op1.fn = add;
  scanf("%s", op1.desc);  // ❶
  printf(op1.desc);       // ❷
  return op1.fn(op1.op_a, op1.op_b);  // ❸
}
```

Listing 1. Reduced example of a sample vulnerable program.

data becomes available. Further, instructions that access the CSR file incur a 5-cycle delay, as the pipeline stalls for the effects of reading and writing to a CSR to take effect.

The observed overhead is largely dependent on the program instrumentation. The tested scenarios were chosen in part because their instrumentation requirements were different. For example, `duff` required only the declaration of a memory region, whereas we instrumented `matmult` to allow us to perform inter-procedural data-flow using registers.

VIII. SECURITY ANALYSIS

To evaluate the security capabilities of FineDIFT, we further test our framework on two programs with security vulnerabilities, a customized program and a real-world application. FineDIFT successfully detected all attacks including data leakage, code-reuse attacks and data corruption. This section discusses how our framework is used to protect programs.

A. Customized Vulnerable Application

For the first test, we design a program vulnerable to data corruption and information leakage. By giving different inputs to the program, different vulnerabilities were exploited. The basics of the program is shown in Listing 1.

When the function executes, it collects user input at ❶ and prints that information in ❷. However, examination of the call to `scanf()` reveals that the buffer where the input data is stored can easily be overflown if the input is more than 24 characters. Overflowing the buffer results in the corruption of the code pointer adjacent to it, giving an adversary the possibility to deploy a code reuse attack. Moreover, inspection of the call to `printf()` in ❷ shows that user input is directly used as the format string. This allows for format string vulnerabilities to occur. This class of vulnerability can be exploited to both corrupt memory through the use of the `%n` format string specifier, and to leak memory through a combination of `%s`, `%c`, `%x`, `%i`, and similar format string specifiers.

Knowing that attacks were possible, we gave the program the minimal instrumentation required to enforce the protection in our platform. We tagged the code pointer region within the **`struct`** as such, and set up a policy that allows the combination of `op\_a` and `op\_b` as long as these share the same kind of tag and flag information, propagating the metadata as required.

1) Data Corruption Detection: Our first test consisted of corrupting the second operand to the function. By overflowing the buffer, the corresponding metadata of `op1.op\_b` will be overwritten by a malicious one. As metadata is propagated to `add()`, the coprocessor raises an exception when both operands are combined within the function.

2) Code-Reuse Attack Detection: Return-oriented programming (ROP) [37] was developed to bypass Data Execution Prevention (DEP). Attackers link the program gadget to form arbitrary computation and gain control of the target program. In order to deal with ROP attacks, different kinds of defense mechanisms to protect `return` instructions in program. The authors in [38] use `jump` instructions to bypass the defense mechanisms which aim at protecting `return` instructions. According to the above two works, we develop our attack vectors. Specifically, we attempted a code reuse attack by overflowing the buffer so that we wrote data over the code pointer area of **`struct`** operation. This resulted in the metadata for the region being overwritten with the input metadata. As a result, when performing the indirect call in ❸, the coprocessor threw an exception indicating that a code pointer was being utilized without being flagged as such.

Even without instrumentation, we were still able to detect a possible code-reuse attack. We crafted an input which overwrites the return address in the stack. Without instrumentation, our coprocessor still creates a region of memory in the CAM-like structure for stored return addresses. The region is lost when writing over this location. We reserve a tag for code pointers and automatically attach it to the link register. Whenever the link register is saved to the stack in a non-leaf function, a memory region is automatically created, providing safeguards against overwriting of that area through a stack buffer overflow. The region is automatically removed when the function returns. When reloading a maliciously modified return address into the return address register an exception will be raised by the coprocessor.

3) Data Leakage Detection: We attempted to use the format string vulnerability in ❷ to leak the second operand to the `add()` function. By using repeated applications of `%x`, we are able to run through the arguments on `printf()` until reaching the one we need. The `printf()` function requires the use of a temporary buffer where converted data is stored. When reaching the location of `op1.op\_b` in memory, metadata is propagated into the buffer. However, we enforced that copying data from `op1.op\_b` into a region of memory that shares the same type of metadata. The buffer in `printf()` has no metadata, thus the coprocessor raises an exception.

B. Real-World Application: OpenSyringePump

We further tested FineDIFT using an open-source application, OpenSyringePump.[1] This program is often used as a sample real-world application in security applications for embedded systems [39]–[41].

OpenSyringePump is an open-source electric pump which uses a syringe as the means of moving fluid. The device makes use of a stepper motor to move a threaded rod which moves a metal piece that is attached to the syringe's plunger. A microcontroller drives the stepper motor and provides the user the means of interaction through a series of push buttons, an LCD module, and a serial terminal. Commands can be entered to the microcontroller's software through the push buttons, or the serial terminal. The microcontroller responds to these commands by moving the stepper motor, causing the syringe to either dispense or absorb a bolus.

The software in OpenSyringePump is written using the wiring set of libraries and Hardware Abstraction Layer (HAL) [42] targeting Arduino-like platforms. All the low-level peripheral handling is delegated to the HAL, with the control system being implemented directly by the programmer. Due to the lack of availability of a HAL for our test platform, we ported the necessary functions to run in the RISC-V core.

We instrumented and evaluated OpenSyringePump with our DIFT framework, including HAL functions. Our instrumentation gated access to the GPIO file to GPIO specific functions, such as `digitalWrite()` and `digitalRead()`, by creating a region over the addresses held by the GPIO registers disallowing access to the GPIO files outside the instrumented functions, as shown in Figure 6. We also instrumented the `String` class to ensure that characters can only be stored in the string buffer.

We tested for both binary size and cycle accurate performance overhead. We measured an increase of 13% in the binary's code and data segments. Most of the overhead comes from the associated initialization routines used by our DIFT framework. For performance overhead, we ran a total of 100 iterations of the main control loop with an emulated serial input. The provided input caused the software to rotate the stepper motor clockwise, then counter clockwise, simulating the actuator moving the syringe's plunger. We measured a performance overhead of about 6%. The main cause to this overhead is the extra instructions needed for set up of registers in the instrumented functions, and the clearing of metadata of registers used by the compiler when doing arithmetic with the stack pointer in some functions. We believe that the effects of the latter can further be optimized by changing the way the compiler utilizes registers within a function. However such analysis is outside the scope of this work.

We then introduced a vulnerability in OpenSyringePump that allowed us to overflow a buffer from the UART's input. This required us to change code in the HAL libraries to introduce the vulnerability. When attempting to overflow the buffer with a mock UART input and corrupt variables in the control system, the DIFT framework was capable of detecting the overflow and halting the corruption.

[1]https://github.com/shoreofwonder/OpenSyringePump

IX. Discussions

A. Limitations

FineDIFT is not without limitation. First, we rely on the programmer to notify the compiler on which variables should be tracked and how the tracking should occur. Moreover, the programmer is also responsible for adding attributes to functions which should take parameters that contain metadata, and functions that return metadata. Although the system is easy to use, programmers still need to be familiar with the management of registers and memory used in the program, and manually set appropriate tags and flags for the corresponding variables.

Variadic functions, those taking a variable number of arguments, present a challenge for our instrumentation approach, like the format string output function `printf()`. The argument list cannot be readily indicated to carry metadata as all parameters are not part of the function definition. For this, we provide an compiler intrinsic which allows programmers to indicate which parameters of a variadic function contain metadata that must be kept on calls.

Library functions that allocate temporary buffers on the heap also present issues. If any data that is copied into this buffer has associated metadata, a new region of memory will be created. When the buffer is deallocated, the function may have no way of identifying and removing metadata from the coprocessor. There is no straightforward process to overcome this limitation other than providing alternate functions with the necessary instrumentation.

Lastly, aggregates such as C **`struct`** which contain contiguous members that are tagged and flagged with the same metadata will be merged into a single allocation by the coprocessor, similar to how an array are handled. As this is a desired functionality of the coprocessor to reduce storage requirements, we opted to rewrite the aggregate so that members that would alias are not contiguous in memory to avoid the unintended merging. This is why our code in Listing 1 places `op\_a` at the start of the **`struct`**, being separated from `op\_b` using a buffer. This limitation is similar to those found in other frameworks, such as AddressSanitizer [43]. Because of the way compilers treat **`struct`**s, extra padding may be

```
la a2, pinout   ;; pin table
slli a0, a0, 4
add a2, a2, a0
lw a3, 0(a2)    ;; get port






lw a5, 0(a3)    ;; get port state
```

(a) Uninstrumented code. Loading the state of the GPIO port can happen at any time during program execution.

```
la a2, pinout   ;; pin table
slli a0, a0, 4
add a2, a2, a0
lw a3, 0(a2)    ;; get port
la t1, GPIO_BASE_ADDR
li t2, 36
dift.memgettag t3, t1, t2
dift.settag a5, t3
dift.memgetflag t3, t1, t2
dift.setflags a5, t3
lw a5, 0(a3)    ;; get port state
```

(b) Instrumented code. With FineDIFT, loading the state of the GPIO must be instrumented lest an exception occur.

Fig. 6. Instrumented code example.

added between members to ensure word alignment usually to increase performance in loads and store instructions. When reordering is not possible, the compiler may be written to add extra padding between members by introducing dummy entries in the `struct` at the cost of memory usage. At this point, this must be done manually.

Another concern is that recursive function calls and nested function calls may not be protected efficiently due to the limited number of CAM-like entries. Instead of simply adding more CAM-like entries, we can either design a new subsystem to handle deep recursion or securely copy the contents of the CAM-like structure to a larger memory.

B. Towards Multicore and Rich OS Systems

Our current work focus on the scenario of single processor and single thread program. For multi-core systems, multiple coprocessors are needed to support the DIFT mechanism. In order to support multi-core and multi-thread applications, the coherency issues between multiple DIFT supported coprocessors need to be solved. An extra coherency protocol need to be added for tag storage (like CAM-like structure and register-files) to ensure the coherency of tag. For example, the coherency protocol of Snoop Control Unit (SCU) [44] in ARM and UltraPath [45] Interconnect (UPI) in Intel are used to deal with this problem. Therefore, extra coherency control unit needs to be added to ensure the coherency of tag between multiple processors. Every coprocessor will reserve one coherency interface to connect the coherency control unit. The coprocessor will raise the request and transmit the tag related information to coherency control unit once the tag in one coprocessor is updated. The coherency control unit will route the related information to other coprocessors to maintain the coherency of tags among multiple coprocessors.

For Out-of-Order (OOO) processor core, the current coprocessor implementation need to be modified to support OOO execution. The coprocessor need to receive the instructions executed out-of-order and committed instructions from processor. Extra hardware components are needed in order to store the instructions that have not committed in processor, like load and store buffer. The security check on speculative instructions and out-of-order instructions will not raise the exception before the corresponding instruction is committed. Also, the coprocessor will determine whether the influenced states need to be rolled back and when these states can be committed to the CAM-like structure according to the committed instructions.

In order to solve the high hardware overhead caused by CAM-like structure, the normal memory will serve as a dictionary or hashmap for the entries of CAM-like structure. Furthermore, the CAM-like structure will act as Cache which makes the contents accessible for OS kernel. The access from processor will be routed to coprocessor through coherency control unit which exposes a slave interface for processor. Also, extra memory protection mechanisms need to be provided to protect the tag storage. In addition, the OS kernel need to be modified to support the above mentioned hardware features. The coprocessor states need to be backed up at context switch. The Process Control Block (PCB) also needs to be modified to save the coprocessor states.

C. Security Policies

The conservative or radical security policies cause different problems, such as false negative rates, false positive rates and taint pollution. The taint tag will be propagated to general purpose registers according to data flow. The situation will be much worse under radical policies. For example, when special registers like stack pointer register are tainted, it will cause taint pollution because many data will be tainted in an unexpected manner. In addition to taint pollution, false positive and false negative cases may happen. In our work, we provide custom coprocessor instructions to untaint the registers and memory. However, the false positive cannot be solved using the automatic method. Some taint related operations need to consider the software context. For example, the tainted index might be used in translation tables. If the index in translation table is tainted, whether or not the corresponding data is tainted in translation table will have different consequences. When more conservative policies are adopted, the false negative case might be triggered because the data in the translation table is not tainted. The false positive case may be caused when radical policies are utilized and thus the data in translation table is tainted. In summary, the taint operation need to be combined with the specified software context. In order to deal with false positive cases and false negative cases, we require the software developer to write hints so that the compiler can generate custom instructions to instruct the security check operations.

X. Conclusion

In this work, we first examined previous DFI approaches in terms of implementation and their security. We then presented FineDIFT as a method of applying DIFT principles to DFI in embedded devices. In designing FineDIFT, we considered the storage requirements for metadata and its use, minimizing the storage and instrumentation requirements of software while still offering a flexible programming model. We examined the overhead of FineDIFT in terms of hardware and software as well as its security benefits. As future work, we will extend FineDIFT to support for multicore platforms for general computing system protections.

References

[1] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proc. 7th Symp. Operating Syst. Design Implement.*, 2006, pp. 147–160.

[2] N. Vachharajani *et al.*, "RIFLE: An architectural framework for user-centric information-flow security," in *Proc. 37th Int. Symp. Microarchitecture (MICRO)*, Dec. 2004, pp. 243–254.

[3] B. Gu *et al.*, "D2Taint: Differentiated and dynamic information flow tracking on smartphones for numerous data sources," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 791–799.

[4] M. Sun, T. Wei, and J. C. S. Lui, "TaintART: A practical multi-level information-flow tracking system for Android RunTime," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 331–342.

[5] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng. (PASTE)*, 2001, pp. 54–61.

[6] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," *ACM Sigplan Notices*, vol. 39, no. 11, pp. 85–96, 2004.

[7] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proc. 37th Int. Symp. Microarchitecture (MICRO)*, Dec. 2004, pp. 221–232.

[8] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 482–493, Jun. 2007.

[9] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "FlexiTaint: A programmable accelerator for dynamic taint propagation," in *Proc. IEEE 14th Int. Symp. High Perform. Comput. Archit.*, Feb. 2008, pp. 173–184.

[10] C. Palmiero, G. D. Guglielmo, L. Lavagno, and L. P. Carloni, "Design and implementation of a dynamic information flow tracking architecture to secure a RISC-V core for IoT applications," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2018, pp. 1–7.

[11] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "TaintTrace: Efficient flow tracing with dynamic binary rewriting," in *Proc. 11th IEEE Symp. Comput. Commun. (ISCC)*, Jun. 2006, pp. 749–754.

[12] H. Chen, X. Wu, L. Yuan, B. Zang, P.-C. Yew, and F. T. Chong, "From speculation to security: Practical and efficient information flow tracking using speculative hardware," in *Proc. Int. Symp. Comput. Archit.*, Jun. 2008, pp. 401–412.

[13] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. London, U.K.: Pearson, 2006.

[14] Z. B. Celik *et al.*, "Sensitive information tracking in commodity IoT," in *Proc. 27th USENIX Secur. Symp. (USENIX Secur.)*, 2018, pp. 1687–1704.

[15] W. Enck *et al.*, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 1–29, 2014.

[16] M. S. Lam, M. Martin, B. Livshits, and J. Whaley, "Securing web applications with static and dynamic information flow tracking," in *Proc. ACM SIGPLAN Symp. Partial Eval. Semantics-Based Program Manipulation (PEPM)*, 2008, pp. 3–12.

[17] U. Dhawan *et al.*, "Architectural support for software-defined metadata processing," in *Proc. 20th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2015, pp. 487–502.

[18] H. Kannan, M. Dalton, and C. Kozyrakis, "Decoupling dynamic information flow tracking with a dedicated coprocessor," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2009, pp. 105–114.

[19] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," in *Proc. USENIX Secur. Symp.*, 2006, pp. 121–136.

[20] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software," in *Proc. NDSS*, vol. 5, 2005, pp. 3–4.

[21] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical dynamic data flow tracking for commodity systems," in *Proc. 8th ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environ.*, 2012, pp. 121–132.

[22] U. Dhawan *et al.*, "PUMP: A programmable unit for metadata processing," in *Proc. 3rd Workshop Hardw. Architectural Support Secur. Privacy*, Jun. 2014, pp. 1–8.

[23] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh, "Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2010, pp. 137–148.

[24] J. Porquet and S. Sethumadhavan, "WHISK: An uncore architecture for dynamic information flow tracking in heterogeneous embedded SoCs," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, Sep. 2013, pp. 1–9.

[25] J. Lee, I. Heo, Y. Lee, and Y. Paek, "Efficient security monitoring with the core debug interface in an embedded processor," *ACM Trans. Design Autom. Electron. Syst. (TODAES)*, vol. 22, no. 1, pp. 1–29, 2016.

[26] K. Chen, Q. Deng, Y. Hou, Y. Jin, and X. Guo, "Hardware and software co-verification from security perspective," in *Proc. 20th Int. Workshop Microprocessor/SoC Test, Secur. Verification (MTV)*, Dec. 2019, pp. 50–55.

[27] A. S. Siddiqui, G. Shirley, S. Bendre, G. Bhagwat, J. Plusquellic, and F. Saqib, "Secure design flow of FPGA based RISC-V implementation," in *Proc. IEEE 4th Int. Verification Secur. Workshop (IVSW)*, Jul. 2019, pp. 37–42.

[28] V. Nagarajan, H.-S. Kim, Y. Wu, and R. Gupta, "Dynamic information flow tracking on multicores," in *Proc. Workshop Interact. Between Compil. Comput. Archit.*, 2008, pp. 1–11.

[29] M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, and T. Suri, "SIFT: A low-overhead dynamic information flow tracking architecture for SMT processors," in *Proc. 8th ACM Int. Conf. Comput. Frontiers (CF)*, 2011, pp. 1–11.

[30] M. A. Wahab, P. Cotret, M. N. Allah, G. Hiet, V. Lapotre, and G. Gogniat, "ARMHEx: A hardware extension for DIFT on ARM-based SoCs," in *Proc. 27th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2017, pp. 1–7.

[31] M. A. Wahab *et al.*, "A small and adaptive coprocessor for information flow tracking in ARM SoCs," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Dec. 2018, pp. 1–8.

[32] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Electron. Notes Theor. Comput. Sci.*, vol. 89, no. 2, pp. 44–66, 2003.

[33] D. Bruening, E. Duesterwald, and S. Amarasinghe, "Design and implementation of a dynamic optimization framework for windows," in *Proc. 4th ACM workshop Feedback-Directed Dyn. Optim. (FDDO)*, 2001, pp. 1–12.

[34] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *LASER Summer School on Software Engineering*. Berlin, Germany: Springer, 2011, pp. 1–30.

[35] *Sifive's Freedom Platforms*. Accessed: Jan. 2022. [Online]. Available: https://github.com/sifive/freedom

[36] J. Pallister, S. Hollis, and J. Bennett, "BEEBS: Open benchmarks for energy measurements on embedded platforms," 2013, *arXiv:1308.5174*.

[37] M. Prandini and M. Ramilli, "Return-oriented programming," *IEEE Secur. Privacy*, vol. 10, no. 6, pp. 84–87, Nov. 2012.

[38] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. 17th ACM Conf. Comput. Commun. Secur. (CCS)*, 2010, pp. 559–572.

[39] T. Abera *et al.*, "C-FLAT: Control-flow attestation for embedded systems software," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 743–754.

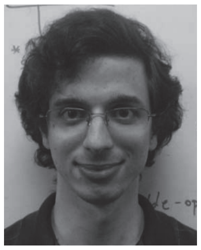
[40] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "LiteHAX: Lightweight hardware-assisted attestation of program execution," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2018, pp. 1–8.

[41] I. D. O. Nunes, S. Jakkamsetti, and G. Tsudik, "Tiny-CFA: A minimalistic approach for control-flow attestation using verified proofs of execution," 2020, *arXiv:2011.07400*.

[42] H. Barragán, *Wiring: Prototyping Physical Interaction Design*. Ivrea, Italy: Interaction Design Institute, 2004.

[43] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2012, pp. 309–318.
[44] *Snoop Control Unit on Arm SoC*. Accessed: Jan. 2022. [Online]. Available: https://developer.arm.com/documentation/ddi0407/e/snoop-control-unit/ab%out-the-scu
[45] *Intel Ultrapath Technology*. Accessed: Jan. 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html

Kejun Chen received the B.S. degree in computer engineering from Liaoning Shihua University, Liaoning, China, in 2015, and the M.S. degree in computer engineering from Northeastern University, Liaoning, where he is currently pursuing the Ph.D. degree. His research interests include hardware security, secure computer architectures, and IP core design and integration.



Orlando Arias received the B.S. and M.S. degrees in computer engineering from the University of Central Florida, Orlando, FL, USA. He is currently pursuing the Ph.D. degree in computer engineering with the University of Florida, Gainesville, FL, USA. His research interests include device security, secure computer architectures, network security, IP core design and integration, and cryptosystems. He was a recipient of the Best Paper Award in the 52nd Design Automation Conference. He was awarded the NSF GRFP in 2015.

Qingxu Deng (Member, IEEE) received the Ph.D. degree from Northeastern University, Shenyang, China, in 1997. He is currently a Full Professor with the School of Computer Science and Engineering, Northeastern University. His research interests include multiprocessor real-time scheduling and formal methods in real-time system analysis.

Daniela Oliveira received the B.S. and M.S. degrees in computer science from the Federal University of Minas Gerais, Brazil, in 1999 and 2001, respectively, and the Ph.D. degree in computer science from the University of California at Davis in June 2010, where she specialized in computer security and operating systems. She is a NAE Frontiers of Engineering Alumni and a NAS Kavli Fellow. She was a recipient of the NSF CAREER Award 2012 and the 2012 United States Presidential Early Career Award for Scientists and Engineers (PECASE) from President Obama. She was also a recipient of the 2017 Google Security and Privacy Research Award.

Xiaolong Guo (Member, IEEE) received the Ph.D. degree in electrical and computer engineering from the University of Florida (UF) in 2019. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, Kansas State University (KSU). His research focuses on detecting hardware or computer vulnerabilities using formal verification and program analysis. He has been recognized with the Best Paper Awards at AsianHOST 2020 and DATE 2019 and the Best Paper Candidate at ASP-DAC 2021.

Yier Jin (Senior Member, IEEE) received the Ph.D. degree in electrical engineering from Yale University in 2012. He is currently an Associate Professor and an IoT Term Professor with the Department of Electrical and Computer Engineering (ECE), University of Florida (UF). His research focuses on the areas of hardware security, embedded systems design and security, trusted hardware intellectual property (IP) cores, and hardware-software co-design for modern computing systems. He was a recipient of the DoE Early CAREER Award in 2016 and the ONR Young Investigator Award in 2019. He received the Best Paper Award at DAC'15, ASP-DAC'16, HOST'17, ACM TODAES'18, GLSVLSI'18, DATE'19, and AsianHOST'20. He is also the IEEE Council on Electronic Design Automation (CEDA) Distinguished Lecturer.