This is a preprint. This paper has been accepted by IEEE Transactions on Computers in Dec/2021.

RVDFI: A RISC-V Architecture with Security Enforcement by High Performance Complete Data-Flow Integrity

Lang Feng*, *Member, IEEE*, Jiayi Huang*, *Member, IEEE*, Luyi Li, Haochen Zhang, Zhongfeng Wang†, *Fellow, IEEE*

Abstract—With the rapid revolution of open-source hardware, RISC-V architecture has been prevalent in both academic research and industrial developments. Due to the increasing threats of information leakage, it is imperative to provide a secure RISC-V ecosystem to defend against malicious software exploits. Toward this goal, data-flow integrity (DFI) is employed as a strict security policy for enforcing the legitimacy of each data access, thereby filtering out most of the attack exploits. However, due to the intensive computations needed by DFI, there are only limited proposals successfully implementing partial DFI with low performance overhead. Moreover, all the previous studies failed to enforce the *complete* DFI policy in a real hardware platform, while trading off security strength for performance efficiency. To provide RISC-V architecture with high security enforcement and low performance overhead, we leverage the open-source Rocket Chip and propose RVDFI, the first complete DFI implementation based on RISC-V architecture with only 17.8% performance overhead on average and 3.9% in minimum, incurring much less performance loss compared to the 166.3% overhead caused by previous complete DFI implementation.

Index Terms—Data-Flow Integrity, RISC-V, Computer Architecture, Security, Rocket Chip.

1 INTRODUCTION

In recent years, the rapid open-source hardware developments have broken the barriers in semiconductor designs, paving the way for the next waveform of computing design and innovation. Various efforts across communities have been made to reshape the ecosystem of open-source hardware, including electronic design automation (EDA) [1], [2], agile development methodology [3], [4], open instruction set architecture (ISA) [5], and so on. To eliminate the access barrier posed by commercial proprietary EDA software, several EDA tools have been open sourced for free usage, such as Verilog to Routing [1] and Icarus Verilog [2]. Additionally, many artificial intelligence hardware accelerator designs and generators have their open accesses, such as NVDLA [6], Gemmini [3], and DNNweaver [4]. In the field of processor design, the open RISC-V ISA [5] has been revolutionizing the processor developments in both academia and industry [7], [8], [9], [10], [11], [12].

In 2010, RISC-V [5] was designed and later became an open ISA standard, which leads to various hardware developments, covering the fields of artificial intelligence [12], [13], high performance computing [8], [14], cryptography [11], etc. However, unlike other architectures, the security enforcement on RISC-V is still in its infancy. For the world leading companies such as Intel, AMD and ARM, their processors are equipped with security modules, such as Intel CET [15], AMD SEV-SNP [16], and ARM CoreSight [17], to provide both security defense and performance efficiency. On the security side, software programs have become more complex and tend to expose more vulnerabilities for wider attack surfaces, which requires strong protection. Although it can be defended with software mechanisms, they usually incur considerable performance overhead. Therefore, it is imperative to provide security support for RISC-V to enforce security while maintaining performance efficiency.

Data-flow integrity (DFI) is a strict security policy that enforces the legitimacy of all the memory access instructions [18]. Since most of the software attacks need to access at least one piece of data in the memory, DFI can be used to identify the illegal data access. Thus, most of the software attacks including control-data attacks [19] and non-control-data attacks [20], [21] can be detected. However, as around 30% of the instructions in a typical software program are memory accesses such as `load` and `store`, DFI verification can be frequent and consume intensive resources. Due to this difficulty, there were only a few follow-on proposals after DFI was proposed and implemented in software in 2006 [18]. If *complete DFI* is defined as the DFI policy proposed in the seminal work [18], all the previous designs either enforce only *partial* DFI or incur large performance overhead. Moreover, none of them implement the complete DFI verification on a real hardware platform.

To mitigate the performance overhead of DFI verification, hardware-assisted approaches have been investigated in recent work [9], [22]. However, security strength has been traded for performance to make the previous designs practical. Although a recent study [23] leverages near-memory processing (NMP) and realizes complete DFI, it incurs high performance overhead and hardware resource consumption. Furthermore, NMP may not be widely accessible, especially for IoT devices. In contrast, our work explores various architectural enhancements along with the compilation flow. The proposed enhancements realize com-

*Both authors contribute equally to this paper.
†The corresponding author.

plete DFI verification without security loss with reasonable hardware resource consumption, meanwhile achieving less than $\frac{1}{9}$ performance overhead compared to the software-based complete DFI implementation.

To provide strong security protection on RISC-V, our work proposes RVDFI, a RISC-V processor design equipped with high security enforcement by performing DFI verification at runtime. With DFI verification, the performance of the proposed RISC-V processor is still comparable with an unsecured baseline. To the best of our knowledge, RVDFI is the first design with complete DFI enforcement and less than 20% (17.8%) performance overhead, and is developed based on the open-source RISC-V based Rocket Chip SoC. In summary, the contributions of this paper are as follows:

- **Architectural Support:** We propose a RISC-V architecture extension and instrumentation approach to transmitting DFI related information and enabling DFI capability for RISC-V processors.
- **Microarchitecture:** A dedicated DFI verification module is designed to provide light-weight accelerations for the simple enforcement logic, which frees the processor pipeline from DFI burden for useful work, thereby reducing performance overhead.
- **Enhancements:** Enhancements are proposed to improve the security and further reduce performance loss, including the support for function return and library protection, the hardware design for dynamic redundant load pruning, and dedicated DFI caches.
- **Evaluation:** We evaluate the performance using SPEC CPU2006 benchmark suite [24]. The results show that RVDFI only incurs 17.8% performance overhead on average, which is less than $\frac{1}{9}$ of the performance overhead of the software-DFI [18] baseline. The security analysis also shows that complete DFI effectively defends 156 control-data attacks generated from the RIPE suite and two real-world non-control-data attacks. The comparisons with previous work show that RVDFI has high security and low hardware resource consumption.

In the following sections, we first introduce the preliminaries of DFI, RISC-V, and the thread model in Section 2. Next, we discuss the related work in Section 3. The basic RVDFI architecture for DFI verification is proposed in Section 4. Then, Section 5 further introduces various enhancements for improving security and performance. Experimental results are analyzed in Section 6. Finally, Section 7 concludes this paper.

2 PRELIMINARIES

2.1 Data-Flow Integrity

Data-flow integrity (DFI) is a policy for ensuring the legitimacy of each data access. Since most software attacks are based on data modification, DFI can identify a wide range of attacks. For example, control-data attacks such as return-oriented programming (ROP) and jump-oriented programming (JOP) can be detected. A control-data attack example is the attack to the indirect branches. Attackers may change the indirect branches' targets illegally, thereby making the processor execute illegal instructions. This illegal data modification cannot pass the DFI verification, and the attacks can be detected. Besides, non-control-data attacks such as Hearbleed [21] and the vulnerability in Nullhttpd [20] can also be identified by DFI verification since these attacks are performed by illegally modifying or reading the data. Compared with control-flow integrity (CFI) [25], which can only protect the control-data attacks and can be bypassed by non-control-data attacks, DFI can enforce the security of both control and non-control-data.

To formalize DFI, given a program, each instruction is assigned an *identifier* (*ID*). We call the instructions that can write data to the memory the *write instruction*s, such as the `store` instruction, and the instructions that can read data from the memory the *read instruction*s, such as the `load` instruction. Note that the instruction here is a general concept. Either each statement of an assembly code or a C program can be called an instruction. When a read instruction with ID `A` reads data from the memory, the data needs to be written by one of the legal write instructions with IDs V_1, V_2, ..., V_n. The set of IDs $\{V_1, V_2, ..., V_n\}$ is called the *reaching definition set* (*RDSet*) of instruction `A`. Each read instruction has its own RDSet, which includes the IDs of all the legal write instructions of this read instruction. The RDSet of each read instruction is generated by the static analysis on the program. Besides, a table that records the ID of the latest write instruction writing to each data is needed by DFI. This table is called *reaching definition table* (*RDTable*). Each time, when a write instruction writes a data, RDTable needs to be updated. When a read instruction reads a data, the latest write instruction ID of the data is read from RDTable, and the obtained ID is compared with each ID in the RDSet of this read instruction. If one of the IDs in the RDSet matches, DFI verification passes, otherwise, there is a DFI violation.

An example of DFI can be illustrated by the example shown in Fig. 1. Assume the ID of each instruction in this example is the line number. After static analysis, the RDSet of instruction 8 is {6} since `printf` reads the data written by instruction 6. Similarly, the RDSet of instruction 11 is {10}; The RDSet of instruction 13 is {5}; The RDSet of instruction 16 is {10, 13}. Therefore, if the variable `len` at line 11 is too large, there will be a buffer overflow and `data` can be illegally written by instruction 11. If so, when instruction 13 (both a write and a read instruction) is executed, the latest write instruction writing `data` recorded in RDTable is 11, which is not in {5}, and a DFI violation occurs.

```
1  int data[32];
2  int data2[32];
3  int data3[32];
4  ...
5  data[pos] = func(pos);
6  data2[pos2] = func(pos2);
7  for (i = 0; i < 32; i++)
8      printf("%d\n", data2[i]);
9  for (i = 0; i < 32; i++)
10     data3[i]=i;
11 memcpy(data2, data3, len);
12 for (i = 0; i < len2; i++) {
13     data3[i] = data[i];
14 }
15 for (i = 0;i < 32; i++)
16     printf("%d\n", data3[i]);
```

Fig. 1. The code example for illustrating DFI.

2.2 RISC-V Architecture and Motivation

Our design RVDFI is based on an open-source RISC-V based SoC—Rocket Chip [7]. However, the key idea of

our design can be easily applied to other processors. The core architecture of Rocket Chip, which is called Rocket tile, is shown in Fig. 2. It is an in-order processor with a 5-stage pipeline, where IF, ID, EX, MEM, WB stand for "instruction fetch", "instruction decode", "execute", "memory access", and "writeback", respectively. Besides, Icache and Dcache represent instruction cache and data cache, respectively. Different from a typical processor, Rocket Chip contains a Rocket Custom Coprocessor (RoCC), which can be customized with specific functions and controlled by customized instructions provided by the Rocket Chip.

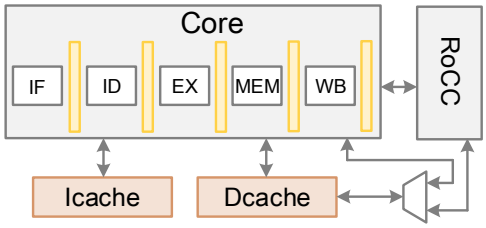


Fig. 2. The architecture of Rocket tile.

For software-DFI [18], general-purpose machine instructions are used for DFI verification, including storing/loading/searching RDTable and DFI checking, which checks if the latest write instruction's ID is in the RDSet or not. Previous study [23] revealed that most of the performance overhead is caused by DFI checking, which contains many comparison instructions and branch instructions. Motivated by this, we allocate the checking tasks from the processor core to a dedicated hardware module. This can release the computation resources required by DFI checking from the core. The dedicated hardware module needs to have three functions: It can communicate with the core to monitor the executed and committed instructions, it can access the memory to access RDTable and RDSet, and it can be customized to perform DFI verification. Therefore, RoCC is leveraged since it perfectly satisfies the architectural requirements.

2.3 Threat Model and Assumptions

In this paper, we address memory corruption based attacks. Therefore, we follow the typical threat model of most related work. We assume the software may have one or more vulnerabilities that can be exploited to attack the system through data alteration. Once the attack succeeds, the attacker would be able to read and write any memory locations in the system. This would allow attackers to perform different measurements or take various actions, which is not limited to any certain types as this capability empowers many attack vectors. The vulnerabilities can be from different places, including operating system kernel, hypervisor, library code, user programs and so on. As long as the binary is generated by the DFI software analysis and compilation workflow, it would be DFI-capable, and thereby under the protection of RVDFI. As a hardware-based solution, we assume the hardware is trusted and bug free. So attacks that exploit hardware vulnerabilities, such as rowhammer [26] attack and cache side-channel attacks [27], are out of scope. These attacks can be mitigated by other hardware-based protection techniques [28], [29].

As DFI is a hybrid approach involving both static software analysis and runtime protection, RVDFI requires the software analysis to assign the instruction IDs and generate the data-flow graph (RDSets) of a program. In addition, since RVDFI extends the instruction set to embed DFI capability, it also needs compilation support for code generation. We assume the DFI software toolchain is also trustworthy and bug free.

3 Previous Work

3.1 RISC-V Architecture

Along with RISC-V [5] developments, coherent RISC-V based system-on-chip (SoC) generators such as Rocket Chip [7] and BlackParrot [30] have been proposed to foster agile RISC-V designs. Recently, several SoC prototypes have been designed with agile development methodology by leveraging the open-source RISC-V ecosystem [8], [14]. For example, Celerity [14] features five general-purpose Rocket cores for controlling a massively parallel 496-core tiled manycore array and ten ultra-low-power cores to achieve both performance and energy efficiency. HammerBlade [8] is another highly programmable and energy efficient manycore RISC-V fabric for accelerating mixed sparse/dense computation through heterogeneity. Besides, RISC-V architecture has been employed to accelerate machine learning workloads. A RISC-V based multicore scheduling approach [13] was proposed for accelerating deep neural networks. XpulpNN [12] is also based on RISC-V and extends the ISA to realize low bitwidth quantized neural networks with low power consumption. In addition, RISC-V is also leveraged for post-quantum cryptography (PQC) such as RISQ-V [11]. In the security field, defense mechanisms have been developed and demonstrated based on RISC-V, such as hardware-assisted data-flow isolation (HDFI) [9], tagged memory supported data-flow integrity (TMDFI) [22], which will be described in detail shortly. Meanwhile, RISC-V is also widely used in industrial developments, such as NVIDIA Falcon controller [10] and Google RISCV-DV [31] instruction generator.

3.2 DFI Variants

There have been few proposals that follow up DFI to reduce its performance overhead while lowering its criteria with partial DFI [9], [32], [33]. Instead of maintaining the full data-flow, write integrity testing (WIT) ensures that each object can only be modified by particular write instructions [32]. However, since WIT only enforces the integrity of write instructions, an unsafe read instruction can read more bytes than the programmer's intention, and consequently lead to information leakage. Attacks such as Heartbleed [21] can bypass WIT. In contrast, RVDFI can defend against it since DFI enforces the integrity of both read and write instructions. Song *et al.* proposed an access control system Kenali with DFI [33]. But it only applies to the operating system kernel, leaving a wide surface in user code and data for exploits. The most recent work PIM-DFI [23] leverages near-memory processing (NMP) for DFI verification offloading. However, it incurs an average of 36.4% performance overhead, which is 1× more than RVDFI.

Moreover, PIM-DFI requires NMP, 238,333 LUTs and 39,994 FFs when implementing on the same platform as RVDFI, which is a magnitude more hardware resources than RVDFI. Another example is hardware-assisted data-flow isolation (HDFI) [9]. Similar to DFI policy, the policy enforced by HDFI also requires the data read by each instruction can only be written by certain instructions. However, HDFI only separates the memory into two regions, which are sensitive and non-sensitive. In contrast, the complete DFI enforced by RVDFI uses 16-bit IDs, which is enough for even large software programs [18], with much finer granularity and thus, higher security. The weakness of HDFI has been discussed in prior work [23], which shows that some attacks that are missed by HDFI can be detected by the complete DFI policy. Consequently, RVDFI is 32768× finer-grained than HDFI. Similarly, tagged memory supported data-flow integrity (TMDFI) [22] also has low granularity, since it only uses 8 bits for the ID and only supports 256 different IDs. As shown in [23], for a typical program, such as the benchmark in SPEC CPU2006, it needs more than 1000 or even 10000 IDs, which is orders of magnitude of what TMDFI can provide. Therefore, RVDFI is 256× finer-grained than TMDFI. Moreover, TMDFI has more than 39% performance overhead, which is 2× as RVDFI.

3.3 Control-Flow Integrity

Control-flow integrity (CFI) [25] is another security policy different from DFI. CFI enforces the legitimacy of each transition between the instruction sequences. For example, CFI requires that each branch instruction in a program should only jump to one of the legal targets generated by static analysis. CFI was first proposed with a software implementation [25], and later assisted with hardware approaches [34], [35] to reduce the performance overhead. Intel proposed control-flow enforcement technology (CET) [15] to enforce CFI, which is a coarse-grained implementation compared to Griffin [34]. Griffin is a CFI design that uses Intel Processor Tracing to generate control-flow traces, which is used for CFI verification in software. Lee *et al.* [35] has proposed using ARM Program Trace Macrocell to generate the control-flow traces, which are sent to an FPGA through ARM Trace Port Interface Unit for CFI verification. In comparison, CFI is able to detect control-data attacks but not non-control-data attacks, while DFI can identify both types of attacks [18].

3.4 Hardware-based Memory Protection

To reduce the high overhead of software memory protection mechanisms, several hardware-based approaches have been proposed [36], [37], [38], [39]. CHERI [36] uses a one-bit tag to indicate whether a memory address stores a valid capability, where 256 bits are used to describe the capability of the stored fat pointer, which can be used for bounds checking. A recent study has shown that CHERI fails to protect intra-object data since the bounds checking is in coarse object granularity [40]. Recently, BOGO [37] leverages memory protection extension (MPX) to provide temporal memory safety in addition to MPX's spatial memory safety. AOS [38] is a low-overhead always-on hardware-assisted approach to protecting heap memory safety with bounds checking, which leaves the stack data an attack surface. All the above approaches focus on pointers for coarse-grained bounds checking, which are limited for fine-grained data protection such as generic and intra-object data. Therefore, they fall short in fine-grained data protection compared to RVDFI that provides instruction-level data access control, covering all data accesses. PHMon is a programmable hardware monitor that can implement different security policies [39]. Despite its flexibility, its generality can incur high overhead for simple checks and miss the opportunities of runtime optimizations offered by RVDFI, which is critical to achieve significant overhead reduction. In contrast to the above solutions, RVDFI provides both fine-grained data protection and low runtime overhead through a complete DFI implementation on RISC-V architecture.

4 Basic RVDFI Architecture

In this section, we introduce the basic RVDFI architecture. The overall DFI verification flow is first presented. Then, two important aspects are described, which are the static analysis for facilitating runtime DFI verification, and the DFI-related information transmission between the processor core and RoCC.

4.1 DFI Verification Flow

The overview of the basic RVDFI architecture and the DFI verification flow is shown in Fig. 3, where the DFI verification is performed in RoCC and managed by the proposed DFI controller. The software program that needs to be verified by DFI is called the *target program*. The verification flow is separated into offline and online parts. In the offline part, the target program is analyzed and instrumented during the compilation flow. For the online part, the DFI verification is performed to check the execution of the target program at runtime. During the execution of the target program, when a memory instruction that needs to be verified is committed, a *DFI-request* is raised by the core and sent to the DFI controller for verification. The instruction that initiates the DFI-request is called the *corresponding instruction* of this DFI-request. RoCC stalls the core once a new DFI-request is raised but RoCC is busy. "Other" in Fig. 3 stands for all the other signals needed by RoCC to complete DFI verification. The circled numbers in Fig. 3 show the steps for the DFI verification flow, which is described as follows (⓪ is not shown in the figure):

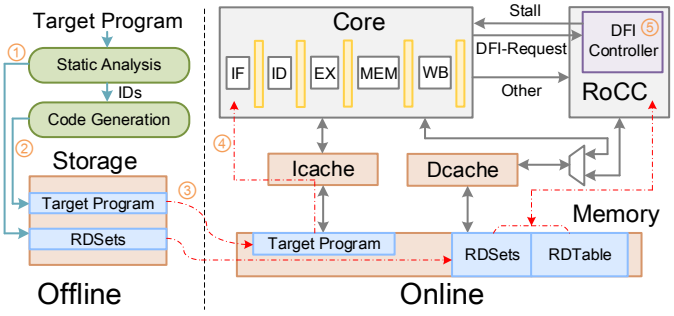


Fig. 3. The DFI verification flow for the basic RVDFI system.

⓪ The operating system (OS) is modified to reserve a part of physical memory dedicated for RDSets and RDTable.

The core can check access bounds to ensure no user instruction can write to the reserved memory.

① Given the source code of the target program, static analysis is performed to assign an ID for each memory instruction and generate the RDSet for each of them.

② The target program is instrumented for sending DFI-requests and related information to RoCC through its customized instruction.

③ When the target program is loaded by the OS, its RDSets are also loaded into the memory.

④ The instructions of the target program are executed on the core, and DFI-requests are raised by memory instructions at the commit stage and sent to RoCC at runtime.

⑤ During the execution of the target program, when a DFI-request is raised, the DFI controller checks the ID (A), type (`write/read`) and the target address ($Taddr$) of the corresponding instruction. If the type is `write`, the DFI controller updates the RDTable to record that the latest instruction writing to $Taddr$ is A. Otherwise, if the type is `read`, the DFI controller first reads the ID (B) of the latest instruction writing to $Taddr$ from the RDTable. Then, the DFI controller reads instruction A's RDSet, and checks if B is in the RDSet. If so, DFI verification passes, otherwise, a violation is detected and an exception is raised.

Therefore, RVDFI is able to enforce DFI at runtime by following the above verification flow.

4.2 Static Analysis and RDSets/RDTable Formats

To facilitate DFI verification of a target program, we use the LLVM [41] compiler infrastructure and the LLVM-based static value-flow analysis (SVF) framework [42] to generate the RDSets through static analysis. First, LLVM [41] is used to compile the target program into the intermediate representation (IR) code. Then we apply Andersen's algorithm (field-sensitive, context- and flow-insensitive) to perform the static analysis on the IR and generate the *def-use* chains, based on which the RDSet for each read instruction is generated. The RDSet of a read instruction (*use*) consists of all the write instructions (*def*s) of the variables that can flow to it. The same RDSets are used in both the software-DFI [18] baseline and RVDFI. Note that the main focus of this work is to reduce DFI performance overhead given the static analysis results. More precise static analysis is beyond our scope and we leave it for future exploration.

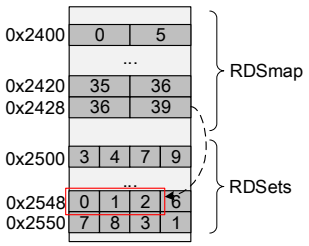


Fig. 4. An example of finding a RDSet using RDSmap.

The RDSets are loaded into the memory when the target program is loaded by the OS upon execution. Since the RDSets have variable sizes, a fixed RDSet size needs to support the largest set and can create significant memory fragmentation. Instead, we adopt an indirect access approach via a RDSmap, whose size is the same as the number of load instruction IDs. Each 64-bit RDSmap entry records the corresponding load ID's RDSet bounds in the RDSet memory region, where the higher and lower 32 bits are used for the higher and lower bounds of the RDSet in the RDSets region, respectively. Fig. 4 shows an example of finding an RDSet using the RDSmap. Suppose the RDSmap starts at address `0x2400`, and the load ID is 5, whose RDSet bounds are stored at the sixth entry of RDSmap at address `0x2428` (`0x2400+8×5`). The entry tells that the RDSet of the load instruction is the 36^{th} – 38^{th} IDs in RDSets, that is {0,1,2}. Same as RDSets, RDSmap is also static information and generated at compile time.

Similar to the `.data` and `.text` sections, RDSmap and RDSets can also be lowered to the binary as `.rdsmap` and `.rdsets` sections that can be loaded into the memory during program loading time. In our evaluation, we store them in a file and load them to a dedicated memory region before the program starts.

For RDTable, it is maintained in the physical memory and each memory location of the remaining physical memory has a corresponding table entry. When one page is shared by multiple processes, the shared page should be read-only, and the corresponding entries in the RDTable will not be updated. When a shared page is about to be written, copy-on-write in OS will copy the shared page to a new physical page and assign it to the writer process. The new physical page corresponds to a different region in the RDTable, so there is no conflict between processes. Similar to the seminal complete DFI implementation [18], the data can be 4-byte aligned. Therefore, the n-th entry of the RDTable can record the ID of the latest instruction writing to the physical target address $n << 2$. Since each ID costs 16 bits, in this case, if the size of the physical memory is N, the RDTable size is $\frac{N}{4} \times 2 = \frac{N}{2}$. Similarly, if the data is 8-byte aligned, the RDTable size is $\frac{N}{4}$. Note that the RDTable cannot be tampered with by user programs, but it can be updated by the proposed DFI controller.

4.3 Information Transmission and Instrumentation

Since DFI verification is performed in RoCC, all the information related to DFI needs to be provided to RoCC. According to Section 2.1, to verify DFI of a memory instruction in a target program, a DFI-request is issued by the core and the following pieces of information of the corresponding instruction are needed:

- **Iid**: The ID of the corresponding instruction, which is static information only related to the corresponding instruction itself.
- **Itype**: The type (`write/read`) of the corresponding instruction, which is also static information.
- **Itaddr**: The target address of the corresponding instruction, which is dynamic information and can only be obtained from the core when the write/read instruction is executed and committed.
- **Iwid**: The ID of the latest write instruction writing the data that is read by the corresponding instruction, if the corresponding instruction is a read instruction. This is also dynamic information.

- **Irds**: The RDSet of the corresponding instruction, if the corresponding instruction is a read instruction. This is static information generated at compile time and retrieved by the RoCC at runtime.

4.3.1 Transmitting Iid and Itype

For Iid and Itype, 16 bits are enough for encoding the ID of each instruction with the offline optimization for reducing the number of IDs [18], so Iid and Itype only need 17 bits to be encoded, with 16 bits for Iid and 1 bit for Itype. Due to the small size and static nature of Iid and Itype, instead of storing Iid and Itype in the memory, we can encode them along with the corresponding instruction. We implement it by leveraging one of the RISC-V's customized instructions: `custom0`, which can be used to control RoCC. When a `custom0` finishes execution in the core's pipeline and retires, its instruction body is sent to RoCC by the core. We directly instrument a `custom0` right after each write/read instruction, and use the body of `custom0` to encode Iid and Itype as the extended additional bits.

We adjust the format of `custom0` to encode Iid and Itype in the instruction body. The original format is shown in Fig. 5. For the white row in Fig. 5, `funct7` is a 7-bit indicator whose meaning can be freely defined by the designers. RoCC can write any data back to the destination register `rd` if `xd` is 1. Besides, `rs1` and `rs2` are the source registers, and the core can pass the data in `rs1`/`rs2` to RoCC if `xs1`/`xs2` is 1, respectively. We change the format of `custom0` as the gray row in Fig. 5, by leveraging `funct7`, `rd`, `rs1` and `rs2` to encode Iid and Itype. The reason why we skip leveraging `xd`, `xs1` and `xs2` is that there are some constraints for their values, so they cannot be arbitrary values as DFI may require.

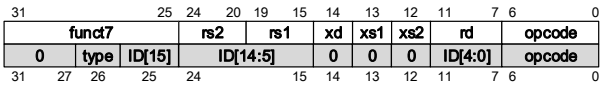


Fig. 5. The original/new (white/gray) format of `custom0` instruction.

Since the static analysis of our work is based on LLVM IR, the only write/read instruction is `store`/`load`. A pseudo code example of the instrumentation can be found in Fig. 6, where the code at line 2/4 stands for store/load data to/from address 0x90/0x70, respectively. In Fig. 6, right after each `store` and `load`, a `custom0` is instrumented. The instrumentation of `store` is named *Inmet St*, while that of `load` is named *Inmet Ld*. We call the instruction, which relates to a `custom0`, the *corresponding instruction* of the `custom0`. For example, the `store` instruction at line 2 is the corresponding instruction of the `custom0` at line 3. When RoCC receives the body of a `custom0`, a DFI-request is raised by the core for DFI verification.

```
1  ...
2  st a2 (0x90)
3  custom0 (Inmet St, with the ID and the type "store")
4  ld a0 (0x70)
5  custom0 (Inmet Ld, with the ID and the type "load")
6  ld a1 (0x80)
7  custom0 (Inmet Ld, with the ID and the type "load")
8  ...
```

Fig. 6. A pseudo code example of the instrumentation for getting Iid, Itype and Itaddr.

By adding Inmet St and Inmet Ld, when `custom0` is committed and its body is transmitted to RoCC, RoCC can get the ID (Iid) and the type (Itype) of the latest committed `store` or `load`.

An alternative approach to transmitting Iid and Itype is to encode them along with each write/read instruction. This may require expanded instruction-length encoding, which we leave for future work.

4.3.2 Transmitting Itaddr

For information Itaddr, it can be obtained at MEM stage while executing a `store`/`load`, since Itaddr is used to access Dcache for the data. However, according to Inmet St and Inmet Ld, the moment of a `store`/`load` accessing the data is before a `custom0` reaches RoCC. Therefore, when RoCC receives Iid and Itype by receiving `custom0` from the core, RoCC needs to be able to obtain the target address (Itaddr) of the latest `store`/`load` that is right before the `custom0`. To realize this, we make the architecture change as shown in Fig. 7 to pass the Itaddr along the CPU pipeline.

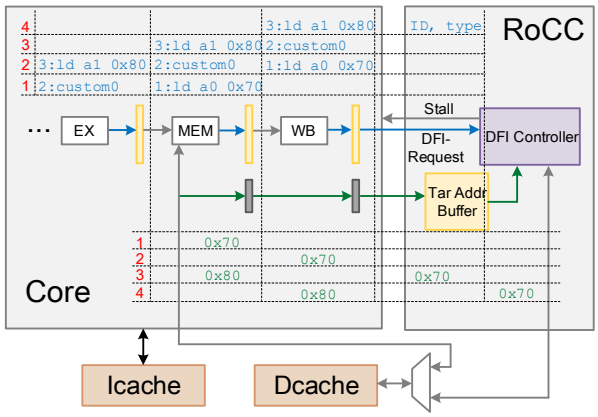


Fig. 7. The modified Rocket Chip and the example for transmitting Itype.

In Fig. 7, two pipeline registers are added to pass the target address from the MEM stage to the commit stage, then the target address is transmitted to RoCC. A target address buffer is added inside RoCC to record the latest target address. With the two registers and the target address buffer, it is ensured that the target address of the corresponding instruction and the `custom0` are synchronized and can reach the DFI controller at the same time. An example is also shown in Fig. 7, where the red numbers stand for the clock cycles, and the vertical dotted lines are for separating the stages. The numbers at the bottom of Fig. 7 are the target addresses. The executing code is the lines 4-6 in Fig. 6. It is shown that when the `custom0` reaches RoCC at cycle 4, the DFI controller is also able to fetch the target address of the corresponding `ld a0 0x70` from the target address buffer. With this modification and instrumentation, RoCC can obtain the correct target address of the latest memory access instruction before `custom0`.

4.3.3 Transmitting Iwid and Irds

For information Iwid, as described in Section 2.1, it is stored in the RDTable and the RDTable is initially empty. The RDTable is updated when a new memory write instruction is committed. As discussed in Section 4.2, RDTable is stored in the memory. RoCC has the memory interface to Dcache, and thus, can access the RDTable. For information Irds, it is

generated through the offline static analysis on the target program, and loaded into the memory before the target program starts. Therefore, RoCC can access Irds from the memory while performing DFI verification.

5 Enhancements on RVDFI System

Although the basic RVDFI architecture moves the computation of DFI verification from the core to RoCC and relieves the large performance overhead, the performance loss is still not negligible. When there is a new DFI-request, if RoCC is performing DFI verification of the previous DFI-request, the core has to be stalled until the previous DFI verification is finished. Otherwise, the new DFI-request would be missed if the core proceeds with execution, and security would be reduced. To mitigate this problem, we can either temporarily store the DFI-requests to avoid stalling, or increase the speed of the DFI verification performed in RoCC. Besides the performance, the security of the basic RVDFI architecture can also be improved by supporting function return and dynamically linked libraries.

We proposed several approaches to further enhancing the RVDFI architecture for supporting complete DFI verification with better performance efficiency. The enhanced RVDFI is shown in Fig. 8, with the additional connections between the registers in the core and RoCC, a FIFO, a load pruning buffer, and a few dedicated DFI caches. The details are introduced in the following subsections.

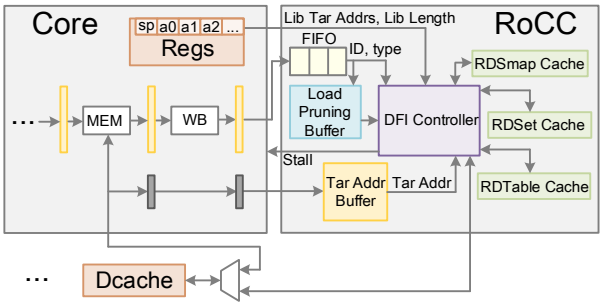


Fig. 8. The modified Rocket Chip for DFI verification with the enhancements, which is the final version of RVDFI.

5.1 Supporting Function Return and Library

As described in Section 4.2, the static analysis and instrumentation are performed based on the LLVM IR. For function return and library function call, they both contain memory accesses. However, these memory accesses are usually implicit in the IR without explicit `load` or `store` available. Therefore, the proposed instrumentation in Section 4.3 cannot protect function returns and library function calls.

We further augment the instrumentation technique and the Rocket Chip architecture to support the DFI verification for function returns and libraries. Although previous works such as [18], [23] also support this, we propose an alternative approach with a more efficient design for RISC-V architectures to achieve less instruction instrumentation.

5.1.1 Function Return Protection

There are two instructions related to function return protection, which are `calls` and `rets`. When a `call` is executed, the return address of the function being called is written on the stack. When `ret` is executed, the return address is read from the stack and the program counter of the core is changed to the return address. DFI requires that the return address of a function can only be written by the function call that calls this function. To enforce this policy, the memory address of the return address needs to be obtained by RoCC when each `call` and each `ret` are executed.

For RISC-V, the memory address of the return address is related to `sp`, which is the stack pointer register. After each function is called, there are 5 steps before it returns:

1) The `call` is executed.
2) The return address is stored at `sp-4`, and the value of `sp` is decreased to enlarge the stack.
3) The instructions in the function body are executed.
4) The value of `sp` is increased to change the size of the stack back to that before this function is called, and the return address at `sp-4` is read by the core.
5) The `ret` is executed.

In this case, right before step 1 and right before step 5, RoCC needs to obtain `sp-4`, which is the memory address where the return address is stored.

To transmit Iid and Itype, and inform RoCC of obtaining the memory address of the return address right before step 1 and 5, we instrument the target program with an additional `custom0` right before each `call` (*Inmet Call*) and `ret` (*Inmet Ret*). When RoCC receives Inmet Call and Inmet Ret, it obtains `sp-4` immediately, which is the target address of implicit `store` and `load`, respectively.

When RoCC receives an Inmet Call, it updates the RDTable entry according to the address `sp-4` with a special ID, -1, which is specifically used for function calls. When RoCC receives Inmet Ret, it reads from the RDTable entry according to the address `sp-4`, and checks if the data is -1. If so, DFI verification passes. If another illegal `store` writes data to the return address at `sp-4`, RoCC updates the corresponding entry of RDTable from `-1` to the ID of the `store`. In this case, DFI verification would fail when the function returns.

5.1.2 Library Protection

For a typical program, there may exist multiple dynamically linked library functions, such as `memcpy`, `memset`, etc. Usually the instructions of dynamically linked library functions are not analyzed during static analysis. However, many attacks can happen in libraries such as `libc` library [43]. Therefore, it is essential to also enforce DFI for library functions. For each library function, the information of the pointer arguments including their memory operation types, the target addresses and the associated memory range that may be accessed, is used during static analysis. This information can also be propagated to the hardware through an extended `custom0` with a new encoding. This new `custom0` is instrumented right before each library function `call`, after function arguments are loaded into registers (*Inmet Lib*). The `custom0` of *Inmet Lib* can specify if the library function writes/reads data to/from the memory, the memory access range. When RoCC receives a DFI-request of *Inmet Lib*, the DFI controller fetches the needed arguments and performs DFI enforcement by verifying the definition IDs of all the memory locations that are pointed by the read pointer

and its range is in the RDSet of the `call` instruction. This procedure is similar to checking n `loads` for a memory range of n. Similarly, the RDTable entries of the write pointer and its memory range are updated with the ID of the current `call`. Note that when a library function needs to load data, sometimes the memory access range can be huge, which may lead to long detection latency. To avoid the core from executing instruction streams that may contain attacks in this detection window, RoCC stalls the core when it begins to process a DFI-request corresponding to a library function that loads data, until this DFI-request is processed.

For complicated library functions, especially system calls, they can be supported similar to the kernel DFI approach in Kenali [33], which is out of the scope of this work.

5.1.3 Instrumentation and Architecture Enhancements

To support both function return and library protections, we further extend the `custom0` instruction in Fig. 5, and the encoding format is shown in Fig. 9, where the white row is for `store` and `load` instrumentation, and the gray row is for function return and library call protections. Whether the format is Inmet St or Ld and Inmet Call, Ret or Lib is decided by the 3rd most significant bit (0 for St/Ld and 1 for Call/Ret/Lib). In Fig. 9, "r", "w", "ret" represent if the corresponding instruction reads data, writes data, and is a function return or not, respectively.

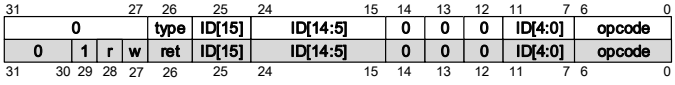


Fig. 9. The format of new `custom0` instruction (gray) for function return and library protection.

An instrumentation example for function returns and libraries is shown in Fig. 10. Inmet Call and Inmet Lib should be right before the `call`, while Inmet Ret should be right before the `ret`, according to the instrumentation policy.

```
1  def func() {
2    ...
3    add sp,sp,0x80
4    custom0 (Inmet Ret)
5    ret
6  }
7  def func2() {
8    ...
9    ld a0 8(sp)
10   add a1,a4,6
11   ld a2 64(sp)
12   custom0 (Inmet Lib)
13   call memcpy
14   custom0 (Inmet Call)
15   call func
16   ld a0 (0x70)
17   custom0 (Inmet Ld)
18   st a2 (0x90)
19   custom0 (Inmet St)
20   ...
21 }
```

Fig. 10. A pseudo code example of the instrumentation for enabling DFI verification for function returns and libraries.

Besides, Rocket Chip is further modified to support function return and library protections in Fig. 8. Since the arguments of a library function `call` are stored in the registers starting from `a0`, the values of `sp`, `a0`, `a1`, `a2`, and other successive registers are passed to RoCC and RoCC fetches the values of these registers upon receiving a `custom0` of Inmet Call, Ret, and Lib.

5.2 DFI-Request FIFO

As described in Section 4.1, the major cause for performance overhead is due to core stalling if it raises a new DFI-request while RoCC is busy with the previous DFI-request. Although dropping the new request can avoid stalling the core, the security can be compromised. Since RoCC may be idle during the processor computation phase when there are no memory instructions for DFI verification, it can result in a *free time slack*. Based on this, we introduce a FIFO inside RoCC to store the incoming DFI-requests, and send them to the DFI controller if it is free. Once the FIFO is full, the core is stalled. The FIFO is not only for temporarily avoiding stalling the core when there is a new DFI-request; more importantly, it also enables a DFI-request to use the free time slacks. This can further reduce the chance of stalling the core and increase the DFI controller utilization, thereby reducing the performance overhead.

5.3 Dynamic Redundant Load Pruning Buffer

In the seminal software-DFI work [18], offline optimizations for pruning redundant DFI verification were proposed to reduce the performance overhead. However, these offline optimizations are conservative due to static analysis. Without the runtime information, they lose the opportunities of pruning for further reducing performance loss. Although previous work PIM-DFI [23] has suggested the runtime optimizations, its optimization implementations incur significant area overhead, as shown in the result analysis in Section 6.6. In RVDFI, besides implementing all the offline optimizations in work [18] during static analysis, we also propose a light-weight hardware design named load pruning buffer for dynamic redundant load pruning, to further prune the redundant DFI-requests of Inmet Ld at runtime.

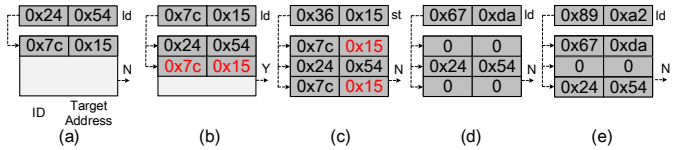


Fig. 11. An example of dynamic redundant load pruning.

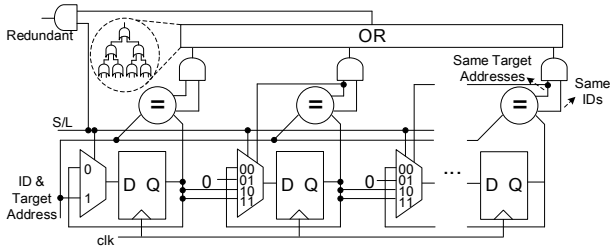


Fig. 12. The hardware structure for dynamic redundant load pruning.

If there are two DFI-requests (L, M) of Inmet Ld with the same ID and target address, and between L and M, there is no other DFI-request of Inmet St with the same target address, nor other DFI-request of Inmet Lib, then, DFI-request M is redundant. Such a runtime pruning is enabled with the proposed load pruning buffer, where each entry is a pair of the target address and ID of an Inmet Ld. As shown in Fig. 8, the load pruning buffer is added between the FIFO and the DFI controller. Once there is a DFI-request output from the FIFO, the load pruning buffer can decide if this DFI-request is redundant or not.

An example is shown in Fig. 11, where the large gray rectangles stand for the load pruning buffer. The ID and the target address of a new DFI-request is shown at the top of each subfigure, with `st` representing Inmet St, and `ld` representing Inmet Ld. In Fig. 11(a), the new DFI-request of Inmet Ld with ID `0x24` and target address `0x54` is sent from the core, and they are compared with the valid buffer entries. Since there is no match, the buffer outputs an "N" to indicate the new DFI-request is not redundant, and its information is pushed into the buffer as shown in Fig. 11(b). In Fig. 11(b), the information of another new DFI-request is simultaneously compared with all the valid buffer entries with a hit. The buffer outputs "Y" to indicate the DFI-request is redundant, and the DFI-request is ignored by the DFI controller. At the same time, the information of this DFI-request is pushed into the buffer. In Fig. 11(c), the target address of the DFI-request of Inmet St matches some entries on target addresses, meaning the corresponding `store` of this DFI-request will change the data-flow and become the most recent definition of the target address. Therefore, the matched entries are stale and are cleared from the buffer, as shown in Fig. 11(d). When the buffer is full and there is another new request, the oldest one is shifted out from the buffer as shown in Fig. 11(d) to (e). Note that the overflow does not make any false alarm since this only reduces the opportunity of pruning redundant DFI processing.

Fig. 12 shows the load pruning buffer design, whose core part is a shift register-like structure realized by a chain of D flip-flops (DFFs). Each circle with "=" is a pair of two comparators, which compare the IDs and the target addresses, respectively. Each comparator outputs `1` if two IDs (or two target addresses) are the same. "S/L" equals `0/1` if the input DFI-request's corresponding instruction is a `store/load`. The "S/L" is used as the higher control bit of the multiplexer. For each pair of comparators, the results of ID and target address comparison are passed to an AND gate, and all the results of such AND gates are passed to a tree of OR gates. If any port Q of the DFFs has the same ID and target address as the input, the output "Redundant" would be `1`, indicating that it is redundant to verify this DFI-request, which can be simply dropped. If "S/L" is 1 (for a `load`), in each clock cycle, one input can be pushed into the leftmost DFF and all the other data at port Qs are shifted right by 1 if the corresponding instruction of the new DFI-request is `load`. However, if "S/L" is 0 (for a `store`), when the port Q of one DFF has the same target address compared with the input, this DFF is reset, which is the procedure in Fig. 11(c)-(d). Otherwise, the DFF remains the same value.

5.4 Dedicated Cache

According to the analysis in the section of experiments, in Fig. 16, one can observe that most of the performance overhead of RVDFI is from memory access. Note that in software-based DFI, most of the performance overhead is due to DFI checking but not memory access [23]. It is because RVDFI has moved the DFI checking to RoCC, which is specialized for DFI verification, thereby greatly reducing the checking overhead. Therefore, memory access changes to be the top performance overhead factor.

Although RoCC can access level-1 (L1) Dcache to reduce the memory access latency, increasing the size of the Dcache is ineffective, since it can interfere with the memory accesses of the core and affect performance. Consequently, as shown in Fig. 8, level-0 (L0) dedicated caches backed by the L1 Dcache are proposed for RoCC to mitigate the Dcache access contention with the processor core. The effectiveness of our L0 dedicated caches is shown in Section 6.3, where even using a larger Dcache still has almost 50% performance overhead than using the dedicated caches with a much smaller cache size in total.

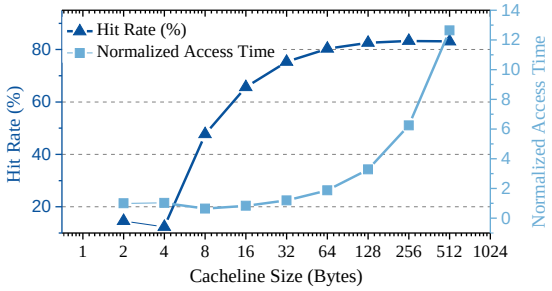


Fig. 13. The relationship of the dedicated cache's cacheline size, the access time, and the hit rate, for SPEC CPU 2006 *429.mcf* benchmark.

As shown in Fig. 8, we added three caches inside RoCC for RDTable, RDSmap, and RDSet, respectively. Note that RDSmap and RDSet cache are read-only since the two pieces of information are static, while RDTable cache involves both read and write since it contains dynamic information. For RDTable cache, write non-allocate and write through policy is adopted for cache write. The cacheline size of each dedicated cache is also an important design parameter for caching performance. We tested the relationship between the cacheline size, the access time, and the hit rate of RDTable using the *429.mcf* benchmark, when the cache size is fixed. The results are shown in Fig. 13, and the access time is normalized to the access time of the 2-byte cacheline configuration. It is shown that the access time is the least when the cacheline is 8 bytes, which is the datapath width between the L0 dedicated caches and the L1 Dcache. The reason is, if the cacheline size is too small, cache hit rate can decrease, as shown in Fig. 13. If the cacheline size is too large, during one cache miss, RoCC needs to access the Dcache multiple times to fill the cacheline, which can increase the miss penalty. Therefore, we choose 8 bytes (64 bits) as the cachline size of each dedicated cache.

Besides, since the cacheline of the RDSet cache contains four 16-bit IDs, DFI checking can be parallelized. When there is a `load`, DFI verification needs to perform the comparison of the ID of the latest `store` writing to the loaded data and the IDs in the `load`'s RDSet. Instead of reading one ID each time, RoCC can read 4 IDs and execute 4 comparisons in parallel, further reducing performance overhead.

Miss status holding registers (MSHRs) can also help reduce performance overhead. Since both the core and RoCC can access the Dcache, we add MSHRs to the Dcache, which provides the opportunities for the core (and RoCC) to access the Dcache when RoCC (and the core) is waiting for the response from memory under a cache miss in the Dcache.

6 Experimental Results

6.1 Experiment Setup

RVDFI is developed based on the Rocket Chip, which is a RISC-V based SoC generator [7]. The full system is based

TABLE 1
Summary of code changes for RVDFI.

	Language	Lines of Code
Static Analysis Tool	C	∼500
Instrumentation Tool	Python	∼4800
Linux System	C	4
Rocket Chip	Chisel	∼1700

on the Freedom project [44]. The 64-bit core of RVDFI has a 5-stage pipeline, with 16KB instruction cache and 16KB data cache. The memory size of RVDFI is 2GB. The system is prototyped on the HyperSilicon VeriTiger-H4000T FPGA platform with a Xilinx Virtex UltraScale XCVU440FLGA2892 FPGA. We use LLVM [41] for software program compilation and SVF [42] for static analysis. A summary of code changes in this work is shown in Table 1.

For security analysis, we use RIPE suite [19], Heartbleed attack [45], the attack code for Nullhttpd [20] to evaluate RVDFI. In addition, we use the SPEC CPU2006 benchmark suite for performance evaluation [24].

6.2 Security Analysis

In this section, we analyze the security of RVDFI. The experiments of control-data attacks, such as return-oriented programming (ROP) and jump-oriented programming (JOP) (indirect branches modification), and non-control-data attacks are discussed.

6.2.1 Control-Data Attacks

RIPE suite [19] is the dominant benchmark of control-data attacks. As discussed in Section 2.1, control-flow attacks can also be identified by DFI since these attacks need to modify the data in the memory. In the experiments, we tested 156 attacks including return-oriented programming (ROP) attacks and jump-oriented programming (JOP) attacks. These attacks change the targets of the indirect branches (such as function pointers), or the return address stored on the stack, to tamper with the control-flow. Results show that RVDFI can detect all the attacks. Besides, we modified RIPE to disable the activation of the attacks, and RVDFI does not report false alarms in this case.

6.2.2 Non-Control-Data Attacks

We also tested two kinds of non-control-data attacks, one is for data leaks while another is for illegal data modification.

Heartbleed (CVE-2014-0160) [21] is a vulnerability in the OpenSSL cryptography library. When a message, including the payload and the length of the payload, is sent to a server, the server echoes back the message with the claimed length. However, it does not check if the actual payload length is the same as the claimed one. As such, an attacker may send a message claiming a length that is larger than the actual payload length. Then, the server sends back not only the original payload but also some additional data, which might be private sensitive data, to fulfill the claimed length. Consequently, the sensitive data is stolen by the attacker. We use the proof-of-concept code based on [45] for the attack, which is successfully detected by RVDFI as the data to be loaded for sending back cannot be most recently written by an instruction not from the sender. An attack-free transaction, where the actual payload length conforms to the claimed one, is also tested and no false alarm is reported.

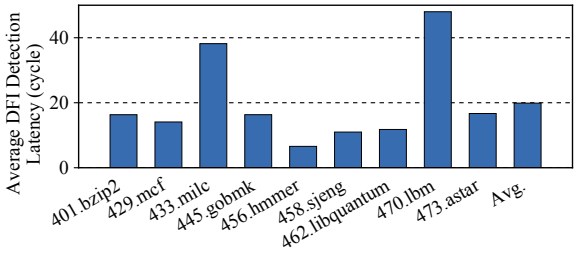


Fig. 14. The average DFI detection latency of each benchmark.

Nullhttpd is a HTTP server that has a heap overflow vulnerability (CVE-2002-1496) [20]. If the server receives a POST request with negative content length L, it should not process the request. However, the server continues to process and allocates a buffer of $L + 1024$ bytes, which is less than 1024 bytes. Later, the server writes data of 1024 bytes into the buffer, and therefore buffer overflow occurs. The experiment shows that RVDFI successfully detects such a buffer overflow. When some `load` instructions attempt to access the data written by overflow, it is found that the data is not written by any instructions in the RDSet of the `load` instruction. An experiment is also conducted to confirm that RVDFI does not produce false alarms in this context.

6.2.3 Detection Latency

Another metric for evaluating security is the detection latency. The latency is defined as the time interval between the moment each DFI-request containing the DFI checking task (by Inmet Ld, Inmet Ret or Inmet Lib) is raised and the moment its processing is finished, excluding the time when the core is stalled. The latency indicates the cycles that can be used by the core to execute instructions during the DFI checking. The results are shown in Fig. 14, which shows that RVDFI only incurs an average 20-cycle latency. This is short enough to prevent an effective attack from being successfully executed before the DFI violation is detected. Besides, the latency varies across different benchmarks, which is not directly related to the performance overhead. These variations are related to the benchmark characteristics and may result from different memory instruction densities and different RDSet sizes. Note that RVDFI defends against software vulnerabilities and enforces DFI for *committed* instructions. The detection window has little to no impact on transient execution attacks. The execution of the instrumented DFI related instructions may in return reduce the window of transient execution attacks.

6.3 Performance Overhead

Table 2 shows the performance overhead. The baseline of Columns NR1 and NR2 (Soft-NoMSHR and RVDFI-NoMSHR) is running the uninstrumented target program on the unmodified Rocket Chip without MSHR, and the baseline of Columns 1–9 (Soft-MSHR, Basic-RVDFI, partially enhanced RVDFI variants and fully enhanced RVDFI) is running the uninstrumented target program on the unmodified Rocket Chip with MSHR. The main result of the proposed RVDFI is at Column 9, with each dedicated cache 8KB, and 24KB in total. As shown in Column 9, RVDFI only incurs 17.8% performance overhead, while the previous complete DFI work based on software implementation at

This is a preprint. This paper has been accepted by IEEE Transactions on Computers in Dec/2021.

TABLE 2
Performance overhead of SPEC CPU 2006 benchmark and hardware resource consumption.
(†The percentage is calculated compared with Column NR1. ‡RDSet, RDSmap, and RDTable caches are implemented.)

Scheme		Soft [18]	RVDFI	Soft [18]	Partial RVDFI							RVDFI
Scheme Name		Soft-NoMSHR	RVDFI-NoMSHR	Soft-MSHR	Basic-RVDFI	RVDFI-FIFO	RVDFI-LdPr	RVDFI-RDSet	RVDFI-RDSmap	RVDFI-RDTable	RVDFI-64KB_D$	RVDFI
Column ID		NR1	NR2	1	2	3	4	5	6	7	8	9
MSHR		×		√								
FIFO		-	√	-	×	√	×	×	×	×	√	√
Load Pruning Buffer		-	√	-	×	×	√	×	×	×	√	√
Dedicated Cache		-	√‡ 24KB	-	×	×	×	RDSet 8KB	RDSmap 8KB	RDTable 8KB	Dcache +48KB	√‡ 24KB
# of LUTs		50025	62376 24.7%†	59163	63676 7.6%	63927 8.1%	66293 12.1%	65692 11.0%	65515 10.7%	65858 11.3%	67315 13.8%	71059 20.1%
# of FFs		38571	49853 29.2%†	41981	48373 15.2%	48047 14.4%	52993 26.2%	48394 15.3%	48373 15.2%	48388 15.3%	54349 29.5%	53209 26.7%
# of BRAMs		81	81	81	81	81	81	81	81	81	117(44.44%)	81
Benchmark	*401.bzip2*	244.6%	26.8%	235.6%	78.9%	77.1%	75.0%	33.1%	69.5%	69.4%	66.2%	16.2%
	429.mcf	130.1%	38.3%	119.0%	29.6%	25.8%	28.2%	22.5%	25.5%	26.4%	10.8%	17.0%
	433.milc	264.6%	31.0%	273.3%	212.5%	207.7%	169.1%	73.8%	187.3%	198.9%	101.7%	24.7%
	445.gobmk	271.0%	36.3%	276.0%	54.4%	42.6%	53.2%	43.9%	42.0%	46.7%	10.8%	26.9%
	456.hmmer	43.3%	3.9%	43.4%	6.1%	6.0%	6.1%	6.2%	5.0%	6.0%	5.8%	3.9%
	458.sjeng	181.4%	15.6%	180.2%	28.9%	24.0%	28.4%	23.4%	22.5%	23.5%	2.1%	11.9%
	462.libquantum	55.7%	20.1%	54.5%	13.1%	13.8%	13.4%	13.4%	13.4%	13.2%	12.4%	12.9%
	470.lbm	113.2%	49.5%	128.5%	36.3%	29.1%	36.4%	34.3%	31.3%	36.1%	19.4%	25.1%
	473.astar	150.9%	35.1%	186.0%	31.5%	26.7%	35.5%	33.2%	30.1%	30.6%	11.4%	21.9%
	Average	**161.6%**	**28.5%**	**166.3%**	**54.6%**	**50.3%**	**49.5%**	**31.5%**	**47.4%**	**50.1%**	**26.7%**	**17.8%**

Column 1 incurs 166.3% overhead, which is more than 9× compared to RVDFI. The effect of each enhancement is also investigated. The Basic-RVDFI implementation incurs nearly 55% performance loss (Column 2). With the FIFO introduced (Column 3), the overhead of RVDFI-FIFO can be reduced by 4.3 percentage points (pp) of the performance overhead. Besides, pruning the `load` instructions at runtime in RVDFI-LdPr (Column 4) can also reduce more than 5pp of the performance overhead. When introducing the RDSet cache in RVDFI-RDSet (Column 5), the overhead is greatly reduced to 31.5%, which proves the effectiveness of the RDSet cache. Similarly, the RDSmap cache enhancement in RVDFI-RDSmap (Column 6) and the RDTable cache enhancement in RVDFI-RDTable (Column 7) are able to reduce the overhead to 47.4% and 50.1%, respectively. The results show that each individual enhancement (Column 3–7) can effectively reduce the performance loss, which results in a low-overhead design when combining them together for a fully enhanced RVDFI (Column 9). We also remove the dedicated caches and barely increase the L1 Dcache size by 48KB (Column 8), which is 1× larger than the dedicated caches in the complete RVDFI design (Column 9). Although a larger Dcache is used, the performance overhead is even 50% worse than that of RVDFI, which demonstrates the effectiveness of the proposed dedicated L0 caches. The existence of MSHRs can also affect the experiment setup of the system and have impacts on performance, especially for RVDFI that has both the core and RoCC access L1 Dcache. Therefore, we separately conduct the experiments with MSHRs enabled and disabled. The results show that MSHRs do not affect the performance overhead of software-DFI (Column NR1). The difference between Column NR1 and Column 1 is only due to noises, indicated by the marginal variations of different benchmarks. However, MSHR-enabled RVDFI (Column 9) can improve the performance compared to its MSHR-disabled counterpart (RVDFI-NoMSHR in NR2) for all the benchmarks.This is because the non-blocking Dcache with MSHRs can eliminate the penalty of subsequent cache access under miss caused by either the core or RoCC.

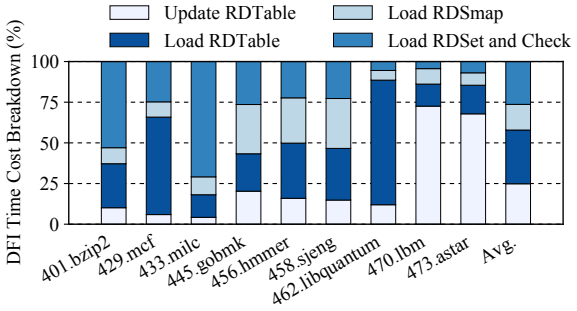


Fig. 15. The percentages of the time cost breakdowns of different DFI verification steps.

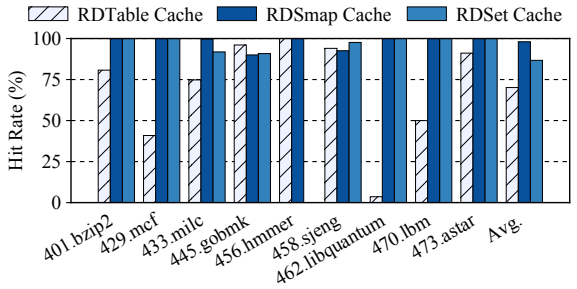


Fig. 16. The cache hit rates of different dedicated caches.

To analyze the cause of the performance overhead of RVDFI, the DFI verification time of RVDFI with all the enhancements is broken down and illustrated in Fig. 15. As shown, although varying from different benchmarks, most of the time is spent on accessing the RDTable. This is due to the relatively low hit rate of the RDTable cache as shown in Fig. 16, where the RDTable cache shows the lowest hit rate among the dedicated caches with the same cache size. For some benchmarks, such as *429.mcf*, *462.libquantum* and *470.lbm*, the hit rates of RDTable cache are lower than 50%. Therefore, RDTable accesses contribute most to the DFI verification time of RVDFI.

Besides, the time cost breakdowns of the setups with individual dedicated caches are shown in Fig. 17. Compared with Basic-RVDFI with no dedicated cache, implementing the RDSet cache inside RVDFI can greatly reduce

This is a preprint. This paper has been accepted by IEEE Transactions on Computers in Dec/2021.

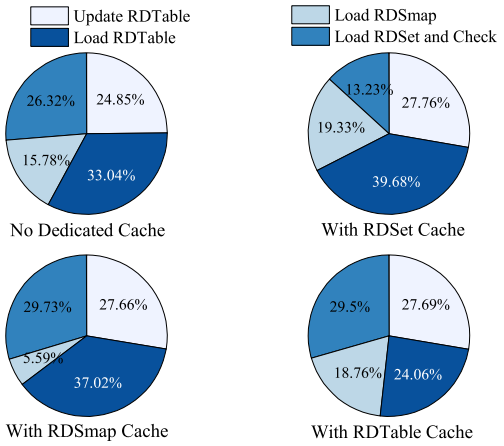


Fig. 17. The percentages of the time cost by different DFI verification steps of different setups.

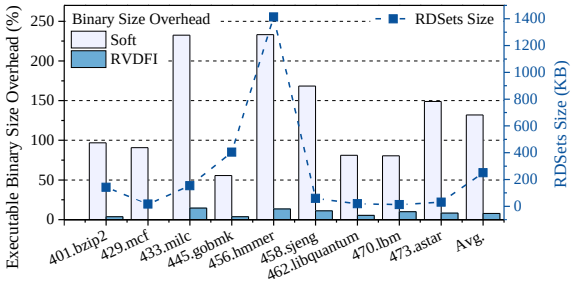


Fig. 18. The executable binary size overhead and RDSets sizes of different benchmarks.

the percentage of the time cost on loading RDSet. Besides, implementing the RDSmap or the RDTable cache can also reduce the percentage of access time of RDSmap or RDTable, respectively. However, RDTable access time is the most challenging one to reduce.

6.4 Hardware Resource and Memory Consumption

The hardware resource consumption is also evaluated and listed in Table 2. It shows that Basic-RVDFI needs 7.6% more look-up tables (LUTs) and 15.2% more flip-flops (FFs), compared with the unmodified Rocket Chip. Each enhancement costs at most around 7,000 LUTs and around 11,000 FFs. The final RVDFI implementation (Column 9) consumes 20.1% more LUTs, 26.7% more FFs. Without MSHR, the LUT and FF consumption of RVDFI are 62,376 and 49,853, which is 24.7% and 29.2% more than the original Rocket Chip without MSHR, respectively.

According to Section 4.2, the memory overhead is 50% and 25%, when the data is 4-byte and 8-byte aligned, respectively, for complete DFI. Although the memory overhead is not low, for some security critical applications such as military and finance applications, memory overhead is less critical while security is one of the top priorities. Compared with other complete DFI work such as software-DFI [18] and PIM-DFI [23], RVDFI realizes much higher performance without more memory overhead.

6.5 Binary Size Overhead and RDSets Size Analysis

Since we instrument the target program for DFI enforcement, the size of the executable binary can increase. Fig. 18 shows the binary size overhead of both software-DFI and RVDFI. It shows that the binary size overhead of software-DFI is more than 125% on average while that of RVDFI is negligible, because the instrumentation of RVDFI only adds 1 instruction for each memory access (or call, return) instruction and some of them are further pruned by offline optimization, while software-DFI needs much more computations including comparison, addition, shifting, and branching. Fig. 18 also depicts the size of the RDSets (including the RDSmaps) of each benchmark. The average size is only around 200KB and the maximum is around 1400KB.

6.6 Comparison with Previous Hardware-based DFI

In this subsection, we compare RVDFI with famous previous hardware-based DFI enforcement. The comparison is shown in Table 3, and the details are discussed in the following:

TABLE 3
Comparison with previous hardware-based DFI enforcement.
†The granularity compared with complete DFI.
‡The result is not reported in the corresponding reference.

Method	DFI Enforcement Completeness	Performance Overhead	Hardware Resource Consumption		Memory Overhead	
			LUT	FF	4B aligned	8B aligned
HDFI [9]	1/32768†	<2%	-‡	-‡	3.1%	1.6%
TMDFI [22]	1/256†	~39%	-‡	-‡	25.0%	12.5%
PIM-DFI [23]	Complete	~36%	238,333	39,994	50.0%	25.0%
RVDFI	**Complete**	**~18%**	**11,896**	**11,228**	**50.0%**	**25.0%**

HDFI [9]: Compared with RVDFI, although the performance overhead of HDFI is lower (<2%) by using a 1-bit tag for each data, its security strength is much weaker and can be easily attacked by the attack model discussed in work [23]. Since a complete DFI implementation such as RVDFI uses 16 bits to separate the memory regions, the memory overhead of HDFI is 1/16 of that of RVDFI, but the data region protection of RVDFI is 32768× finer-grained than HDFI.

TMDFI [22]: For TMDFI, RVDFI is 256× finer-grained than TMDFI, since TMDFI can only separate the memory region into 256 regions. Although TMDFI consumes 1/2 less memory overhead, it incurs a much higher performance overhead (39%) than RVDFI (18%).

PIM-DFI [23]: For security, both schemes realize complete DFI. However, PIM-DFI has 36.4% performance overhead while RVDFI only has 17.8% overhead. In terms of memory usage, both schemes incur the same overhead as that of software-DFI [18]. PIM-DFI's most significant disadvantage is that it requires either a PIM processor or a normal CPU core, in addition to an extra 238,333 LUTs and 39,994 FFs overhead when implementing at the same platform as RVDFI. In contrast, RVDFI only needs 11,896 LUTs and 11,228 FFs to realize DFI verification, which is a magnitude fewer. Therefore, RVDFI is more efficient in both performance and hardware resources than PIM-DFI. Another difference is that PIM-DFI is evaluated using simulation while RVDFI is a real hardware prototype.

6.7 Sensitive Study

We also studied the detailed effectiveness of each enhancement by varying their sizes. Fig. 19 shows the normalized performance overhead compared to the baseline, which is an unmodified Rocket Chip with MSHR (Column 1 in Table. 2). It shows that by adding the enhancements and the hardware resources, performance overhead can be mitigated. Specifically, increasing the RDSet cache size has the most positive impact.

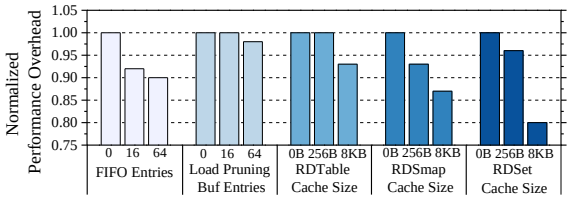


Fig. 19. The effectiveness of the enhancements.

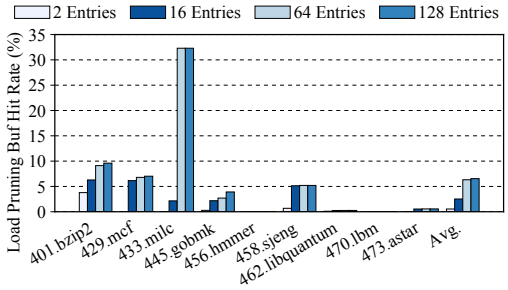


Fig. 20. The load pruning buffer hit rates under different buffer sizes.

For dynamic redundant load pruning, we use "hit rate" to represent the ratio of the number of the redundant DFI-requests identified by the load pruning buffer over the total number of Inmet Ld DFI-requests. The pruning hit rate increases as the load pruning buffer size increases, as shown in Fig. 20. After the point where the load pruning buffer has 64 entries, the hit rate only increases marginally. Therefore, the load pruning buffer is implemented with 64 entries in the main RVDFI results (Column 9 of Table 2). Besides, the hit rates vary from benchmark to benchmark. Although the average hit rate is around 6%, the effectiveness of the load pruning buffer is relatively higher for some benchmarks, such as *433.milc* and *401.bzip2*. Therefore, the dynamic redundant load pruning can increase the chance to reduce the performance overhead for certain programs.

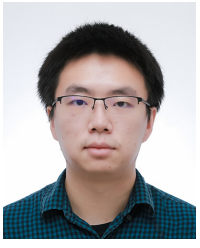
7 Conclusions

In this paper, a secure RISC-V architecture named RVDFI is proposed, which enables hardware-assisted complete DFI verification through specialized DFI verification architecture design. The system stacks consisting of compilation, customized instruction instrumentation, and operating system are augmented to enable a secure DFI-capable RISC-V SoC. In addition, several enhancements are proposed to improve the security and reduce the performance overhead, including the DFI request FIFO, the load pruning buffer, the dedicated DFI caches, etc. The evaluation shows that RVDFI not only realizes complete DFI that can detect both control-data and non-control-data attacks, but also is practical due to its low performance overhead. In summary, RVDFI is the first RISC-V architecture with complete DFI verification that incurs only 17.8% performance overhead.

References

[1] Verilog to Routing, https://verilogtorouting.org/, 2012.
[2] Icarus Verilog, http://iverilog.icarus.com/, 1998.
[3] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister, Y. S. Shao, B. Nikolic, I. Stoica, and K. Asanovic, "Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures," *arXiv preprint*, 2019.
[4] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From High-Level Deep Neural Models to FPGAs," *IEEE/ACM International Symposium on Microarchitecture*, pp. 1–12, 2016.
[5] RISC-V: The Free and Open RISC Instruction Set Architecture, https://riscv.org/, 2010.
[6] NVDLA, http://nvdla.org/, 2018.
[7] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep., 2016.
[8] HammerBlade RISC-V Manycore, https://riscv.org/news/2020/07/the-hammerblade-risc-v-manycore-a-programmable-scalable-risc-v-fabric-michael-taylor-and-max-h-ruttenberg-fosdem/, 2020.
[9] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-Assisted Data-Flow Isolation," *IEEE Symposium on Security and Privacy*, pp. 1–17, 2016.
[10] RISC-V in NVIDIA, https://riscv.org/wp-content/uploads/2017/05/Tue1345pm-NVIDIA-Sijstermans.pdf, 2017.
[11] T. Fritzmann, G. Sigl, and J. Sepúlveda, "RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 4, pp. 239–280, 2020.
[12] A. Garofalo, G. Tagliavini, F. Conti, D. Rossi, and L. Benini, "XpulpNN: Accelerating Quantized Neural Networks on RISC-V Processors Through ISA Extensions," *Design, Automation and Test in Europe Conference*, pp. 186–191, 2020.
[13] Y. Zhang, B. Du, L. Zhang, and J. Wu, "Parallel DNN Inference Framework Leveraging a Compact RISC-V ISA-Based Multi-Core System," *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, p. 627–635, 2020.
[14] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor, "The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips," *IEEE Micro*, vol. 38, no. 2, pp. 30–41, 2018.
[15] Intel CET, https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2019.
[16] AMD Secure Encrypted Virtualization, https://www.amd.com/en/processors/amd-secure-encrypted-virtualization, 2020.
[17] CoreSight Program Flow Trace, http://infocenter.arm.com/help/topic/com.arm.doc.ihi0035b/IHI0035B_cs_pft_v1_1_architecture_spec.pdf, 2011.
[18] M. Castro, M. Costa, and T. Harris, "Securing Software by Enforcing Data-Flow Integrity," *Symposium on Operating Systems Design and Implementation*, pp. 147–160, 2006.
[19] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime Intrusion Prevention Evaluator," *Computer Security Applications Conference*, pp. 41–50, 2011.
[20] Null HTTPd Remote Heap Overflow Vulnerability, https://www.securityfocus.com/bid/5774.
[21] The Heartbleed Bug, http://heartbleed.com/.
[22] T. Liu, G. Shi, L. Chen, F. Zhang, Y. Yang, and J. Zhang, "TMDFI: Tagged Memory Assisted for Fine-Grained Data-Flow Integrity Towards Embedded Systems Against Software Exploitation," *IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ IEEE International Conference On Big Data Science And Engineering*, pp. 545–550, 2018.
[23] L. Feng, J. Huang, J. Huang, and J. Hu, "Toward Taming the Overhead Monster for Data-Flow Integrity," *arXiv preprint*, 2021.
[24] SPEC CPU 2006 Benchmark, https://www.spec.org/cpu2006/.
[25] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow Integrity," *ACM Conference on Computer and Communications Security*, pp. 340–353, 2005.
[26] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.
[27] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and

Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," *IEEE Symposium on Security and Privacy*, pp. 1–19, 2019.
[28] Y. Park, W. Kwon, E. Lee, T. J. Ham, J. H. Ahn, and J. W. Lee, "Graphene: Strong yet Lightweight Row Hammer Protection," *IEEE/ACM International Symposium on Microarchitecture*, pp. 1–13, 2020.
[29] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," *IEEE/ACM International Symposium on Microarchitecture*, pp. 428–441, 2018.
[30] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri, T. Guarino, A. Joshi, M. Oskin, and M. B. Taylor, "BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 93–102, 2020.
[31] RISCV-DV, https://github.com/google/riscv-dv, 2019.
[32] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing Memory Error Exploits with WIT," *IEEE Symposium on Security and Privacy*, pp. 263–277, 2008.
[33] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee, "Enforcing Kernel Security Invariants with Data Flow Integrity," *Network and Distributed System Security Symposium*, pp. 1–15, 2016.
[34] X. Ge, W. Cui, and T. Jaeger, "GRIFFIN: Guarding Control Flows Using Intel Processor Trace," *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 585–598, 2017.
[35] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, "Using CoreSight PTM to Integrate CRA Monitoring IPs in an ARM-Based SoC," *ACM Transactions on Design Automation of Electronic Systems*, vol. 22, no. 3, pp. 52:1–52:25, 2017.
[36] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," *IEEE Symposium on Security and Privacy*, pp. 20–37, 2015.
[37] T. Zhang, D. Lee, and C. Jung, "BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free," *International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 631–644, 2019.
[38] Y. Kim, J. Lee, and H. Kim, "Hardware-based always-on heap memory safety," *IEEE/ACM International Symposium on Microarchitecture*, pp. 1153–1166, 2020.
[39] L. Delshadtehrani, S. Canakci, B. Zhou, S. Eldridge, A. Joshi, and M. Egele, "PHMon: A Programmable Hardware Monitor and Its Security Use Cases," *USENIX Security Symposium*, pp. 807–824, 2020.
[40] N. Joly, S. ElSherei, and S. Amar, "Security Analysis of CHERI ISA," https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf, 2020.
[41] LLVM, https://llvm.org/.
[42] Y. Sui and J. Xue, "SVF: Interprocedural Static Value-flow Analysis in LLVM," *International Conference on Compiler Construction*, pp. 265–266, 2016.
[43] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," *ACM conference on Computer and communications security*, pp. 552–561, 2007.
[44] Freedom, https://github.com/sifive/freedom, 2016.
[45] The Source Code for Triggering Heartbleed Bug, https://github.com/mykter/afl-training/tree/master/challenges/heartbleed.



Lang Feng Lang Feng received his B.E. degree in electronic science and technology (microelectronic technology) from University of Electronic Science and Technology of China, Chengdu, China, in 2016, and his Ph.D. degree in computer engineering from Texas A&M University, College Station, in 2020. In Nov. 2020, he joined the School of Electronic Science and Engineering of Nanjing University, where he is an associate research fellow. His research interests are computer architecture, security, etc.

Jiayi Huang (Member, IEEE) received the BEng degree in information and communication engineering from Zhejiang University, China, in 2014, and the PhD degree in computer engineering from Texas A&M University, in 2020. He is currently a postdoctoral researcher with the Department of Electrical and Computer Engineering, UC Santa Barbara. His research interests include computer architecture, computer systems, and security. He is a member of the ACM and the IEEE Computer Society.

Luyi Li Luyi Li is currently working towards the B.E. degree with integrated circuit design and integrated system from Nanjing University, Nanjing, China. His research interests focus on hardware acceleration, computer architecture, security, etc.

Haochen Zhang Haochen Zhang is currently working towards the B.E. degree with integrated circuit design and integrated system from Nanjing University, Nanjing, China. His research interests are computer architecture, security, etc.

Zhongfeng Wang Zhongfeng Wang (Fellow, IEEE) received both B.E. and M.S. degrees from Tsinghua University. He obtained the Ph.D. degree from the University of Minnesota, Minneapolis, in 2000. He has been working for Nanjing University, China, as a Distinguished Professor since 2016. Previously he worked for Broadcom Corporation, California, from 2007 to 2016 as a leading VLSI architect. Before that, he worked for Oregon State University and National Semiconductor Corporation.

Dr. Wang is a world-recognized expert on Low-Power High-Speed VLSI Design for Signal Processing Systems. He has published over 200 technical papers with multiple best paper awards received from the IEEE technical societies. In the current record, he has had many papers ranking among top 25 most (annually) downloaded manuscripts in IEEE Trans. on VLSI Systems. In the past, he has served as Associate Editor for IEEE Trans. on TCAS-I, T-CAS-II, and T-VLSI for many terms.