Toward Taming the Overhead Monster for Data-flow Integrity

LANG FENG, Nanjing University, China and Texas A&M University, USA
JIAYI HUANG, University of California, Santa Barbara, USA
JEFF HUANG and JIANG HU, Texas A&M University, USA

Data-Flow Integrity (DFI) is a well-known approach to effectively detecting a wide range of software attacks. However, its real-world application has been quite limited so far because of the prohibitive performance overhead it incurs. Moreover, the overhead is enormously difficult to overcome without substantially lowering the DFI criterion. In this work, an analysis is performed to understand the main factors contributing to the overhead. Accordingly, a hardware-assisted parallel approach is proposed to tackle the overhead challenge. Simulations on SPEC CPU 2006 benchmark show that the proposed approach can completely enforce the DFI defined in the original seminal work while reducing performance overhead by 4×, on average.

CCS Concepts: • **Computer systems organization → Architectures**; • **Security and privacy → Systems security;**

Additional Key Words and Phrases: Data-flow integrity, processing in memory

ACM Reference format:
Lang Feng, Jiayi Huang, Jeff Huang, and Jiang Hu. 2021. Toward Taming the Overhead Monster for Data-flow Integrity. *ACM Trans. Des. Autom. Electron. Syst.* 27, 3, Article 25 (November 2021), 24 pages.
https://doi.org/10.1145/3490176

1 INTRODUCTION

Data-Flow Integrity (DFI) is a regulation to ensure that data to be accessed are written by legitimate instructions [7]. As such, DFI enforcement can identify unwanted data modifications that are not consistent with the programmer's intention. It can detect a wide variety of security attacks, including control data attacks such as **Jump-Oriented Programming (JOP)** [6] and **Return-Oriented Programming (ROP)** [27], and non-control data attacks such as Heartbleed [34] and the heap overflow attack to Nullhttpd [23]. As a large number of software attacks rely on data modifications, DFI is a single principle that is effective for many different attack scenarios, including future potential ones. In fact, its defense scope is a much bigger superset of **Control-Flow Integrity (CFI) [1]**, which is another well-known software security approach.

This work is partially supported by NSF (CNS-1618824) and NSF (CCF-1815583).
Authors' addresses: L. Feng, Nanjing University, 163 Xianlin Road, Qixia District, Nanjing, China, 210023, Texas A&M University, 400 Bizzell Street, College Station, TX, 77840; email: flang@nju.edu.cn; J. Huang, University of California, Santa Barbara, CA, 93106; email: jyhuang@ucsb.edu; J. Huang and J. Hu, Texas A&M University, 400 Bizzell Street, College Station, TX, 77840; emails: {jeffhuang, jianghu}@tamu.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Association for Computing Machinery.
1084-4309/2021/11-ART25 $15.00
https://doi.org/10.1145/3490176

The concept of DFI was introduced in 2006 by the seminal work cited in Reference [7] and has received a great amount of attention thereafter due to its potential of being a powerful security measure. To differentiate from the approaches in terms of granularity, the DFI enforcement in Reference [7] is named **complete DFI** in this article. However, a complete DFI enforcement as in Reference [7] incurs more than 100% performance overhead even though several optimization techniques have been applied. Indeed, the huge overhead seems inevitable, as every data access needs to be examined. Due to this intrinsic difficulty, there have been few follow-up works on DFI despite its widely recognized importance. This is in sharp contrast to CFI [1], which has much more published studies [10–13, 19, 21, 39].

The few later works on DFI [2, 20, 29, 30] reduce the overhead by exploiting partial DFI, whose criteria are substantially lower than the original DFI definition [7]. The **Hardware-assisted Data-Flow Isolation (HDFI)** [30] is one example. It partitions data into two regions and only requires that data to be read and written must be consistent in the same region. In other words, it reports a violation only when data intends to be in one region but is actually written by an instruction for another region. Although its overhead is very small, the enforcement granularity is very coarse and may miss attacks that mingle different data within the same region, By contrast, a complete DFI [7] can isolate data among dozens of thousands of regions, i.e., a resolution >30,000× higher than HDFI. Therefore, the security price that HDFI paid for its overhead reduction can be very high.

Enforcing the complete DFI [7] with practically acceptable overhead is a huge challenge. Different from most of existing overhead reduction techniques [2, 20, 29, 30], which rely on lowering the DFI criterion, we pursue a new approach that exploits additional hardware while the original DFI [7] can still be completely enforced. As hardware cost becomes increasingly affordable along with the progress of semiconductor technology, reducing performance overhead at the expense of extra hardware is a promising direction.

We first conduct an extensive performance analysis of DFI. Surprisingly, the frequent DFI data access does not lead to frequent memory access and thus, memory access is not a bottleneck, but the other computations involved in DFI enforcement contribute the most to the overhead. We propose a parallel and asynchronous approach, where most of the DFI computations are performed in another processor core. However, a straightforward software-based parallel computing still experiences huge overhead resulted from runtime information collection and communications with the other processor core. Therefore, we develop a new hardware technique to further trim down the overhead. This hardware-assisted parallel approach also includes new software instrumentation techniques, lossless data compression, and runtime optimization techniques. For the ease of deployment, we intend to minimize the dependence on computing infrastructure changes. Except the necessary circuits and software instrumentation, our approach does not rely on using new instructions or OS modifications.

Overall, the proposed approach reduces performance overhead from 161% of Reference [7] to an average of 36% on the same SPEC CPU 2006 benchmarks. As it is a complete DFI enforcement, it can detect a wide range of security attacks and cover cases that cannot be handled by the previous low-overhead methods [2, 20, 29, 30]. Our approach provides a solution with a security-overhead tradeoff in complement to existing methods [2, 20, 29, 30]. A brief comparison with existing methods is summarized in Table 1. The contributions of this work are as follows:

- An overhead breakdown analysis is performed to understand the main performance bottlenecks in software DFI.
- This is the first hardware approach for complete DFI enforcement, to the best of our knowledge.
- Two variants of the proposed approach are investigated, one for **Processing-In-Memory (PIM)** and the other for **Chip Multiprocessor (CMP)**.

Table 1. Comparison between Our Work and Others

Method[1]	Performance Overhead	DFI Enforcement Completeness	Approach	New Instruction	OS Change
SW DFI [7]	161%	Complete	SW	×	×
KENALI [29]	7%–15%	Partial	SW	×	√
WIT [2]	7%	Partial	SW	×	×
CHERI [36]	5%–20%	Partial	HW	√	√
TMDFI [20]	39%	Partial	HW	√	×
HDFI [30]	<2%	Partial	HW	√	√
Our work	**36%**	**Complete**	**HW**	×	×

- The tradeoff between DFI violation detection latency and performance overhead is studied.
- Our approach achieves about 4× overhead reduction, which is a major progress for complete DFI since 2006.

The rest of this article is organized as follows: Section 2 introduces the background. The threat model and system assumptions are introduced in Section 3. The related work is briefly reviewed in Section 4, and characterizations are performed in Section 5 to uncover the key factor for the performance overhead. Section 6 provides an overview of our approach. Next, Sections 7, 8, and 9 describe the three critical parts of our design. The experiment results are shown in Section 10. Section 11 discusses the tradeoff of our approach and broader applications. Finally, we conclude in Section 12.

2 BACKGROUND ON DATA-FLOW INTEGRITY

Data-flow integrity requires that data to be loaded from memory can only be stored by legitimate instructions that are consistent with the programmer's original intention [7]. Every instruction in a program is assigned a numerical **identifier** through automatic code instrumentation. If the data loaded by instruction `A` was most recently stored by instruction `B`, then the **reaching definition** of `A` is `B` and is represented by the identifier of `B`. Each instruction that can load data from memory has its own **Reaching Definition Set (RDS)**, which consists of all the allowed reaching definitions of this instruction. A static software analysis can be performed for a program to obtain the RDSs for all relevant instructions. In the example of Figure 1, "`store x y`" means storing variable `x` at address `y`, "`load x y`" is to load the data at address `y` to variable `x`, "`cmp x y`" is to compare the values of variable `x` and variable `y`, and "`jne label`" implies a conditional branch to the location marked by `label`, if the values in the previous comparison are different. If the identifier of each instruction is the same as its line number, then the RDS of instruction "`load x3 addr1`" (line 6) is {5}, and the RDS of instruction "`load x4 addr1`" (line 8) is {1, 5}. DFI requires that all the instructions that can load data from memory are consistent with their RDSs, i.e., when executing an instruction `A` that loads data from memory, the data should indeed be most recently stored by one of the instructions in the RDS of `A`. Hence, the identifier of the latest instruction that stores a data needs to be tracked for the data. Such identifiers for all data form a **Reaching Definition Table (RDT)**.

In summary, given a program, the information required for DFI enforcement and their locations are as follows:

[1] All the listed works need code static analysis and instrumentation.

```
1   store x1 addr1
2   store x2 addr2
3   cmp x1 x2
4   jne label
5   store x2 addr1
6   load x3 addr1      // RDS: {5}
7   label:
8   load x4 addr1      // RDS: {1, 5}
```

Fig. 1. A code example for illustrating DFI, where the line numbers are used as identifiers (IDs) for the corresponding instructions for simplicity.

(1) **RDS (Reaching Definition Set)** for all `load` instructions in the program. This information never changes throughout the program execution, and it can be loaded into the memory once in the beginning.
(2) **RDT (Reaching Definition Table)**. This information changes dynamically during a program execution. It is stored in the memory and maintained by the computing resource for DFI enforcement.
(3) Target instruction information. A **target instruction** is an instruction in the program to be enforced for DFI. Mainly two types of instructions are involved: `load` instructions for which DFI enforcement is performed and `store` instructions that affect RDT. These two pieces of information change at runtime and need to be obtained by the computing resource for DFI enforcement. It consists of the following components:
   - Instruction identifier.
   - Instruction type: either `load` or `store`.
   - Target address of `load` or `store`.

After all the three kinds of information are obtained, DFI enforcement can be performed.

3 THREAT MODEL AND SYSTEM ASSUMPTIONS

Following the typical threat model of most related work, it is assumed that the attackers are able to leverage the possible software vulnerabilities to corrupt any locations in the memory. Once the attack is successfully performed, the attackers can take any desired actions. The software vulnerabilities may exist in any places of the user programs. Note that any attacks that leverage hardware vulnerabilities are not considered in our threat model. For example, rowhammer [16] and cache side-channel attacks [17] are out of the scope. Meanwhile, the attacks that maliciously modify the instructions can be simply protected by Write **XOR Execute (W⊕X)** technique [38], and they are not included in this work.

For our system, we assume that all the software can be static analyzed, and the static analysis is assumed to be accurate. The static analysis tool can provide the software programs' RDSs of all the instructions that can load data. The DFI software toolchain and the hardware of our system are assumed to be trusted and bug-free.

Under this threat model, DFI is a superset of **Control-Flow Integrity (CFI)** [1], which only regulates instruction flow transitions toward target addresses conforming to the original design intention. Attackers have to modify the control data, such as the target address for an indirect branch, to change the control flow. By protecting all the data, DFI can also prevent control-flow attacks. Additionally, DFI can protect non-control data that cannot be covered by CFI.

4 PREVIOUS WORK

The concept of DFI was proposed in Reference [7] in 2006. This work also provides a software implementation technique and optimization techniques for overhead reduction. Although the DFI

enforcement procedure is simple, its performance overhead is intrinsically huge as the enforcement needs to be conducted for tremendous data.

The few later previous works [2, 20, 29, 30, 36] achieved much lower overhead by focusing on partial DFI. The work of Reference [29] is restricted to only certain selected data for kernel software. One of its main contributions is the techniques on how to select data to be protected. Although its performance overhead is only 7%–15%, its application is restrictive and misses many attacks at user programs. For example, Nullhttpd [23], Heartbleed [34], and data-oriented programming [14] are conducted at user level and thus not handled by this technique. By contrast, our approach covers both kernel- and user-level programs.

While DFI involves both `load` and `store` instructions, the scope of **Write Integrity Testing (WIT)** [2] is restricted to `store`. It requires that each `store` instruction can only write to certain data objects, and each indirect call can only call certain functions. Although its overhead is at most 25%, it does not cover `load` instructions. Thus, an unsafe `load` instruction may read more bytes than the programmer's intention, and consequently information leak may occur, e.g., Heartbleed [34] is an attack that WIT would fail to detect. Another related work, HardScope [24], also restricts the memory access behaviors of each function with different memory access rules, which only allow certain data to be accessed. Thus, it prevents the instructions in the unprivileged functions from accessing the privileged data. While HardScope has low performance overhead, it does not distinguish each `store` and `load`, thus, it is less fine-grained than complete DFI.

Data isolation is another approach to protecting data with relatively low overhead. A hardware solution for data-flow isolation, called HDFI, is proposed in Reference [30]. It designates two data regions, a sensitive one and a non-sensitive one. A 1-bit tag is employed to tell the region that a data belongs to. Instruction set is modified such that the tags can be read and set. Moreover, the processor hardware and the operating system also need changes. If data belongs to one region, then it cannot be written by an instruction for the other region. Although the isolation helps security, it cannot handle the case where `load/store` instructions for different data of the same region are mingled. Consider the example in Figure 2, where input data are first written into `u0` and `u1` at lines 10 and 11. Later, the data are copied to buffers at lines 13–15. If there is buffer overflow when executing line 10, i.e., the input data size exceeds 256, then offset `u0->off` is modified unintentionally. Then, line 13 may copy `user0`'s data to other users' buffers through the modified `u0->off`. Meanwhile, `user1` can write to `user2`'s buffer at line 14 in the same way. As HDFI partitions data into only two regions, one of the user pairs—(`user0`, `user1`), (`user0`, `user2`), or (`user1`, `user2`)—must share the same region. Consequently, the former user in a pair can attack the latter in the pair without being detected by HDFI. By contrast, to a certain degree, the original DFI [7] can be regarded as data isolation among individual instructions. If 16 bits are used for each instruction identifier, then it is equivalent to isolation among up to 2^{16} regions. Compared to the only two regions of HDFI [30], the resolution of the original DFI is $2^{15} = 32{,}768$ times higher. Thus, its low overhead of <2% comes with the price of very coarse-grained security resolution. Similar to HDFI, TMDFI [20] also enforces DFI by a tag-based approach, and it results in 39% overhead. However, TMDFI only uses 8 bits for the tag and can only isolate $2^8 = 256$ regions, which are much coarser grained than the resolution of our approach. For a typical program, such as each benchmark in SPEC CPU 2006, it needs at least >1,000 and sometimes >10,000 identifiers, which cannot be isolated by 256 regions, so TMDFI is not sufficient to support complete DFI for a typical program, while our approach is.

There are also other tag-based isolation techniques. The work of Reference [8] uses 1-bit tag for each word of data to indicate its integrity level in Biba's low-water-mark integrity policy [5], which requires that an instruction can only modify data with integrity level no higher than that of the instruction. In Reference [8], processor hardware is modified to enforce this policy for control data

```c
struct vuln{
  char data[256];
  int off=0;
  int size=0;
}*u0, *u1, *u2;
/* ============== */
char user0_buffer[256];                              //user0's buffer for storing u0's data
char user1_buffer[256];                              //user1's buffer for storing u1's data
char user2_buffer[256];                              //user2's buffer for storing u2's data
read_user_input(u0, user0_input);                    //store user0's input to u0
read_user_input(u1, user1_input);                    //store user1's input to u1
...
memcpy(user0_buffer+u0->off, u0->data, u0->size); //copy u0's data to user0's buffer
memcpy(user1_buffer+u1->off, u1->data, u1->size); //copy u1's data to user1's buffer
memcpy(user2_buffer+u2->off, u2->data, u2->size); //copy u1's data to user1's buffer
```

Fig. 2. An example of vulnerability that HDFI cannot detect.

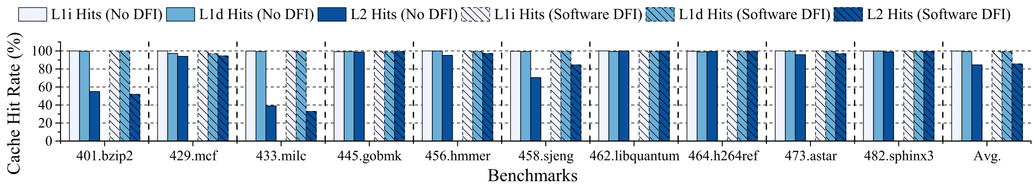

Fig. 3. Cache hit rates of user programs with and without software DFI. The high cache hit rates indicate that memory access is probably not a bottleneck.

protection. In Reference [36], a 256-bit tag is employed to specify if each data can be referred by certain instructions. However, the 256-bit in Reference [36] has a different meaning from the 16-bit identifier in our approach. For security, the approach in Reference [36] only handles the permission of pointers. In contrast, our approach handles the permission of every store/load instruction. Overall, the tag-based techniques cited in References [8, 20, 30, 36] provide only coarse-grained isolation as different data/instructions with the same tag cannot be isolated from each other.

5 PERFORMANCE OVERHEAD ANALYSIS

We analyze the source of performance overhead of software DFI [7]. The experiment setup of the analysis is the same as that in Section 10.1. We call the program to be checked by DFI enforcement the **user program**. For a user program, when each `store` or `load` is executed, RDT needs to be accessed and consequently data transfer with memory may be greatly increased. A memory access typically takes hundreds of clock cycles and can cause huge overhead. Thus, we first tested the cache hit rate to understand the DFI's impact on memory accesses.

When testing with SPEC CPU 2006 benchmark [31], the cache hit rates of user programs without DFI enforcement and with software DFI are shown in Figure 3. One can see that the cache hit rates are usually greater than 95% regardless of applying DFI enforcement or not. This indicates that memory access is probably not a bottleneck.

We further investigated the overhead breakdown of software DFI, of which the results are shown in Figure 4, where "RDT Search" represents the execution of the instrumented instructions for finding the RDT entry of the corresponding user `load` or `store`. "Bounds Check" means the check for preventing RDT from illegal modification. "Library Loop" represents the execution of the loop-related instructions (such as comparison and branch instructions) in the instrumentation for each

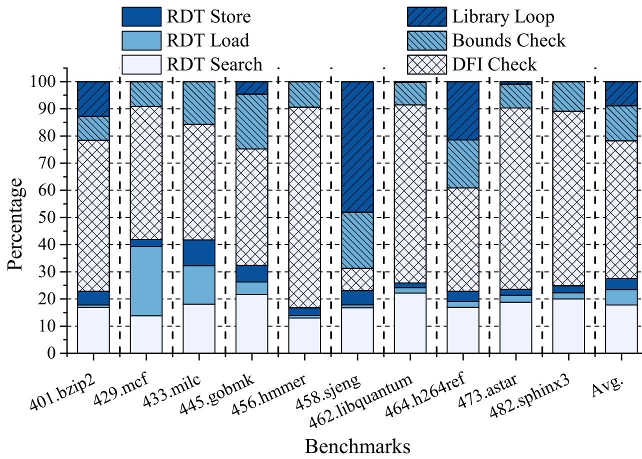

Fig. 4. Overhead breakdown of software DFI. The results indicate that the bottleneck is not memory access but "DFI check" instructions.

library function. "DFI Check" indicates the comparisons checking if the identifier found in RDT Search is in the RDS of the corresponding user `load` or not.

According to Figure 4, most of the overhead is from "DFI check." It also shows that RDT access (excluding "RDT search") contributes little to the overhead. This confirms that the bottleneck is not memory access but "DFI check" instructions. Specifically, many comparison and branch instructions are executed for each "DFI check," which compares the identifier found in RDT with each identifier in RDS of the corresponding user `load`. Although this check computation is fairly simple, it is performed for a huge volume of data.

6 OVERVIEW OF PROPOSED APPROACH

Our approach is to delegate DFI enforcement to another computing resource external to the main processor where the user program is executed. The delegated resource can be a processor core in a **Chip Multiprocessor (CMP)** or a **Processing-In-Memory (PIM)** processor [4]. The two options are similar in terms of the overhead reduction. We will use PIM as an example platform to describe our approach, while the same idea is applicable to the CMP core option.

The PIM processor undertakes most of the DFI verification components analyzed in Section 2 and can quickly access RDSs and RDT in its vicinity. As such, what remains for the main processor to do is to collect target instruction information discussed in Section 2 and send it to PIM. Although the information collection and transmission can be implemented with software in a way same as multithreading, our study shows that such a software approach still experiences huge or even worse performance overhead. Thus, we propose a hardware approach to minimize extra software executions at the main processor.

The proposed system is depicted in Figure 5, which consists of three main blocks:

(1) Offline program analysis and instrumentation (Section 7).
(2) Runtime information collection (Section 8).
(3) PIM-based DFI Enforcement (Section 9).

Offline Program Analysis and Instrumentation: According to Section 2, RDSs of the user program are required by DFI enforcement. Since RDS of each instruction is static, offline program analysis can be applied to the user program once and RDSs can be loaded into the

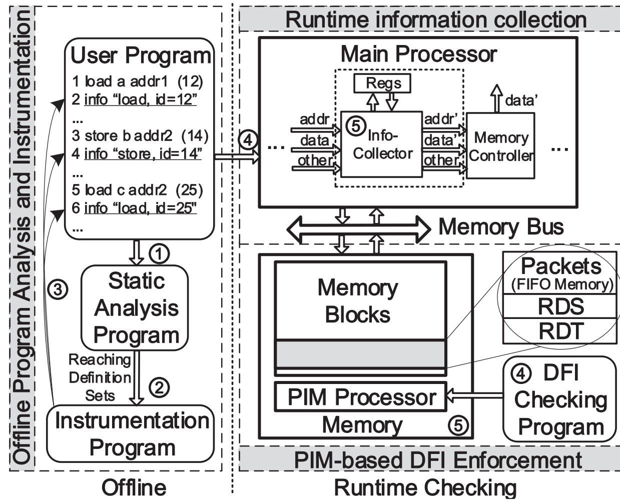

Fig. 5. The proposed system and the flow of PIM-based DFI enforcement.

memory when the user program starts. Besides, the software instrumentation is introduced for generating the target instruction information, which can be used by the hardware in the Runtime Information Collection block. In Figure 5, the underlined instructions are for instrumentation. One example is `info “load, id = 12,”` which indicates the former instruction is `load`, with the identifier 12.

Runtime Information Collection: The target instruction information generated from the instrumentation needs to be transferred to the PIM processor. The information transfer is performed by the dedicated hardware module named **info-collector** designed in the main processor. Info-collector parses the instrumented instruction for the target instruction information, and it can optimize the size of the information, which is sent to the memory while the software program is being executed.

PIM-based DFI Enforcement: This block contains the PIM processor, which performs the DFI checking after receiving the target instruction information from the main processor and accessing RDS and RDT in the memory. With the collaboration of the three blocks, complete runtime DFI enforcement is realized.

Putting It All Together: Figure 5 shows the overall flow of the proposed DFI enforcement, where the circled numbers indicate the step ID:

① Static analysis is performed for a user program.
② RDSs are obtained from the static analysis.
③ The codes are instrumented automatically. The main instrumentation is to add instructions for encoding the target instruction information after each target instruction so as to help collect its information. The instrumented instructions are underscored in Figure 5.
④ The DFI checking program and RDS are loaded onto the PIM processor before the user program execution starts on the main processor.
⑤ During program execution, the info-collector parses each instrumented instruction, collects target instruction information accordingly, forms a **DFI packet**, and sends it to the PIM processor, where enforcement computations are performed or RDT is updated.

In the following sections, the details of the three blocks are elaborated.

7 SOFTWARE INSTRUMENTATION

Instrumentation is to add additional code into a user program to facilitate the DFI enforcement. The software instrumentation in our approach helps not only to extract the necessary information but also avoid changing the instruction set.

As shown in the Offline Program Analysis and Instrumentation block in Figure 5, a user program is automatically instrumented by the software we developed after getting the **Reaching Definition Sets (RDSs)** from the static analysis. The instrumentation mainly adds the instructions that generate the runtime information of target instructions, which can be parsed by the info-collector. Assuming the instrumented instruction is `info`, the basic syntax is

```
info runtime_info.
```

When `info` is executed in the main processor, the `runtime_info` is transferred to the info-collector. A simple way to realize the `info` instruction is to extend the instruction set. However, Instruction Set Architecture (ISA) extension requires much more changes across the computing stack including both software and hardware, thereby introducing more engineering efforts and cost. To avoid this, we propose another approach to implement the `info` instruction by overloading the `store` instruction. These instrumentation `store` instructions are called **DFI `store`**, of which we overload the use with underlying semantics different from ordinary `store` instructions. Our key technique is to differentiate between ordinary `store` and DFI `store` without adding new instructions. The basic syntax of the DFI `store` is

```
store runtime_info dfi_global,
```

where `runtime_info` is a constant value including the runtime information, and `dfi_global` is the address (the pointer) of a global variable declared at the beginning of a program and serves as a signature to indicate a DFI `store`. The address of this global variable is set by writing a dummy value at the beginning of a program as

```
store dfi_dummy dfi_global.
```

The `dfi_dummy` is a dummy value that has a fixed value to obtain the destination address of `dfi_global`. The info-collector can obtain `dfi_global` by identifying the first `store` in a program that stores `dfi_dummy` to an address. For example, `dfi_dummy` can be designed as `123456`. Once `store 123456 11122` is executed, the info-collector assigns `11122` to `dfi_global`, and `dfi_global` can only be assigned one time for a user program.

The info-collector (dotted box in Figure 5) checks if the target address of a `store` instruction is the same as `dfi_global`. If so, then the instruction is a DFI `store` and the runtime information is extracted and sent to PIM.

There are three scenarios where the instrumentation is needed:

(1) For each ordinary `store` or `load` instruction, its target instruction information is required by DFI, thereby instrumentation is needed.
(2) The source code of a library function is not necessarily accessible to the users, but instrumentation can still be performed to obtain the target instruction information if the library functions are for memory accesses. This is similar to the wrapper [7] in theory, but our implementation is hardware-based but not software-based.
(3) Function return addresses are stored on stack and vulnerable to attacks such as **Return-Oriented Programming (ROP)** [27]. We treat their accesses as implicit `load`/`store` instructions and perform DFI enforcement accordingly. When a parent function `parent_func()` calls a child function `child_func()`, the return address is stored on the stack by an instruction `parent_inst`. When function `child_func()` returns, the return

```
1  /* =====beginning of the program====== */
2  (instructions for allocating FIFO memory)
3  (instructions for storing RDS to memory)
4  store dfi_dummy dfi_global
5  store packet_dummy packet_mem_addr
6  ...
7  store x1 addr1         //identifier: 12
8  store (0<<16)+12 dfi_global
9  ...
10 load x2 addr2          //identifier: 25
11 store (1<<16)+25 dfi_global
```

Fig. 6. An example of code instrumentation.

address is loaded by a return instruction `child_inst`. DFI ensures that the return address used by `child_inst` should be the latest value stored by `parent_inst`. To enforce the DFI of function return addresses, we need instrumentation for generating the target instruction information of function calls and returns.

7.1 Instrumentation for DFI Enforcement

The instrumentation is mainly to extract the runtime information of the `load`/`store` instructions in a user program related to DFI checking and sent to the PIM processor. The information includes instruction identifier, instruction type, and target address of `load`/`store`. Instruction identifiers are automatically assigned by the instrumentation tool.

Every `store` and `load` instruction in a user program, called target instruction, is followed by a DFI `store`. The `runtime_info` of the DFI `store` contains the instruction type and identifier of the preceding target instruction. For example, in Figure 5, line 2 is an instrumentation instruction `info "load, id = 12"` that is implemented by `store "load, id = 12" dfi_global`, which tells the instruction type and identifier of the target instruction in line 1. To encode the instruction type and identifier, according to Reference [7], 16 bits are sufficient for representing instruction identifiers in a large program. We use an additional bit to indicate instruction type, where 0 means `write` and 1 means `read`. When the info-collector recognizes a DFI `store`, it extracts the target address of the preceding target instruction. The target address and the `runtime_info` form a **DFI packet** to be sent to PIM.

At the beginning of code execution, a memory space is dynamically allocated at the PIM processor for DFI enforcement. This includes the memory space for storing incoming packets, which is called packet **FIFO memory**. The starting address of packet FIFO memory is `packet_mem_addr`, which is also a dynamic value. Similar to `dfi_global`, `packet_mem_addr` is also set by writing a dummy packet at the beginning of a program as

```
store packet_dummy packet_mem_addr.
```

Later during the code execution, all DFI packets are sent to FIFO memory based on `packet_mem_addr`. Please note that `dfi_global` and `packet_mem_addr` are generated by the automatic code instrumentation and not visible to security attackers. Besides, only the info-collector is allowed to control the memory controller to send data to the FIFO memory after `packet_mem_addr` is set. Any other attempts for accessing the FIFO memory in the main processor are identified as violations.

An example of the instrumentation is shown in Figure 6, where lines 7 and 10 are the original instructions in the user program, while lines 2, 3, 4, 5, 8, and 11 are instrumentations. The identifiers of the instructions at lines 7 and 10 are in the parentheses (12 and 25). The data of a DFI `store` (lines 8 and 11 in Figure 6) has bit 16 for instruction type and bits 15–0 for an instruction identifier.

```
1  store (1<<20)+(1<<19)+(0<<18)+(1<<17)+7 dfi_global
2  store (y1's addr) dfi_global
3  store (x1's addr) dfi_global
4  store 40 dfi_global
5  memcpy(x1, y1, 40)          //identifier: 7
6  ...
7  store (1<<20)+(0<<19)+(1<<18)+(1<<17)+15 dfi_global
8  store (x2's addr) dfi_global
9  store 12 dfi_global
10 store 9 dfi_global
11 memset(x2, 3, (9<<32)+12)    //identifier: 15
```

Fig. 7. The instrumentation for library functions.

7.2 Handling Library Functions

A program often calls library functions, whose source code is not necessarily directly accessible. This makes it hard to directly instrument the DFI `stores` inside the library functions. Reference [7] proposed a wrapper approach for solving this problem, and we propose its implementation scheme to effectively enforcing DFI for library functions by instrumenting the DFI `stores` outside a library function. As a library function call may involve a multi-byte data block in general, the instrumentation needs to keep track of data-length besides data address. To include these kinds of information in `runtime_info` of the DFI `store`, multiple DFI `stores` are required. In detail, our approach is illustrated using the example in Figure 7.

In this example, the target instructions are the function calls in lines 5 and 11, with their identifiers in parentheses. The instrumentation for each library function call includes multiple DFI `store` instructions like lines 1–4 for the target instruction of line 5. The first DFI `store` keeps the corresponding identifier in its lower 16 bits. Its bits 17–20 are four binary indicators telling if the target instruction is a library function call or not, if the data-length needs 64 bits to represent or not, and if the function loads/stores data or not. The info-collector parses these indicators and then takes corresponding actions. Additional DFI `store` instructions are added to send other information. For example, lines 2 and 3 send load and store addresses. Depending on if the data-length is represented in 32 or 64 bits, the data-length needs to be sent through a single or two DFI `store` instructions. For example, line 4 sends the data-length in a single DFI `store`, while lines 9 and 10 send in two DFI `store` instructions.

If the arguments of a library function call do not include the data address or data-length, or the function is called by an indirect branch instruction, the approach in Figure 7 cannot be directly applied. Instead, more complex instrumentation schemes and library function overriding are needed. Since enforcing DFI for library functions is not the main focus of this work, we leave it as the future work.

8 HARDWARE DESIGN

In Figure 5, the info-collector in the Runtime Information Collection block is the key hardware component to be added to the main processor. It detects DFI `store` instructions, collects runtime information of target instructions, generates DFI packets, and sends them to PIM. Due to the existence of three instrumentation scenarios, the info-collector needs to first identify the scenario and generate a DFI packet accordingly. The DFI packet is then sent to the FIFO memory. Besides, since transferring data to the memory can lead to performance overhead, data compression and runtime optimization are applied. The design details are described in the following subsections.

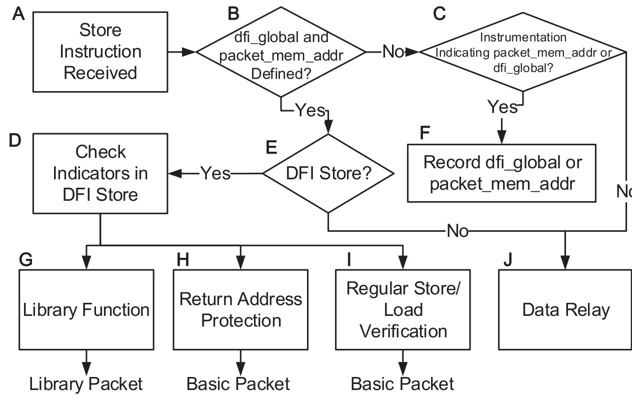

Fig. 8. Operations of info-collector.

8.1 DFI Packet Generation

Info-collector can be realized as a hardware circuit through synthesizing Verilog description. Its basic operations are depicted in Figure 8.

The info-collector acts only when a `store` instruction is executed. In step B of Figure 8, it checks if `dfi_global` and `packet_mem_addr` have already been defined. If not, then it proceeds to step C to capture `dfi_global` or `packet_mem_addr`. Please note "`store dfi_dummy dfi_global`" and "`store packet_dummy packet_mem_addr`" are instrumented at the beginning of a program. Moreover, both `dfi_dummy` and `packet_dummy` have signature values that can be recognized by the info-collector. If they have already been defined, then the info-collector further checks if the `store` is a DFI `store`. This is by examining if the target address is the same as that of `dfi_global`.

If this `store` is a DFI `store`, then the info-collector parses the indicators in the data part of the DFI `store` and tells if this is to verify `load/store` for DFI enforcement, function return, or a library function call. If this instrumentation is for a `load/store` instruction, then the info-collector collects instruction type and identifier from this DFI `store` instruction, and the target address from the previous instruction. These pieces of information form a **basic packet** ("`data`" in Figure 5) to be sent to PIM, which stores the packet to the address of the allocated packet FIFO memory ("`addr`" in Figure 5).

If this DFI `store` is for a return address protection (step H in Figure 8), then the info-collector takes the identifier and instruction type from this DFI `store` and extracts the pointer to the return address from the next DFI `store`. This information also forms a **basic packet**. If this DFI `store` is for a library function (step G), then the indicators of this `store` tell if the library function is to load data, store data or not, and if the data-length needs to be encoded by 64 bits or not. Next, the info-collector continues to collect additional information from subsequent DFI `store` instructions and generates a **library packet** to be sent to PIM.

If the `store` instruction is a part of the user program (step J), i.e., not a DFI `store`, then its `data` is relayed to memory without any change and its target address is stored in a local register for future use.

8.2 Packet Transfer to PIM

A memory space is allocated to store DFI packets sent from the main processor. It is used as a packet FIFO to store and process the packets in a first-come-first-serve manner. To maintain the FIFO nature using a region of random access memory with low overhead, we develop circuit design

```
1  /* ========Example A========== */
2  int aa[1024];
3  for(int i=0;i<1024;i++)
4    aa[i]=i;
5  /* ========Example B========== */
6  int bb[1024][1024];
7  for(int i=0;i<1024;i++)
8    for(int j=0;j<1024;j++)
9      bb[j][i]=i+j;
```

Fig. 9. Examples of address locality.

techniques to maintain the head and tail pointers in hardware, where the head pointer is updated by PIM (consumer) and tail pointer is updated by the main processor (producer).

8.3 Lossless Data Compression

The main reason for performance overhead of PIM-based DFI is transferring DFI packets to memory. Although each DFI packet has only a few bytes, the number of DFI packets is huge and the overall impact is significant. We propose to compress target addresses and identifiers by exploiting locality. The compression is realized in the info-collector hardware.

Consider the two C program examples in Figure 9. For example A, assume the starting memory address of `aa` is 0×8000, then the program stores data at 0×8000, 0×8004, 0×8008, and so on. Starting from `i = 1`, each target address increases by 4 compared to the previous one. Thus, we only need to send the increment in 4 bits, which include 1 sign bit, instead of a 32-bit address. Example B in Figure 9 is similar, but has an address pattern of 0×8000, 0×8400, 0×8800, and so on. Although the address increment 0×400 is relatively large and needs 11 bits to represent, the lower bits of the increment are all 0s. Thus, instead of using integer compression, we use a format similar to floating point number representation to further reduce the bitwidth of the address increment. This format consists of a sign bit, significand, and exponent of 16. To represent 0×400, the sign bit is 0, there are 3 bits for significand to represent 4 and the exponent is 2. Overall, the bitwidth is 6, which is shorter than the 11-bit binary encoding. The floating point number representation contains 8-bits, 1 sign bit, 4 bits of significand, and 3 bits of exponents (the power of 16). This representation can cover the range from -15×2^{28} to 15×2^{28}. The info-collector calculates the difference between two target addresses. If the difference is within this range and the significand is within -15 to 15, then the difference is represented by an 8-bit floating point number. Note that the difference is compressed only when it can be represented in this format with a 16-basis exponent.

Identifiers can also be compressed based on their value locality. However, they rarely have the patterns like example B, where the increment is at the middle bits of an address. Thus, the difference between two identifiers is represented by a binary number. Overall, a DFI packet can be compressed to 15 bits. Thus, we can pack two **compressed packet**s into one word.

8.4 Runtime Optimization

We develop packet pruning techniques and a technique for increasing the opportunity of locality for data compression. These optimization techniques help reduce the amount of data sent to PIM and thereby further decrease performance overhead. Some pruning techniques described here are similar to those in Reference [7]. However, the pruning techniques in Reference [7] are offline, while our hardware approach allows pruning at runtime. As more information, such as target address, is available at runtime, the opportunity of pruning is increased.

Similar to data transfer between memory and cache in cache lines, we pack multiple DFI packets into a block of hundreds of bytes before sending them to PIM. The packets in a block are organized in a **transmission buffer**, which is implemented as a register file. The optimizations are performed for packets in the buffer before they are sent out. Note that waiting for other packets to form a block increases DFI enforcement latency but does not increase performance overhead.

Consider two pairs of basic packets in the transmission buffer, (P_1, P_2) and (Q_1, Q_2). Each basic packet is for instruction `load`, `store`, or function return. Packet P_1 (Q_1) precedes P_2 (Q_2). The packets of each pair share the same target address and there is no other DFI packet for `store` of the same target address between them. There are five optimization techniques described using the packet pairs:

A: If P_1 and P_2 are for `store` instruction, and there is no other DFI packet for a `load` with the same target address between them, then packet P_1 is redundant and can be pruned out without being sent to PIM.

B: If P_1 and P_2 are both for `store` instruction, and their identifiers are the same, then P_2 can be pruned out.

C: If P_1 and P_2 are both for `load` instruction, and their identifiers are the same, then P_2 can be pruned out.

D: P_1/P_2 are for `store/load` of the same target address $Addr_1$. After P_1 and P_2, packets Q_1 and Q_2 are for `store/load` of the same target address $Addr_2$. P_1/P_2 have identifiers α/β, respectively. If Q_1/Q_2 also have identifiers α/β, respectively, then Q_1 and Q_2 are redundant. This is to make sure that the same `store/load` pair appears only once in the transmission buffer. An example is shown in Figure 10(a), where the table is the packets in the transmission buffer, with each line representing a basic packet. The last line represents the latest packet. "S/L" represents the instruction type ("S" for `store` and "L" for `load`), and "Tar Addr" represents the target address. In this example, Q_1 and Q_2 are redundant.

E: All basic packets in the transmission buffer are sorted according to their target addresses. If two packets have the same target address, their relative order keeps unchanged. If there is a library packet, the basic packets before and after this library packet are sorted separately. After sorting, the target address difference between two adjacent packets is examined to find if data compression can be performed. The sorting helps find opportunities for data compression. The verifications in DFI enforcement for `load/store` of different target addresses are independent of each other and hence sorting does not affect DFI enforcement results. An example is shown in Figure 10(b), where the left table is the packets in the transmission buffer before sorting. Before sorting, the difference of the target addresses between each pair of adjacent packets is large, which is hard for compression. After sorting, we found two groups of packets that can be easily compressed by our compression approach. Note that the sorting is performed before and after the library packet separately.

Among the optimizations, A, B, and C are similar to those in Reference [7] except that they can be performed both offline and at runtime while those in Reference [7] are restricted to offline. Techniques D and E are newly developed in this work. After the optimizations are performed, a packet is compressed if possible.

8.5 Circuit Implementation of the Optimizations

All the five optimizations can be realized in circuits for runtime use in the main processor. We illustrate the circuit designs by using optimization C as an example.

The schematic of combinational circuit implementation of optimization C is shown in Figure 11. Assume there are n basic packets in the transmission buffer, Pi represents the ith packet, and Ri

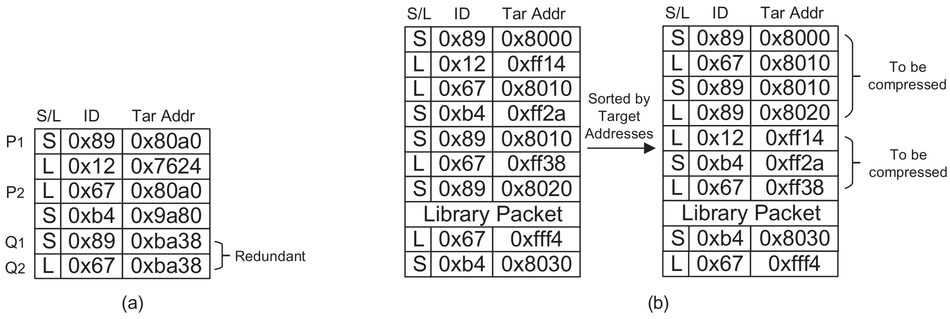

Fig. 10. (a) The example for illustrating optimization D. (b) The example for illustrating optimization E.

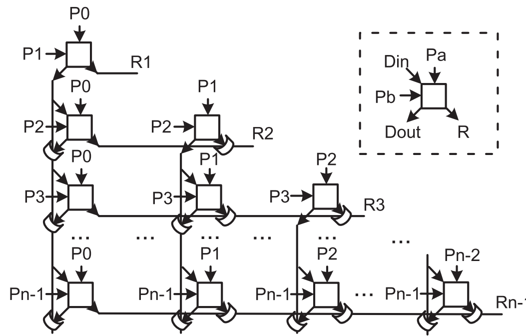

Fig. 11. Circuit for implementing optimization C.

indicates if the ith packet is redundant or not. Each square in Figure 11 is a **Processing Element (PE)** that computes if a packet is redundant or not. In each column of Figure 11, a packet Pi is compared with all later packets $Pj, j > i$ and attempts to find a redundant Pj to be pruned. If there are multiple packets that are redundant with respect to Pi, then only the topmost one (with the smallest $|j - i|$) is asserted for pruning and the others can be pruned later in other columns to the right. The R signals in a row are ORed such that a packet in a row can potentially be pruned by any proceeding packets organized in columns. For example, $P3$ in row 3 can be potentially pruned by $P0$, $P1$, or $P2$ in the left three columns. Like illustrated in the dotted box, a PE compares two input packets Pa and Pb. A necessary but insufficient condition for asserting $R = TRUE$ is that Pa and Pb are both for `load` with the same target address and identifier. The final result of R also depends on Din, which is a disable signal for the pruning. The value of $R = TRUE$ when $Din == 0$ and the necessary condition holds. There are two scenarios where the disable signal asserts: (1) there is a `store` at the same target address between the two `load` instructions of Pa and Pb, and thus the conditions for optimization C is not completely satisfied; (2) a redundant packet has already been found and no further pruning is needed in a column. For scenario (1), $Dout = 1$ when Pa is for `load` while Pb is for `store`. For scenario (2), $Dout = 1$ if $R = TRUE$ for the same PE.

9 DFI CHECKING PROGRAM AT PIM

In the PIM-based DFI Enforcement block in Figure 5, the DFI checking program at the PIM processor continuously reads DFI packets from the FIFO memory and either performs DFI enforcement or updates RDT. For different types of DFI packets, the PIM processors take different actions.

The DFI checking program is written in C language, and its binary code is executed on the PIM processor. The RDT memory space is allocated by the instrumentation code. Same as in Reference [7], all program data are organized in words, each of which requires one RDT entry. If the data memory for the user program has N bytes, there are $N/4$ entries in the RDT [7]. Since each identifier has 16 bits = 2 bytes, the RDT uses $\frac{N \times 2}{4} = N/2$ bytes of memory.

There are three kinds of DFI packets to be processed by the DFI checking program.

- **Basic packet for `store` or `load`:** The DFI checking program extracts instruction type, identifier α and target address β from the packet. If the instruction type is `store`, then identifier α is stored at entry $\beta >> 2$ of RDT. The right shift is performed because RDT is organized in words. If the instruction type is `load`, then the DFI checking program reads identifier γ from entry $\beta >> 2$ of RDT and loads the RDS of identifier α. Then, the program checks if γ is in the RDS of α or not. If not, then a DFI violation is reported. Finally, identifier α and target address β are saved in registers for future decompression of compressed packets.
- **Compressed packet for `store` or `load`:** The process is similar to handling basic packets except that decompression is performed.
- **Library packet:** The DFI checking program extracts target address α if there is `load` in the library function call, and target address β if there is `store`. Then, data-length γ (in words) of the `load` and/or `store` and identifier δ of this function are also extracted. If there is an address α, then the DFI checking program loads the identifiers ϵ_0, $\epsilon_1 \ldots \epsilon_{\gamma-1}$ from entries $\alpha >> 2$, $(\alpha >> 2) + 1, \ldots (\alpha >> 2) + \gamma - 1$ in the RDT and checks if every ϵ_i is in the RDS of identifier δ. If there is address β, then the program stores identifier δ to all the entries from $\beta >> 2$ to $(\beta >> 2) + \gamma - 1$ in the RDT.

10 EXPERIMENT

10.1 Experiment Setup

This section describes the experimental setup and the modeling and evaluation methodology.

Software Analysis and Instrumentation: The programs used in our article are based on C/C++ and compiled by LLVM [22] and the static analysis is performed by an extended SVF [32]. The instrumentation is performed on the program's LLVM **Intermediate Representation (IR)** by our software without interacting with LLVM. Then, the instrumented program is further compiled into binary code. Besides, our techniques are general and directly applicable to other programming languages supported by LLVM. Note that compilers, static analysis tools, and the static analysis itself is out of the scope of this work.

System Configuration and Modeling: We evaluate our approach and the proposed techniques using architecture simulations through SMCsim [4, 28], which is an extension to the gem5 simulator [33] for accommodating PIM. The main processor is an ARM Cortex-A15 with 2 GHz frequency, 32 KB L1 instruction cache, 64 KB L1 data cache, 2 MB L2 cache, and 512 MB memory. A single PIM processor is used and operates at 2 GHz frequency [25, 40]. 64 MB memory is allocated for RDT, which is sufficient for the test cases in our experiment. Other details of the PIM can be found in References [4, 28]. Please note that the PIM configuration has little impact on the user program execution.

Security and Performance Evaluation: For security analysis, we used the RIPE benchmark suite [37] for control-data attacks and tested on Heartbleed vulnerability [34] and Nullhttpd heap overflow vulnerability [23] for non-control data attacks. In addition, we used SPEC CPU 2006 [31] benchmark suite for performance evaluation.

Table 2. The Configurations of RIPE Tests

Dimensions	Overflow Technique	Attack Code	Target Code Pointer	Location	Function Abused
Configurations	direct, indirect	rop, createfile, returnintolibc	ret, baseptr, funcptrstackvar, funcptrstackparam, structfuncptrstack, funcptrheap, structfuncptrheap, structfuncptrbss, funcptrdata, structfuncptrdata	stack, heap, bss, data	memcpy, strncpy, strncat

10.2 Security Analysis

Our approach enforces the same DFI as defined in Reference [7] and thus achieves similar security as Reference [7] except that our approach is asynchronous monitoring [9, 19, 39], where detection of DFI violation can trigger system interrupt for further security measures, rather than synchronous enforcement like Reference [7]. This difference is a tradeoff between security and service availability. Synchronization inevitably entails extra performance overhead, as DFI enforcement blocks user program executions.

10.2.1 RIPE Benchmark. RIPE [26, 37] is a well-known benchmark containing various control-flow attacks, and all control-flow attacks can be identified by DFI. RIPE is originally designed for X86 architecture and modification is required for executions on an ARM processor. We implemented 156 attacks of the benchmark for our system, including **Return-Oriented Programming (ROP)** [27] attacks and **Jump-Oriented Programming (JOP)** [6] attacks. Table 2 shows the configuration dimensions and possible configurations of RIPE, where "Overflow Technique" indicates whether the attack target can be directly reached by sequentially overflowing from a buffer. "Attack Code" is what the attack payload is. "Target Code Pointer" is the target to be attacked. "Location" is the location of the attack target. "Function Abused" is the function used to modify the data [37].

We tested all the valid combinations of the five dimensions' configurations, which results in 156 valid attacks in total. No configuration is ignored in each dimension, except "Function Abused," which is only for copying data by different functions, and this does not affect the key idea of the attack. Different functions in "Function Abused" need different dedicated instrumentation to obtain the store/load addresses and data-lengths, as described in Section 7.2, which needs manual design. To avoid too many engineering efforts, we tested three typical functions: `memcpy`, `strncpy`, and `strncat`.

In addition, we prepared a RIPE program without any attack to test if there is any false alarm or not. It is observed that our DFI system successfully identifies all the 156 attacks and does not make any false alarm for the case without the attack.

10.2.2 Heartbleed. Heartbleed (CVE-2014-0160) [34] is a vulnerability in 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 is not checked if the actual payload length is the same as the claimed one. As such, an attacker may send a message with the actual payload length smaller than the claimed one. 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, sensitive data is stolen by the attacker. We use the source code in Reference [35] to simulate such an attack. This attack is successfully detected by our DFI enforcement 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 made by our approach.

10.2.3 Nullhttpd. Nullhttpd is an HTTP server that has heap overflow vulnerability (CVE-2002-1496) [23]. If the server receives a POST request with negative content length L, then it should not

Table 3. Scenarios for Figure 2 where HDFI Fails

HDFI				Our approach
u0	u1	u2	Missed overflow	detect?
Tag 0	Tag 0	Tag 0	u0 ⇒ u1, u0 ⇒ u2, u1 ⇒ u2	Yes
Tag 0	Tag 0	Tag 1	u0 ⇒ u1	Yes
Tag 0	Tag 1	Tag 0	u0 ⇒ u2	Yes
Tag 0	Tag 1	Tag 1	u1 ⇒ u2	Yes
Tag 1	Tag 0	Tag 0	u1 ⇒ u2	Yes
Tag 1	Tag 0	Tag 1	u0 ⇒ u2	Yes
Tag 1	Tag 1	Tag 0	u0 ⇒ u1	Yes
Tag 1	Tag 1	Tag 1	u0 ⇒ u1, u0 ⇒ u2, u1 ⇒ u2	Yes

process the request. However, the server continues to process and allocates a buffer of $L + 1,024$ bytes, which is less than 1,024 bytes. Later, the server writes data of 1,024 bytes into the buffer, and therefore buffer overflow occurs. The experiment shows that our method successfully detects such buffer overflow. When some `load` instruction attempts to access the data written by overflow, it is found that the data is not written by any instructions in the RDS of the `load` instruction. An experiment is also conducted to confirm that our approach does not produce false alarm in this context.

10.2.4 Comparison with HDFI and TMDFI. To compare the security between our approach and **Hardware-assisted Data-Flow Isolation (HDFI)** [30], we exhaustively tested different tag schemes of HDFI for the example of Figure 2, which are listed in the left three columns of Table 3. For each tag scheme, there is some overflow that cannot be detected by HDFI as shown in column 4, where `u0` ⇒ `u1` means some data of `user0` is written into `user1`'s space through overflow. By contrast, our approach can successfully detect all these overflows.

For TMDFI [20], although there is a significant improvement over HDFI, its enforcement resolution is still far from enough in many applications. Figure 12 shows the numbers of identifiers needed for several benchmarks, which are hundreds or tens of hundreds. Hence, the gap between the 256 regions by TMDFI [20] and the actual needs is large. By contrast, our approach can accommodate all identifiers in these benchmarks and achieve complete DFI with an overhead similar to TMDFI.

In conclusion, although both HDFI and TMDFI are able to identify the attacks in RIPE benchmark, Heartbleed attack, and the attack to Nullhttpd, the data can only be separated into the limited regions. Therefore, for a typical real-world program, it is possible that the attacks performed in the same region can pass the checking of HDFI and TMDFI, while our approach can defend against.

10.3 Performance Overhead

Performance overheads of the following methods are evaluated through simulations on the SPEC CPU 2006 benchmark [31]:

- Software. This is the original software DFI by Reference [7].
- HBM. This is similar to Reference [7] except that High Bandwidth Memory [15, 18] is employed.
- CMP. This is a parallel approach, where DFI enforcement is performed in another core in CMP with two versions: the software version **CMP-S** (multithreading) and the hardware version **CMP-H** using our info-collector circuit.
- PIM. This is the proposed hardware-assisted parallel approach using PIM.

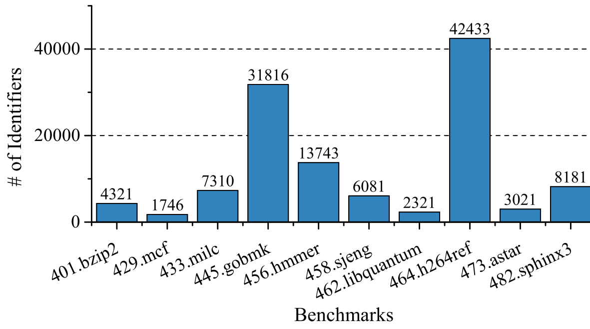

Fig. 12. The number of identifiers of each benchmark.

Table 4. Performance Overhead of DFI

		Software [7]	HBM	CMP-S	CMP-H	PIM (No Compression or Optimization)				PIM (512B Buffer)		PIM (2KB Buffer)		
Column ID		1	2	3	4	5	6§	7	8	9	10	11	12	13
Compression		×	×	×	√	×	×	√	×	√	√	√	√	√
Transmit Buf Size		-	-	-	2KB	-	-	2KB	2KB	512B	512B	2KB	2KB	2KB
Runtime Optimization		×	×	×	All	×	×	E	A,B,C,D	All	C,E	All	C,E	C,E
#Gates in Info-Collector		-	-	-	†	<2908†	†	†	†	†	116,769‡	†	†	753,666‡
Benchmark	*401.bzip2*	218.7%	219.5%	543.3%	43.7%	313.4%	34.6%	40.0%	44.7%	40.8%	43.2%	37.9%	38.6%	39.8%
	429.mcf	105.0%	105.6%	320.5%	28.8%	191.6%	18.9%	24.3%	27.1%	25.7%	26.8%	23.9%	24.1%	24.8%
	433.milc	80.9%	82.7%	256.6%	24.1%	150.0%	22.5%	25.5%	24.1%	25.3%	26.6%	23.4%	24.4%	25.0%
	445.gobmk	179.0%	179.0%	463.0%	59.4%	272.3%	46.9%	54.8%	56.5%	55.9%	57.7%	53.5%	54.3%	55.3%
	456.hmmer	233.4%	233.5%	1087.6%	60.9%	510.7%	47.2%	55.5%	64.2%	57.9%	60.8%	53.0%	53.0%	55.0%
	458.sjeng	372.6%	374.2%	226.9%	29.4%	128.6%	24.6%	28.0%	28.5%	28.6%	29.4%	27.3%	27.6%	28.2%
	462.libquantum	61.2%	61.2%	262.2%	22.5%	156.4%	21.9%	23.9%	23.2%	24.2%	25.0%	22.6%	22.7%	23.3%
	464.h264ref	205.3%	205.8%	544.0%	44.5%	275.7%	33.9%	42.1%	42.4%	42.8%	45.2%	39.9%	41.0%	43.1%
	473.astar	116.6%	116.6%	442.0%	38.2%	255.7%	31.6%	36.9%	38.4%	37.2%	39.1%	35.2%	35.5%	36.5%
	482.sphinx3	41.4%	41.6%	123.0%	18.7%	74.4%	32.1%	33.4%	33.4%	33.6%	33.9%	33.1%	33.1%	33.3%
Average		**161.4%**	**162.0%**	**426.9%**	**37.0%**	**232.9%**	**31.4%**	**36.4%**	**38.2%**	**37.2%**	**38.8%**	**35.0%**	**35.4%**	**36.4%**

†Computation time of optimizations and compression is neglected.
‡Computation time of optimizations and compression is considered.
§No DFI packet is sent to the memory.

Our **proposed approach** has two variants: CMP-H and PIM. As architectural simulations using gem5 are many orders of magnitude slower than real system runs for accurately modeling hardware behaviors, we manage to terminate the simulations to run the region of interest of the program for sufficiently long time while accounting for a reasonable simulation time. To ensure a fair comparison, each application was terminated at the same point in the simulations. The results are summarized in Table 4.

On average, the performance overhead of software DFI [7] is 161%, as shown in column 1. Column 2 shows the result of software DFI using HBM, where the memory bandwidth is abundant and memory access latency is fairly low. One can see that using HBM brings almost no overhead reduction. This result confirms the analysis in Section 5. The results of the parallel approach using another CMP core are summarized in columns 3 and 4, for software and our hardware version, respectively. Without dedicated hardware, the parallel approach actually increases the overhead due to the expensive communication in software. CMP-H reduces the overhead to 37%.

The PIM results are listed in columns 5–13, where “All” means all of the five optimization techniques are applied and “C, E” corresponds to the results where only the two most effective

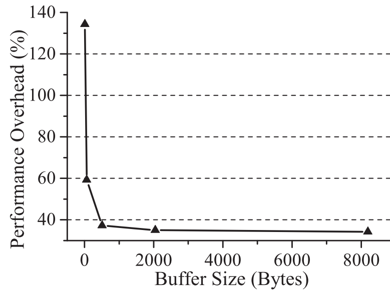

Fig. 13. Overhead vs. buffer size.

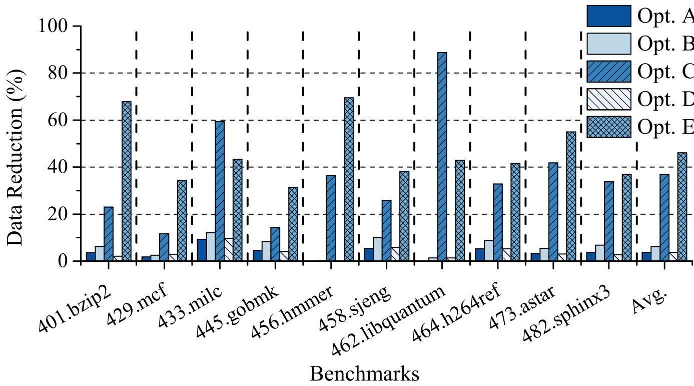

Fig. 14. Effects of optimization techniques.

optimizations are employed. In column 5, the overhead is 233% although the offline optimization has been applied. This tells the importance of our hardware-based optimization and compression. In column 6, we dropped all DFI packets without sending them out by simulating only instruction fetching but not executions of instrumentation. This is not realistic for DFI, but is to obtain a lower bound for the overhead, which is about 31%. Column 7 shows that the joint effect of data compression and optimization E is dramatic. Please note optimization E is designed for increasing the chance of data compression. The setup for column 11 is very similar to column 4, except that one is by PIM and the other is by CMP. Examining the results of the two columns that their overhead reductions are similar. PIM is a little better, as it causes less cache contentions as CMP. Column 13 takes the two most important optimizations and considers the compression/optimization delay, showing an overhead of about 36%.

The effect of transmission buffer size on reducing performance overhead is plotted in Figure 13. It shows that an increase of buffer size from 0 quickly brings down the overhead. However, the reduction soon diminishes as buffer size reaches 2K bytes, and this is why we limit the buffer size to be no more than 2K in our experiments.

The effects of the five optimization techniques described in Section 8.4 on data reduction are evaluated separately, and the results are depicted in Figure 14. It shows that optimizations C and E always lead to more data reduction than the other techniques. For *462.libquantum*, optimization C can reduce data by over 80%, while optimization E reduces data by more than 60% for both *401.bzip2* and *456.hmmer*. Optimization E is designed to facilitate compression, and one can observe that its average data reduction is 46%, which is also the average **compression ratio**.

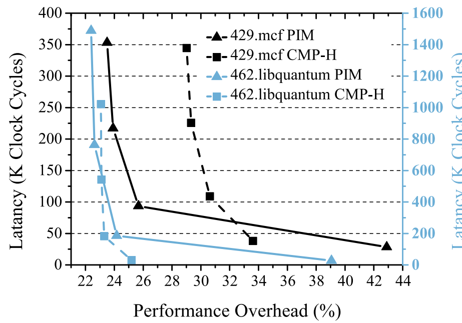

Fig. 15. Detection latency vs. overhead for *429.mcf* and *462.libquantum*.

10.4 Tradeoff between Detection Latency and Overhead

Ideally, the latency for detecting DFI violations need to be minimized so attackers have less time to complete damaging operations. In Figure 15, we show that the latency can be managed by a tradeoff with the overhead via varying the buffer size. The results also indicate that the PIM approach performs better for low overhead, while the CMP-H approach is slightly better for obtaining low latency. The reason is, as discussed in Section 10.3, PIM causes fewer cache contentions as CMP and leads to less overhead. However, due to the data transferring latency from the main processor to the PIM and the relatively low performance of PIM, using PIM has longer latency than using CMP.

10.5 Binary Size and Hardware Circuit Overhead

Performing instrumentation can increase the size of the executable binary of the user program. As shown in Figure 16, our approach only increases the binary size by 10%, on average, while software-DFI can increase the size by 35%, which is 3× as our approach. Note that the amounts of the instrumentation are the same for PIM, CMP-S, and CMP-H.

The info-collector circuit is implemented by synthesizing Verilog using Synopsys Design Compiler and ASAP 7nm cell library [3]. The info-collector with basic operation and compression costs only 2,908 gates and less than 30 ps circuit delay. Hence, its area and delay are negligible. We also implemented the circuit for optimization C/E. The results with these implementations are in columns 10 and 13 of Table 4, where the gate counts of the info-collector with different buffer sizes are listed. The hardware circuit overhead is dominated by the optimization part. The gate count of 754K is not trivial, but still a small fraction of a modern microprocessor, which often has hundreds of millions of gates. Moreover, our DFI can isolate data among 64K regions, and the hardware cost per region is no more than 12 gates. Although the hardware overhead of our approach is still high for practical use of many current civil applications, there can be niche applications, where security is super critical while hardware cost is of little concern, such as some military applications. Moreover, as security problems become increasingly prevalent and hardware cost continues to decrease, there can be a point in future where the hardware cost becomes justified for the achieved security. The works of CHERI [36] and HDFI [30] did not describe their hardware details. However, HDFI can isolate only between 2 regions, and its hardware cost is almost impossible to be less than 24 gates. Therefore, it is highly possible that the hardware cost per region of our approach is less than HDFI.

The memory overhead of our approach is $N/2$ and the size of the RDSs if the user program needs N bytes data memory, and CHERI and HDFI have much less memory overhead than us. In our design, we trade more memory space for higher security.

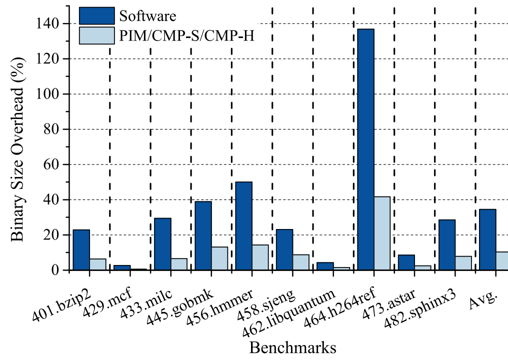

Fig. 16. The executable binary size overhead of different benchmarks.

11 DISCUSSIONS

In this section, we discuss the tradeoff and limitation of the asynchronous enforcement, as well as potential supports to OS.

The Asynchronous Enforcement: Different from other works such as the software DFI [7], CHERI [36], and HDFI [30], of which the DFI enforcement approaches are synchronous, our approach is asynchronous and has detection latency. The security risk due to asynchronous checking is a price paid for the performance overhead reduction compared to the software DFI. However, the proposed approach arguably still provides a higher level of security than CHERI and HDFI, where a large amount of attacks are completely undefended. Specifically, if an attacker violates DFI for two data in one region for HDFI, then this violation cannot be detected by HDFI. Although our asynchronous check leaves a brief window before stopping the program execution, any followup attacks must be carried out within this window and the threshold for such attacks is significantly raised. In contrast, in many cases, attackers can launch attacks without such restrictions for systems with HDFI.

The Supports to the OS: User programs and OS are two main application scenarios of DFI. Our main focus is the principle techniques for reducing DFI overhead and working out the details for user programs. The same principles are applicable for OS, but the implementation details are quite different, which is out of the scope of this work.

12 CONCLUSIONS AND FUTURE RESEARCH

Data-Flow Integrity (DFI) is potentially a very powerful security measure that can detect a large number of software attacks. However, it requires checking a large volume of data and thus intrinsically entails huge performance overhead. We propose a hardware-assisted parallel approach to address this challenge. This approach can reduce the overhead by more than 4× compared to the original software DFI while enforcing complete DFI. In future research, we will study how to further reduce the performance overhead and detection latency.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. *ACM Conf. Comput. Commun. Secur.* (2005), 340–353.

[2] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing memory error exploits with WIT. In *IEEE Symposium on Security and Privacy*. 263–277.

[3] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. 2016. ASAP 7nm Predictive PDK. Retrieved from http://asap.asu.edu/asap/.

[4] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2016. Design and evaluation of a processing-in-memory architecture for the smart memory cube. *International Conference on Architecture of Computing Systems*. 19–31.

[5] Ken Biba. 1977. Integrity considerations for secure computer systems. *Defense Technic. Inf. Cent.* (1977), 68.

[6] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: A new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security*. 30–40.

[7] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing software by enforcing data-flow integrity. In *Symposium on Operating Systems Design and Implementation*. 147–160.

[8] Jedidiah R. Crandall and Frederic T. Chong. 2004. Minos: Control data attack prevention orthogonal to memory model. In *IEEE/ACM International Symposium on Microarchitecture*. 221–232.

[9] Sanjeev Das, Yang Liu, Wei Zhang, and Mahintham Chandramohan. 2016. Semantics-based online malware detection towards efficient real-time protection against malware. *IEEE Trans. Inf. Forens. Secur.* 11, 2 (Feb. 2016), 289–302.

[10] Lucas Davi, Ra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad reza Sadeghi. 2012. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Symposium on Network and Distributed System Security*.

[11] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding control flows using intel processor trace. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 585–598.

[12] Zonglin Guo, Ram Bhakta, and Ian G. Harris. 2014. Control-flow checking for intrusion detection via a real-time debug interface. In *International Conference on Smart Computing Workshops*. 87–92.

[13] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing unique code target property for control-flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*. 1470–1486.

[14] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy*. 969–986.

[15] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. 2017. HBM (High Bandwidth Memory) DRAM technology and architecture. In *IEEE International Memory Workshop*. 1–4.

[16] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Comput. Archit. News* 42, 3 (2014), 361–372.

[17] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre attacks: Exploiting speculative execution. *IEEE Symposium on Security and Privacy*. 1–19.

[18] Dong Uk Lee, Kyung Whan Kim, Kwan Weon Kim, Kang Seol Lee, Sang Jin Byeon, Jae Hwan Kim, Jin Hee Cho, Jaejin Lee, and Jun Hyun Chun. 2015. A 1.2 V 8 Gb 8-Channel 128 GB/s High-Bandwidth Memory (HBM) stacked DRAM with effective I/O test circuits. *IEEE J. Solid-state Circ.* 50, 1 (2015), 191–203.

[19] Yongje Lee, Jinyong Lee, Ingoo Heo, Dongil Hwang, and Yunheung Paek. 2017. Using CoreSight PTM to integrate CRA monitoring IPs in an ARM-based SoC. *ACM Trans. Des. Autom. Electron. Syst.* 22, 3 (2017), 52:1–52:25.

[20] Tong Liu, Gang Shi, Liwei Chen, Fei Zhang, Yaxuan Yang, and Jihu Zhang. 2018. TMDFI: Tagged memory assisted for fine-grained data-flow integrity towards embedded systems against software exploitation. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications/IEEE International Conference on Big Data Science and Engineering*. 545–550.

[21] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Transparent and efficient CFI enforcement with Intel processor trace. In *IEEE International Symposium on High Performance Computer Architecture*. 529–540.

[22] 2003. LLVM. LLVM Team. Retrieved from https://llvm.org/.

[23] 2002. Null HTTPd Remote Heap Overflow Vulnerability. Netric Security. Retrieved from https://www.securityfocus.com/bid/5774.

[24] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N. Asokan, and Ahmad-Reza Sadeghi. 2017. HardScope: Thwarting DOP with hardware-assisted run-time scope enforcement. *arXiv preprint* (2017).

[25] Seth H. Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. NDC: Analyzing the impact of 3D-stacked Memory+Logic devices on MapReduce workloads. In *IEEE International Symposium on Performance Analysis of Systems and Software*. 190–200.

[26] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. 2011. RIPE. Retrieved from https://github.com/johnwilander/RIPE.

[27] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security*. 552–561.

[28] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2016. SMCsim. Retrieved from https://iis-git.ee.ethz.ch/erfan.azarkhish/SMCSim.

[29] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. 2016. Enforcing kernel security invariants with data flow integrity. In *Network and Distributed System Security Symposium*.

[30] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. 2016. HDFI: Hardware-assisted Data-Flow Isolation. In *IEEE Symposium on Security and Privacy*. 1–17.

[31] 2006. SPEC CPU 2006 Benchmark. Standard Performance Evaluation Corporation. Retrieved from https://www.spec.org/cpu2006/.

[32] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural static value-flow analysis in LLVM. In *International Conference on Compiler Construction*. 265–266.

[33] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[34] 2014. The Heartbleed Bug. Google Security. Retrieved from http://heartbleed.com/.

[35] Michael Macnair. 2014. The Source Code for Triggering Heartbleed Bug. Retrieved from https://github.com/mykter/afl-training/tree/master/challenges/heartbleed.

[36] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symposium on Security and Privacy*. 20–37.

[37] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. 2011. RIPE: Runtime Intrusion Prevention Evaluator. In *Computer Security Applications Conference*. 41–50.

[38] 2003. Exploit Mitigation Techniques. Retrieved from https://www.openbsd.org/papers/ven05-deraadt/index.html.

[39] Yubin Xia, Yutao Liu, H. Chen, and B. Zang. 2012. CFIMon: Detecting violation of control flow integrity using performance counters. In *International Conference on Dependable Systems and Networks*. 1–12.

[40] Xu Yang, Yumin Hou, and Hu He. 2019. A processing-in-memory architecture programming paradigm for wireless Internet-of-Things applications. *Sensors* 19, 1 (2019), 140.

Received May 2021; revised August 2021; accepted October 2021