TOWARD TAMING THE OVERHEAD MONSTER FOR DATA-FLOW INTEGRITY

Lang Feng[*§], **Jiayi Huang**[†], **Jeff Huang**[¶], **Jiang Hu**[§],
[*]*School of Electronic Science and Engineering, Nanjing University*
[†]*Department of Electrical and Computer Engineering, University of California, Santa Barbara*
[§]*Department of Electrical & Computer Engineering, Texas A&M University*
[¶]*Department of Computer Science & Engineering, Texas A&M University*
flang@nju.edu.cn; jyhuang@ucsb.edu; jeffhuang@tamu.edu; jianghu@tamu.edu;

February 22, 2021

ABSTRACT

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 verify the DFI defined in the original seminal work while reducing performance overhead by 4× on average.

***K*eywords** Data-Flow Integrity · Architecture · Security

1 Introduction

Data-Flow Integrity (DFI) is a regulation to ensure that data to be accessed are written by legitimate instructions [1]. As such, DFI verification can identify unwanted data modifications that are not consistent with programmer's intention. It can detect a wide variety of security attacks including control data attacks such as Jump-Oriented Programming (JOP) [2] and Return-Oriented Programming (ROP) [3], and non-control data attacks such as Heartbleed [4] and the heap overflow attack to Nullhttpd [5]. 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) [6], which is another well-known software security approach.

The concept of DFI was introduced in 2006 by the seminal work [1], and has received a lot of attention thereafter due to its potential of being a powerful security measure. However, a complete DFI enforcement as in [1] 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 [6], which has much more published studies [7, 8, 9, 10, 11, 12].

The few later works on DFI [13, 14, 15, 16] reduce the overhead by exploiting partial DFI, whose criteria are substantially lower than the original DFI definition [1]. The Hardware-Assisted Data-Flow Isolation (HDFI) [13] 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 verification granularity is very coarse and may miss attacks that mingle different data within the same region. Consider the example in Figure 1, where input data are first written into `u0` and `u1` in lines 10 and 11. Later, the data are copied to buffers in 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 in 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, a complete DFI [1] can isolate data among dozens of thousands of regions, i.e., a resolution $> 30000\times$ higher than HDFI. Therefore, the security price that HDFI paid for its overhead reduction can be very high.

```
1  struct vuln{
2    char data[256];
3    int off=0;
4    int size=0;
5  }*u0, *u1, *u2;
6  /* ============== */
7  char user0_buffer[256];
8  char user1_buffer[256];
9  char user2_buffer[256];
10 read_user_input(u0, user0_input);
11 read_user_input(u1, user1_input);
12 ...
13 memcpy(user0_buffer+u0->off, u0->data, u0->size);
14 memcpy(user1_buffer+u1->off, u1->data, u1->size);
15 memcpy(user2_buffer+u2->off, u2->data, u2->size);
```

Figure 1: An example of vulnerability that HDFI cannot detect.

Verifying the complete DFI [1] with practically acceptable overhead is a huge challenge. Different from most of existing overhead reduction techniques [13, 14, 15, 16], which rely on lowering the DFI criterion, we pursue a new approach that exploits additional hardware while the original DFI [1] can still be completely verified. 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 and, surprisingly, we discover that the frequent DFI data access does not lead to frequent memory access and thus, memory access is not a bottleneck, but the DFI kernel computations usually contribute the most to the overhead. We propose a parallel approach, where kernel 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/compiler modifications.

Overall, the proposed approach reduces performance overhead from 161% of [1] to an average of 36% on the same SPEC CPU2006 benchmarks. As it is a complete DFI verification, it can detect a wide range of security attacks and cover cases that cannot be handled by the previous low-overhead methods [13, 14, 15, 16]. Our approach provides a solution with a security-overhead tradeoff in complement to existing methods [13, 14, 15, 16]. 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 to complete DFI verification, 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).
- The tradeoff between DFI violation detection latency and performance overhead is studied.
- Our approach achieves about $4\times$ overhead reduction, which is a major progress for complete DFI since 2006.

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 [1]. Every instruction in a program is assigned a numerical **identifier** through automatic code instrumentation. The **reaching definition** of an instruction `A` is the latest instruction `B` that stores the data loaded by `A` 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.

Table 1: Comparison between our work and others.

Method	Performance Overhead	DFI Verification Completeness	Approach	New Instruction	OS Change	Compiler Change	Instrument
SW DFI [1]	161%	Complete	SW	×	×	×	√
KENALI [14]	7-15%	Partial	SW	×	√	×	√
WIT [15]	7%	Partial	SW	×	×	×	√
CHERI [17]	5-20%	Partial	HW	√	√	√	×
TMDFI [16]	39%	Partial	HW	√	×	×	×
HDFI [13]	<2%	Partial	HW	√	√	√	×
Our work	**39%**	**Complete**	**HW**	×	×	×	√

In the example of Figure 2, "`store x y`" means storing variable `x` at address `y`, "`load x y`" is to load the data at address `y` to variable `x`, and "`jump label`" implies an unconditional branch to the location marked by `label`. If the identifier of each instruction is the same as its line number, the RDS of line 7's instruction is $\{1\}$. 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 be indeed 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)**.

```
1  store x1 addr1
2  store x1 addr2
3  jump label
4  store x2 addr1
5  load x3 addr1
6  label:
7  load x4 addr1
```

Figure 2: A code example for illustrating DFI.

DFI is a superset of Control-Flow Integrity (CFI) [6], which only regulates instruction flow transitions toward target addresses conforming to the original design intention. Attackers have to modify control data, such as the target address for an indirect branch, to change a control flow. By protecting all the data, DFI can also prevent all control-flow attacks. Additionally, DFI can protect non-control data that cannot be covered by CFI.

A general threat model for DFI is that computer hardware and OS are secure, while attackers can manage to view the binary code and opportunistically modify some program data, e.g., through buffer overflow.

3 Previous Works

The concept of Data-Flow Integrity (DFI) was proposed in the seminal work [1] in 2006. This work also provides a software implementation technique and optimization techniques for overhead reduction. Although the DFI verification procedure is simple, its performance overhead is intrinsically huge as the verification needs to be conducted for tremendous data.

The few later previous works [14, 15, 13, 17, 16] achieved much lower overhead by focusing on partial DFI. The work of [14] 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 [5], Heartbleed [4] and data-oriented programming [18] 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) [15] 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 [4] is an attack that WIT would fail to detect.

Data isolation is another approach to protecting data with relatively low overhead. A hardware solution for data-flow isolation, called HDFI, is proposed in [13]. 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, processor hardware, operating system and compiler also need changes. If data belongs to one region, 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. Thus, its low overhead of $< 2\%$ comes with the price of very coarse grained security resolution. To a certain degree, the original DFI [1]

can be regarded as data isolation among individual instructions. If 16 bits are used for each instruction identifier, it is equivalent to isolation among up to 2^{16} regions. Compared to the only 2 regions of HDFI [13], the resolution of the original DFI is $2^{15} = 32768$ times higher. Similar to HDFI, TMDFI [16] also verifies 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 >1000 and sometimes >10000 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 [19] uses 1-bit tag for each word of data to indicate its integrity level in Biba's low-water-mark integrity policy [20], which requires that an instruction can only modify data with integrity level no higher than that of the instruction. In [19], processor hardware is modified to enforce this policy for control data protection. In [17], a 256-bit tag is employed to specify if each data can be referred by certain instructions. However, the 256-bit in [17] has different meaning from the 16-bit identifier in our approach. For security, the approach in [17] only handles the permission of pointers. In contrast, our approach handles the permission of every store/load instruction. Overall, the tag-based techniques [13, 19, 17, 16] provide only coarse-grained isolation as different data/instructions with the same tag cannot be isolated from each other.

4 Performance Overhead Analysis

We analyze the source of performance overhead of software DFI [1]. We call the program to be checked by DFI verification 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.

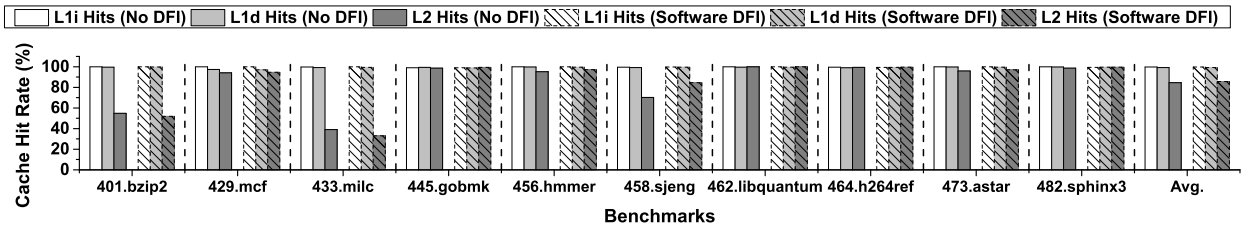

Figure 3: Cache hit rates of user programs with and without software DFI.

The cache hit rates of user programs without DFI verification and with software DFI are shown in Figure 3. One can see that the cache hit rates are usually greater than 95% regardless with DFI verification or not. This indicates that memory access is probably not a bottleneck.

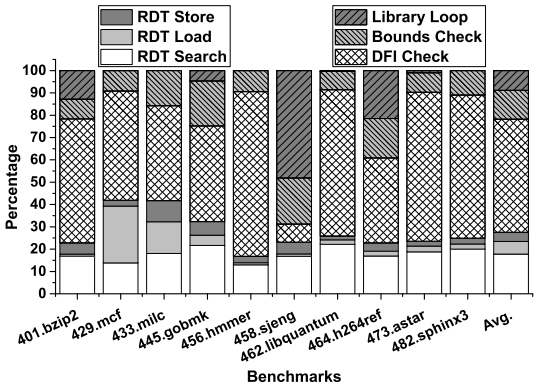

Figure 4: Overhead breakdown of software DFI.

We further investigated the overhead breakdown of software DFI, of which the results are shown in Figure 4, where "RDT Search" represents the instrumentation execution to find the RDT entry of the corresponding user `load` or `store`. "Bounds Check" means the check for preventing RDT from illegal modification. "Library Loop" is the additional loop

in the instrumented wrapper of each library function. "DFI Check" indicates the comparisons verifying the identifier found in RDT Search is in the RDS of the corresponding user `load`.

According to Figure 4, most of the overhead is from DFI check. It also shows that RDT access itself (excluding the RDT search part) 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.

5 Overview of Proposed Approach

Our approach is to delegate DFI verification 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 [21]. The two options are similar in terms of the overhead reduction. A non-essential yet non-trivial difference is that the PIM approach entails less data movement as RDSs and RDT reside in memory. Thus, the PIM approach is more power-efficient. Moreover, the DFI kernel computation is simple and a PIM processor would suffice, whereas a CMP core is usually an overkill. An approximate estimate [22, 23] tells that the circuit area of a PIM processor is often $< 10\%$ of the main processor under the same technology. We will use PIM as a platform to describe our approach while the same idea is applicable to the CMP core option.

We first summarize the information required for DFI verification by PIM and their locations as follows.

1. RDS (Reaching Definition Set) for all user `load` instructions in the program. This information does not change throughout the program execution and can be loaded into PIM once at the beginning.
2. RDT (Reaching Definition Table). This information changes dynamically during a program execution. It is maintained by the PIM processor, and therefore is local to DFI verification at PIM .
3. Target instruction information. A **target instruction** is an instruction in a user program to be verified for DFI. Mainly two types of instructions are involved: `load` instructions for which DFI verification is performed, and `store` instructions that affect RDT. These two pieces of information change at runtime and need to be transferred from the main processor to memory. It consists of the following components:
   - Instruction identifier.
   - Instruction type: either `load` or `store`.
   - Target address of `load` or `store`.

The PIM processor undertakes most of the DFI verification components analyzed in Section 4, 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 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. Moreover, the hardware approach facilitates runtime application of optimizations described in Section 7.4.

The overall flow of the proposed DFI verification is depicted in Figure 5, where green numbers indicate 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 `store` instructions, which are called **DFI store**s and in red font in Figure 5, after each target instruction so as to help collect its information.
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, a dedicated hardware, called **info-collector** in Figure 5, parses each DFI `store`, collects target instruction information accordingly, forms a **DFI packet**, and sends it to the PIM processor, where verification computations are performed or RDT is updated.

6 Software Instrumentation

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

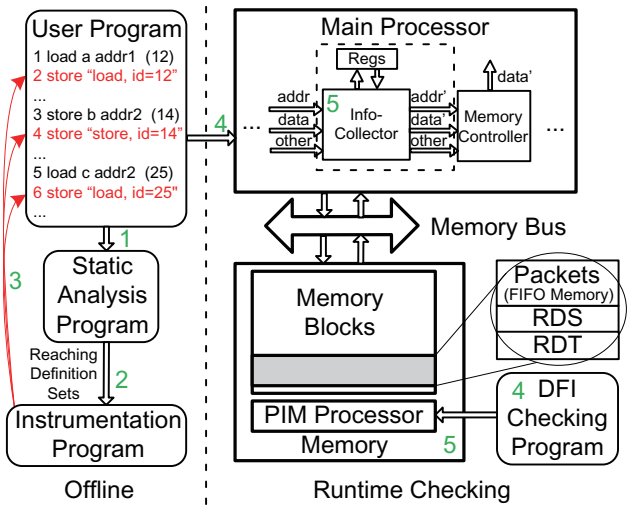

Figure 5: The flow of PIM DFI verification.

the instruction set. The description here is based on C/C++ programs compiled by LLVM [24] and the static analysis is performed by SVF [25]. However, our techniques are general and directly applicable to other software languages, compilers and static analysis tools.

Given a program's LLVM Intermediate Representation (IR), static analysis is performed to obtain its reaching definition sets (RDSs), which will be sent to PIM at the beginning of code execution. The instrumentation is automatically performed on an IR by a software that we developed. Then, the instrumented IR is further compiled into binary code. Although the instrumentation is inserted in the middle of a compiling flow, it does not require any changes to compiler code. In absence of source code such as proprietary software, our method can still be applied by employing a binary code analysis tool and instrumentation for the binary code.

6.1 Instrumentation for DFI Verification

The instrumentation is mainly to extract the runtime information of target instructions, which are 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. An example of code instrumentation is shown by the red font instructions in Figure 5. These instrumentation `store` instructions are called **DFI** `store`, which we overload its 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 `dfi_global` is the address 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 info-collector (dotted box in Figure 5) checks if a `store` instruction has a target address the same as that of `dfi_global`. If yes, then the instruction is a DFI `store`.

Every `store` and `load` instruction in a user program, called target instruction, is followed by a DFI `store`. The `runtime_info` contains the instruction type and identifier of the proceeding target instruction. For example, in Figure 5, line 2 is an instrumentation instruction `store "load, id=12"`, which tells the instruction type and identifier of the target instruction in line 1. To encode the instruction type and identifier, according to [1], 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 proceeding 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 verification. 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. We specify it by adding the following instruction at the beginning of each user program:

```
store packet_dummy packet_mem_addr
```

The `packet_dummy` is a dummy packet that has a fixed value to obtain the destination address for future DFI packets. The info-collector can obtain `packet_mem_addr` by indentifing the first `store` in a program that stores `packet_dummy` to an address. For example, `packet_dummy` can be designed as `123456`. Once `store 123456 11122` is executed, the info-collector assigns `11122` to `packet_mem_addr`, and `packet_mem_addr` can only be assigned one time for a program. 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.

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

Figure 6: An example of code instrumentation.

6.2 Handling Library Functions

A software program often calls library functions, whose source code or IR is not directly accessible. However, instrumentation can still be performed to obtain target instruction information, which is a library function call. This is similar to the wrapper [1] in spirit, but our realization is quite different. 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. Our approach is illustrated using the example in Figure 7.

```
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)     //(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)     //(15)
```

Figure 7: The instrumentation for library functions.

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.

6.3 Function Return Protection

Function return addresses are stored in stack and vulnerable to security attacks such as Return-Oriented Programming (ROP) [3]. We treat their accesses as implicit `load/store` instructions and perform DFI check accordingly. When

a parent function `parent_func()` calls a child function `child_func()`, the return address is stored in the stack by an instruction `parent_inst`. When function `child_func()` returns, the return 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`. However, function return is not covered by some static analysis tools like SVF [26]. Thus, we develop a dedicated instrumentation technique different from that for ordinary `load/store` instructions. Although a similar idea was also proposed in [1], our instrumentation is quite different.

```
1  /* =====beginning of the function====== */
2  p_ret_addr = instruction_getting_ret_addr_pointer
3  store (1<<21)+(max_id+thread_id) dfi_global
4  store p_ret_addr dfi_global
5  ...
6  store (1<<21)+(1<<16)+(max_id+thread_id) dfi_global
7  store p_ret_addr dfi_global
8  return
```

Figure 8: Instrumentation for function return.

The instrumentation for function return is illustrated in Figure 8. At the beginning (line 2), the pointer to return address `p_ret_addr` is obtained. For a C/C++ program, this can be realized by calling built-in function `__builtin_frame_address(0)` and adding 4 to the returned result. We designate the identifier of the implicit `store` instruction (function call) `parent_inst` as the maximum identifier from the static analysis plus the thread ID (lines 3 and 6). This ensures that the identifier of `parent_inst` is unique. Bit 21 of the data in the DFI `store` in line 3 is set to 1, to inform the info-collector that this is for function return. Then, the info-collector expects a subsequent DFI `store` for the pointer to return address. The info-collector combines instruction type (implicit `load/store`), identifier and the pointer to form a DFI packet. At the end of the child function (lines 6 and 7), similar instrumentation instructions are added for the implicit `load` (function return). For each `load` whose identifier is larger than the maximum identifier of static analysis, DFI requires the identifier of the latest `store` to be the same as the identifier of this `load`.

7 Hardware Design

7.1 DFI Packet Generation

Info-collector is the key hardware component to be added at the main processor. It detects DFI `store` instructions, collects runtime information of a target instruction, generates DFI packets and sends them to PIM. It can be realized as a combinational circuit through synthesizing Verilog description. Its basic operations are depicted in Figure 9.

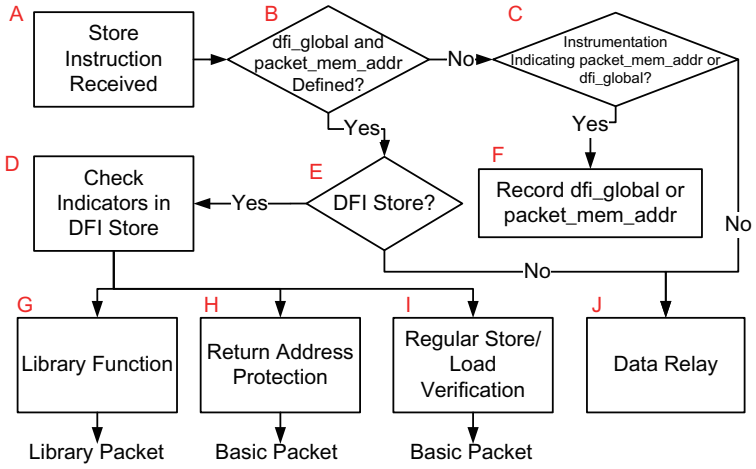

Figure 9: Operations of info-collector.

The info-collector acts only when a `store` instruction is executed. In step B of Figure 9, it checks if `dfi_global` and `packet_mem_addr` have already been defined. If not, it proceeds to step C to capture `dfi_global` or `packet_mem_addr`. Please note "`store dfi_dummy dfi_glabal`" 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, 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`, the info-collector parses the indicators in the data part of the DFI `store` and tells if this is to verify `load/store`, function return or a library function call. If this instrumentation is for a `load/store` instruction, 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 9), 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), 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`, its `data` is relayed to memory without any change and its target address is stored in a local register for future use.

7.2 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. In order to maintain the FIFO nature using a region of random access memory with low overhead, we develop circuit design 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). Due to space limit, we omit the detailed description for brevity.

7.3 Lossless Data Compression

A main reason for performance overhead of PIM 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 10. For example A, assume the starting memory address of `aa` is 0x8000, then the program stores data at 0x8000, 0x8004, 0x8008, 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 10 is similar, but has an address pattern of 0x8000, 0x8400, 0x8800, etc. Although the address increment 0x400 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 0x400, 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.

```
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;
```

Figure 10: Examples of address locality.

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.

7.4 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 [1]. However, the pruning techniques in [1] 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 other packets to form a block increases DFI verification 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. After P_1 and P_2, if packets Q_1 and Q_2 are for `store`/`load` of another same target address, and Q_1/Q_2 have the same identifiers as P_1/P_2, 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.

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. DFI verifications for `load`/`store` of different target addresses are independent of each other and hence sorting does not affect DFI verification results.

Among the optimizations, A, B and C are similar to those in [1] except that they can be performed both offline and at runtime while those in [1] 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.

7.5 Circuit Implementation of the Optimizations

All the 5 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.

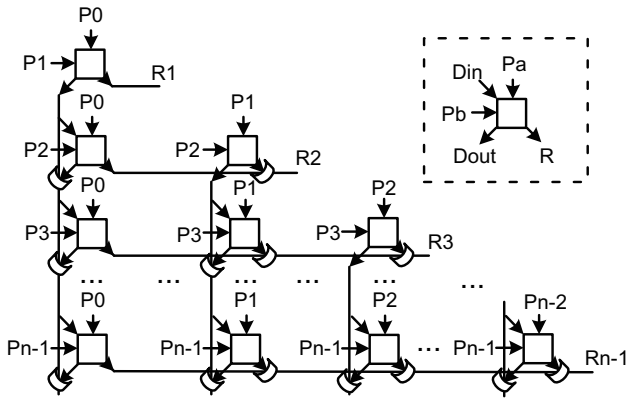

Figure 11: Circuit for implementing optimization C.

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 i-th packet, and *Ri* indicates if the i-th 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, 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 *OR*ed 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.

8 DFI Verification Program at PIM

The DFI verification 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 [1], all program data are organized in words, each of which requires one RDT entry. If the data memory for user program has N bytes, there are $N/4$ entries in the RDT [1]. Since each identifier has 16 bits = 2 bytes, the RDT uses $\frac{N\times 2}{4} = N/2$ bytes of memory. The verification program at the PIM processor continuously reads DFI packets from the FIFO memory, and either performs DFI verification or updates RDT. There are three kinds of DFI packets to be processed by the verification program.

- **Basic packet for `store` or `load`:** The verification program extracts instruction type, identifier α and target address β from the packet. If the instruction type is `store`, 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`, the verification 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, 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 verification 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 address α, the verification program loads the identifiers ε_0, ε_1... $\varepsilon_{\gamma-1}$ from entries $\alpha >> 2$, $(\alpha >> 2)+1$, ... $(\alpha >> 2)+\gamma-1$ in the RDT, and checks if every ε_i is in the RDS of identifier δ. If there is address β, the program stores identifier δ to all the entries from $\beta >> 2$ to $(\beta >> 2)+\gamma-1$ in the RDT.

9 Experiment

9.1 Experiment Setup

We evaluate our approach and the proposed techniques using architecture simulations through SMCsim [27, 21], which is an extension to the gem5 simulator [28] for accommodating PIM. The main processor is an ARM Cortex-A15 with 2GHz frequency, 32KB L1 instruction cache, 64KB L1 data cache, 2MB L2 cache, and 512 MB memory. A single PIM processor is used and operates at 2GHz frequency [29, 30]. 64MB memory is allocated for RDT, which is sufficient for the testcases in our experiment. Other details of the PIM can be found in [21, 27]. Please note that the PIM configuration has little impact on the user program execution.

9.2 Security Analysis

Our approach verifies the same DFI as defined in [1] and thus achieves similar security as [1] except that our approach is asynchronous monitoring [11, 31, 7], where detection of DFI violation can trigger system interrupt for further security measures, rather than synchronous enforcement like [1]. This difference is a tradeoff between security and service availability. Synchronization inevitably entails extra performance overhead as DFI verification blocks user program executions.

9.2.1 Comparison with HDFI and TMDFI

Hardware-assisted Data-Flow Isolation (HDFI) [13] verifies partial DFI at a very coarse granularity. It uses a 1-bit tag to differentiate a sensitive region and a non-sensitive data region, and only ensures that data in one region are not lastly written by an instruction for the other region. As such, it cannot detect attacks that mingles data within the same region. For the example of Figure 1, we exhaustively tested different tag schemes of HDFI, which are listed in the left three columns of Table 2. For each tag scheme, there is some overflow that cannot be detected by HDFI as shown in column 4, where $u0 \Rightarrow u1$ means some data of user0 is written into user1's space through overflow. By contrast, our approach can successfully detect all these overflows.

Table 2: Scenarios for Figure 1 where HDFI fails.

HDFI				Our approach
$u0$	$u1$	$u2$	Missed overflow	detect?
Tag 0	Tag 0	Tag 0	$u0 \Rightarrow u1$, $u0 \Rightarrow u2$, $u1 \Rightarrow u2$	Yes
Tag 0	Tag 0	Tag 1	$u0 \Rightarrow u1$	Yes
Tag 0	Tag 1	Tag 0	$u0 \Rightarrow u2$	Yes
Tag 0	Tag 1	Tag 1	$u1 \Rightarrow u2$	Yes
Tag 1	Tag 0	Tag 0	$u1 \Rightarrow u2$	Yes
Tag 1	Tag 0	Tag 1	$u0 \Rightarrow u2$	Yes
Tag 1	Tag 1	Tag 0	$u0 \Rightarrow u1$	Yes
Tag 1	Tag 1	Tag 1	$u0 \Rightarrow u1$, $u0 \Rightarrow u2$, $u1 \Rightarrow u2$	Yes

TMDFI [16] employs an 8-bit tag and thus can differentiate data among 256 regions. Although this is a significant improvement over HDFI, its verification 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 [16] 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.

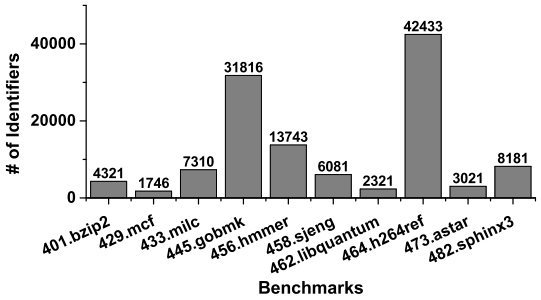

Figure 12: The number of identifiers of each benchmark.

9.2.2 RIPE Benchmark

RIPE [32, 33] 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) [3] attacks and Jump-Oriented Programming (JOP) [2] attacks. In addition, we also prepared a RIPE program without any attack. It is observed that our DFI system successfully identifies all the 156 attacks and does not make false alarm for the case without attack.

9.2.3 Heartbleed

Heartbleed (CVE-2014-0160) [4] 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 [34] to simulate such attack. This attack is successfully detected by our DFI verification 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.

9.2.4 Nullhttpd

Nullhttpd is a HTTP server that has heap overflow vulnerability (CVE-2002-1496) [5]. 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 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.

9.3 Performance Overhead

Performance overheads of the following methods are evaluated through simulations on the SPEC CPU 2006 benchmark [35].

- Software. This is the original software DFI by [1].
- HBM. This is similar to [1] except that High Bandwidth Memory [36, 37] is employed.
- CMP. This is a parallel approach, where DFI verification 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.

Our **proposed approach** has two variants: CMP-H and PIM. To ensure a fair comparison, each application was terminated at the same point in the simulations. The results are summarized in Table 3. As the static analysis tool failed in some applications, results are only shown for the successful runs.

Table 3: Performance overhead of DFI. †Computation time of optimizations and compression is neglected. ‡Computation time of optimizations and compression is considered. §No DFI packet is sent to the memory.

		Software [1]	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‡
Benchmarks	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%**

On average, the performance overhead of software DFI [1] 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 4. The results of 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 5 optimization techniques are applied and "C, E" corresponds to the results where only the two most effective 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 CPM. 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.

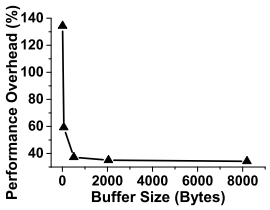

Figure 13: Overhead vs. buffer size.

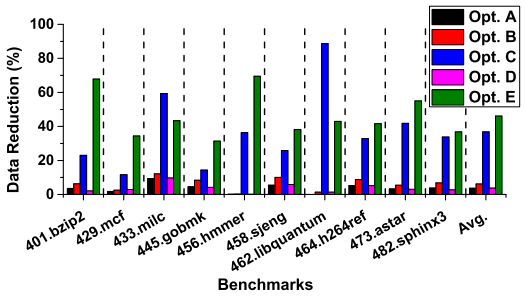

Figure 14: Effects of optimization techniques.

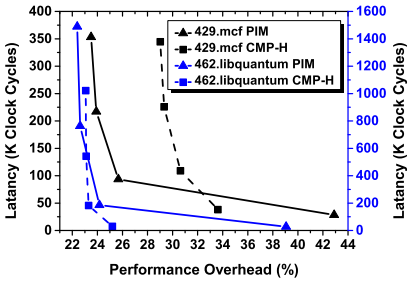

Figure 15: Detection latency vs. overhead for *429.mcf* and *462.libquantum*.

The effects of the 5 optimization techniques described in Section 7.4 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**.

9.4 Tradeoff Between Detection Latency and Overhead

Ideally, the latency for detecting DFI violations need to be minimized so that 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 slight better for obtaining low latency.

9.5 Hardware Circuit Overhead

The info-collector circuit is implemented by synthesizing Verilog using Synopsys Design Compiler and ASAP 7nm cell library [38]. The info-collector with basic operation and compression costs only 2908 gates and less than 30ps 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 3, where the gate counts of the info-collector with different buffer sizes are listed. The 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 that 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. The works of CHERI [17] and HDFI [13] did not describe their hardware details. However, they can isolate only between 2 regions, and their hardware cost is almost impossible to be less than 24 gates. Therefore, the hardware cost per region of our approach is less than CHERI and HDFI.

10 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 to check 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 verifying complete DFI. In future research, we will study how to further reduce the performance overhead and detection latency.

References

[1] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-Flow Integrity. *Symposium on Operating Systems Design and Implementation*, pages 147–160, 2006.

[2] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented Programming: A New Class of Code-reuse Attack. *ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[3] Hovav 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*, pages 552–561, 2007.

[4] The Heartbleed Bug. `http://heartbleed.com/`.

[5] Null HTTPd Remote Heap Overflow Vulnerability. `https://www.securityfocus.com/bid/5774`.

[6] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity. *ACM Conference on Computer and Communications Security*, pages 340–353, 2005.

[7] Yongje Lee, Jinyong Lee, Ingoo Heo, Dongil Hwang, and Yunheung Paek. Using CoreSight PTM to Integrate CRA Monitoring IPs in an ARM-Based SoC. *ACM Transactions on Design Automation of Electronic Systems*, 22(3):52:1–52:25, 2017.

[8] Zonglin Guo, Ram Bhakta, and Ian G. Harris. Control-flow Checking for Intrusion Detection via a Real-time Debug Interface. *International Conference on Smart Computing Workshops*, pages 87–92, 2014.

[9] Xinyang Ge, Weidong Cui, and Trent Jaeger. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 585–598, 2017.

[10] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Transparent and Efficient CFI Enforcement with Intel Processor Trace. *IEEE International Symposium on High Performance Computer Architecture*, pages 529–540, 2017.

[11] Yubin Xia, Yutao Liu, Haibo Chen, and Zang Binyu. CFIMon: detecting violation of control flow integrity using performance counters. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, 2012.

[12] Lucas Davi, Ra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad reza Sadeghi. MoCFI: A Framework to Mitigate Control-flow Attacks on Smartphones. *Symposium on Network and Distributed System Security*, 2012.

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

[14] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. *Network and Distributed System Security Symposium*, 2016.

[15] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing Memory Error Exploits with WIT. *IEEE Symposium on Security and Privacy*, pages 263–277, 2008.

[16] Tong Liu, Gang Shi, Liwei Chen, Fei Zhang, Yaxuan Yang, and Jihu 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*, pages 545–550, 2018.

[17] 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. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. *IEEE Symposium on Security and Privacy*, pages 20–37, 2015.

[18] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. *IEEE Symposium on Security and Privacy*, pages 969–986, 2016.

[19] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. *IEEE/ACM International Symposium on Microarchitecture*, pages 221–232, 2004.

[20] Ken Biba. Integrity Considerations for Secure Computer Systems. *Defense Technical Information Center*, page 68, 1977.

[21] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. Design and Evaluation of a Processing-in-Memory Architecture for the Smart Memory Cube. *International Conference on Architecture of Computing Systems*, pages 19–31, 2016.

[22] Youngmin Shin, Hoi-Jin Lee, Ken Shin, Prashant Kenkae, Rajesh Kashyap, DongJoo Seo, Brian Millar, Yohan Kwon, Ravi Iyengar, Min-Su Kim, Ahsan Chowdhury, Sung-Il Bae, Inpyo Hong, Wookyeong Jeong, Aaron Lindner, Uk-Rae Cho, Keith Hawkins, Jae-Cheol Son, and Sung-Ho Park. 28nm high-K metal gate heterogeneous quad-core CPUs for high performance and energy-efficient mobile application processor. In *Proceedings of the IEEE International SoC Design Conference*, 2013.

[23] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. The mondrain data engine. In *Proceedings of the ACM International Symposium on Computer Architecture*, pages 639–651, 2017.

[24] LLVM. `https://llvm.org/`.

[25] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 265–266, New York, NY, USA, 2016. ACM.

[26] SVF for Reaching Definition Analysis. `https://github.tamu.edu/jyhuang/SVF`.

[27] SMCsim. `https://iis-git.ee.ethz.ch/erfan.azarkhish/SMCSim`.

[28] The gem5 Simulator. `http://www.gem5.org/Main_Page`.

[29] Xu Yang, Yumin Hou, and Hu He. A Processing-in-Memory Architecture Programming Paradigm for Wireless Internet-of-Things Applications. *Sensors*, 19(1):140, 2019.

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

[31] Sanjeev Das, Yang Liu, Wei Zhang, and Mahintham Chandramohan. Semantics-based online malware detection towards efficient real-time protection against malware. *IEEE Transactions on Information Forensics and Security*, 11(2):289–302, February 2016.

[32] RIPE. `https://github.com/johnwilander/RIPE`.

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

[34] The Source Code for Triggering Heartbleed Bug. `https://github.com/mykter/afl-training/tree/master/challenges/`

[35] SPEC CPU 2006 Benchmark. `https://www.spec.org/cpu2006/`.

[36] 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. A 1.2 V 8 Gb 8-Channel 128 GB/s High-Bandwidth Memory (HBM) Stacked DRAM With Effective I/O Test Circuits. *IEEE Journal of Solid-State Circuits*, 50(1):191–203, 2015.

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

[38] ASAP 7nm Predictive PDK. `http://asap.asu.edu/asap/`.