HDFI: Hardware-Assisted Data-flow Isolation

Chengyu Song*, Hyungon Moon†, Monjur Alam*, Insu Yun*, Byoungyoung Lee*,
Taesoo Kim*, Wenke Lee*, Yunheung Paek†

*Georgia Institute of Technology
†Seoul National University

***Abstract*—Memory corruption vulnerabilities are the root cause of many modern attacks. Existing defense mechanisms are inadequate; in general, the software-based approaches are not efficient and the hardware-based approaches are not flexible. In this paper, we present *hardware-assisted data-flow isolation*, or, HDFI, a new fine-grained data isolation mechanism that is broadly applicable and very efficient. HDFI enforces isolation at the machine word granularity by virtually extending each memory unit with an additional tag that is defined by data-flow. This capability allows HDFI to enforce a variety of security models such as the Biba Integrity Model and the Bell–LaPadula Model. We implemented HDFI by extending the RISC-V instruction set architecture (ISA) and instantiating it on the Xilinx Zynq ZC706 evaluation board. We ran several benchmarks including the SPEC CINT 2000 benchmark suite. Evaluation results show that the performance overhead caused by our modification to the hardware is low ($< 2\%$). We also developed or ported several security mechanisms to leverage HDFI, including stack protection, standard library enhancement, virtual function table protection, code pointer protection, kernel data protection, and information leak prevention. Our results show that HDFI is easy to use, imposes low performance overhead, and allows us to create more elegant and more secure solutions.**

I. Introduction

Memory corruption vulnerabilities are the root cause of many modern attacks. To defeat such attacks, many security features have been commoditized, including NX-bit (No-eXecute), Supervisor Mode Execution Protection (SMEP), Supervisor Mode Access Prevention (SMAP), Memory Protection Extension (MPX), which have provided a strong foundation for security in today's computer systems. However, while these hardware-based security features are very efficient, they do not provide adequate protection against modern, complex memory-corruption-based attacks. For example, NX-bit can eliminate simple forms of code injection attacks, but cannot stop code-reuse attacks such as return-to-libc attack [26], return-oriented programming (ROP) [65], COOP [62], and non-control data attacks [14, 35, 66].

To defeat these new attacks, researchers continue to develop new hardware-based mechanisms. For example, hardware-based shadow stacks have been proposed to protect return addresses from tampering by adversaries [46, 59, 81]. Hardware-based control-flow integrity (CFI) has also been proposed to prevent code-reuse attacks, with various trade-offs [18, 23, 24, 41]. Furthermore, a number of other approaches have been proposed to eliminate the root cause of these memory corruption vulnerabilities [27, 51, 52, 77].

Our work also aims to prevent memory corruption based exploits. Towards this end, we take the direction of developing a new hardware feature that provides both flexibility (i.e., applicable to broad use cases) and performance (i.e., very efficient when activated and otherwise near-zero performance overhead on common execution paths). Our key observation is that even with hardware support, enforcing memory safety for the whole application is still too expensive for practical use, e.g., WatchDogLite [52] imposes 29% slowdown on SPEC CINT 2006 benchmarks. To further reduce performance overhead, one promising direction is to divide the memory into different regions—one for sensitive data (e.g., function pointers) and the other for the rest (e.g., application data). Then, we enforce memory safety only over the sensitive region [43, 63, 66]. There are two major advantages of this approach. First, sensitive data is usually a smaller set than normal data, and less data implies fewer checks and less performance overhead. Second, the safety of memory operations over sensitive data is easier for static verification. For example, because pushing/popping data onto/from stack is always safe, once we isolate the stack slots used to store return addresses, we can guarantee memory safety for return addresses without any runtime check. With these two advantages, we can significantly reduce the number of runtime checks, thereby making memory safety more affordable. However, implementing this strategy on commodity hardware is non-trivial due to the lack of *an efficient, fine-grained mechanism for data isolation*.

Table I compares existing software-based (top half) and hardware-based (bottom half) isolation mechanisms on *commodity hardware*. The most apparent problem is that the two most efficient hardware-based mechanisms—segment in x86 and access domain in ARM processors, are absent on 64-bit mode. As a result, security solutions in modern processors must make a trade-off between security and performance—solutions that opt for performance, e.g., by using randomization based protection, are usually subject to information disclosure or brute-force based attacks [16, 32], while solutions that opt for security, e.g., by leveraging context switch or masking, usually yield poorer performance [19, 63, 66].

However, even if we managed to bring back the segment and access domain, these mechanisms are still inadequate. Specifically, because they are all coarse-grained, if we want to isolate data at a smaller granularity (e.g., function pointers) and preserve a program's original memory layout, then we must perform *data shadowing*. Unfortunately, data shadowing breaks

	Mechanism	(1) Data shadowing	(2) Context switch	(3) Liveness tracking	(4) Availability on 64-bit	(5) Self protection	(6) Vulnerable to info leak	(7) Performance overhead
Software-based	Randomization [43, 63]	Y	N	N	Y	Y	Y	low
	Masking [19, 43, 63]	Y	N	N	Y	Y	N	moderate
	Access control list [11, 30]	N	N	Y	Y	Y	N	high
Hardware-based	x86 memory segment [43, 83]	Y	N	N	N	Y	N	low
	ARM access domain [87]	Y	Y	N	N	Y	N	moderate
	Virtual address space [66]	Y	Y	Y	Y	Y	N	high
	Privilege level	Y	Y	Y	Y	N	N	moderate
	Virtualization [64]	Y	Y	Y	Y	N	N	high
	TrustZone [4]	Y	Y	Y	Y	N	N	very high
	ADI [57]	N	N	Y	Y	Y	N	low[1]
	HDFI	N	N	N	Y	Y	N	low

TABLE I: Comparison between HDFI and other isolation mechanisms, based on (1) whether data shadowing is required, (2) whether context switch is required for data access, (3) whether liveness tracking is required, (4) is available on 64-bit mode, (5) whether they can be used for self-protection, (6) is vulnerable to information leak, and (7) performance overhead. Self-protection means whether the mechanism can be used to prevent attacks from the same privilege-level. Performance overhead is measured by comparing one instrumented read/write operation against a normal memory read/write operation. [1]There is no public benchmark result for ADI, so this conclusion is purely based on their presentation [57].

data locality and requires extra steps to retrieve the shadow data.This introduces additional performance overhead [21]. Furthermore, data shadowing also introduces unavoidable memory overhead.

To overcome these limitations, we propose *hardware-assisted data-flow isolation* (HDFI), a new fine-grained data isolation mechanism. To eliminate data shadowing, HDFI enforces isolation at machine word granularity by virtually extending each memory unit with an additional tag. We choose to enforce isolation at this granularity because it balances the memory overhead (finer granularity requires more spaces for tags) and application requirements—a majority of sensitive data like pointers are at this granularity, and the rest can be easily aligned through software approaches. Please also note that HDFI's tags are associated with memory units' *physical addresses*, so attackers cannot tamper or bypass the protection by mapping the same physical page to different virtual addresses. Moreover, instead of using static partition, the tag is defined by data-flow. Inspired by the idea of data-flow integrity [10], HDFI defines the tag of a memory unit by the last instruction that writes to this memory location; then at memory read, it allows a program to check if the tag matches what is expected. This capability allows developers to enforce different security models. For example, to protect the *integrity* of sensitive data, we can enforce the Biba Integrity Model [6]. In particular, we can use the tag to indicate integrity level (IL) of the corresponding data: sensitive data has `IL1` and normal data has `IL0`. Next, we assign IL to write operations based on the data-flow. That is, we use static analysis to identify write operations that can manipulate sensitive data, and allow them to set the memory tag to `IL1`; all other write operations will assign to the tag to `IL0`. Finally, when loading sensitive data from memory, we check if the tag is `IL1` (see §III for a concrete example). HDFI can also be used to enforce *confidentiality*, i.e., the Bell–LaPadula Model [5]. For instance, to protect sensitive data like encryption keys, we can set their tag to `SL1` (secret level 1), and enforce that all untrusted read operations (e.g., when copy data to an output buffer) can only read data with tag `SL0`.

We implemented a prototype of HDFI by extending the RISC-V instruction set architecture (ISA) [72], an open-source, license-free ISA that is designed with direct hardware implementation and practical applications in mind. Our prototype implementation was designed to support one-bit tag for two reasons: (1) it limits the amount of required resources; and (2) as discussed above, in most security applications, one-bit tags are sufficient.

To evaluate the performance of HDFI, we instantiated it on the Xilinx Zynq ZC706 evaluation board [80] and ran several benchmarks including the SPEC CINT 2000 [68] benchmark suite. Evaluation results show that the performance overhead caused by our modification to the hardware is low ($< 2\%$).

In order to demonstrate the benefit of HDFI to security solutions, we developed and ported six representative security mechanisms to leverage HDFI, including stack protection, standard library enhancement (protection for `setjmp/longjmp`, heap metadata, GOT, and the exit handler), virtual function table protection, code pointer separation, kernel data protection, and information leak prevention. Our development experience shows that HDFI is easy to use and usually allows us to create more elegant solutions. We also evaluated the security and performance benefits of HDFI. The results show that HDFI can help improve the security guarantees. At the same time, by eliminating data shadowing and context switching, HDFI can also help reduce the performance overhead for security mechanisms like CPS [43] and Kenali [66].

To summarize, this paper makes the following contributions:

- **Design**: We present a new hardware security mechanism, which is general, efficient, backwards compatible, and only requires small hardware modification. We also present optimization techniques to minimize HDFI's performance impact on normal operations (§IV).
- **Applications**: To demonstrate the benefits of HDFI, we developed/ported six security mechanisms that utilize HDFI and analyzed how HDFI can enhance their security and performance (§V).

- **Implementation**: We implemented the proposed hardware with realization on FGPA board. We also implemented all six security applications (§VI).
- **Evaluation**: We quantitatively evaluated: (1) the performance impact of HDFI and the effectiveness of our optimization techniques and (2) the performance improvement delivered to the security mechanisms we implemented (§VII).

The rest of this paper is organized as follows. §II defines the threat model and the problem scope. §III uses a concrete example to explain how HDFI works, and discusses the differences between HDFI and similar work. §IV presents the the design of HDFI. §V describes the security applications we have developed. §VI provides some implementation details. §VII describes the evaluation of HDFI and its security applications. §VIII analyzes the security guarantee provided by HDFI, its attack surface, and discusses best practices. §IX discusses the limitations of our current design and future work. §X concludes the paper.

II. Threat Model and Assumptions

In this work, we focus on preventing memory corruption based attacks; therefore we follow the typical threat model of most related work. That is, we assume that software may contain one or more memory vulnerabilities that, once triggered would allow attackers to perform arbitrary memory reads and writes. We do not limit what attackers would do with this capability, as there are many different attack vectors given this capability. As a hardware-based solution, we also do not limit where the vulnerabilities are: they can be in user-mode applications, OS kernel, hypervisor, etc. However, we assume all hardware components are trusted and bug free, so attacks that exploit hardware vulnerabilities, such as the row hammer attack [42], are out-of-scope.

Similar to NX-bit, HDFI requires software modifications to obtain its benefits. This can be done in many ways: manual modification, compiler-based modification, static binary rewriting, dynamic binary rewriting, etc. For the example applications we demonstrated in this paper, we either manually modified the source or leveraged compiler-based approaches. However, we must emphasize that this is not a limitation of HDFI and source code is not always necessary.

III. Background and Related Work

This section provides the background of HDFI and compares HDFI with related work.

A. Data-flow Integrity

The goal of HDFI is to prevent attackers from exploiting memory corruption vulnerabilities to tamper/leak sensitive data. To achieve this goal, we leverage data-flow integrity (DFI) [10]. DFI ensures that the runtime data-flow cannot deviate from the data-flow graph generated from static analysis. In particular, DFI assigns an identifier to each write instruction and records the ID of the last instruction that writes to a memory position; then at each read instruction, DFI checks whether the ID of the last writer belongs to the set allowed by static analysis. Take Example 1. This code snippet contains a buffer overflow vulnerability at line 6, which allows attackers to use `strcpy()` to overwrite the return address saved at line 3 and launch control-flow hijacking attacks. Such attacks can be prevented by checking if the return address read at line 8 is defined by the store instruction at line 3.

```
1  main:
2    add      sp,sp,-32
3   ⋆sdset1   ra,24(sp)
4    ld       a1,8(a1)       ; argv[1]
5    mv       a0,sp          ; char buff[16]
6    call     strcpy         ; strcpy(buff, argv[1])
7    li       a0,0
8   ⋆ldchk1   ra,24(sp)
9    add      sp,sp,32
10   jr       ra             ; return
```

Example 1: A typical stack buffer overflow example, in RISC-V assembly, in which HDFI prevented by replacing load and store instructions with two new load and store instructions (line 3 and 8). `strcpy()` at line 6 can overflow the return address saved at line 3, and HDFI can accordingly detect the overflow when it is loaded back at line 8.

In HDFI, we extend the ISA to perform DFI-style checks with hardware. Specifically, we leverage memory tagging to record the last writer of a memory word and provide new instructions to set and check the tag. However, instead of trying to fully replicate DFI, which would require supporting arbitrary tag size, we focus on providing isolation, i.e., using a one-bit tag to indicate the trustworthiness of the writer. Using the same example, HDFI can be utilized to prevent the attack by (1) using a new instruction `sdset1` (store and set tag) to set the tag of memory used to store return address to `1` (line 3); and (2) when loading the return address from memory for function return, using another instruction `ldchk1` (load and check tag) to check if the memory tag is still `1`. Since normal store instructions (e.g., `sd`) would set the tag to `0`, if attackers try to overwrite the return address, the `ldchk1` instruction would fail and generate a memory exception.

B. Tag-based Memory Protection

Tag-based memory protection is not new and has been explored in many previous works. For example, lowRISC [8] uses a 2-bit tag to specify if a memory address is readable and writable. Loki [84] also allows developers to specify permission with a memory address, but is more flexible, as the permission is related to the current protection domain. The problem with these approaches (including the Mondriaan protection model [79]) is that, although the objects (memory addresses) are fine-grained, the subjects are still coarse-grained—the access permissions are applied to the whole program or the whole protection domain. However, the subjects are individual instructions in HDFI.

An alternative approach is to associate the access permission with pointers instead of memory locations. For example, Watchdog [51] and the application data integrity (ADI) [57] mechanism on SPARC M7 processors allow a program to associate memory addresses and pointers with versions (tags) and require that when accessing the memory the version of the pointer must match the version of the memory. The tricky part of this approach is how to maintain the tag of a pointer, because every pointer should have two tags: one indicating the tag of the target memory, and the other indicating the tag of the memory where the pointer is stored. Without this, attackers can still tamper with the pointers. Watchdog handles this by using shadow memory to maintain the first type of tags, but it is unclear whether or how ADI handles this issue.

Write integrity test [2] is another tag-based memory safety enforcement mechanism. It enforces that each write operation (instead of pointer) can only write to objects that are allowed by the static data-flow graph. However, since the integrity test is only enforced on write operations, WIT can only enforce data integrity, but not data confidentiality.

A common issue with all the aforementioned approaches is that they must track the liveness of memory objects, which makes the protection more complicated. For instance, in Example 1, to protect the return address, all aforementioned systems must tag the memory used for return address at prologue. Here we must pay special attention to the order of tagging and store: if store happens before tagging, the system would be vulnerable to time-of-check-to-time-of-use (TOCTTOU) attack, because the address might be modified unless the two operations are guaranteed to be atomic. Then, after the function finishes execution and returns, the current stack frame is *freed*, so the old memory position used to store the return address must be unprotected for future re-use. Here is another tricky part—if the capability system is location-based, or does not assign a new version for every memory allocation (which is very challenging for fixed tag size), then it would be subject to use-after-free (UAF) based attacks. Moreover, for software that heavily utilizes custom memory allocators, such as browsers and OS kernels, tracking object allocation is non-trivial. Fortunately, HDFI does not need to track liveness of memory objects.

Among existing hardware features, Minos [18] and CHERI [77] are the closest to HDFI. Specifically, Minos uses one-bit tags to indicate the integrity of code pointers and updates the tag based on the Biba model [6]. CHERI [77] also uses one-bit tags to indicate whether a memory address stores a valid capability (fat pointer). This bit can only be set when the memory content is written by a capability-related instructions and is cleared when written by normal store instructions. Comparing to them, the advantage of HDFI is flexibility—as will be shown in §V, besides pointers, HDFI can also be used to protect generic data like `uid`; and along with the Biba model, HDFI can also be used to enforce the Bell–LaPadula model [5].

C. Tag-based Hardware

Because memory tagging is widely used for dynamic information flow tracking (DIFT), which can be very expensive when purely done in software [56]. For this reason, numerous hardware solutions have been proposed, including pure DIFT-oriented [18, 20, 40, 69], and more general, programmable metadata processing [13, 25, 28, 75]. The most significant difference of HDFI from these solutions is our emphasis on minimizing hardware changes so as to make HDFI more likely to be adopted in practice. In particular, HDFI does not require modifying register files, ALU, main memory, or the bandwidth between cache and main memory. More importantly, instead of requiring half of all physical memory dedicated to store tags (i.e., an overhead of 100%), HDFI only impose 1.56% memory overhead.

D. Memory Safety

Since memory safety issues are the root cause of many attacks [70], researchers have proposed many solutions to address this problem, including automated code transformation [55], instrumentation-based [2, 10, 53, 54], and hardware-based [27, 36, 51, 52, 77]. The biggest hurdle for adopting these solutions is their performance overhead—even with hardware assistance, the average overhead is still 29% on benchmark workloads [52]. To help further reduce the overhead, HDFI is designed to enable another optimization direction—using isolation to limit the protection scope and only enforcing memory safety over the isolated data. Such data could be security sensitive, e.g., code pointers [43], generic pointers [17, 77], or important kernel data [66]. It could also be data that can be statically proved to be memory safe, e.g., safe stack [43]. We believe such a combination would allow us to build powerful yet efficient solutions to eliminate all memory corruption based attacks.

IV. HDFI ARCHITECTURE

In this section, we present the design of HDFI, which includes two major components: the ISA extension and the memory tagger. Our current design tags memory at machine-word granularity because most sensitive data we want to protect are of this size (e.g., pointers). For data not of this size, we can manually extend the size, or leverage compilers. To prevent attackers from creating inconsistent views of data and its corresponding tag and launching TOCTTOU attacks in a multi-core/-processor system, we require all HDFI instructions to be *atomic* (i.e., data and tag must always be loaded and stored together) and comply with the same cache consistency model as other memory accessing instructions. To avoid changing the main memory system and the data link between main memory and the processor, our current design stores all the tag information at a dedicated area called *tag table*. In our current design, tag table is allocated and initialized by the OS kernel during boot, similar to how Intel SGX reserves the secure pages (i.e., EPC pages) for enclaves [36]. Once allocated, the memory region for the tag table will be protected from malicious modification (§IV-D).

A. ISA Extension

To enforce DFI, the authors added two high-level instructions: `SETDEF` and `CHECKDEF` [10]. Since HDFI only supports one-bit tags, in order to allow programs to use DFI-style checks to enforce the integrity/confidentiality level of memory contents, we introduce three new instructions:

- `sdset1 rs,imm(rb)`: store word and set tag to `1`.

- `ldchk0 rd,imm(rb)`: load word and check if tag equals `0`.
- `ldchk1 rd,imm(rb)`: load word and check if tag equals `1`.

Note that we do not have an instruction that explicitly sets the tag to `0`. Instead, HDFI implicitly sets the tag of the destination memory to `0` when written by regular store instructions. However, HDFI preserves the semantics of regular load instructions, i.e., tag is not checked on regular load operations. To check the tag bit of the target memory region, HDFI provides `ldchk0` and `ldchk1`. To enable the OS kernel to capture tag mismatch, we also introduced a new memory exception, which is similar to other memory faults except for the error code.

HDFI also provides a special instruction alias `mvwtag` [1] for copying the memory from a source to a destination along with the corresponding tag bits. This special operation is necessary to achieve optimal performance in modern system software. Specifically, modern OS kernels like Linux use copy-on-write (CoW) to share memory between the parent process and its child processes. However, if we use normal `sd` operations to perform the copy, it could break HDFI-protected applications because the tag information is lost; on the other hand, we also cannot use `sdset1` because it allows attackers to abuse this feature to tag arbitrary data. To solve this problem, we introduced the `mvwtag` instruction to allow OS kernels to copy data while preserving the tag. Please also note that because `memcpy` can cause memory corruption, we do not recommend using `mvwtag` to implement `memcpy` unless the developer can guarantee memory safety for all the invocations of `memcpy`.

B. Memory Tagger

Our hardware extension is similar to lowRISC [8]. Specifically, to simplify the implementation of the new instructions and support atomicity in a multi-core/-processor system, we modified the interface between the processor core and the cache system (including the coherence interconnect) to associate each data with its tag. In particular, when the processor core executes a memory related instruction such as `sd`, `sdset1` or `ld`, it sends a request to the data cache(s). This request includes a data field and a command field. HDFI adds one tag bit to the data field, so for every memory write request, data is always stored with the tag; and for memory load requests, tags can be (not always, see §IV-C for detail) loaded with data.

To facilitate this, we augmented the caches to hold the tags for the cached memory units, as shown in Figure 1. To hold the tag bits for the cached memory units, the caches have a one-bit register for each machine word to store the corresponding tag. When the processor core sends a store request, the L1 cache can simply update the data and tag value with the incoming value from the core; and when the core sends a read request, the L1 data cache provides the core with the tag bit, with which the core can check whether the tag matches expected value or not.

[1]Since we do not extend general register files with tag, this operation is an alias for two instructions: load data and tag from source into a special register then store them to the destination.

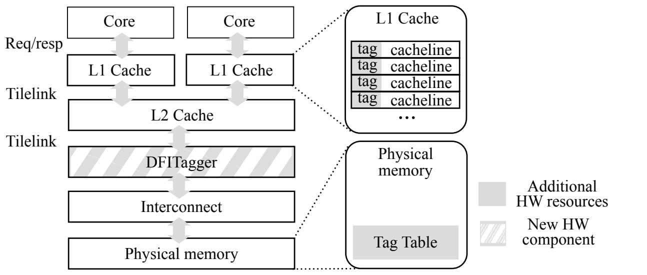

Fig. 1: Design of HDFI. The processor core and caches are augmented and the DFITAGGER is added.

While the L2 cache can also be augmented similarly to hold the tags for each memory unit, we believe it is not feasible to add the tag bits physically to the external main memory. For this reason, we added an additional module DFITAGGER in between the L2 cache and the main memory, which decomposes memory accesses from the L2 cache to separate data accesses and tag accesses. Data accesses are handled as usual and tag accesses are handled as follows. HDFI preserves a memory chunk to be used as tag table (Figure 1), which acts as a huge bit vector to store tag bits. When the L2 cache issues a memory access, DFITAGGER maps the physical address to a table entry of the tag table and generates a tag access.

C. Optimizations

Unfortunately, the additional memory accesses to the tag table introduce non-negligible performance overhead. More specifically, without any optimization, HDFI will double the memory accesses because for every cache miss, DFITAGGER needs to issue one data access and another tag access. To minimize this impact, we developed several optimization techniques.

1) Tag Cache: The most straightforward way of reducing the overhead is caching, so we introduced a tag cache within the DFITAGGER to exploit the locality of memory accesses. Moreover, tag cache also allows DFITAGGER to fetch a set of tags from the main memory in the cache line granularity to reuse the existing memory interface. For example, a cache line in the Rocket Core is 64 bytes. To handle one cache miss, DFITAGGER only needs 8 tag bits (one bit per eight bytes), but for the fixed size of memory interface, it has to fetch 64 bytes from the tag table. In fact, this 64-byte unit, which we call one *tag table entry*, naturally stores the tags for a 4 KB memory block; so tag cache allows us to generate only one memory access per 4 KB data access.

2) Tag Valid Bits: The second optimization technique takes advantage of the fact that most of the memory loads are *not* checked, so there is no need to always refill the cache line with corresponding tag bits. Leveraging this observation, we add a *Tag Valid Bit* (TVB) to each memory unit in the caches to further reduce unnecessary accesses to the tag table. TVB is updated as follows. When the cache has to refill a line but the request from the inner cache or the processor core does not explicitly asks for tag bits, the cache generates a refill request to the outer cache or DFITAGGER, and clears the TVB for the memory units in the line. Later, if an incoming load (with tag) request hits in the cache, but the TVB for the corresponding memory unit is not set, the cache will refill the line again with the valid tags. Note that any write hit will set the TVB because store operations always update the tag bit. Finally, when a cache line is evicted and written back to main memory, the cache forwards TVB to DFITAGGER, so the later can update the tag cache accordingly.

3) Meta Tag Table: The third technique leverages the fact that most of the memory units are tagged with `0` and only a few ones will be tagged with `1`. This means that most tag table entries would be filled with `0`. To take advantage of this observation, DFITAGGER maintains a *Meta Tag Table* (MTT) in the main memory and a *Meta Tag Directory* (MTD) as a register. Each bit of the MTT entries is set to `1` if the corresponding tag table entry contains `1`, and each bit of MTD is set to `1` if the corresponding MTT entry has `1`. Utilizing them, DFITAGGER can avoid fetching tag table entries from the main memory if they are filled with `0`. It also enables DFITAGGER to avoid (1) updating the tag table entry for a given write miss if that entry is filled with `0`; and (2) write back to main memory if both the evicted tag cache and the main memory copy are filled with `0`.

D. Protecting the Tag Tables

The design of HDFI requires that the tag table and the meta tag table in the main memory are protected from the malicious modifications. To do so, we leverage the fact that DFITAGGER is sitting between the cache and the main memory, hence we can use it to mediate all modifications to the main memory. That is, once the memory chunk used for tag tables are assigned to DFITAGGER, it drops any access to this memory chunk. Because tag is always provided by DFITAGGER, this effectively prevents any malicious modifications to the tag tables. Note that our current design cannot prevent DMA-based attacks; we will discuss this issue in §IX.

V. SECURITY APPLICATIONS

In this section, we demonstrate how HDFI can be utilized to build security solutions with simplified designs, improved performance, and better security. We want to use these examples to highlight the generality of HDFI (i.e., the ability to support different security applications), as well as its ease of adoption. Regarding backward compatibility, it completely depends on the security solution. Some security mechanisms like shadow stack could allow mixing protected and unprotected code, but other solutions like VTable protection will not allow such mixing.

In each application example, we focus on protecting one type of security critical data, such as return addresses, function pointers, etc. However, as there is no overlapping between the protected data (i.e., the meaning of the tag bit is not ambiguous), we can integrate all mechanisms together to maximize the defense against memory corruption based attacks.

To implement these examples, we either directly modified the source code or augmented compilers to emit HDFI's new instructions. However, we want to emphasize again that this is not a limitation of HDFI—as long as a security solution can make the target program use HDFI's new instructions, it will be able to leverage the isolation provided by HDFI.

A. Shadow Stack

In Example 1, we have demonstrated how to use HDFI to implement a virtual shadow stack for protecting the return addresses. To implement this scheme, we just need to change 6 lines in GCC (Example 2). Implementation in the LLVM toolchain is similarly simple, with only 4 lines of changes—in function `storeRegToStackSlot/loadRegFromStackSlot`, which are invoked at function prologues/epilogues, we use `sdset1/ldchk1` instead of normal store/load. Because these functions are also used to handle register spills/restores, our (LLVM-based) shadow stack also protects spilled registers, which can also be an attack vector [16].

```
char *riscv_output_move (rtx dest, rtx src) {
    // if dest == REG && src == MEM
    if (flag_safe_stack && (REGNO (dest) == RETURN_ADDR_REGNUM))
        return "ldchk1\t%0,%1";
    else
        return "ld\t%0,%1";
    // if dest == MEM && src == REG
    if (flag_safe_stack && (REGNO (src) == RETURN_ADDR_REGNUM))
        return "sdset1\t%z1,%0";
    else
        return "sd\t%z1,%0";
}
```

Example 2: How to use HDFI to implement shadow stack in GCC, with only 6 lines of changes.

Supporting context saving and restoring like `setjmp/longjmp` has always been a challenge for hardware-based shadow stacks [46, 59, 81]. However, for a HDFI-based shadow stack, supporting this feature is straightforward—just like saving registers to the stack, when saving current context to `jmp_buf`, we set the tag of the corresponding memory to `1`. Then, when restoring the context, we check if the memory tag is still `1`. If attackers try to overwrite `jmp_buf`, the load check will fail. Furthermore, because HDFI-based shadow stack is still memory-based, it naturally supports deep recursion. It can even support modifying return addresses as long as they are always stored using `sdset1` and loaded with `ldchk1`. Finally, unlike SmashGuard [59], because HDFI is orthogonal to the execution privilege level, HDFI-based shadow stack does not need any support from the OS kernel and can also be used to protect kernel stacks.

B. Standard Library Enhancement

Runtime libraries like the dynamic linker (`ld.so`) and the standard C library are important parts of every program's runtime security. Unfortunately, many compiler-based security solutions neglected them, thus leave holes for attacks [9, 39, 60]. In this subsection, we describe enhancements made to the libraries to prevent attacks.

1) Heap Metadata Protection: Many standard C libraries like `glibc` (GNU C Library) uses a variant of Doug Lea's Malloc [45] that supports multi-threading, called (`ptmalloc`). `ptmalloc` uses double-linked lists to manage freed memory chunks. When removing a memory chunk from this list, it performs a general unlinking process. If there exist a heap buffer overflow vulnerability, attackers can exploit this vulnerability to tamper with these metadata (pointers), which will allow attackers to overwrite an arbitrary address with arbitrary data [39]. Moreover, despite that many integrity checks have been applied to the heap implementation to stop heap-based attacks, attackers still find their ways to bypass them [31, 33].

To prevent such attacks, we can leverage HDFI to protect the integrity of these metadata—similar to return addresses, when linking a freed chunk, we set the tags of forward and backward pointers to `1`; then when unlinking a chunk, we check if the tag is still `1`. By doing so, if attackers overwrite these pointers (with normal writes), the tag will be set to `0`, which will be captured by the load check.

2) Global Offset Table Protection: Global Offset Table (GOT) is a data structure for dynamic linking. Since GOT is modifiable by default and affects the program's control flow, GOT overwriting [60] has been used for changing control flow with memory corruption based attacks. To protect the GOT, we enforce that whenever a dynamic linked function is invoked, the target address is loaded by `ldchk1`. To tag the initial pointer (i.e., the call to the resolver), we leveraged the fact that for position-independent executables (PIE), GOT table entries need to be patched due to address space layout randomization (ASLR); so we modified the relocation routine to tag the initial GOT values with `1`. Then during runtime, after resolving a real function address, we make the loader use `sdset1` to update the GOT value.

3) Exit Handler Protection: Another attack surface is the exit handler [9]. To prevent attackers from manipulating the exit handler, pointer encryption [29] is applied in `glibc`. However, because performance was top priority when designing this scheme, the encryption is implemented in an ad hoc manner and can be easily bypassed with information leakage. To protect the exit handler, we use HDFI to enforce that it is always registered with `sdset1` and loaded with `ldchk1`. Since attackers cannot tag an exit handler with `1`, they cannot abuse it to execute arbitrary code.

C. VTable Pointer Protection

As virtual function calls comprise a large portion of indirect control transfer in large C++ programs like browsers [71], virtual function table pointers (a.k.a., `vfptr`) have become a popular attack target [86]. In these attacks, attackers try to exploit memory corruption vulnerabilities to control the `vfptr` so as to invoke arbitrary code, which has been demonstrated to be very powerful [62]. For this reason, many systems have been proposed to defeat such attacks [7, 38, 71, 85, 86].

Leveraging HDFI, we also implemented a protection mechanism based on one security invariant: *only a constructor function can initialize a* `vfptr`. This invariant can be enforced in two simple steps: (1) when initializing a C++ object, we use `sdset1` to initialize its `vfptr`; and (2) when performing a virtual call, we always use `ldchk1` to load the `vfptr`.

Compared with existing protection mechanisms, our implementation is much simpler in that it requires no sophisticated static analysis and/or runtime instrumentation. At the same time, it is also very effective. More specifically, there are two typical attacks against VTable: injection attacks and reuse attacks. In VTable injection attacks, attackers try to forge a `vfptr` pointing to a crafted VTable. With our protection, this is no longer feasible because the values assigned to `vfptr` are always static/constant. In VTable reuse attacks, attackers try to make the `vfptr` point to an existing VTable, but usually at a wrong offset [62]. Although our mechanism cannot fully prevent all VTable reuse attacks, it significantly increases the difficulty of attacks, because (1) making the `vfptr` point to a wrong offset is no longer feasible, because constructors always assign the correct value; and more importantly, (2) crafting a counterfeit object is also much more difficult, i.e., once combined with techniques that can prevent illegal jumping to the middle of a function (e.g., shadow stack and CPS), the only way to modify the `vfptr` is to invoke a constructor, who will initialize a legitimate object and overwrites the crafted data from attackers.

D. Code Pointer Separation

Control flow hijacking is one of the most popular and powerful attacks. In all control flow hijacking attacks, attackers seize control by corrupting one or more code pointers. Based on this observation, researchers have proposed code pointer separation (CPS) [43], a technique that isolates code pointers into a safe region to prevent attackers from tampering with them. In their original implementation, the isolation is enforced using segment on 32-bit x86 processors or randomization (or masking) on 64-bit x86 processors and ARM processors. As discussed in §I, these approaches introduce (1) additional memory overhead for data shadowing, and (2) additional performance overhead for shadow data lookup, which is very problematic on benchmarks where code pointer dereference is more frequent, such as C++ programs and language interpreters. Moreover, their randomization-based approach is subject to brute-force attacks [32], and their masking-based approach introduces an additional 5% performance overhead [43].

By utilizing HDFI, we can eliminate all these drawbacks. Specifically, using the same static analysis from CPS, we can identify all code pointers that need to be protected. With this information, instead of instrumenting the target program to load/store code pointers from the safe region with an additional runtime library, we instrument the program to (1) always use `sdset1` instructions to store code pointers, and (2) always use `ldchk1` instructions to load code pointers. Because no other instructions can store code pointers, our approach has the same effectiveness as segments and masking based approaches. However, because there is no additional lookup step(s), the performance of our approach is better when there are many indirect calls.

One drawback of our solution is that we need to add one additional step to tag static code pointers that are initialized by the OS kernel or the dynamic loader, e.g., virtual function pointers in the VTables. For PIE code, we can reuse our modification to the relocation procedure to perform this task.

E. Kernel Protection

Although control flow hijacking attacks are the most popular attack type, non-control data attacks are also feasible [14], especially for kernel attacks [66]. More importantly, existing kernel-wide protection mechanisms all imposes very high performance overhead; regardless of whether it is masking-based [19] or context-switch-based [66]. As a generic data isolation mechanism, HDFI can also be used to replace those expensive isolation mechanisms thus reduce the performance overhead of these solutions.

Similar to CPS, porting Kenali [66] to utilize HDFI is straightforward. Specifically, we replace: (1) its randomization-based stack protection with the shadow stack described in §V-A; (2) the expensive, context switch-based update operations with `sdset1`; (3) all read to sensitive data with `ldchk1`; (4) global object shadowing with tagging (i.e., similar to function pointers in the VTable, we wrote a small early initialization routine to tag sensitive global object); and (5) we eliminate its complicated object shadowing mechanism.

F. Information Leak

In all of the above applications, we try to prevent attackers from injecting data into the trusted region, but HDFI can also be used to prevent attackers from reading sensitive data from the trusted region. For example, in the Heartbleed attack [15], attackers exploited a buffer overread vulnerability in the OpenSSL library to steal the private key associated with the website's certificate. To prevent such attacks, we can (1) tag the memory used to store the private key as `1`, (2) replace all legitimate read access to the key with `ldchk1`, and (3) implement a simple sanitation routine that uses `ldchk0` to check if the buffer to be written to network contains any data with tag `1`.

VI. Implementation

Components	Language	Lines of Code		
		Modified	Added	Total
Architecture	Scala (Chisel)	395	1,803	2,198
Assembler	C	-	16	16
Linux Kernel	C	8	52	60
Total		403	1,871	2,274

TABLE II: Components of HDFI and their complexities in terms of their lines of code.

In this section, we provide the implementation detail of HDFI. Table II shows the lines of code used to implement HDFI, excluding empty lines and comments.

A. Hardware

We implemented a prototype of HDFI by modifying the Rocket Chip Generator [73]. The generated system includes a Rocket Core [74] as its main processor, which has 16KB of L1 instruction and data caches. Modifying the generator itself instead of a generated instance allows us to generate and evaluate multiple versions of HDFI with various features

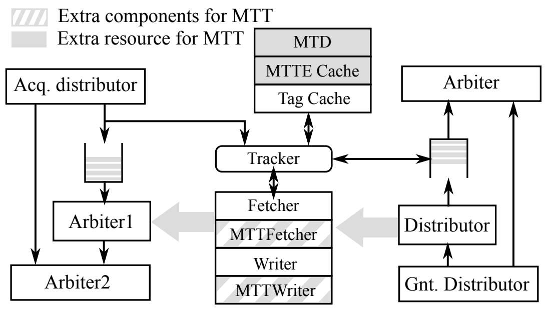

Fig. 2: A simplified diagram of DFITAGGER on a Rocket Chip.

and parameters, e.g., different optimization techniques and configuration parameters.

1) ISA Extensions: Following the design pattern of RISC-V, we assign a new opcode to our new instructions that is similar to the RV64I load/store instructions [76].

`sdset1`: We extend the memory request unit's data field by one-bit to include the tag. To determine whether the tag should be `0` or `1`, we introduce a new configuration to the set of control signals for memory command type that is unique to `sdset1`.

`ldchkx`: We add a new, one-bit field to the memory response unit for the tag bit loaded with the machine word. To determine whether the tag bit should be loaded, we assigned a new memory command to these two instructions. Upon a valid response from cache, HDFI compares the tag to the expected value. This expected value is extracted from bit `12` of the `ldchkx` instruction. A tag mismatch generates a new memory exception; otherwise, the pipeline continues normally.

`mvwtag`: At the execution stage, HDFI first calculates the source address from the second register's value and the immediate offset using the ALU, and sends out a memory read request to load the data and tag. The result is stored in a new internal register that is capable of storing both data and tag. Simultaneously, HDFI calculates the destination address from the destination register's value and the same offset using a separate adder. Finally, we issue a memory store request to store the internal register's data and tag to the destination address.

2) DFITAGGER: To avoid adding the tag bits physically to the main memory, which is usually a set of DRAMs, we implemented DFITAGGER to translate memory accesses with tags from inner caches into data accesses and tag accesses. Figure 2 shows the DFITAGGER we implemented for the Rocket Chip. The DFITAGGER is designed to handle the memory accesses that comply with the *TileLink* protocol which the rocket chip uses to implement the cache coherence interconnect. Among the five channels that the protocol defines, DFITAGGER handles two of them because they are used to connect the L2 caches and the outer memory system.

To initiate a memory access, the inner cache generates one or more *beats* of transaction through the *Acquire* channel, and the DFITAGGER selectively intercepts the beats using the *Acquire Distributor*. When the option `tagger` is enabled, the Acquire Distributor bypasses the device accesses, drops the access to the tag table or meta tag table (for protection purpose), and forwards all the transactions heading to the memory to the *Acquire Queue*, which simply forwards the incoming transactions to memory. The *Acquire Arbiter1* drops all the tag bits in the transactions; the resulting memory accesses only contain the data part of the incoming accesses.

In the mean time, the *Tracker* duplicates the required field of incoming transactions, including the tag bits, the transaction id, the type of the transaction, and the address. When the incoming transaction writes to memory, the Tracker updates the corresponding tag bits in the tag table with the tag bits in the transaction. To do this, the Tracker first check the *Tag Cache*, and uses the *Fetcher* and the *Writer* modules to fetch and evict tag table entries.

Handling memory read accesses is similar, but the Tracker need to intervene in the *Grant* channel as well. In the Rocket Chip, the memory interface (which is a protocol converter) uses the Grant channel to provide the caches with the read data. To attach the tag bits to the Grant transactions, the Acquire Queue changes the transaction id of read accesses so that the corresponding Grant transactions are forwarded to the *Grant Queue*. In the mean time, the Tracker accesses the Tag Cache and uses other modules to prepare the corresponding tag bits. Once the tag bits become available, the Grant Queue forwards the transaction from the memory interface, after changing the transaction id back to the original one and attaching the correct tag bits for each machine word.

3) Tag Valid Bits: To reduce the number of tag table accesses, HDFI adds a TVB for each machine word in the caches. Using TVB, the cache can avoid fetching the tag bits when it refills a cache line. To take advantage of this, the cache uses the `union` field of an Acquire transaction to mark if the response to the transaction should have valid tag bits or not. The Acquire Distributor then uses this field to decide whether a transaction could be directly forwarded to the *Acquire Arbiter2* and bypass the Acquire Queue.

The location of TVBs is also important. A simple solution is to put the TVBs in the metadata array, where the cache holds the *cache tags* and the coherence information. However, this approach would increase the latency of write hits because the cache has to update the metadata for every write operation. To address this issue, we choose to put a *tag fetched bit* in the metadata array for each cache line and extend the size of the data array to store the TVBs for each word. The tag fetched bit is set/cleared by the miss handler, which is called *MSHR* in the Rocket Chip. When the handler fetches the cache line with tags, it sets the bit; otherwise the bit is cleared. Since every write operation should update the tag, the cache also sets the TVB whenever a machine word is written.

Adding TVBs also requires the DFITAGGER to consider a memory write access whose tag bits are partially valid. To handle this, the cache attaches the TVBs for each machine word to the Acquire transactions for memory writes. With the TVBs, the DFITAGGER can selectively update the tag bits in the corresponding tag table entry.

An important drawback of this implementation is that the cache refills a cache line to handle an incoming load with tag access even when the TVB of the requested machine word is set, but if the Tag Fetched Bit is not set. We believe that we can avoid these cache refills by augmenting the miss handler, by letting it to consider the TVBs before evicting and refilling the cache, but the current implementation does not include such feature.

4) Meta Tag Table: Enabling the Meta Tag Table adds the shaded components and resource in Figure 2 to the DFITAGGER. When handling an incoming tag table read access, the Tracker checks whether the MTT cache and the tag cache has a matching entry. If the Tracker fails to find a matching tag table entry, it checks the MTD and the matching MTT entry (loaded into MTT cache if does not exist) to see if the corresponding tag table entry is all zero. If so, the Tracker handle the incoming tag table access without really fetching the entry from the memory. To minimize the miss penalty, the *MTTFetcher* and the *MTTWriter* handles the access to the MTT in the memory in parallel with the existing Writer and Fetcher.

After updating the tag table entry and the MTT entry, the Tracker checks if it can clear the corresponding MTT entry bit and MTD bit. In particular, the Tracker clears the corresponding bit in MTT entry if the updated tag table entry is filled with zeros, and clears the MTD bit if the MTT entry is filled with zeros.

B. Software Support

To utilize HDFI, we made the following changes to the software.

1) Assembler: We modified the GNU assembler (`gas`) so that it recognizes the new instruction extension and can generate the correct binary.

2) Kernel Support: Our modifications to the OS kernel include three parts. First, we modified its exception handler to recognize the new tag mismatch exception. To handle this exception, we reused the same logic as normal load/store faults, i.e., generate a segment fault (`SIGSEGV`) for user mode applications, and panic if the exception happens in kernel space. Second, as mentioned in §IV, we implemented a special memory copy routine with the new `mvwtag` instruction and modified the CoW handler to invoke this routine to copy page content, so that the tag information are preserved. Last, we added routines to allocate the tag table and meta tag table, and initialize the DFITAGGER with the base addresses of the tables.

C. Security Applications

Most security applications mentioned in §V were implemented based on the llvm-riscv toolchain [61] (RISCV branch). Table III summarizes the effort of implementation/porting.

1) LLVM Shadow Stack: LLVM-based shadow stack is implemented as part of the frame lowering process. Specifically, we modified the `getLoadStoreOpcodes` function to return the opcode of `sdset1` for the `storeRegToStackSlot` function; and return the opcode of `ldchk1` for the `loadRegFromStackSlot` function.

Solutions	Language	LoC
Shadow Stack	C++ (LLVM 3.3)	4
VTable Protection	C++ (LLVM 3.3)	40
CPS	C++ (LLVM 3.3)	41
Kernel Protection	C (Linux 3.14.41)	70
Library Protection	C (glibc 2.22)	10
Heartbleed Prevention	C (OpenSSL 1.0.1a)	2

TABLE III: Required efforts in implementing or porting security schemes in terms of lines of code. Given a software-based solution, HDFI is easy to adopt or extend in practice.

2) VTable Pointer Protection: VTable pointer protection is implemented in two steps. First, during compilation, we enable the TBAA (type-based alias analysis) option so Clang will annotate VTable load/store operations with corresponding TBAA metadata ("`vtable pointer`"). This metadata will be propagated to machine instruction, so in the second step, we leveraged the DAG to DAG transformation pass to replace `sd` instructions with `sdset1` instructions, and to replace `ld` instructions with `ldchk1` instructions, if the machine instruction has the corresponding TBAA of VTables.

3) Code Pointer Separation: To port CPS [44] to our architecture, we performed the following modifications. (1) Because code pointers are now protected by HDFI, we removed the runtime library required by its original implementation. (2) We modified the instrumentation, so when a code pointer is stored to or loaded from memory, we annotate the corresponding operations with a special TBAA metadata and removes the original invocation to the runtime library. (3) Using the same DAG to DAG transforming function, we replace the `sd` and `ld` instructions with `sdset1` and `ldchk1`, respectively. Unfortunately, lacking link time optimization support in the llvm-riscv toolchain, we cannot port the original CPS and CPI implementations.

4) Kernel Protection: Due to the limitation of llvm-riscv toolchain, even though we were able to generate LLVM bitcode for the target kernel and apply the static analysis of Kenali [66], we cannot use Clang to compile the kernel into executable binary. As a result, we cannot perform automated instrumentation to protect all the discover data. For proof-of-concept, we utilize the analysis results to manually instrumented the kernel to protect the `uid` fields in the `cred` structure, which are the most popular target for kernel exploits. Since we have implemented the shadow stack in GCC, we were able to replace Kenali's randomization-based stack protection with our stack shadow.

The rest of the protection mechanisms are implemented through manual modification.

5) Standard libraries: To protect the integrity of saved context of `setjmp/longjmp`, we modified `setjmp.S` and `__longjmp.S` so general registers are saved with `sdset1`, and restored with `ldchk1` to enforce its integrity. To protect the integrity of heap metadata, we manually modified the linking and unlinking routine to use `sdset1` for assigning pointers and `ldchk1` for loading pointers. To set the tag of static code pointers to `1`, we modified the dynamic loader (`elf_machine_rela`) so that during the relation process, it stores the patched code pointer with tag `1`. And to protect code pointers in GOT table and the exit handler, we modified the dynamic loader to use `sdset1` to set these pointers, and `ldchk1` to load these pointers.

6) Heartbleed: To protect sensitive data from Heartbleed attacks, we modified OpenSSL so that (1) the private key is stored with `sdset1`; and (2) when building the response buffer, `ldchk0` is used to make sure that all content copied to this buffer has tag `0`. To implement this protection, we used background knowledge about Heartbleed to decide where to put the checking routine (i.e., when constructing the response buffer). For a prototype implementation, we believe this is a reasonable limitation. To thoroughly protect the sensitive data, one could use data flow analysis or taint analysis [82] to determine where to tag sensitive data, and where to put the check.

D. Synthesized Attacks

To evaluate the effectiveness of the security applications we implemented/ported, we developed/ported several synthesized attacks against different targets.

1) RIPE Benchmark: RIPE [78] is an open sourced intrusion prevention benchmark. It provides five testbed dimensions: location of the buffer overflow, target code pointers, overflow technique, attack payload and abused function. Since RIPE was developed for the x86 platform, we need to modify it to make it work on the RISC-V architecture. However, due to time limitations, we could not port all the features of RIPE. Specifically, our ported RIPE benchmarks support all locations of buffer overflow, all target code pointers except the frame pointer, both overflow techniques (direct and indirect), one attack payload (return-to-libc), and one abused function (`memcpy`).

2) Heap Exploit: To evaluate heap metadata protection, we ported the example exploit from [39] to overwrite the return address of a function.

3) VTable Hijacking: Due to the limitations of the FPGA, we could not use real-world cases like browser attacks to evaluate our VTable pointer protection mechanism. Instead, we developed a simple attack that overwrites the VTable pointer with a fake one, so the next invocation of the virtual function will invoke the attacker controlled function.

4) Format String Exploit: Because the RIPE benchmark does not cover attack targets used in recent attacks, we implemented a simple program with format string vulnerability to evaluate the ported CPS mechanism. We chose a format string vulnerability because it is one of the most powerful vulnerabilities that can be used as local stack read (`%x`), arbitrary memory read (`%s`), and arbitrary memory write (`%n`). For attack targets, we implemented two new attacks: GOT overwriting and atexit handler overwriting.

5) Kernel Exploit: In the kernel, overwriting non-control data is sufficient to obtain root permissions without hijacking control flow. To test the feasibility of using HDFI to defend against data-only attacks in the kernel, we back ported `CVE-2013-6282` [1], an arbitrary memory read and write vulnerability to our target kernel. Leveraging this vulnerability, an attackers can modify the `uid` of a process and escalate their privilege.

6) Heartbleed: Heartbleed (`CVE-2014-0160`) [15] is a heap out-of-bounds read vulnerability in OpenSSL caused by missing input validation when parsing malicious TLS heartbeat request. This bug was marked as extremely critical, because researchers have proved that it can be exploited to reveal private keys [34]. To reliably[2] simulate such attacks, we modified vulnerable OpenSSL (`1.0.1a`) to insert special characters as a decoy private key. Since the decoy data is inserted in the affected range of Heartbleed, it can always be leaked in default settings through a Heartbleed attack.

VII. Evaluation

In this section, we evaluate our prototype of HDFI by answering the following questions:

- **Correctness**. Does our prototype comply with the RISC-V standard (i.e., no backward compatibility issue)? (§VII-A)
- **Efficiency**. How much performance overhead does HDFI introduce compared to the unmodified hardware? (§VII-B)
- **Effectiveness**. Can HDFI-powered security mechanisms accurately prevent attacks? (§VII-C)
- **Benefits**. Compared to their original implementation, does HDFI-powered implementation perform better and/or is it more secure? (§VII-D)

Experimental setup. All evaluations were done on the Xilinx Zynq ZC706 evaluation board [80]. The OS kernel is Linux 3.14.41 with support for the RISC-V architecture [58]. Unless otherwise stated, all programs (including the kernel) were compiled with GCC 5.2.0 (`-O2`) and binutils 2.25, with a set of patches to support RISC-V (commit `572033b`) and default kernel configuration of RISC-V. While the board is equipped with 1GB of memory, the Rocket Chip can only use 512MB because the co-equipped ARM system requires 256MB. At boot time, the kernel reserves 8MB for tag tables and 128KB for the meta tag table, respectively. Following the environment that the RISC-V community built, we use the *Frontend Server* that runs on the ARM system and the *Berkeley Boot Loader* that runs on the Rocket Chip to boot vmlinux. The Rocket Chip accesses an ext2 file system in an SD card via the Front-end Server.

Although the tape-out Rocket Core chip can operate on 1GHz or higher, the synthesized FPGA on the ZC706 board can only operate at the maximum frequency of 50MHz. In addition, because the L2 cache is not mature enough for memory-mapped IO [47], we only evaluated with the L1 caches. In place of the L2 cache, we used the *L2BroadcastHub* that interconnects the L1 caches and the outer memory system. Due to the above limitation and the memory limitation of the evaluation board, we were not able to run most SPEC CINT 2006 benchmarks, so we used the much lighter SPEC CINT 2000 [68]. For SPEC CINT 2000, some benchmarks (`gzip` and `bzip`) cannot run successfully with the reference inputs. For these benchmarks, we adjusted the parameters of the reference inputs to reduce the size of the buffer they use to 3MB. We have annotated the results to clarify this.

We used pseudo-LRU (Least Recently Used) as the replacement policy for both tag and meta tag caches, and set the size of each cache to 1KB, allowing up to 16 entries of 512-bit cachelines.

A. Verification

HDFI passes the RISC-V verification suite provided by the RISC-V teams, which means our modifications to the RISC-V complies with the RISC-V standard so unmodified programs can still run correctly on our modified hardware.

B. Performance Overhead

In this subsection, we evaluate the performance impact of our hardware extension, as well as the effectiveness of our optimization techniques. This evaluation includes two part: the impact of new instructions on the processor core and the impact on memory access. Since HDFI did not introduce many changes to the pipeline of the processor core, the focus will be on memory access.

1) Pipeline: The `sdset1` and two `ldchk` instructions are treated identically to their normal store and load counterparts in the pipeline, with the exception of `ldchk` doing a comparison at the end of the memory stage. These three instructions can stall the pipeline in the same manner as their counterparts. However, the special register dedicated to `mvwtag` for preserving tags introduces a structural hazard to the pipeline. Because there is only one special register available, a series of `mvwtag` instructions have to wait for the previous `mvwtag` to finish, stalling the pipeline. Other memory instructions do not have to wait on previous ones to issue memory requests.

2) Memory Access: While the ISA extension does not affect the performance of the processor core, HDFI inevitably introduces additional memory accesses to fetch/update the tag table.

Micro benchmark. To measure the performance impact of these additional memory accesses and the logics to deal with them, we used `lat_mem_rd` from LMBench [49] to measure memory access latency and STREAMBench [48] to measure memory bandwidth. Table IV shows the result of the five configurations. The first row shows that HDFI does not affect the cache access latency. As the system operates at 50MHz, the 40ns latency means that it takes two clock cycles to read from the L1 cache. The second column shows that HDFI does increase the memory access latency. When `TVB` is enabled, DFITAGGER simply bypasses the incoming memory read access unless it explicitly requests the tag bits. However, the access should be examined by the Acquire Distributor and the Grant Distributor (Figure 2), which adds 2 clock cycles latency. For

[2]Attacking a OpenSSL-powered HTTPS server cannot always reveal the private key because the buffer used to store the privately may at a lower address, so it cannot be read by a buffer over read.

Benchmark	Baseline	Tagger	TVB	MTT	TVB+MTT
L1 hit	40ns	40ns (0%)	40ns (0%)	40ns (0%)	40ns (0%)
L1 miss	760ns	870ns (14.47%)	800ns (5.26%)	870ns (14.47%)	800ns (5.26%)
Copy	1081MB/s	939MB/s (13.14%)	1033MB/s (4.44%)	953MB/s (11.84%)	1035MB/s (4.26%)
Scale	857MB/s	766MB/s (10.62%)	816MB/s (4.79%)	776MB/s (9.45%)	817MB/s (4.67%)
Add	1671MB/s	1598MB/s (4.37%)	1650MB/s (1.26%)	1602MB/s (4.13%)	1651MB/s (1.2%)
Triad	818MB/s	739MB/s (9.66%)	802MB/s (1.96%)	764MB/s (8.8%)	803MB/s (1.83%)

TABLE IV: Impact on memory bandwidth and read latency, with different optimization techniques. The load does not include tag check and store does not include tag set.

Benchmark	Baseline	Tagger	TVB	MTT	TVB+MTT
164.gzip	963s	1118s (16.09%)	984s (2.18%)	1029s (6.85%)	981s (1.87%)
175.vpr	14404s	18649s (29.51%)	14869s (3.26%)	15513s (7.71%)	14610s (1.43%)
181.mcf	8397s	11495s (36.89%)	8656s (3.08%)	9544s (13.66%)	8388s (−0.11%)
197.parser	21537s	25005s (16.11%)	22025s (2.27%)	23177s (7.61%)	21866s (1.53%)
254.gap	4224s	4739s (12.19%)	4268s (1.04%)	4500s (6.53%)	4254s (0.71%)
256.bzip2	716s	820s (14.52%)	735s (2.65%)	742s (3.63%)	722s (0.84%)
300.twolf	22240s	28177s (26.71%)	22896s (2.97%)	23883s (7.37%)	22323s (0.36%)

TABLE V: Performance overhead of a subset of SPEC CINT 2000 benchmarks. Due to the limited computing power of the Rocket Chip on FPGA, we chose relatively lighter benchmark. In addition, to be fair, we included relatively memory bound benchmarks. According to a paper [37], 181.mcf, 175.vpr and 300.twolf are memory bound and showing higher overhead. We used reduced version of reference input to run 164.gzip and 256.bzip2.

memory bandwidth, our results also show that the optimizations we implemented can effectively reduce overhead.

SPEC CINT 2000. In addition to the micro benchmarks, we also ran a subset of SPEC CINT 2000 benchmarks on the five configurations of HDFI, *without* any security applications (i.e., no load check and no `sdset1`). Table V shows that even though the unoptimized version of HDFI causes non-negligible performance overhead, our optimizations successfully eliminated a large portion of overhead. Specifically, since there is no load check, TVB eliminated all read access requests to the tag table; and since there is no `sdset1`, MTT eliminated all the write access to the tag table. Table VI shows the number of memory accesses reduced by TVB and MTT. Please note that the 0.11% performance gain on `mcf` is due to fluctuations.

C. Security Experiments

In this subsection, we evaluate the effectiveness of HDFI-powered protection mechanisms. We evaluated all the security applications described in §V, with synthesized attacks described in §VI-D. The evaluation result is shown in Table VII, all HDFI-powered protection mechanisms can successfully mitigate the corresponding attack(s).

RIPE benchmark. With our ported RIPE benchmark, there are 112 possible combinations, with 54 that could proceed and 58 are not possible. Please note that although we did not port all combinations, all attack targets are supported except the frame pointer, which behaves quite differently on RISC-V. The supported targets are: return address, stack function pointer, heap function pointer, `.bss` section function pointer, `.data` section function pointer, `jmp_buf` on stack, `jmp_buf` as stack parameter, `jmp_buf` in heap, `jmp_buf` in `.bss` section, `jmp_buf` in `.data` section, function pointer in a structure on stack, in heap, in `.bss` section and in `.data` section. With our ported CPS, we can prevent all 54 attacks.

Heap exploit. Without protection, our basic version of heap attack targeting newlibc (a lightweight libc) was able to overwrite the return address to launch a return-to-libc attack to invoke the "evil" function. With our enhanced library, we were able to stop the attack.

VTable hijacking. Without protection, our simple VTable hijacking attack was able to invoke the "evil" function. With our VTable protection mechanism, we were able to prevent the loading of attacker-crafted `vfptr`.

Format string exploit. Without protection, our format string exploit can overwrite the GOT table entry and the exit handler to invoke the "evil" function. With our enhanced library, both attacks were stopped.

Kernel exploit. Without protection, the exploit can change the `uid` of the attack process to a arbitrary number. With our protection, the attack causes a kernel panic when trying to access the `uid`.

Heartbleed. : without protection, we can leak the decoy secret by exploiting the Heartbleed vulnerability. With our protection, the attack was stopped when constructing the response buffer.

D. Impact on Existing Security Solutions

As a fine-grained hardware-based isolation mechanism, we expect HDFI to provide the following benefits:

I **Security**: HDFI should provide non-bypassable protection for the isolated data;

II **Efficiency**: HDFI should provide the protection with low performance overhead;

III **Elegance**: HDFI should enable the building of elegant security solutions, e.g., no data shadowing, which as discussed in the introduction, has many drawbacks;

Benchmark	**Type**	**Baseline**	**Tagger**	**TVB**	**MTT**	**TVB+MTT**
164.gzip	**Read**	590M	799M (35.25%)	606M (2.71%)	589M (−0.17%)	588M (−0.34%)
	Write	380M	1,217M (220.26%)	453M (19.21%)	1,017M (167.63%)	378M (−0.53%)
175.vpr	**Read**	9,816M	17,200M (75.15%)	10,930M (11.35%)	9,760M (−0.57%)	9,792M (−0.25%)
	Write	7,908M	37,480M (373.83%)	12,420M (57.06%)	31,890M (303.16%)	7905M (0%)
181.mcf	**Read**	9,778M	14,310M (46.35%)	10,503M (7.41%)	9,778M (0%)	9,778M (0%)
	Write	5,588M	23,720M (324.33%)	8,490M (1.11%)	20,300M (263.15%)	5,588M (0%)
197.parser	**Read**	12,770M	17,610M (37.9%)	13,220M (3.52%)	12,850M (0.63%)	12777M (0.01%)
	Write	8,290M	27,490M (231.6%)	9,640M (16.28%)	24,440M (194.81%)	8299M (0.11%)
254.gap	**Read**	2,233M	2,872M (28.61%)	2,239M (0.27%)	2,225M (0%)	2,206M (−1.21%)
	Write	1,594M	4,237M (165.81%)	1,701M (6.71%)	3,926M (146.3%)	1,592M (−0.13%)
256.bzip2	**Read**	228M	390M (71.05%)	268M (17.54%)	229M (0.44%)	229M (0.44%)
	Write	249M	896M (259.84%)	407M (63.45%)	730M (193.17%)	249M (0%)
300.twolf	**Read**	13,600M	22,350M (64.34%)	15,820M (16.32%)	13,600M (0%)	13,610M (0%)
	Write	13,680M	48,650M (255.63%)	22,510M (64.55%)	38,090M (178.43%)	13,610M (−0.51%)

TABLE VI: The number of total memory read/write access from both the processor and DfiTagger.

Mechanism	**Attacks**	**Result**
Shadow stack	RIPE	✓
Heap metadata protection	Heap exploit	✓
VTable protection	VTable hijacking	✓
Code pointer separation (CPS)	RIPE	✓
Code pointer separation (CPS)	Format string exploit	✓
Kernel protection	Privilege escalation	✓
Private key leak prevention	Heartbleed	✓

TABLE VII: Security applications utilizing Hdfi can effectively prevent various attacks including Heartbleed (CVE-2014-0160).

IV **Usability**: Hdfi should be flexible, capable of supporting different security solutions; it should also be easy to use, so as to increase the chance of real-world adoption.

In this subsection, we evaluate whether Hdfi achieves these design goals or not. As described in §V, none of the Hdfi-powered security applications requires data shadowing, including three solutions (stack protection, CPS and Kenali) whose previous implementations rely heavily on data shadowing. For this reason, we consider Hdfi to have achieved goal III. And as shown in Table III, implementing/porting security solutions with Hdfi is very easy, so we consider goal IV to be achieved as well. Next, we analyze the security and efficiency benefit.

1) Security Improvement: Compare with software-based shadow stacks [21], our stack protection provides better security than platforms that do not have efficient isolation mechanisms, such as x86_64 and ARM64. Compared with existing hardware-based shadow stacks [46, 59, 81], our solution provides the same security guarantee but is more flexible and supports kernel stack. Compared to active callsite based solutions [23, 24], our stack protection provide better security. For standard libraries, existing heap metadata integrity checks can be bypassed under certain conditions. For example, Google project zero team has successfully compromised `ptmalloc` with NULL off-by-one [31]; and existing encryption-based exit handler protection is vulnerable to information leak based attacks. However, Our Hdfi-based library enhancement cannot be bypassed because attackers cannot control the hardware-managed tags. Compared with existing VTable protection mechanisms [7, 38, 71, 85, 86], our Hdfi-based solution has both advantages and limitations. On the positive side, our approach makes it much harder to overwrite the `vfptr`; while in all other solutions, attackers can easily tamper with `vfptr`. However, because our approach does not involve any class hierarchy analysis, we cannot guarantee type safety (i.e., semantic correctness). Compared to the original CPS implementation, our ported version provides the same security guarantee as segment-based isolation but is stronger than its randomization-based isolation, which has been proven to be vulnerable [32]. Compared to the original implementation of Kenali [66], our ported version provides stronger guarantees than its randomization-based stack. Based on the above analysis, we also consider Hdfi to achieve goal I.

2) Performance Improvement: Because we can neither fully port the original implementation of CPS and Kenali to our testbed due to problems with the official llvm-riscv toolchain nor run the C++ benchmarks of SPEC CINT 2000, we designed the following benchmarks to evaluate the performance improvement of Hdfi-based security solutions.

Micro benchmarks. Compared with the original implementation of CPS, our ported version would be more efficient because it does not need to access the shadow data. To demonstrate this benefit, we implemented a micro benchmark that measures the overhead for performing an indirect call for 1,000 times. To simulate CPS, we used their own hash table implementation and performed the same look up before the indirect call. For our implementation, we just replaced the load instruction with a checked load. Note, although our implementation sounds simpler, it provides the same level of security guarantee as the original segment-based CPS implementation. The result showed that our protection only incurs 1.6% overhead, whereas the hash table lookup incurred 971.8% overhead. Note, this micro benchmark only shows the worst case performance of both approaches. Depending on the running application, the real end-user performance impacts could be much less than this.

Because we cannot perform automated instrumentation to fully replicate Kenali, here we only measured the performance overhead of kernel stack protection. The result is shown in Table VIII. Although our prototype implementation has higher a performance overhead, it is also more secure than the randomization-based stack protection used in the original implementation.

Benchmark	Baseline	Kernel Stack Protection
null syscall	8.91μs	8.934μs (0.27%)
open/close	160.6μs	168.7μs (5.04%)
select	285.6μs	287.5μs (0.67%)
signal install	19.3μs	21.5μs (11.4%)
signal catch	99.8μs	105.6μs (5.81%)
pipe	273.6μs	306.6μs (12.06%)
fork+exit	5892μs	6308μs (7.06%)
fork+execv	6510μs	6972μs (7.1%)
page fault	50.0μs	52.6μs (5.2%)
mmap	800μs	880μs (10%)

TABLE VIII: LMBench results of baseline system and HDFI with kernel stack protection.

Benchmark	GCC	Shadow Stack	Clang[1]	CPS+SS[1]
164.gzip	981s	992s (1.12%)	1734s	1776s (2.42%)
181.mcf	8388s	8536s (1.76%)	11014s	11403s (3.54%)
254.gap	4254s	4396s (3.34%)	20783s	23526s (13.23%)
256.bzip2	722s	744s (3.05%)	1454s	1521s (4.61%)

TABLE IX: Performance overhead of HDFI-based shadow stack CPS. [1]Please note that because Clang cannot compile the benchmark with `-O2`, they were compiled with `-O0`.

SPEC CINT 2000. To measure the performance overhead of HDFI under the existence of load check and store set, we ran four benchmarks from SPEC CINT 2000 with two security protections: GCC-based shadow stack and CPS plus LLVM-based shadow stack. The result is shown in Table IX. As we can see, the performance overhead is also low. Please note that because Clang cannot compile the benchmarks with `-O2`, they are compiled with `-O0`. As a result, the performance is much worse than GCC. More importantly, because Clang did not optimize redundant stack access with `-O0`, it caused trouble for our current implementation of TVB (§VI-A); this is the reason why the `gap` benchmark behaved so badly on CPS.

VIII. Security Analysis

Being an isolation mechanism, HDFI cannot guarantee memory safety by itself, so it cannot prevent all memory corruption-based attacks. In this section, we analyze the security guarantee provided by HDFI and provide our recommendations on how to utilize HDFI properly in security solutions.

A. Attack Surface

The security guarantee of HDFI is in data-flow isolation, i.e., preventing data flowing from one region to another. This is enforced by (1) partitioning write operations into two groups: those who can set the memory tag to `1`, and those who set the memory tag to `0`; and (2) when loading, checking if the tag matches the expected value. In this regard, HDFI has the following attack surfaces:

1) Inaccuracy of Data-flow Analysis: The first challenge for utilizing HDFI is how to correctly perform partitioning and checking. To do so, we rely on data-flow analysis. For some security-critical data, such as return addresses and VTable pointers, their data-flow is quite simple, so the accuracy can be easily guaranteed even without any program analysis. For data like code pointers, because their data-flow is more complicated, it would require thorough static analysis to guarantee the accuracy. Fortunately, because these data are usually self-contained, i.e., not provided by external input, the accuracy, to some extent, can still be guaranteed. However, for data that exhibits complicated data-flow, it may not always be possible to guarantee the accuracy of static analysis. In this case, the common strategy is to avoid false positives by allowing false negatives, i.e., allowing some attacker controllable write operations to set the memory tags. As a result, HDFI itself is not sufficient to guarantee data integrity, so one must employ other runtime protection techniques to compensate for such inaccuracies.

2) Deputy Attacks: After partitioning, the next challenge is how to guarantee the trustworthiness of each write operation. More specifically, a write operation takes two parameters, a *value* and an *address*. The integrity of a write operation thus relies on the integrity of both the value and the address. If either of them can be controlled by attackers or the instruction gets executed under wrong context (e.g., via control flow hijacking), then they can launch *deputy attacks*. Please note that the control here means both *direct* and *indirect* control. For example, if attackers can control the object pointer used to invoke a C++ constructor, then even though our VTable pointer protection can prevent them from directly overwriting the VTable pointer, they can still leverage this constructor to overwrite the VTable pointer of an existing C++ object. Similarly, if a piece of sensitive data may propagate from one memory location to another, and one forgets to check the tag of the source before setting the tag of the destination to `1`, then an attack can leverage this bug to overwrite sensitive data with a value controlled by the attacker.

B. Best Practices

To mitigate the aforementioned attacks, we recommend utilizing HDFI in the following ways:

1) To prevent write operations from executing under the wrong context, it is important to enforce the integrity of the control flow, which is also required by other systems that enforces write capability [2, 10]. With HDFI, this can be easily achieved through protecting all the control data (e.g., CPS + shadow stack).

2) To prevent attackers from controlling the address parameter of write operations, it is important to recursively protect all pointers that are part of the dereference chain [43, 66]. It is worth noting that because HDFI is designed to be fine-grained and its protection is enforced efficiently by hardware, including more pointers would not be a big performance issue.

3) To prevent attackers from controlling the value parameter of write operations, one must ensure that the value is trusted. A value is trusted if any of these conditions hold: (1) it is a constant; (2) it is from a trusted register (e.g., the link register); (3) it is loaded from a memory location with the expected tag; or (4) the semantic of the current program context guarantees the trustworthiness of the value (e.g., during early kernel initialization or when the program is being initialized by the dynamic loader). Moreover, if the value may have both tags (e.g., unions in C), one should use the special memory copy instruction to propagate data with the tag when the data is not modified or leverage a exception handler when the data needs to be modified between load and store.

4) To compensate the potential inaccuracy of data-flow analysis, we recommend combining HDFI with a runtime memory safety enforcement mechanism like [36, 52]. By doing so, even if we allow attackers to control some write operations, the memory safety protection mechanism would prevent attackers from abusing those write operations to launch attacks.

IX. Limitations and Future Work

Direct Memory Access (DMA). Since our current prototype of HDFI only handles memory accesses from the processor core, it is vulnerable to DMA-based attacks. Attackers can leverage DMA to (1) corrupt the data without changing the tag and (2) directly attack the tag table. To mitigate this threat, we could leverage features like IOMMU to confine the memory that can be accessed through DMA [64]. Alternatively, we can choose to add our own hardware module in between the interconnect and the memory controller such that all memory accesses would pass through the hardware module. By doing so, our hardware module would be able to determine whether or not the access is from DFITAGGER, thus prevents malicious access to the tag table. It is worth noting that similar hardware modules have already been introduced [50] and deployed in commodity hardware [3, 36].

Configurable Tag Table. Our current implementation completely blocks accesses to the tag table. Although this provides a stronger security guarantee, it also comes with some drawbacks. The first problem is that we cannot save the page to disk because the tag information will lost. To support these features, we must allow the kernel to access the tag table. However, to protect the tag table from tampering, we must implement some protection techniques like [22] or integrity measurements like [36]. Another drawback of our current design is that we must allocate the whole tag table in advance. In the future, we could provide other options for the OS kernel or the hypervisor to manage the tag table depending on the security requirement by users. On such a model, we can implement an on-demand allocation mechanism to reduce the memory overheads, i.e., we allocate the tag memory only when DFITAGGER modifies a tag entry.

Opportunities for Further Optimizations. Although the Rocket Chip Generator is a great tool for prototype verification, the Rocket Core is a very limited processor compared to x86 processors. With a more powerful processor core like the Berkeley out-of-order machine (BOOM) [12] and a more sophisticated cache, we could further reduce the memory access overhead using the following techniques.

Tag prefetch: Just like prefetching data that is likely to be used in the future due to program locality, we could also prefetch the tag. We could both prefetch the tag from DFITAGGER to avoid possible read miss hit due to TVB and prefetch the tag entries from the main memory when the bus is free.

Delayed check: Just like speculating a branch, as most tag checks should not triggering the exception, with an out-of-order machine we could speculate the execution even when the tag is not ready (i.e., TVB miss hit). By doing so, we could avoid stalling the pipeline and further reduce the overhead of HDFI.

Better cache design: In our prototype implementation, we did not extend our modification to the L2 cache. At the same time, as mentioned in §VI-A, out current design of TVB is not ideal, which may cause some obvious performance overhead for unoptimized programs (§VII-D). For future work, we plan to extend our modification to the L2 cache with better TVB implementation.

Dynamic Code Generation. Dynamic code generation is an important technique that has been widely utilized in browsers and OS kernels to improve performance. However, because this technique requires memory to be both writable and executable, it may be vulnerable to code injection attacks [67]; and unlike static code, it is not always possible to detect malicious modification to the generated code. In the future, we can perform tag checking for instruction fetching, i.e., provide a configuration flag that once enabled, only allows tagged memory to be fetched as code.

X. Conclusion

In this paper, we have presented HDFI, a new fine-grained data isolation mechanism. HDFI uses new machine instructions and hardware features to enforce isolation at the machine word granularity, by virtually extending each memory unit with an additional tag that is defined by data-flow. To implement HDFI, we extended the RISC-V instruction set architecture and instantiated it on the Xilinx Zynq ZC706 evaluation board. Our evaluation using benchmarks including SPEC CINT 2000 showed that the performance overhead due to our hardware modification is low ($< 2\%$). We also implemented security mechanisms including stack protection, standard library enhancement, virtual function table protection, code pointer protection, kernel data protection, and information leak prevention on HDFI. Our results show that HDFI is easy to use, imposes low performance overhead, and improves security.

XI. Acknowledgment

We thank Manuel Costa and the anonymous reviewers for their helpful feedback, as well as our operations staff for their proofreading efforts. This research was supported by the NSF award CNS-1017265, CNS-0831300, CNS-1149051, and DGE-1500084, by the ONR under grant No. N000140911042 and N000141512162, by the DHS under contract No. N66001-12-C-0133, by the United States Air Force under contract No. FA8650-10-C-7025, by the DARPA Transparent Computing program under contract No. DARPA-15-15-TC-FP-006, by the ETRI MSIP/IITP[B0101-15-0644]. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, ONR, DHS, United States Air Force or DARPA.

References

[1] "CVE-2013-6282," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-6282.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[3] ARM, "CoreLink^{TM} TrustZone Address Space Controller TZC-380," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0431c/DDI0431C_tzasc_tzc380_r0p1_trm.pdf, 2010.

[4] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[5] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," DTIC Document, Tech. Rep., 1973.

[6] K. J. Biba, "Integrity considerations for secure computer systems," DTIC Document, Tech. Rep., 1977.

[7] D. Bounov, R. Kici, and S. Lerner, "Protecting c++ dynamic dispatch through vtable interleaving," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[8] W. S. A. Bradbury and R. Mullins, "Towards general purpose tagged memory," http://riscv.org/workshop-jun2015/riscv-tagged-mem-workshop-june2015.pdf, 2015.

[9] K. Bulba, "Bypassing stackguard and stackshield," *Phrack Magazine*, vol. 10, no. 56, 2000.

[10] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[11] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[12] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor," http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html, UCB, Tech. Rep. UCB/EECS-2015-167, 2015.

[13] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible hardware acceleration for instruction-grain program monitoring," in *Annual International Symposium on Computer Architecture (ISCA)*, 2008.

[14] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Usenix Security Symposium (Security)*, 2005.

[15] Codenomicon and N. Mehta, "The Heartbleed Bug," http://heartbleed.com/, 2014.

[16] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A. R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[17] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard tm: protecting pointers from buffer overflow vulnerabilities," in *Usenix Security Symposium (Security)*, 2003.

[18] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *International Symposium on Microarchitecture (MICRO)*, 2004.

[19] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[20] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[21] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.

[22] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[23] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "HAFIX: Hardware-assisted flow integrity extension," in *Annual Design Automation Conference*, 2015.

[24] L. Davi, P. Koeberl, and A. R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Annual Design Automation Conference*, 2014.

[25] D. Y. Deng and G. E. Suh, "High-performance parallel accelerator for flexible and efficient run-time monitoring," in *International Conference on Dependable Systems and Networks (DSN)*, 2012.

[26] S. Designer, "Getting around non-executable stack (and fix)," http://seclists.org/bugtraq/1997/Aug/63, 1997.

[27] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the C programming language," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[28] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, T. F. Knight Jr, B. C. Pierce, and A. DeHon, "Pump: a programmable unit for metadata processing," in *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2014.

[29] U. Drepper, "Pointer encryption," http://udrepper.livejournal.com/13393.html, 2007.

[30] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[31] C. Evans and T. Ormandy, "The poisoned NUL byte, 2014 edition," 2014.

[32] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point(er): On the effectiveness of code pointer integrity," in *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[33] J. N. Ferguson, "Understanding the heap by breaking it," 2007, black Hat USA.

[34] J. Graham-Cumming, "Searching for the prime suspect: How heartbleed leaked private keys," https://blog.cloudflare.com/searching-for-the-prime-suspect-how-heartbleed-leaked-private-keys/, 2014.

[35] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *Usenix Security Symposium (Security)*, 2015.

[36] Intel Corporate, "Intel architecture instruction set extensions programming reference," https://software.intel.com/en-us/intel-architecture-instruction-set-extensions-programming-reference, 2013.

[37] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," http://www.glue.umd.edu/~ajaleel/workload/, 2008.

[38] D. Jang, Z. Tatlock, and S. Lerner, "SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[39] M. Kaempf, "Smashing the heap for fun and profit," *Phrack Magazine*, vol. 11, no. 57, 2001.

[40] H. Kannan, M. Dalton, and C. Kozyrakis, "Decoupling dynamic information flow tracking with a dedicated coprocessor," in *International Conference on Dependable Systems and Networks (DSN)*, 2009.

[41] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," in *Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[42] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *Annual International Symposium on Computer Architecture (ISCA)*, 2014.

[43] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer Integrity," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[44] ——, "Code pointer integrity (early technology preview)," https://dslabpc10.epfl.ch/ssl_read/levee/levee-early-preview-0.2.tgz, 2014.

[45] D. Lea and W. Gloger, "A memory allocator," 1996.

[46] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi, "Enlisting hardware

architecture to thwart malicious code injection," in *Security in Pervasive Computing*. Springer, 2004, pp. 237–252.

[47] H. Mao, "make sure l2 passes no-alloc acquires through to outer memory," https://github.com/ucb-bar/uncore/commit/e53b5072caf12a2c18245cecd709204a4231d2d9, 2015.

[48] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[49] L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in *ATC Annual Technical Conference (ATC)*, 1996.

[50] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: Toward snoop-based kernel integrity monitor," in *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[51] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[52] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *International Symposium on Code Generation and Optimization (CGO)*, 2014.

[53] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: highly compatible and complete spatial memory safety for C," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[54] ——, "CETS: compiler enforced temporal safety for C," in *International Symposium on Memory Management*, 2010.

[55] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[56] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[57] Oracle, "Introduction to SPARC M7 and application data integrity (ADI)," https://swisdev.oracle.com/_files/What-Is-ADI.html, 2015.

[58] A. Ou, A. Waterman, Q. Nguyen, darius bluespec, and P. Dabbelt, "RISC-V Linux Port," https://github.com/riscv/riscv-linux, 2015.

[59] H. Ozdoganoglu, T. Vijaykumar, C. E. Brodley, B. Kuperman, and A. Jalote, "SmashGuard: A hardware solution to prevent security attacks on the function return address," *Computers, IEEE Transactions on*, 2006.

[60] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib(c)," in *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[61] C. Schmidt, "Low Level Virtual Machine (LLVM)," https://github.com/riscv/riscv-llvm, 2014.

[62] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the difficulty of preventing code reuse attacks in C++ applications," in *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[63] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary CPU architectures," in *Usenix Security Symposium (Security)*, 2010.

[64] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[65] H. Shacham, "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 (CCS)*, 2007.

[66] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity (to appear)," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[67] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski, "Exploiting and protecting dynamic code generation," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[68] Standard Performance Evaluation Corporation, "SPEC CPU2000 benchmark descriptions - CINT 2000," https://www.spec.org/cpu2000/CINT2000/, 2003.

[69] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[70] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[71] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Usenix Security Symposium (Security)*, 2014.

[72] UC Berkeley Architecture Research, "The RISC-V instruction set architecture," http://riscv.org/, 2015.

[73] ——, "Rocket chip generator," https://github.com/ucb-bar/rocket-chip, 2015.

[74] ——, "Rocket microarchitectural implementation of RISC-V ISA," https://github.com/ucb-bar/rocket, 2015.

[75] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[76] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual, volume I: User-level ISA, version 2.0," http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html, UCB, Tech. Rep. UCB/EECS-2014-54, 2014.

[77] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[78] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: runtime intrusion prevention evaluator," in *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[79] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[80] Xilinx, "ZC706 evaluation board for the Zynq-7000 XC7Z045 all programmable SoC user guide," http://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf, 2015.

[81] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer, "Architecture support for defending against buffer overflow attacks," in *Workshop on Evaluating and Architecting Systems for Dependability*, 2002.

[82] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis," in *International Workshop on Computer Science and Engineering(WCSE)*, 2012.

[83] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *IEEE Symposium on Security and Privacy (Oakland)*, 2009.

[84] N. Zeldovich, H. Kannan, M. Dalton, , and C. Kozyrakis, "Hardware enforcement of application security policies using tagged memory," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[85] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song, "VTrust: Regaining trust on virtual calls," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[86] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "VTint: Protecting virtual function tables' integrity," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[87] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "ARMlock: Hardware-based fault isolation for ARM," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.