CHINESE
JOURNAL OF ELECTRONICS

Design, Realization, and Evaluation of FastDIM to Prevent Memory Corruption Attacks

Jian Huang, Yanbo Li, Hao Han

Citation: Jian Huang, Yanbo Li, Hao Han, Design, Realization, and Evaluation of FastDIM to Prevent Memory Corruption Attacks, Chinese Journal of Electronics, 2025, 34(4), 1233–1246. doi: 10.23919/cje.2024.00.218.

View online: https://doi.org/10.23919/cje.2024.00.218

Chinese Journal of Electronics
vol. 34, no. 4, pp. 1233–1246, July 2025
https://doi.org/10.23919/cje.2024.00.218

RESEARCH ARTICLE

Design, Realization, and Evaluation of FastDIM to Prevent Memory Corruption Attacks

Jian Huang, Yanbo Li, and Hao Han

Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China

Corresponding author: Hao Han, Email: hhan@nuaa.edu.cn
Manuscript Received August 15, 2024; Accepted February 8, 2025; Published Online March 26, 2025

Abstract — Software vulnerabilities, particularly memory corruption, are significant sources of security breaches. Traditional security measures like data-execution prevention, address space layout randomization, control-flow integrity, code-pointer integrity/separation, and data-flow integrity provide insufficient protection or lead to considerable performance degradation. This research introduces, develops, and scrutinizes FastDIM, a novel approach designed to safeguarding user applications from memory corruption threats. FastDIM encompasses an low-level virtual machine (LLVM) instrumentation mechanism and a distinct memory monitoring module. This system modifies applications in user space into a more secure variant, proactively reporting vital memory operations to a memory monitoring component within the kernel to ensure data integrity. Distinctive features of FastDIM compared to prior methodologies are twofold: FastDIM's integrated out-of-band monitoring system that secures both control-flow and non-control data within program memory, and the creation of a dedicated shared memory space to enhance monitoring efficiency. Testing a prototype of FastDIM with a broad spectrum of real-life applications and standard benchmarks indicates that FastDIM's runtime overhead is acceptable, at 4.4% for the SPEC CPU 2017 benchmarks, while providing the defense against memory corruption attacks.

Keywords — Memory corruption, Control-flow integrity, Data-flow integrity, Compiler, Software hardening.
Citation — Jian Huang, Yanbo Li, and Hao Han, "Design, realization, and evaluation of FastDIM to prevent memory corruption attacks," *Chinese Journal of Electronics*, vol. 34, no. 4, pp. 1233–1246, 2025. doi: 10.23919/cje.2024.00.218.

I. Introduction

Within the domain of software security, memory corruption lays the foundation for numerous vulnerabilities and attacks that undermine contemporary computing environments. Memory corruption emerges as a persistent issue, particularly in low-level, memory-unsafe programming languages like C/C++. Attackers leveraging this flaw can illicitly modify data in memory, leading to unauthorized code execution, elevation of privileges, data breaches, and more. Such software flaws are central to a multitude of vulnerabilities and actual attacks. Addressing the intricacies of memory corruption demands advanced protective strategies, especially as adversaries continually refine their methods to exploit these weaknesses.

Memory corruption attacks fall into two primary classifications: control-oriented and data-oriented. Control-oriented attacks aim at memory objects linked to control transfers, including return addresses, function pointers, and pointers within virtual function tables. An intruder altering these specific data points can redirect or alter the control flow of a program, leading to the execution of unauthorized code. Conversely, data-oriented attacks, which focus on data not related to control, represent an evolving security challenge that has gained prominence recently.

Control-oriented cyberattacks initially exploited code injection techniques, but more recent strategies have employed return-oriented programming (ROP) [1] and just-in-time (JIT) [2] methods to execute code-reuse attacks. Various defensive mechanisms have been introduced to counteract control-flow hijacking, showing a trend towards enhanced practicality and efficiency. Such defenses encompass data-execution prevention [3], strategies for memory randomization including address space layout randomization (ASLR) [4], data space randomization [5], the insertion of randomized no-operation (NOP) instruc-

Associate Editor: Prof. Shen Wang, Harbin Institute of Technology.

tions [6], and timely address space randomization [7]. Moreover, control-flow integrity (CFI) along with its various adaptations [8], as well as measures for safeguarding code pointers like cryptographically enforced control flow integrity (CCFI) [9] and code-pointer integrity/separation (CPI/CPS) [10], are notable examples of these protective measures.

While control-flow attacks are significant, they are not the sole adverse effects stemming from memory corruption vulnerabilities. Non-control data, which is more prevalent in a program's memory, can also be a target for attackers, who might modify it to elevate privileges or circumvent security measures. The initial examples of non-control-flow (or data-oriented) attacks were introduced by [11], [12], highlighting that these attacks necessitate specific knowledge about the application's semantics, challenging even for advanced attackers. The true potential of data-oriented attacks was later unveiled by Hu *et al.* [13], who devised a novel data-flow stitching approach, enabling the systematic creation of data-oriented attacks without an in-depth understanding of program semantics. As data-oriented attacks do not alter control flows, they evade detection by leading control-flow protection mechanisms. To counter such attacks, methods like dynamic taint analysis [14], which tracks suspicious data flows and is often employed in malware detection and vulnerability identification, were adapted for real-time use. Data-flow integrity (DFI) [12] is another strategy, ensuring that data flows during execution do not stray from a pre-determined data-flow graph established via static analysis. Nonetheless, these defensive strategies can lead to significant overhead if specialized hardware support is unavailable.

Furthermore, most existing defenses against memory corruption attacks use inlined reference monitors (IRM), where monitor code is inlined into target programs. While this enables program-specific optimizations, one problem is that monitor code cannot run in parallel with target programs; this can cause a large performance overhead when the number of checks inlined into the program is significant.

To tackle these issues associated with memory corruption, we unveil FastDIM, an innovative framework engineered to enhance program defenses against memory corruption threats. FastDIM is comprised of two principal elements: an low-level virtual machine (LLVM)-based *instrumentation toolchain* and an out-of-band *memory observer*. The toolchain involves the collection of tools and the instrumentation process integrated into the LLVM compiler framework, which inserts security checks and monitoring capabilities into the code. A memory monitor, functioning as a kernel module, safeguards the integrity of these objects by keeping a shadow copy. If any discrepancies or unusual sequences of operations are detected, FastDIM intervenes by halting the program to avert potential memory corruption attacks. Notably, earlier CFI strategies utilized a comparable shadow stack to impose dynamic constraints on function returns. However, due to the significant overhead, the shadow stack was later discarded in favor of enhanced performance. FastDIM addresses this challenge by employing on-chip random access memory (OCRAM) to create shared memory, enabling rapid shadow copying and verification. This design allows the monitoring code to operate in parallel with the target programs, significantly improving efficiency compared to traditional inlined reference monitor designs. Additionally, we implement several optimization techniques, including loop-invariant code motion and caching strategies.

FastDIM underwent rigorous testing using the runtime intrusion prevention evaluator (RIPE), standard performance evaluation corporation (SPEC) central processing unit (CPU) 2017 benchmarks, and real-world applications with known vulnerabilities. The evaluations demonstrated that FastDIM effectively identified all attack attempts within the RIPE benchmarks and those targeting real-world susceptible applications. In terms of runtime performance, FastDIM demonstrated a geometric mean overhead of 4.4% and reached a maximum overhead of 27% on the SPEC CPU 2017 benchmarks.

In conclusion, our research presents the following key contributions:

- We have crafted an out-of-band memory observer that addresses both control- and data-oriented attacks on memory integrity. We have also created an LLVM instrumentation tool to allow applications to be protected by the monitor.
- Our development of OCRAM-based shared memory facilitates fast communication between the memory observer and the programs it safeguards. This, along with various optimization techniques, substantially lowers the monitoring overhead.
- A prototype of FastDIM has been developed and assessed using benchmarks and real-world scenarios on the development board with model IMX6DQ6DSL. The outcomes of these evaluations indicate that our method effectively identifies memory corruption attacks while maintaining a manageable overhead.

The following parts of this paper are structured as follows: Section II explores prior research, highlighting established methodologies and their constraints. Section III describes the design principles of FastDIM, with Section IV describing a detailed implementation. Section V offers an extensive assessment of FastDIM's performance. Finally, Section VI summarizes our insights and outlines prospective research avenues.

II. Related Work and Background

1. Related work

Control-flow attacks and defenses CFI is recognized as a systematic security approach [8] to counter control-flow attacks. Notably, CFI has been integrated within Clang and Visual Studio [15], respectively. Essentially, CFI ob-

structs an attacker's ability to alter control flows at will during a program's execution, maintaining adherence to a predefined control-flow graph while the program runs. Over decades, original CFI has been improved in various ways. For example, compact control flow integrity and randomization [16] enhances performance by randomizing legitimate transfer target layouts. Opaque CFI [17] is designed so that even if attackers analyze the modified code, they can't gain insights into the legitimate control jump targets. A CFI method for binaries (bin-CFI) [18] offers a CFI implementation relying solely on a stripped binary without needing the program's source code. Modular CFI [19] applies CFI principles to programs composed of distinct compilation units, enhancing modular security.

Besides, hardware capabilities [20]–[28] have been leveraged to enhance the effectiveness of CFI. Control flow integrity montior (CFIMon) [20] utilizes the branch tracing features of the processor to gather transfer targets selected by the application, verifying these against a points-to-analysis outcome. kBouncer proposed in [21] intervenes during system call executions, utilizing the last branch record (LBR) available on Intel processors to identify transfer target patterns indicative of an ROP attack. ROPecker proposed in [22] also intervenes at critical security checkpoints, reviewing the LBR while integrating past branch choice reviews with a forward-looking analysis. Address-based CFI [24] presents a hardware-assisted fine-grained CFI design that reformulates labels as the lower bits of addresses. Slot-based CFI [27] proposes a new fine-grained forward CFI implementation by combining existing coarse-grained instruction set architecture extensions with software modifications.

Non-control attacks and defenses The initial demonstrations of data-oriented attacks [29] introduced a classification where attacks focusing solely on crucial data structures were labeled as pure data attacks. Subsequent studies [11] delved deeply into this type of assault, uncovering their profound effects on practical applications. Identifying susceptible critical data within source code for these exploits traditionally demanded manual labor. However, an innovative approach by research [13] enabled the automated generation of data-oriented exploits. Research [30] introduced a more nuanced technique for crafting such exploits, using specific vulnerable functions to develop Turing-complete challenges to CFI. Additionally, data-oriented programming [31] has enhanced the depth of non-control data attacks by identifying data-oriented gadgets within the software, enriching the potential for such attacks.

Numerous defensive strategies have been developed to thwart data-oriented attacks. Data-flow integrity, for instance, aims to ensure that a program's data flow remains consistent with a predefined data-flow graph [12], [32]–[35]. Similar to CFI, DFI has also been accelerated by hardware features. For example, runtime scope type integrity [36] leverages ARM Pointer Authentication to enforce both code and data pointers to conform to the original programmer's intent. Alternatively, some defenses focus on enhancing memory safety. For example, Softbound [37] introduces memory safety to the inherently unsafe C language through bound checking and fat pointers. Compiler enforced temporal safety [38] advances this approach by specifically targeting the prevention of memory errors, while CCured (a program transformation system that adds type safety guarantees to existing C programs) [39] introduces a type-safe system that can statically identify potential memory errors and apply dynamic bound checks to mitigate them. However, the significant performance overhead associated with these methods can hinder their practical application.

2. Motivating examples

Figure 1 shows an instance of a program with vulnerabilities that are challenging for the aforementioned solutions to secure effectively. In this example, the readPacket() function (line 6) has a buffer overflow vulnerability. It may overwrite adjacent memory of packet and change the value of authenticated from 0 to 1 intentionally. In that case, the attacker can bypass the authentication branch (line 7) without providing a valid credential. While path-sensitive analysis [40] might resolve this particular issue, the set pertaining to any control-flow transfer must be precisely approximated. Any errors in this approximation could result in the program malfunctioning even when protective measures are in place.

```
1  int server() {
2      int authenticated = 0;
3      char packet[1000];
4      void (*handler) (char *);
5      while(!authenticated) {
6          readPacket(packet); //vulnerable function
7          if(Auth(packet)) {
8              authenticated = 1;
9              if (getUser(packet) == ADMIN){
10                 handler = priv;
11             }else{
12                 handler = unpriv;
13             }
14         }
15     }
16     if (authenticated == 1)
17         handler(packet);
18     ⋮
19 }
```

Figure 1 A demonstration showing the impact of non-control data.

Figure 2 showcases a scenario derived from a known real-world vulnerability (from common vulnerabilities and exposures, CVE-2016-2059) involving a linked list of compound structures, each embedded with a function pointer. In this example, a susceptible function (line 18) interacts with a structure, potentially removing some structures from the list and deallocating them from the heap. Consequently, when the function pointer func is accessed at line 21, there's a risk it could reference a freed location on the heap or, worse, an address manipulated by an attacker post-deallocation of the compound structure. This situation underscores the difficulty static

analysis faces in accurately determining the legitimacy of a function pointer's content at a specific moment in the program's execution.

```
struct compound {
    struct list_head list;
    ⋮
    int (*func)(void *data);
};//this structure contains a function pointer
  //and is attached to a list
    ⋮
int foo(){
    ⋮
    struct compound mylist;
    INIT_LIST_HEAD(&mylist.list);
    struct compound *tmp;
    ⋮
    tmp= (struct compound *)malloc(sizeof(struct compound));
    tmp->func = &valid_func;
    list_add(&(tmp->list), &(mylist.list));
    ⋮
    vulnerable(tmp);//this function may free tmp from the heap
    ⋮
    if (tmp->func)
        tmp->func(data);
        /*this function pointer may point to an invalid address
        forged by attacker*/
}
```

Figure 2 A case study demonstrating how use-after-free vulnerabilities can circumvent conventional CFI defenses.

3. Threat models

User programs are compiled using a specialized LLVM toolchain and are under the surveillance of our memory observer. While these applications are generally secure, they may contain vulnerabilities that could enable memory corruption by an attacker, who then aims to undermine the execution integrity and take control of the program's behavior. Echoing prior research, we postulate that the attacker is restricted to corrupting writable memory areas, meaning they cannot alter read-only sections, such as the executable code. Applications not compiled with our toolchain are considered unreliable and are not allowed to execute without restrictions. Consequently, we exclude the possibility that malware under the attacker's command can execute any operations within the system. Employing code scanning and anti-virus tools is advised to mitigate these risks.

Our trust is placed solely on the security of operating system kernels. Consistent with preceding research, this paper does not account for physical intrusions, including cold boot [41] attacks and bus monitoring [42], [43], nor does it address denial of service (DoS) attacks or cache side-channel exploits [44]–[47].

III. System Design

1. Overall architecture

FastDIM represents an integrated approach designed to safeguard the integrity of security-sensitive memory objects, thereby thwarting data-oriented attacks. This protection is facilitated by *compiler instrumentation* coupled with a *memory observer*. As depicted in Figure 3, the tailored compiler inserts programs with a shared memory space between the application and the observer for logging the activities of safeguarded memory objects. This arrangement enables the observer to verify data integrity perpetually. The observer within the kernel space keeps a shadow copy of the safeguarded memory objects, synchronizing these copies with the program's actions. If a memory object is compromised through a vulnerability, it will go unreported to the observer, causing discrepancies between the shadow copy and the actual data. Moreover, the observer will identify any abnormal reporting sequence that deviates from the program's anticipated control flows, prompting program termination and an alert signal.

Our methodology distinguishes itself from prior efforts by offering a segregated environment for program monitoring. A standalone observer, operational within the

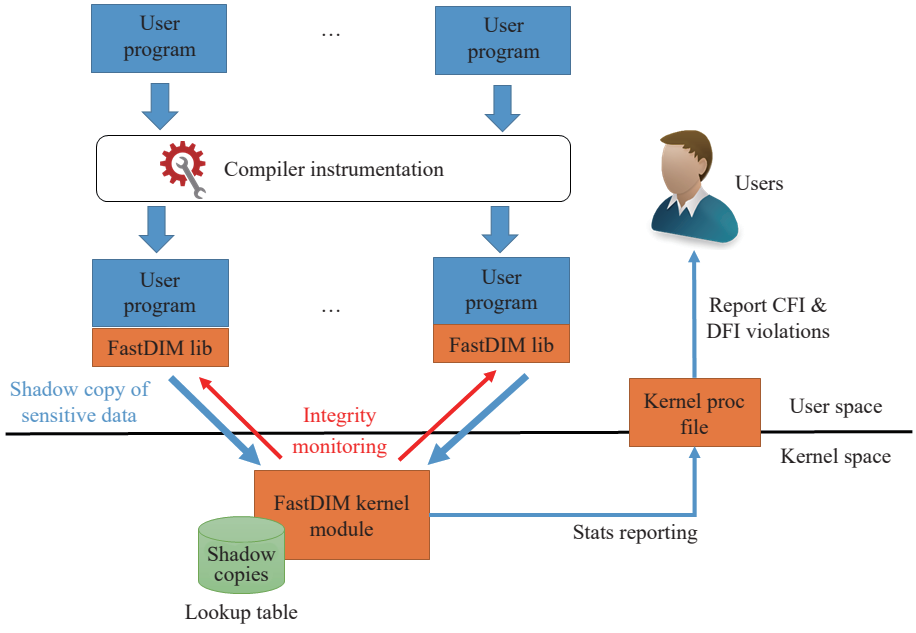


Figure 3 An overall system architecture of FastDIM.

kernel, is tasked with verifying data integrity. The use of OCRAM as shared memory facilitates swift interaction between the program and the observer. Individual programs, identified by their task identifier (tid), concurrently post messages to their designated shared memory areas. The observer accesses these messages from the shared memory, ensuring the program's operation remains uninterrupted. Interaction with the shared memory is restricted to well-defined application programming interfaces (APIs), with any unauthorized attempts being intercepted. To ensure security, a message authentication code (MAC) verifies each message, preventing programs from sending unauthorized messages to shared memory regions not assigned to them. Subsequent sections will delve into the specifics of each FastDIM component.

2. Compiler-time instrumentation

The objective of the instrumentation is to integrate reporting instructions into applications via our specialized compiler. Initially, this compiler is tasked with pinpointing all the memory elements that require protection. In this context, sensitive information encompasses elements such as function pointers, virtual function tables (vptr), return addresses, and additional non-control data that users have marked for attention.

- Return addresses. The return address, a well-known focal point for control-flow attack vectors, is stored on the stack each time a function is invoked. If the return address has been compromised, it provides an avenue for attackers to execute chosen code, such as injected scripts or ROP gadgets. Despite the employment of existing defense mechanisms like stack canaries, ASLR, and Safestack can address this problem to some extent, the latest research [48] reveals that more sophisticated attacks can still bypass these measures. Therefore, protecting return addresses continues to be a critical component of our FastDIM.
- Function pointers. Function pointers are a vital component of control-flow transfers, encapsulating the address of functions. Typically situated in the stack, heap, or various data segments, they play an essential role in the functionality of programs. If an attacker manages to tamper with a function pointer, they can manipulate the program's control flow to divert to a specific location when the program performs an indirect call via that pointer.
- Vtables pointers. In C++, classes with virtual functions, or those inheriting from such classes, utilize a virtual function table (vtable) during execution. This vtable serves as a look-up table containing pointers to the code of each virtual function within the class, facilitating dynamic dispatch. While the vtable is located in read-only memory segments, the pointer to the vtable's base (commonly known as vptr) resides in a writable memory area. An attacker can modify the program's control flow by modifying this vptr, thereby affecting the behavior of C++ programs.
- Annotated non-control data. As shown in Figure 1, data that does not control program flow can still be pivotal. For instance, altering parameters in crucial system calls (e.g., setuid, execute) could result in unauthorized privilege elevation or execute unexpected program actions. Modifying data in memory, especially data derived from configuration files, might enable the circumvention of built-in access controls in server applications. Moreover, if data that influences decision-making is altered, an attacker could steer a program's control flow towards unintended pathways, potentially evading security mechanisms. Identifying such critical non-control data automatically remains a challenging issue, necessitating a deep understanding of the program's semantics, which is beyond the scope of this discussion. Nonetheless, some methodologies from prior research, such as [49], [50], offer strategies for automatically detecting sensitive data.

Locating sensitive data in the program To identify various types of sensitive data, we employ distinct strategies. For example, a return address is typically identified by a fixed offset from the stack's frame pointer. Due to optimizations like function inlining and leaf-function optimization, not all functions store their return addresses on the stack. If the return addresses are not on the stack, monitoring them is unnecessary, assuming that an attacker cannot modify the registers directly. For locating function pointers, we depend on the type information provided by the compiler. For function pointers embedded within complex data structures (such as structures, arrays, or pointers to function pointers), we conduct a recursive search through all fields of an aggregate type and its subtypes. Special attention is required when a function pointer undergoes casting to or from other types, for instance, a void* type. We apply flow analysis to track memory objects that are not initially identified as function pointers but derived from or converted to function pointers. Last, identifying vtable pointers is done via the C++ application binary interface, using naming conventions for virtual function calls to locate all vtables and vptrs associated with each class object. For sensitive non-control data, we use attribute annotations (__attribute__((annotate("sensitive")))) in the source code to mark them explicitly.

Verifying sensitive data Once FastDIM identifies the relevant memory objects, the next step is to insert the code of security reporting within the program. This process involves monitoring all store instructions and memory duplication operations, like memcpy, to notify the observer, enabling the creation of a shadow copy of the objects' legitimate contents. Nonetheless, this approach encounters two practical challenges. First, determining the content of a pointer reference is complex. For instance, program A utilizes memcpy(i8*<dest>, i8*<src>, i32<len>) to copy an array of function pointers. Program B uses the same function to duplicate structured data containing a function pointer field. Relying solely on memcpy's provided data is inadequate. Therefore, we

use backward analysis to ascertain the original pointer type. Second, reporting the contents of memory objects initialized within a function is straightforward. We can insert reporting instructions right after the object initialization. For memory objects initialized statically, we place reporting instructions at the main function's start, ensuring the observer is informed of their correct values prior to their initial use. However, how do we handle memory objects initialized in libraries without a main function? To solve this, we devise a helper function with a distinct naming convention, avoiding the need for unstable link-time optimization (LTO) or cross-dynamic shared object methods. This helper function encompasses all necessary reporting instructions for memory objects initialized within libraries. Upon the program's launch, we search for these helper functions in the libraries' symbol tables. If found, we dynamically invoke these functions at the main function's commencement. Section IV delves deeper into the implementation specifics.

To maintain the integrity of sensitive data in a program, it's crucial to inform the observer for validation checks whenever such data is utilized. In the case of non-return data like function pointers, vptrs, and specially annotated memory objects, FastDIM monitors every instruction that loads data, as these elements need to be loaded into registers prior to their use. Reporting instructions are strategically placed just before each load instruction. Additionally, when memory is transferred from one location to another, we integrate instructions before the copy operations to verify the source's integrity. For return addresses, the reporting instructions are placed prior to the ret instructions in functions that are unoptimized. It's important to note that no additional instrumentation is required for sensitive data already present in registers, as such data would have been verified when loading into the registers.

Lifetime of sensitive data To mitigate the attack shown in Figure 2, FastDIM monitors the lifespan of sensitive data. Such data, when allocated on a function's stack, is eradicated once the function concludes. FastDIM aids in this process by integrating reporting instructions prior to the function's ret instruction. Similarly, when deallocating a memory segment on the heap, FastDIM dispatches an alert to the observer, who then expunges the associated shadow copies. During a fork() system call, which clones the memory pages of the parent process for the child, it's crucial to inform the observer about the new memory locations. While both processes initially seem to hold separate instances of sensitive data, the copy-on-write (COW) mechanism delays the duplication of the parent's memory pages until they are altered. Therefore, FastDIM incorporates a post-fork reporting instruction to signify the commencement of the sensitive data's new cycle within the child process.

3. Runtime integrity monitoring

The responsibilities of the observer encompass: 1) Overseeing the shared memory that facilitates communication across user and kernel spaces; 2) Confirming the legitimacy of the actions on safeguarded data as reported by the applications; 3) Executing appropriate measures depending on operations; 4) Halting the execution of a compromised program if an infringement is identified.

Figure 4 depicts how an OCRAM is utilized to configure shared memory, incorporating several circular buffers to speed up the data transfer process to the observer. This shared memory is structured as a block matrix, with each column of the matrix representing an individual circular buffer. Within these circular buffers, each element is organized into a specific data structure, detailed as follows:

$$\text{Msg} := \begin{cases} \text{tid, address, value, OP\_STORE, MAC} \\ \text{tid, address, value, OP\_LOAD, MAC} \\ \text{tid, address, \_, OP\_PUSH, MAC} \\ \text{tid, address, \_, OP\_POP, MAC} \\ \text{tid, \_, parent's tid, OP\_FORK, MAC} \\ \text{tid, \_, parent's tid, OP\_FREE, MAC} \end{cases}$$

Each circular buffer is allocated to programs sharing the same hash of their tid. When a program needs to report an operation involving protected data, it inserts an entity into its designated circular buffer. This approach permits various processes to interact with distinct circular buffers simultaneously. A locking mechanism is employed to prevent race conditions when multiple programs share a single circular buffer. Each entry within the circular buffer includes a MAC. Initially, our instrumentation tool injects additional code to acquire a key from the observer when setting up the shared memory for each program. This key is used to append a MAC to every message sent to the shared memory. The observer, possessing the corresponding key, verifies these messages. Unique keys are assigned to different circular buffers, preventing attackers without the correct key from manipulating the buffers. Furthermore, interaction with the shared memory is restricted to specific APIs, with any unauthorized attempts being intercepted. To circumvent a scenario where a malfunctioning program could monopolize the shared memory, a timeout feature is in place, ensuring that the memory is not locked indefinitely.

The protected program communicates with the observer through six distinct operation types: OP_STORE, OP_LOAD, OP_PUSH, OP_POP, OP_FORK, and OP_FREE. For instance, to indicate the initialization or modification of a function pointer, vtable pointer, or other non-control data, the program sends an OP_STORE message, which includes the data's address and content, to the shared memory. The observer then creates a shadow copy of this data in a specialized lookup table. When dealing with return addresses, an OP_PUSH message is dispatched to shared memory, prompting the observer to replicate the return address in a shadow stack. Notably, an OP_PUSH message contains only the address, unlike

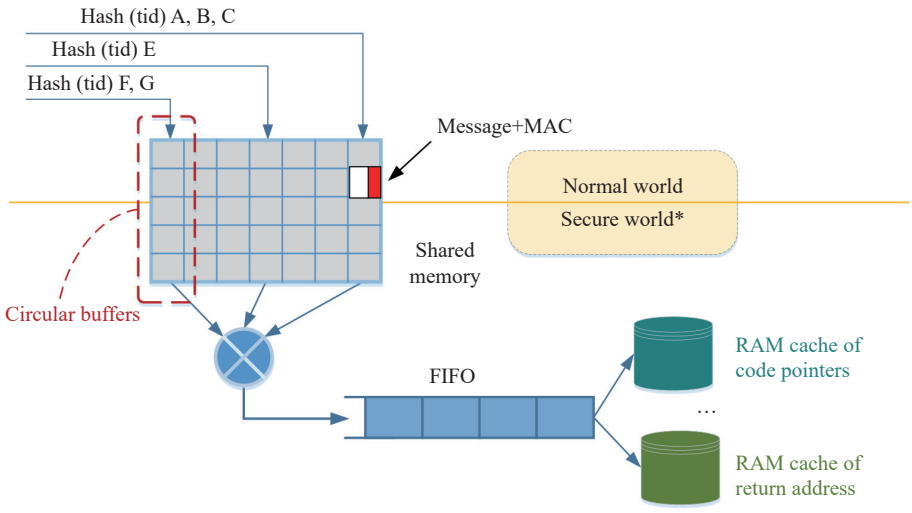


Figure 4 Circular buffers and look-up tables in shared memory (The FIFO in the figure represents a first in first out queue).

an OP_STORE message, which carries both address and value. For usage of sensitive data, the program sends either an OP_LOAD or OP_POP message to the shared memory, signaling the observer to check data integrity. Depending on the type of operation, the observer consults the shadow lookup table for non-return data or the shadow stack for return addresses. If the observer finds that the shadow copy is missing or differs from the program's reported runtime value, a violation is flagged. In the case of an OP_FORK message, the observer duplicates all shadow copies from the parent to the child process, safeguarding the sensitive data in the new process. Lastly, an OP_FREE message triggers the observer to eliminate any related shadow copies from the tables or stacks, thwarting use-after-free vulnerabilities.

The observer operates a random access memory (RAM) cache dedicated to a specific category of safeguarded memory objects. These objects are organized using hash tables to enable $O(1)$ search efficiency, although the implementation nuances vary.

Shadow lookup table Function pointers are maintained in a two-tier lookup table. As illustrated in the left part of Figure 5, a hash bucket contains the task ID, the data address, the data's value, and a timestamp. Whenever a new message is written to the shared memory by the program, the observer verifies the message and calculates the hash key using the task ID and data address included in the message. Once the key is determined, the observer interacts with the appropriate hash bucket depending on the message type. For example, if the type is OP_STORE, the observer will create a new bucket to make a shadow copy of sensitive data given that no match is found or update the data value of the existing bucket in the table. If the message type is OP_LOAD, the observer will compare the runtime value of sensitive data to the stored shadow copy. The observer will issue a violation if a mismatch or no bucket is found.

To enhance the efficiency of replicating or deleting groups of shadow copies during task forking or exit, an auxiliary task hash table is established, as depicted in the right part of Figure 5. Within this table, keys are derived solely from the task ID, and each hash bucket holds a list of shadow copy references sharing that task ID. Consequently, this structure enables the deletion of obsolete entries or the duplication of entries from a parent task to be executed as an $O(1)$ operation.

Shadow stack Similar to the hash table for indirect calls, the observer manages a shadow stack of each process under monitoring in another hash table. A hash key is generated from the task ID, and a hash bucket holds the task ID along with a shadow stack that exclusively contains return addresses. When a user-space program (process) pushes a return address onto the stack, the observer will do the same operation on the shadow stack associated with that process. When a return address is popped when a callee function returns to the caller function, the observer will also pop the stored address from the shadow stack and compare if the two addresses are equal. A return address violation will be claimed if they are not equal, and the observer will take subsequent actions. When a task exits, the module will remove all remaining entries from the hash table. Unlike the indirect call, deleting the task entries and their local stacks is straightforward, and no auxiliary hash table is needed.

4. Discussion

Security guarantees FastDIM ensures the integrity of security-related memory elements such as return addresses, function pointers, vtable pointers, and annotated sensitive data (e.g., keys and configures), thereby thwarting various memory corruption attacks. The common types include buffer overflow in stack or heap, format string vulnerability, use-after-free, and integer overflow. These attacks are typically the cause of more advanced attacks such as ROP attacks, JIT attacks, and code reuse attacks. Moreover, FastDIM functions akin to an intrusion detection system, identifying irregular sequences in the

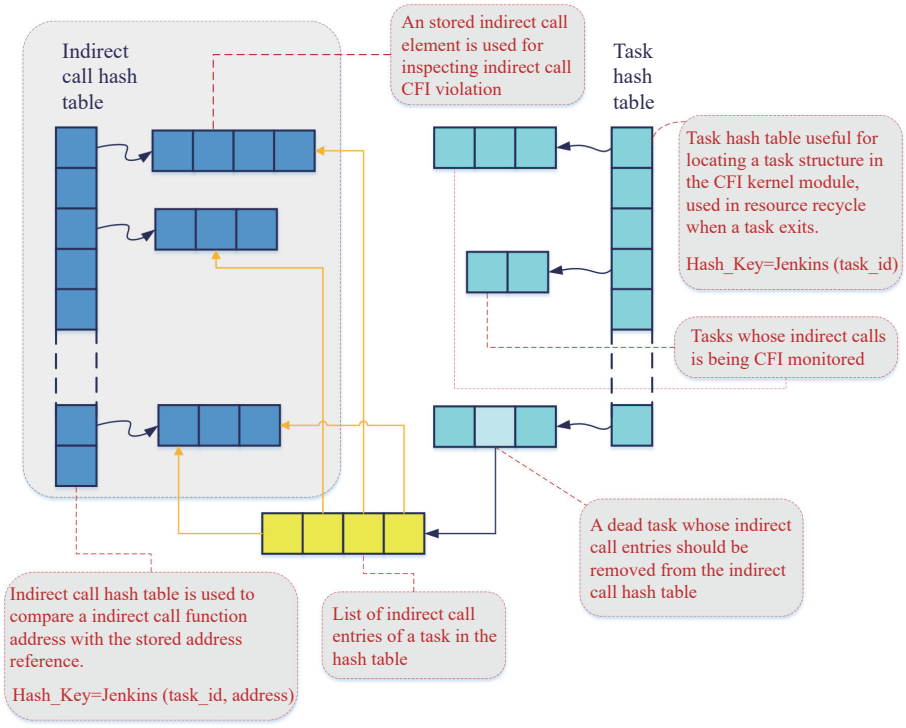


Figure 5 Dual-lookup table for indirect calls and the basic workflow.

operations of protected data that deviate from a program's anticipated control flows.

FastDIM implements stringent measures to ensure attackers cannot misuse shared memory to evade detection. It restricts the program's interaction with shared memory exclusively to predefined APIs. Any unauthorized access to shared memory outside these APIs is blocked because the OCRAM is not mapped to those unauthorized processes. Furthermore, each program receives a unique key when initializing shared memory communication with the observer. This key is used to create a MAC for every message sent to the shared memory, preventing attackers without the key from altering the circular buffers undetected.

Limitations Our methodology prioritizes runtime efficiency by allowing programs to run without pausing for integrity verification. This approach might briefly enable an attacker to execute exploits before the observer detects memory corruption. Nonetheless, the window for such exploitation is minimal, given that the observer operates on a dedicated core, continuously monitoring messages from circular buffers in an uninterrupted round-robin sequence. Consequently, an attacker's attempt to flood its circular buffer won't hinder the observer's operation or lessen the likelihood of detection. To minimize the potential for DoS attacks, future enhancements will focus on implementing access controls for protected programs, such as establishing a whitelist and setting limitations on the rate of message writing.

A potential issue is shared memory, which acts as a system bottleneck. To manage the shared memory region's performance, FastDIM uses an OCRAM-based shared memory with circular buffers and a locking mechanism to prevent race conditions when multiple programs attempt to write simultaneously. Circular buffers help reduce collisions by ensuring that each program has dedicated space, while the locking mechanism prevents buffer overflow and ensures orderly access. For environments with many concurrent processes, scalability is further enhanced by adjusting buffer sizes based on observed usage patterns. Regarding adversarial attacks, the shared memory region is fortified by exclusive access through specific APIs and message authentication codes, as mentioned above. If the platform has hardware support such as trusted execution environment, shared memory can be physically isolated or set up with dedicated access controls that enforce stronger separation.

For further robustness, we are exploring techniques like rate-limiting for message writes and adding quotas per process to prevent DoS scenarios within shared memory. Additionally, our further work will extend our framework to monitor shared memory usage patterns actively, triggering alerts if anomalous behaviors indicative of adversarial actions are detected.

IV. Implementation

The FastDIM framework has been designed with scalability and portability in mind. Although the current implementation is developed within LLVM 3.9, the memory observer component is integrated into the Linux kernel. It can be adapted across different operating systems, compilers, and hardware architectures. That is because FastDIM is primarily designed for portable operating system interface (POSIX)-compliant operating systems.

Its use of LLVM for instrumentation ensures compatibility with any operating system that supports the LLVM toolchain. The out-of-band monitoring framework can also be ported to other Unix-like systems with kernel-level modules, as FastDIM primarily relies on generic kernel-space memory operations. Later in Section V, FastDIM has been evaluated on x86 and ARM architectures.

1. LLVM transform pass

Within the LLVM compiler structure, a program's source code is initially transformed by the front end (utilizing Clang/Clang++ for C/C++ languages) into an intermediate representation (IR). Subsequent to this, a series of LLVM passes are applied for optimization purposes. Following these optimizations, the back end is responsible for producing assembly code tailored to particular platforms, utilizing the refined IR. To ensure compatibility and performance, FastDIM conducts its instrumentation at both the IR and assembly stages. Instrumentation concerning non-return data occurs at the IR stage, while return address instrumentation is conducted at the assembly stage, allowing for targeted protection decisions post-optimization.

2. Support of libraries without LTO

Traditionally, LTO is utilized to facilitate the exchange of information between modules during the linking phase, enabling the sharing of target data from libraries to the program. However, leveraging LTO in LLVM typically requires substituting system files like ld, ar, and ranlib with versions that support the gold plugin. Moreover, both the program and its associated libraries need to be compiled using LTO. To circumvent these limitations, we devised a method allowing each module to be compiled separately while still supporting modularity. This approach involves embedding a helper function whenever target data is detected. While not altering the library's core functionality, these functions offer a straightforward method for the program to communicate with the monitor. Upon the program's initiation or when libraries are dynamically brought in using dlopen, the program is set up to inspect the symbol tables of the linked libraries for our custom helper functions. Detected helper functions are then executed right after the library loads. This technique enables the static incorporation of sensitive data details into the code and the dynamic transfer of this data across modules upon execution, all without the necessity for LTO.

3. Communication between application and monitor

An OCRAM-based shared memory is established between the applications and the monitor, which enables simultaneous program execution and integrity verification. Specifically, the monitor allocates a segment of OCRAM memory within the kernel. User applications are configured to access the monitor device (e.g., /dev/monitor) and link the designated ring buffer (see Figure 4) to their virtual address space, utilizing their tid. When transmitting an operation message to the monitor, the program puts the message into the ring buffer, functioning as a producer. Concurrently, the monitor, serving as a consumer, retrieves messages from the ring buffer. This shared memory arrangement allows both entities to function autonomously, eliminating the need for context switching. This setup considerably diminishes the latency associated with integrity checks for user applications. However, it is worth noting a minor drawback: a slight delay exists between the usage of sensitive data and its integrity verification. To minimize this lag, the verification point is strategically positioned before the actual load instructions. Experimentation has shown this delay to be negligible.

To minimize overhead, OCRAM is employed as the shared memory in specific hardware setups, like the i.MX6 processor. This OCRAM, integrated into the memory address space, can be directly accessed through the advanced extensible interface bus, bypassing the need for a memory management unit required for regular RAM access. This approach substantially lowers run-time overhead. The memory observer efficiently polls data from this shared memory, ensuring prompt processing of write operations. A challenge with using OCRAM is the risk of data clashes or corruption when multiple cores attempt simultaneous writes to the OCRAM. To counteract this, circular buffers are utilized to decrease collision chances, and a locking mechanism is employed to secure OCRAM segments during access by any core. This polling method, while quicker than interrupt-driven approaches, does have a trade-off in terms of consuming certain CPU resources. Detailed results from this approach are discussed in the subsequent section.

4. Further optimization

We noticed that certain programs access function pointers or vptrs repetitively within loops, even though the pointers' values seldom change. Thus, it is inefficient to verify the integrity of these pointers during each iteration of the loop. Leveraging this insight, we applied a compiler strategy known as *hoisting* to execute loop-invariant code movement. When a function pointer is identified as invariant within a loop, our technique relocates the integrity verification code outside the loop, ensuring the check is executed just once per loop execution. For cases where a function pointer's value is altered within the loop, we introduced a caching mechanism to prevent redundant checks. This cache stores all data written to the shared memory in an additional buffer. If a value is reused, the system first verifies its presence in the cache. If found, it bypasses notifying the integrity monitor for a new check (i.e., writing the runtime value to the shared memory with operator type OP_LOAD), thereby reducing the frequency of shared memory write operations. This approach considerably decreases the number of shared memory write activities.

V. Evaluation

We conducted a series of detailed experiments to assess our prototype, with the objective of addressing these inquiries:

(Q1) Correctness Does our security mechanism adversely impact the intended operations of the programs?

(Q2) Effectiveness Does our prototype successfully detect memory corruption attacks targeting return addresses, vtable pointers, function pointers, and other non-control data types?

(Q3) Efficiency What is the extent of performance overhead that FastDIM introduces to the protection process?

1. Performance on benchmarks

SPEC CPU 2017 (Q1&Q3) SPEC CPU 2017 [51] has a set of programs and additional commands/scripts for benchmarking. Each program was compiled using Clang/LLVM 3.9 and tested on an Ubuntu 14.04.5 system with an Xeon (R) CPU E5-1620 and 16 GB of RAM. It should be noted that Clang/LLVM 3.9 is the latest stable version as this work began. Since the LLVM is backward compatible, the toolchain can be easily ported to the latest version (e.g., LLVM 10).

We leverage the runcpu command to validate the correctness (Q1) of the generated executables. Such a command will set up all of the benchmarks using the test workload, run them, and verify whether we get correct answers. The results show that all the hardened programs passed the correctness tests. Table 1 presents the overhead of running the SPEC CPU 2017 benchmarks. The "KLoc" column in the table indicates the benchmark's code size in thousands of lines. The "Original" column records the average execution time in seconds across three iterations of the reference workload. To apply FastDIM's protection, we altered the CMake files to include our LLVM pass (noted in LVM_MODULE_PATH) and library (in link_directories) and modified the CFLAGS, CXXFLAGS, and LDFLAGS. The table's fourth column details the overhead when FastDIM secures only function pointers, and the fifth column displays the overhead for protecting all specified target types.

Table 1 Performance cost of SEPC CPU2017 benchmarks (n/a: not available)

Benchmark	Measured performance				Reported performance			
Programs	Original (s)	KLoc	**Ours**	**Ours (FP)**	πCFI	Lockdown	CCFI	binCFI
557.xz_r	391 ± 17	33	+3.6%	+3.3%	n/a	n/a	n/a	n/a
541.leela_r	491 ± 1	21	+8.3%	+0.9%	n/a	n/a	n/a	n/a
531.deepsjeng_r	323 ± 3	10	+0.3%	+0.7%	n/a	n/a	n/a	n/a
525.x264_r	350 ± 1	96	+10.5%	+6.2%	n/a	n/a	n/a	n/a
523.xalancbmk_r	375 ± 3	520	+29.6%	+15.3%	+10.3%	+118%	+170%	n/a
520.omnetpp_r	489 ± 4	134	+32.6%	+21.3%	+6.7%	n/a	n/a	+45%
505.mcf_r	395 ± 6	3	+1.8%	+0.1%	+4.0%	+2.0%	+10%	0
502.gcc_r	315 ± 3	1304	+14.5%	+8.9%	+6.1%	+50%	n/a	+4.5%
500.perlbench_r	488 ± 5	362	+26.8%	+21.9%	+8.2%	+150%	n/a	+12%
544.nab_r	457 ± 0	24	+0.4%	+0.4%	n/a	n/a	n/a	n/a
538.imagick_r	556 ± 1	259	+0.2%	+0.5%	n/a	n/a	n/a	n/a
519.lbm_r	274 ± 3	1	+1.4%	+0.1%	−0.2%	+2.0%	n/a	−2.5%
511.povray_r	540 ± 4	170	+27.3%	+27.4%	+11.3%	+90%	n/a	+37%
510.parest_r	421 ± 3	427	+19.9%	+3.1%	n/a	n/a	n/a	n/a
508.namd_r	282 ± 5	8	+1.2%	+1.4%	−0.3%	+3.0%	n/a	−2%
Geo.Mean	400	57	+4.4%	+2.1%	+4.0%	+20%	+45%	+8.5%

The outcomes indicate that FastDIM introduces an average overhead of 2.1% when safeguarding only function pointers and 4.4% when protecting all designated targets across the 15 SPEC CPU 2017 benchmarks, with the highest overhead being 32.6% on 520.omnetpp_r. It's acknowledged that variances in overhead among different programs are expected, as seen with cutting-edge solutions, influenced by the quantity and utilization of protected memory objects within the program. For an equitable evaluation, the geometric mean is utilized to compare FastDIM with other methods: virtual-table verification (VTV) at 9.6%, per-input control-flow integrity (πCFI) at 3.3%, modular control flow integrity (MCFI) [19] at 2.9%, practical context-sensitive CFI (PathArmor) at 3.0%, dynamic control-flow integrity method (Lockdown) at 20%, CCFI [9] at 45%, ROPecker [22] at 2.6%, bin-CFI [18] at 8.5%, path-sensitive variation of CFI (PITTYPAT) [40] at 12.7%, DFI [12] at 200%, key property based DFI (KPDFI) [34] at 9.53%, against FastDIM (ours) at 4.4%, and FastDIM with function pointer (FP) only at 2.1%. Note that the reported performance was cited from the survey [52] with SEPC CPU

2006. We did not compare our approach with hardware-based solutions.

Academic example (Q1&Q2) For unit tests, we use an example in the work [53] to demonstrate the capability of FastDIM of protecting user-annotated sensitive data. As shown in Figure 6, the program takes a username and a text input from the user, greets the user with the greeter function, initializes a key, and encrypts the text with the key. This program has a format-string vulnerability in the greeter function, which could allow an attacker to take over the program and modify the key. The global key is the sensitive data, so we marked it using a LLVM attribute. By applying FastDIM, we successfully protect the integrity of the encryption keys.

```
1  char __attribute__((annotate("sensitive"))) *key;
2  char *ciphertext;
3  unsigned int i;
4
5  void greeter (char *str) {
6      printf(str); printf(", welcome!\n");
7  }
8
9  void initkey (int sz) {
10     key = (char *) (malloc (sz));
11     // init the key randomly; code omitted
12     for (i=0; i<sz; i++) key[i]= ...;
13 }
14
15 void encrypt (char *plaintext, int sz) {
16     ciphertext = (char *) (malloc (sz));
17     for (i=0; i<sz; i++)
18     ciphertext[i]=plaintext[i] ^ key[i];
19 }
20
21 void main (){
22     char username[20], text[1024];
23     printf("Enter username: ");
24     scanf("%19s",username);
25     greeter(username);
26     printf("Enter plaintext: ");
27     scanf("%1023s",text);
28     initkey(strlen(text));
29     encrypt(text, strlen(text));
30 }
```

Figure 6 Demonstration of user annotation to prevent data-oriented attacks.

RIPE benchmark (Q2) The RIPE benchmark [54] encompasses various vulnerable points, such as function pointers and return addresses located in the stack, heap, .bss, and .data segments, alongside numerous attack scenarios. We applied FastDIM to the RIPE benchmark, a comprehensive C program designed to simulate diverse attack methodologies through buffer overflows in different memory segments (stack, heap, .bss, and .data segments). By default, RIPE is compiled for 32-bit systems using the -m32 flag. Our testing environment was an x86_32 Ubuntu 16.04 virtual machine. The benchmark explores 3840 attack permutations, with 83 initially successful, 767 unsuccessful, and 2990 deemed non-feasible. Under FastDIM's shield, all previously successful attacks were effectively intercepted.

2. Performance on real-world programs

TORQUE resource manager (Q2) Terascale open-source resource and queue manager (TORQUE) [55] serves as a distributed resource manager, orchestrating batch jobs and compute nodes within high-performance computing clusters. It was identified that TORQUE resource manager versions 2.5.x to 2.5.13 are susceptible to a stack-based buffer overflow vulnerability, as highlighted in CVE-2014-8729 and CVE-2014-8787. In our evaluation, we aimed to ascertain whether FastDIM could thwart attacks leveraging these vulnerabilities. We also employed two security enhancements from Clang/LLVM, using the compile options "-fsanitize=safe-stack" and "-fsanitize=cfi". However, simply compiling TORQUE with these options led to executable failures, with crashes attributed to illegal instructions triggered by CFI checks—possibly due to CFI misidentifying legitimate function pointer calls as indirect call breaches, causing the program counter to jump to an invalid operation. Conversely, FastDIM did not encounter this issue, and the TORQUE executables modified by FastDIM were capable of executing tasks such as job submission, queuing, dispatching, and deletion without any hitches. To verify FastDIM's effectiveness, we executed an overflow attack using a Python script (test_overflow.py) that dispatches a transmission control protocol (TCP) packet with a 148-byte payload, inducing a buffer overflow in the TORQUE server (pbs_server) that crashes the application. FastDIM successfully identified and countered this assault.

Null HTTPd (Q2) Null HTTPd [56] is a Linux-based multi-threaded web server that was found to have a remotely exploitable heap overflow vulnerability. This vulnerability arises when an attacker supplies a negative length to the server, influencing the allocated size for the read buffer and causing a heap overflow. This flaw permits an attacker to overwrite memory locations arbitrarily via the free() function, as outlined in CVE-2002-1496 [56]. The risk involves corrupting the CGI-BIN configuration viariable stored in memory, which stores the directory path of executable programs processed during HTTP request handling. Consequently, by altering this configuration string, an attacker gains the capability to execute arbitrary code surreptitiously.

In the *main.h* file of the program, we marked the string variable CONFIG config as sensitive by using the _attribute_((annotate("sensitive"))) annotation. After integrating FastDIM to secure this program, the data attack targeting this sensitive information was effectively thwarted.

Apache HTTP server (Q3) Our prototype was further tested on Apache HTTP server version 2.4.27, utilizing the integrated ApacheBench (ab) tool for assessment. This evaluation took place on an x86 Ubuntu 14.04.5 virtual machine equipped with 4 cores and 4 GB of RAM, where we initiated the Apache server using the subsequent command line:

apachectl -f /local-path/conf/httpd.conf

On the host system, equipped with a Xeon(R) CPU

E5-1620 and 16 GB of RAM, we executed the benchmark using this command line:

$$ab - n5000 - c1000http : //127.0.0.0 : 80/$$

This command dispatches 5000 HTTP GET requests to the Apache server on the virtual machine, handling as many as 1000 requests simultaneously. The processing time for each request averaged around 128 ms, with an overall data transfer rate of approximately 1798 KB/s. Table 2 presents a comparative analysis across ten iterations. On average, FastDIM with function pointer-only protection and FastDIM with full protection resulted in an overhead of 12%–24% and 12%–39%, respectively, on the Apache HTTP server.

Table 2 Performance outcomes on Apache HTTP server

Method	Average connection time (ms)	Total time (s)	Longest connection time (ms)	Transfer rate (kpbs)
Original FastDIM	66.69	0.84	808.69	1798.43
FastDIM with function pointer-only protection	82.77	0.95	908.45	1497.15
FastDIM with full protection	92.96	0.97	936.86	1488.97

3. Performance on overall system

In order to evaluate the performance of FastDIM on protecting the overall system with concurrent programs, we ported our work to harden the Linux/Android kernel 3.1 and a legacy Android (v4.3) on ARMv7 platform. This version of Android has several memory corruption vulnerabilities, such as CVE-2014-3100 in the KeyStore service (Q2), in which a stack buffer is created by the function KeyStore::getKeyForNamelocated located in system/security/keystore/keystore.cpp. In this function, the filename array is allocated on the stack and the input parameter keyName is copied into this array by calling a function. However, that function does not verify the size of the input parameter keyName, allowing attackers to execute arbitrary code and consequently obtain sensitive key information or bypass intended restrictions on cryptographic operations. Our experiment showed that FastDIM was able to detect the modification of the return address caused by this vulnerability.

Overall, the security-hardened system image is 2.5% larger than the original one. The binder mechanism is part of the Android kernel and serves as the major interprocess communication mechanism for Android. FastDIM increased the binder latency by 1.4% on average (Q3). Lastly, AnTuTu is a well-known benchmark tool for mobile platforms widely used to evaluate the overall system performance across different hardware platforms. On average, the overhead of the security-hardened kernel protected by FastDIM is around 3% (Q3).

VI. Conclusion

This paper introduces, develops, and assesses FastDIM, a system devised to shield user applications from control-related and data-centric attacks. To minimize runtime overhead, we introduce optimization strategies such as OCRAM-based shared memory and hoisting. FastDIM was rigorously tested through comprehensive experiments, demonstrating its accuracy, effectiveness, and efficiency. Our methodology advances towards a deterministic security defense, aiming for complete immunity against memory corruption attacks.

References

[1] J. Li, Z. Wang, X. Jiang, *et al.*, "Defeating return-oriented rootkits with "*Return-Less*" kernels," in *Proceedings of the 5th European Conference on Computer Systems*, Paris, France, pp. 195–208, 2010.

[2] K. Z. Snow, F. Monrose, L. Davi, *et al.*, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, pp. 574–588, 2013.

[3] Microsoft, "Data execution prevention (DEP)," Available at: https://cyberpedia.reasonlabs.com/EN/data%20execution%20prevention%20(dep).html, 2023-09-15.

[4] Pax Team, "Pax address space layout randomization (ASLR)," Available at: http://pax.grsecurity.net/docs/aslr.txt, 2004-09-10.

[5] S. Bhatkar and R. Sekar, "Data space randomization," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Paris, France, pp. 1–22, 2008.

[6] T. Jackson, A. Homescu, S. Crane, *et al.*, "Diversifying the software stack using randomized NOP insertion," in *Moving Target Defense II*, S. Jajodia, A. K. Ghosh, V. S. Subrahmanian, *et al.* , Eds. Springer, New York, NY, USA, pp. 151–173, 2013.

[7] D. Bigelow, T. Hobson, R. Rudd, *et al.*, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22nd SIGSAC Conference on Computer and Communications Security*, Denver, CO, USA, pp. 268–279, 2015.

[8] M. Abadi, M. Budiu, Ú. Erlingsson, *et al.*, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, article no. 4, 2009.

[9] A. J. Mashtizadeh, A. Bittau, D. Boneh, *et al.*, "CCFI: Cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, Denver, CO, USA, pp. 941–951, 2015.

[10] V. Kuznetzov, L. Szekeres, M. Payer, *et al.*, "Code-pointer integrity," in *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, P. Larsen and A. R. Sadeghi, Eds. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, pp. 81–116, 2018.

[11] S. Chen, J. Xu, E. C. Sezer, *et al.*, "Non-control-data attacks are realistic threats," in *Proceedings of the 14th Conference on USENIX Security Symposium*, Baltimore, MD, USA, pp. 177–191, 2005.

[12] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, USA, pp. 147–160, 2006.

[13] H. Hu, Z. L. Chua, S. Adrian, *et al.*, "Automatic generation

of Data-Oriented exploits," in *Proceedings of the 24th USENIX Conference on Security Symposium*, Washington, D. C. , USA, pp. 177–192, 2015.

[14] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, USA, pp. 317–331, 2010.

[15] Microsoft, "Control flow guard," Available at: https://answers.microsoft.com/en-us/windows/forum/all/control-flow-guard-cfg-not-enabled-in-c/a7f8d6fd-34ac-4a2c-a118-c2e636de428b, 2025-06-07.

[16] C. Zhang, T. Wei, Z. F. Chen, *et al.*, "Practical control flow integrity and randomization for binary executables," in *Proceedings of IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, pp. 559–573, 2013.

[17] V. Mohan, P. Larsen, S. Brunthaler, *et al.*, "Opaque control-flow integrity," in *Proceedings of the 22nd Network and Distributed System Security Symposium*, San Diego, CA, USA, pp. 27–30, 2015.

[18] M. W. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proceedings of the 22nd USENIX Conference on Security*, Washington, D. C. , USA, pp. 337–352, 2013.

[19] B. Niu and G. Tan, "Modular control-flow integrity," *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Edinburgh,UK, pp. 577–587, 2014.

[20] Y. B. Xia, Y. T. Liu, H. B. Chen, *et al.*, "CFIMon: Detecting violation of control flow integrity using performance counters," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks*, Boston, MA, USA, pp. 1–12, 2012.

[21] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proceedings of the 22nd USENIX Conference on Security*, Washington, D. C. , USA, pp. 447–462, 2013.

[22] Y. Q. Cheng, Z. W. Zhou, M. Yu, *et al.*, "ROPecker: A generic and practical approach for defending against ROP attack," in *Proceedings of the 21st Network and Distributed System Security Symposium*, San Diego, CA, USA, 2014.

[23] V. van der Veen, D. Andriesse, E. Göktaş, *et al.*, "Practical context-sensitive CFI," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, Denver, CO, USA, pp. 927–940, 2015.

[24] J. F. Li, L. W. Chen, G. Shi, *et al.*, "ABCFI: Fast and lightweight fine-grained hardware-assisted control-flow integrity," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3165–3176, 2020.

[25] A. J. Gaidis, J. Moreira, K. Sun, *et al.*, "FineIBT: Fine-grain control-flow enforcement with indirect branch tracking," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, Hong Kong, China, pp. 527–546, 2023.

[26] R. T. Gollapudi, G. Yuksek, D. Demicco, *et al.*, "Control flow and pointer integrity enforcement in a secure tagged architecture," in *Proceedings of IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, pp. 2974–2989, 2023.

[27] C. R. She, J. F. Li, L. W. Chen, *et al.*, "SCFI: Efficient forward fine-grained control flow integrity based on coarse-grained ISA extensions," *Computers & Security*, vol. 140, no. C, artilce no. 103800, 2024.

[28] M. Ammar, A. Abdelraoof, and S. Vlasceanu, "On bridging the gap between control flow integrity and attestation schemes," in *Proceedings of the 33rd USENIX Conference on Security Symposium*, Philadelphia, PA, USA, pp. 6633–6650, 2024.

[29] J. Pincus and B. Baker, "Beyond stack smashing: Recent advances in exploiting buffer overruns," *IEEE Security & Privacy*, vol. 2, no. 4, pp. 20–27, 2004.

[30] N. Carlini, A. Barresi, M. Payer, *et al.*, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proceedings of the 24th USENIX Conference on Security Symposium*, Washington, D. C. , USA, pp. 161–176, 2015.

[31] H. Hu, S. Shinde, S. Adrian, *et al.*, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Proceedings of IEEE Symposium on Security and Privacy*, San Jose, CA, USA, pp. 969–986, 2016.

[32] T. Liu, G. Shi, L. W. Chen, *et al.*, "TMDFI: Tagged memory assisted for fine-grained data-flow integrity towards embedded systems against software exploitation," in *Proceedings of 2018 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/ 12th IEEE International Conference on Big Data Science and Engineering*, New York, NY, USA, pp. 545–550, 2018.

[33] L. Feng, J. Y. Huang, L. Y. Li, *et al.*, "RvDfi: A RISC-V architecture with security enforcement by high performance complete data-flow integrity," *IEEE Transactions on Computers*, vol. 71, no. 10, pp. 2499–2512, 2022.

[34] X. F. Nie, L. W. Chen, H. L. Wei, *et al.*, "KPDFI: Efficient data flow integrity based on key property against data corruption attack," *Computers & Security*, vol. 128, article no. 103183, 2023.

[35] J. X. Peng, Y. Wang, J. F. Xue, *et al.*, "Fast Cross-Platform Binary Code Similarity Detection Framework Based on CFGs Taking Advantage of NLP and Inductive GNN," *Chinese Journal of Electronics*, vol. 33, no. 1, pp. 128–138, 2024.

[36] M. Ismail, C. Jelesnianski, Y. Jang, *et al.*, "Enforcing C/C++ type and scope at runtime for control-flow and data-flow integrity," in *Proceedings of 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, La Jolla, CA, USA, pp. 283–300, 2024.

[37] S. Nagarakatte, J. Z. Zhao, M. M. Martin, *et al.*, "SoftBound: Highly compatible and complete spatial memory safety for c," *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[38] S. Nagarakatte, J. Z. Zhao, M. M. K. Martin, *et al.*, "CETS: Compiler enforced temporal safety for C," in *Proceedings of International Symposium on Memory Management*, Toronto, ON, Canada, pp. 31–40, 2010.

[39] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, OR, USA, pp. 128–139, 2002.

[40] R. Ding, C. X. Qian, C. Y. Song, *et al.*, "Efficient protection of path-sensitive control security," in *Proceedings of the 26th USENIX Conference on Security Symposium*, Vancouver, BC, Canada, pp. 131–148, 2017.

[41] J. A. Halderman, S. D. Schoen, N. Heninger, *et al.*, "Lest we remember: Cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[42] A. Huang, "Keeping secrets in hardware: The microsoft Xbox^{TM} case study," in *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, Redwood Shores, CA, USA, pp. 213–227, 2002.

[43] M. G. Kuhn, "Cipher instruction search attack on the bus-encryption security microcontroller DS5002FP," *IEEE Transactions on Computers*, vol. 47, no. 10, pp. 1153–1157, 1998.

[44] D. Gruss, C. Maurice, K. Wagner, *et al.*, "Flush+Flush: A fast and stealthy cache attack," in *Proceedings of the 13th International Conference on Detection of Intrusions and*

Malware, and Vulnerability Assessment, San Sebastián, Spain, pp. 279–299, 2016.

[45] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Topics in Cryptology-CT-RSA 2006*, D. Pointcheval, Ed. Springer, Berlin Heidelberg, Germany, pp. 1–20, 2006.

[46] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proceedings of the 24th USENIX Conference on Security Symposium*, Washington, D. C. , USA, pp. 897–912, 2015.

[47] K. Shang, W. Z. He, and S. Zhang, "Review on security defense technology research in edge computing environment," *Chinese Journal of Electronics*, vol. 33, no. 1, pp. 1–18, 2024.

[48] E. Göktaş, A. Economopoulos, R. Gawlik, *et al.*, "Bypassing clang's SafeStack for fun and profit," in *Proceedings of Black Hat Europe, Black Hat Europe*, London, UK, artilce no. 21, 2016.

[49] C. Y. Song, B. Lee, K. J. Lu, *et al.*, "Enforcing kernel security invariants with data flow integrity," in *Proceedings of the 23rd Network and Distributed System Security Symposium*, San Diego, CA, USA, 2016.

[50] J. W. Huang, H. Han, F. Y. Xu, *et al.*, "SAPPX: Securing COTS binaries with automatic program partitioning for intel SGX," in *Proceedings of 34th IEEE International Symposium on Software Reliability Engineering*, Florence, Italy, pp. 148–159, 2023.

[51] JohnHenning, "Spec cpu®2017 overview/what's new?" Available at: https://ssg.spec.org/cpu2017/docs/overview.html, 2024-09-03.

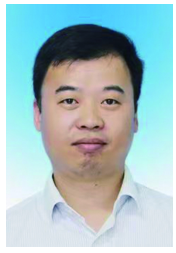
[52] N. Burow, S. A. Carr, J. Nash, *et al.*, "Control-Flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, article no. 16, 2017.

[53] S. Liu, G. Tan, and T. Jaeger, "PtrSplit: Supporting general pointers in automatic program partitioning," in *Proceedings of 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas, TX, USA, pp. 2359–2371, 2017.

[54] J. Wilander, N. Nikiforakis, Y. Younan, *et al.*, "RIPE: Runtime intrusion prevention evaluator," in *Proceedings of the 27th Annual Computer Security Applications Conference*, Orlando, FL, USA, pp. 41–50, 2011.

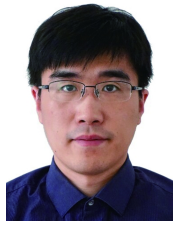
[55] Adaptive Computing Inc., "Torque resource manager," Available at: https://adaptivecomputing.com/cherry-services/torque-resource-manager/, 2025-06-07.

[56] CVE, "Learn more at National Vulnerability Database (NVD)," Available at: https://www.cve.org/CVERecord?id=CVE-2002-1496, 2025-06-07.



Jian Huang is currently pursuing a Ph.D. degree in the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China. His research interests include cyber security in industrial control systems.
(Email: ellison.huang@nuaa.edu.cn)

Yanbo Li is currently pursuing a M.S. degree at the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China. His research interests include software security.
(Email: faxcaelestisccc@gmail.com)



Hao Han received the B.S. degree in computer science and technology from Nanjing University, Nanjing, China, in 2005, and the Ph.D. degree in computer science from the College of William and Mary, Williamsburg, USA, in 2014. He is currently a Professor with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China. His research interests include system and software security against various types of threats.
(Email: hhan@nuaa.edu.cn)