HyperSpace: Data-Value Integrity for Securing Software

Jinwoo Yom

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

April 28, 2020

Blacksburg, Virginia

Keywords: Data Value Integrity, Value Invariant, Security Policy

HyperSpace: Data-Value Integrity for Securing Software

Jinwoo Yom

(ABSTRACT)

Most modern software attacks are rooted in memory corruption vulnerabilities. They redirect security-sensitive data values (*e.g.*, return address, function pointer, and heap metadata) to an unintended value. Current state-of-the-art policies, such as Data-Flow Integrity (DFI) and Control-Flow Integrity (CFI), are effective but often struggle to balance precision, generality, and overhead.

In this thesis, we propose *Data-Value Integrity (DVI)*, a new defense policy that enforces the integrity of "data value" for security-sensitive control and non-control data. DVI breaks an essential step of memory corruption based attacks by asserting the compromised security-sensitive data value. To show the efficacy of DVI, we present HyperSpace, a prototype that enforces DVI to provide four representative security mechanisms. These include Code Pointer Separation (DVI-CPS) and Code Pointer Integrity (DVI-CPI) based on HyperSpace. We evaluate HyperSpace with SPEC CPU2006 and real-world servers. We also test HyperSpace against memory corruption based attacks, including three real-world exploits and six attacks that bypass existing defenses. Our evaluation shows that HyperSpace successfully detects all attacks and introduces low runtime performance and memory overhead: 0.9% and 6.2% performance overhead for DVI-CPS and DVI-CPI, respectively, and overall approximately 15% memory overhead.

HyperSpace: Data-Value Integrity for Securing Software

Jinwoo Yom

(GENERAL AUDIENCE ABSTRACT)

Many modern attacks originate from memory corruption vulnerabilities. These attacks, such as buffer overflow, allow an adversary to compromise a system by executing arbitrary code or escalating their access privilege for malicious actions. Unfortunately, this is due to today's common programming languages such as C/C++ being especially prone to memory corruption. These languages build the foundation of our software stack thus, many applications such as web browsers and database servers that are written using these vulnerable programming languages inherit these shortcomings. There have been numerous security mechanisms that are widely adopted to address this issue but they all fall short in providing complete memory security. Since then, security researchers have proposed various solutions to mitigate these ever-growing shortcomings of memory safety techniques. Nonetheless, these defense techniques are either too narrow-scoped, incur high runtime overhead, or require significant additional hardware resources. This results in them being unscalable for bigger applications or requiring it to be used in combination with other techniques to provide a stronger security guarantee. This thesis presents *Data Value Integrity (DVI)*, a new defense policy that enforces the integrity of "data value" for sensitive C/C++ data which includes, function pointers, virtual function table pointers, and inline heap metadata. DVI can offer wide-scoped security while being able to scale, making it a versatile and elegant solution to address various memory corruption vulnerabilities. This thesis also introduces HyperSpace, a prototype that enforces DVI. The evaluation shows that HyperSpace performs better than state-of-the-art defense mechanisms while having less performance and memory overhead and also providing stronger and more general security guarantees.

Dedication

To my parents and grandparents.

For their endless love and sacrifice.

Acknowledgments

First and foremost, I am thankful for my amazing research advisor Dr.Changwoo Min, a professor in The Bradley Department of Electrical and Computer Engineering at Virginia Tech, for his continuous guidance on my Master's thesis. His patience, kindness, and motivation helped guide me through my research and this thesis. Dr.Min always kept his door open and never hesitated to answer my questions regarding my research at any time of the week. I could not have done this without his mentorship.

I thank Dr. David Raymond, Dr. Joseph G. Tront, Prof. Randy Marchany, and Dr. Haining Wang for their service as my thesis committee members and also for their valuable review and feedback on my thesis.

A very special gratitude goes out to our remote research collaborator Dr. Yeongjin Jang from Oregon State University. His expertise in the field of computer security was an incredible help towards the completion of my thesis.

I am grateful for my security teammates in Computer Systems, Memory, and Security (COSMOSS) Lab. They made valuable contributions to my research and supported me in my research endeavors.

Thank you to the rest of the COSMOSS lab members for all of the coffee, tea, snacks and laughter along the way.

Last but not least, I thank my parents, grandparents, sister, extended family members, friends, and my lovely parrots for providing me with support and encouragement. This would not have been possible without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The foundation of most software stacks is written in unsafe languages such as C/C++. This jeopardizes not only the security of programs written in those languages but also the security of programs written in modern type-safe languages as they often utilize libraries written in unsafe languages. This problem affects our common applications to be prone to memory corruption vulnerabilities.

Most memory attacks modify the intended value of security-sensitive data. For example, control data attacks exploit buffer overflows to overwrite code pointers. In most cases, these targeted code pointers are return addresses [10, 11, 75, 78], function pointers [14, 21, 30, 36], or virtual function table pointers in C++ [72, 95]. Meanwhile, non-control data attacks aim to overwrite other security-sensitive data, such as heap metadata [6, 27, 28, 65, 76, 77, 93] and security credentials. Furthermore, advanced control and non-control data attacks, such as return-oriented programming (ROP) [75] and data-oriented programming (DOP) [43], respectively, are powerful as they can construct arbitrary Turing-complete computations.

Memory-corruption Defense Landscape. In response, many defenses have been proposed to thwart memory corruption-based attacks, however, they suffer from high runtime overhead, can only protect control data, or are imprecise thus susceptible to attacks. Full memory safety enforcement, such as [60, 73, 97], prevents all memory corruption as they enforce spatial and temporal memory safety. However, these approaches fall short in practicality due to their high runtime performance and memory overhead. For example, a state-of-the-art system BOGO [97] has 60% runtime overhead and 36% memory overhead. Control-flow integrity (CFI) [1, 12, 17, 24, 34, 39, 40, 44, 50, 54, 63, 64, 69, 71, 82, 86, 88, 89, 94, 96] provides control data protection by guaranteeing the integrity of expected control flow based on the program's control-flow graph (CFG). However, not only CFI fails to defend non-control data attacks, but it also struggles to balance between precision and runtime overhead. Numerous CFI proposals suffer from over-conservative precision in the form of large equivalence classes (EC) [49], which are a set of indistinguishable code targets for each indirect transfer. In this case, CFI cannot accurately detect illegally bent control transfer within a given EC [14, 30, 72]. Recent work [44] attempts to address this inherent problem by enforcing a unique code target (UCT) property (*i.e.*, EC = 1). However, this technique requires background threads to process Intel PT packets, which consumes additional hardware resources (*e.g.*, dedicating an analysis CPU core) and scarifies scalability and ability to be widely adopted. Code-pointer integrity (CPI) [53] protects *all* code pointers in a program. Similar to CFI, CPI defends against control data attacks but neglects non-control data attacks. CPI protects code pointers via isolation relying on information hiding, however, attacks against information hiding [29, 37, 66] can break its security guarantee. Furthermore, CPI has high memory overhead (105% on average) to keep track of metadata for all sensitive pointers. Data-flow integrity (DFI) [15, 79, 80] prevents both control and non-control data. DFI ensures that the data-flow at runtime does not deviate from a statically computed data-flow graph (DFG). DFI is a generic defense with broad coverage but suffers from high runtime overhead due to frequent instrumentation of *all* load and store instructions. It has an average performance overhead of 104% and memory overhead of 50%.

Data Value Integrity. Our goal is to protect software against both control and non-control data attacks with low performance and memory overhead. To this end, we propose *Data-Value Integrity (DVI)*, a new defense policy that enforces the integrity of *data value* for both control and non-control security-sensitive data (*e.g.*, function pointers for an indirect call, virtual function table pointers in C++ objects, and heap metadata). The key idea of DVI is to prevent the software from accepting maliciously altered security-sensitive data via memory corruption attacks. DVI achieves this by having a secure copy of such security-sensitive data, the copy which is immutable to memory corruption attacks. DVI detects an attack by checking if the values of the data and its secure copy mismatches. By coincidence, if the compromised value happens to be the same as the original value, DVI will allow the program to continue because the attack will not affect program execution. However, if the values mismatch, DVI will raise a security exception. In this regard, DVI mitigates both control and non-control data corruption by breaking an essential step of the attack chain (*i.e.*, subverting security-sensitive data to unintended values). DVI differentiates itself from *flow-integrity approaches*, such as CFI and DFI, that struggles with a fundamental trade-off between precision and runtime overhead. In flow-integrity, it is inevitable to improve the precision of flow tracking without sacrificing runtime overhead. Instead, DVI is a new design that detaches from such trade-offs in flow-integrity approaches and provides a *generic and efficient defense.*

HyperSpace. We present our prototype, HyperSpace, a defense mechanism that enforces DVI to guarantee a wide-scope of protection while maintaining low runtime overhead and minimal additional hardware resources. HyperSpace mirrors values of security-sensitive data into a safe memory region and validates values before use to enforce value integrity. HyperSpace manages memory status to enforce spatial and temporal safety of security-sensitive data. The safe memory region is protected by Intel Memory Protection Keys (MPK) [45, 51, 70], a per-thread memory protection mechanism, and is efficiently accessible using hardware segmentation.

To demonstrate the versatility of DVI, HyperSpace enforces DVI to implement four state-of-the-art security mechanisms where the first three mechanisms protect various types of control data while fourth addresses non-control data, respectively: 1) code pointer separation (DVI-CPS), 2) virtual function table pointer protection for C++ objects (DVI-VTPtr), 3) code pointer integrity (DVI-CPI), and 4) inline heap metadata protection. To avoid manual annotation, HyperSpace includes an LLVM compilation pass that automatically instruments DVI mechanisms to source code.

We further extended HyperSpace with six optimization techniques aimed to lower HyperSpace's runtime overhead via reducing: the cost of each DVI instrumentation, the number of security-sensitive data to protect, and the number of costly permission changes of the safe memory region. Additionally, they also optimize the access to safe memory region to be as efficient as possible.

We protect several benchmarks and real-world applications using HyperSpace to evaluate its efficiency and effectiveness. These include all C/C++ SPEC CPU2006 programs, NGINX web server, and PostgreSQL database server. Furthermore, we test HyperSpace against three real-world exploits and six synthesized attacks that include: virtual function pointer table hijacking attacks, a COOP attack [72], and a heap exploit. We detail how these attacks are successfully detected and blocked by HyperSpace as it kills the process when an attempt for a corrupted sensitive data usage is detected. HyperSpace incurs a small performance and memory overhead; even when programs are armored with DVI-CPI, which guarantees the *full* integrity of all code pointers, DVI on average incurs 6.35% performance overhead (median: 0.67%) and 15.6% memory overhead (median: 6%). Therefore, data-value integrity enforcement can be used to protect software from both control and non-control data attacks efficiently and effectively.

To summerize, our contributions include:

- We propose a new defense policy, Data-Value Integrity (DVI), which mitigates both control and non-control data attacks.
- We built a full prototype of the defense mechanism that enforces DVI, HyperSpace, and implemented four state-of-the-art security mechanisms to demonstrate how DVI and HyperSpace can be used to provide securities to various well-known security sensitive data. We also introduce six optimization techniques to reduce the overhead of HyperSpace instrumentation.
- We evaluate HyperSpace and its security mechanisms on benchmarks, real applications, and synthesized attacks. Our results show HyperSpace can protect control and non-control data with 6.35% performance overhead and 15.6% memory overhead.

Chapter 2

Background

It is well known that the root cause of many modern attacks originates from memory corruption vulnerabilities. These attacks, such as buffer overflow, allow an adversary to compromise our systems by executing arbitrary code or escalating their access privilege for malicious actions. Common programming languages like C/C++ are especially susceptible to these memory corruption attacks due to programmers being responsible for handling memory management. Unfortunately, these languages are often used to build the foundation of our software stacks. Thus, many applications such as internet browsers that are written using these vulnerable programming languages inherit these shortcomings, making them vulnerable to countless variations of memory corruption attacks as well.

To address this issue, a security mechanism called Data Execution Prevention (DEP) [47, 57] was widely adopted to prevent the adversary from injecting malicious code [2, 3, 48] into the program address space to be executed. To bypass this, attackers took control of the code pointers and executes a chain of existing code fragments called ”gadgets”. This type of gadget chaining attack is a form of code reuse attack called Return Oriented Programming (ROP) [10, 75, 78]. To thwart ROP attacks from locating such gadgets, another security mechanism called Address Space Layout Randomization (ASLR) [7, 8, 84] was introduced to randomize the code layout during the load-time a program. However, due to having poor granularity of only randomizing base address of the program, ASLR can easily be bypassed by disclosing a single pointer and re-adjusting the ROP gadgets based on the calculated

ASLR randomized offset.

At this point, the security research branched off mainly into two directions. One is code hiding, and the other is guaranteeing code integrity. Although both methods aim to protect the same sensitive data such as code pointers from being abused by adversary their approaches are drastically different from one another.

Code Hiding. The main goal of code hiding is to hinder attackers from successfully disclosing the location of code pointers. To prevent single pointer disclosure attacks from succeeding ROP, techniques such as fine-grained randomization approaches were introduced. These techniques extend ASLR by randomizing code layout at finer granularity such as function, or basic-block level making it harder for adversaries to re-adjust their ROP gadget based on single pointer disclosure [5, 22, 42, 52, 68].

To get around fine-grained randomization defense mechanisms, indirect code disclosure was introduced. Attacks such as Just-in-Time ROP (JIT-ROP) [78] use indirect code disclosure to read the application code after it has been randomized to build an ROP attack that is unique to target the newly randomized layout during run-time. To combat this attack, researchers disabled read permission to code pages since programs do not need to read their own code in most cases. This defense technique was coined as Execute-only Memory (XoM) [19, 81]. Nevertheless, attackers found that there remain code pointers inside of readable memory, such as return addresses, that point into the execute-only memory. This is called indirect code disclosure and it allows attackers to, yet again, carry out their ROP attack.

This takes us to the current state of code hiding research which is largely sub-divided into two techniques. The first technique is called continuous re-randomization [9, 16, 33, 55, 91] where code layout is continuously randomized during runtime to expire the attacker's knowledge on code layout. Although this technique can successfully defend against current state-of-the-art memory disclosure attack, it suffers from having high overhead, high resource usage, and being time-sensitive. The second technique is called code-pointer hiding [19, 46, 55, 90, 91] where they internally encrypt and decrypt code pointers to prevent attackers from gaining any knowledge of program layout from code-pointers. Much like continuous re-randomization techniques, code-pointer hiding has drawbacks such as having high-overhead.

Code Integrity. Code Integrity aims to guarantee the integrity of the code by preventing illegal usage of program execution that deviates the program from its expected behavior. Currently, there exists four major categories of code integrity: Control-Flow Integrity (CFI) [1, 17, 34, 39, 40, 44, 50, 54, 63, 64, 69, 71, 82, 86, 88, 89, 94, 96], Data-Flow Integrity (DFI) [15, 79, 80], Code-Pointer Integrity (CPI) [53], and Object-Type Integrity (OTI) [13].

CFI is a defense mechanism that addresses control-flow hijacking attacks. It relies on the static analysis of Control-Flow Graph (CFG) to generate a set of possible code targets called Equivalence Class (EC) [49] for each indirect transfers such as call, jump, and return instructions. These ECs are used during runtime of the program to constraint control-flow transfers of basic blocks to only those that exist within its EC. However, CFI struggles against mimicry attacks that ”bend” the control flow within an EC targets to carry out control-flow hijacking attacks. Additionally, it also does not protect against data leaks or data corruption attacks.

Similar to CFI, DFI ensures that the data-flow from the runtime of the program does not deviate from its statically generated data-flow graph. To do this, it first assigns every write instruction a unique ID. Then, for every load instruction, it checks for the ID to be within the allowed set generated during compilation. Due to the requirement of every load and store instructions being instrumented, DFI imposes a very high overhead, making it undesirable for wide adoption.

CPI enforces memory safety of control data, such as code pointers, by isolating them in a separate safe region protected via memory hiding. It uses these isolated code pointers to provide runtime code pointer verification during every control-flow transfers. Although CFI provides an efficient solution for providing memory safety, it does not protect against non-control data corruption. Furthermore, it has a low entropy for hiding safe regions which, once discovered, can easily be corrupted [29, 37, 66].

OTI focuses on securing virtual function table pointers in C++ applications from being corrupted. The virtual function table pointer (VTPtr) is used for C++'s dynamic dispatch for polymorphism. VTPtr exists in writable memory, but is only assigned once inside of the object's constructor. Thus, attackers could corrupt this pointer to modify the object's type or point to a malicious object. Due to its narrow scope, it relies on being used with CFI for the rest of the memory safety.

2.1 State-Of-the-Art in Memory Corruption

Below are summaries of various types of state-of-the-art attack techniques as well as some of their recent variations that relate to memory corruption. These attacks take advantage of memory corruption to formulate an exploit that allows the adversary to skew a program to behave unexpectedly.

2.1.1 Related Attacks

Return Oriented Programming (ROP). ROP attacks were originally derived from a Linux code reuse exploit technique known as Return-to-Libc [75]. ROP utilizes a memory corruption exploitation concept to bypass DEP by chaining together a series of existing code gadgets that end in corruptible sensitive code transfers such as code pointers and return addresses. Due to various ROP mitigation techniques, ROP attacks have evolved into many varieties of forms that target specific loopholes in those techniques. Few of the well-known ROP variants are Just-In-Time ROP (JIT-ROP) [78] and Blind ROP (BROP) [10]. JIT-ROP can map out the application's runtime memory layout by continuously abusing its memory disclosure vulnerability to bypass static load-time randomization. Once disclosed, the memory layout can be used to dynamically discover gadgets and are used to JIT-compile a target program to a serialized payload that is usable by the exploit scripts. BROP attacks synthesize ROP exploits without the target's binary. It does this by bypassing ASLR using a technique known as stack reading. In stack reading, it aims to disclose return addresses and stack canaries by continuously abusing buffer overflow vulnerabilities. Using this information, BROP targets the write system call to dump the target's binary which is then used to synthesize the traditional ROP payload.

Spatial Memory Safety Exploit. Insuring a dereference to be within valid object bounds is the main idea behind spatial memory safety. Attack techniques such as buffer overflows are a typical example of an attack that violates the spatial memory. To mitigate, defense mechanisms such as stack canary [18, 56] enforce object bounds checking to guarantee the spatial property.

Temporal Memory Safety Exploit. Valid memory dereferences are the focus of temporal memory safety property. Attacks such as use-after-free attempts to violate this property by dereferencing dangling pointers. Object location-based technique is one of the ways to thwart temporal safety exploit by having an auxiliary data structure to keep track of valid allocated/deallocated status of each object's memory location [59].

Control/Non-Control Data Attack. Control and non-control data attacks are broad categorization of various memory corruption attacks. Control data attacks involve attacks that corrupts data, such as code pointers, to change the behavior of the program control flow. ROP is one of many attacks that are categorized as a control data attack. Non-Control attacks corrupt any other data that are not control data such as heap metadata to change program's behavior.

Counterfeit Object-oriented Programming (COOP). COOP attack [72] takes advantage of the C++ semantics of virtual functions to construct ROP attacks. It defines two important components: vfgadgets and counterfeit objects. Virtual functions used to carry out the attack are vfgadgets. Whereas, counterfeit objects are maliciously injected objects in a attack-controlled memory location. To carry out the attack, it first finds the counterfeit objects and vfgadgets. Afterward, it is essential to locate a main loop gadget where a virtual function iterates over an array of C++ object pointers that invokes virtual functions. The main loop is manipulated to continuously loop over itself to provide a platform for orchestrating the rest of the attack. Finally, for each iteration of a corrupted array that carries counterfeit objects, various combinations of vfgadgets are used to carry out a payload of the exploit.

2.1.2 Relevant Memory Protection Features

Stack Cookies/Stack Canary. Stack canary [18, 56] is a security mechanism used to protect return pointers from being corrupted via buffer overflow attacks. When stack canary is enabled, a randomized integer value called ”canary value” is chosen at the start of a program. This canary value is placed just before the return pointer to provide a way to check for corruption detection before each return instructions. The main concept is that, if buffer overflow was to corrupt the return pointer, the canary value that is placed in between the buffer and return address will be corrupted therefore triggering an invalid canary value. However, attacks such as BROP that can perform memory disclosure by repeatedly abusing buffer overflow vulnerabilities can bypass stack canaries.

Address Space Layout Randomization (ASLR). ASLR [7, 8, 84] is a security technique that is widely adopted to deny an attackers knowledge of the code layout. To achieve this, it randomizes the arrangement of the base address of executable, stack, heap, and libraries once during load-time. Although this technique increases entropy to memory layout, it has some limitations that allow attackers to bypass its security. Attack techniques such as Just-In-Time ROP [78] that uses memory disclosure vulnerabilities to read the code layout of a program during runtime. Not only this mechanism lacks in randomization granularity but also, further randomization during runtime is required for better security guarantees.

Data Execution Prevention (DEP)/NX-bit (No-eXecute). DEP [47, 57] is a security feature that utilizes an additional inverted permission bit called nx-bit for supporting execute permission of memory page. It is used to prevent the execution of machine instructions for memory space used for regular data. Unfortunately, this mitigation technique can be easily be bypassed using return-to-libc and other variations of return-oriented programming attacks.

Memory Protection Extension (MPX). Intel MPX [67] is an extension to the x86 architecture and introduces four new 128-bit bounds registers (BND0-BND3). Each register contains a pair of 64-bit lower and upper bounds (LB and UB) values of a buffer for checking pointer references. MPX is intended to provide efficient protection against memory attacks such as buffer overflows and out-of-bounds access. It was not widely adopted due to the overhead of bounds checking.

Memory Protection Keys (MPK). MPK [45, 51, 70] is Intel's new hardware primitive that utilizes previously unused four bits in each page table to assign sixteen independent permission key values to any given memory page. Thus, using this feature, a process can partition it's memory up to sixteen regions and independently assign memory access permissions to those regions. Additionally, since the control register is local to each thread, each thread's permission can be unique from another. Thus, MPK allows thread-local permission control on groups of pages without modifying the page tables.

Chapter 3

Data Value Integrity

In this chapter, we first present the *value invariant property*, which is our common observation on security-sensitive data. We then present how the value invariant property holds for virtual function table pointers in C++. Lastly, we introduce *data-value integrity (DVI)*.

3.1 Value Invariant Property

Data-Value Integrity (DVI) is a new security policy that protects security-sensitive data, such as function pointers and variables that store security credentials. DVI achieves this by enforcing the integrity guarantee for those *values*, *i.e.*, preventing variables from being overwritten to different values by memory corruption attacks.

DVI is inspired by many previous defense approaches that aim to protect program's integrity including Data-Flow Integrity (DFI) [15, 79, 80], Control-Flow Integrity (CFI) [1, 17, 34, 39, 40, 44, 50, 54, 63, 64, 69, 71, 82, 86, 88, 89, 94, 96], Code Pointer Integrity (CPI) [53], and Object Type Integrity (OTI) [13]. Prior defense approaches can protect only control data (*i.e.*, code pointers) [1, 13, 17, 34, 39, 40, 44, 50, 53, 54, 63, 64, 69, 71, 82, 86, 88, 89, 94, 96] or suffer from high performance overhead by tracking complex data flow [15, 79, 80].

In contrast to existing approaches, we aim to provide a generic security policy that can be utilized to protect both control and non-control data with low runtime overhead. While

```
// Register a sensitive memory region
// starting at addr with size
void dvi_register(void *addr, int size);
// Unregister a sensitive memory region
void dvi_unregister(void *addr, int size);
// Write the current value in a sensitive memory
// region to the corresponding safe memory region
void dvi_write(void *addr, int size);
// Same as dvi_write() but do not allow further writes
void dvi_write_final(void *addr, int size);
// Check if the sensitive memory value is the same
// as the safe memory value
void dvi_assert(void *addr, int size);
```

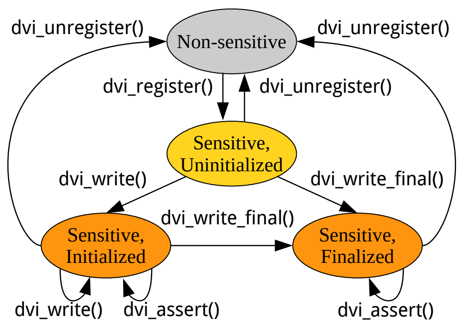

Figure 3.1: **DVI primitives (left) and the state transition diagram (right) for DVI protected memory.** DVI primitives trigger state transitions for a specified memory location. DVI manages the intended value of sensitive data for integrity checking. Mismatching values of sensitive data or an illegal state transition indicates a value integrity violation (dvi_assert).

"generic" and "fast" are often considered an oxymoron, our key insight behind DVI is to enforce *value integrity* instead of tracking *flow integrity* of control or data transfers, which is expensive and complex. Our reasoning for this is simple: even if a program is compromised by attackers, if security-sensitive values (*e.g.*, function pointer) remain the same as the original by coincidence, the attack will not affect program execution. Therefore, such memory corruption attacks can be mitigated by breaking an essential step of their attack chain: subverting security-sensitive data to unintended values. Attackers may attempt to exploit a set of security vulnerabilities, but they cannot achieve arbitrary code execution because code pointers cannot be altered. Privilege escalation also cannot occur because memory blocks storing a program's privilege cannot be altered, etc.

Our key intuition behind DVI originates from a common pattern in programs: the value of security-sensitive data *does not frequently change*, and there exists a period that values should *never be changed*. These two patterns form the basis of our *value invariant property* for security-sensitive data. That is, during the life cycle of an object, it is prevalent that values of security-sensitive data never change after their legitimate assignments.

3.2 VTPtr in a C++ Object

We illustrate our observation of the value invariant property using a virtual function table pointer (VTPtr) in a C++ object.

Virtual Function Table Pointer (VTPtr). The VTPtr pointer is a hidden member variable for dynamic polymorphism used for virtual function calls in C++. Each object type has a virtual function table. When an object is created, a constructor assigns the address of the appropriate virtual function table to its VTPtr variable according to its object type. After initialization, the program can invoke virtual function calls by indexing this table (*e.g.*, this->VTPtr[idx]()).

Attacking VTPtr. Because VTPtr determines which virtual function to call at runtime, the value stored in VTPtr is security-sensitive. Modification to its value, *e.g.*, pointing to a fake table structure (*e.g.*, FakeVTPtr), allows attackers to execute arbitrary code (*e.g.*, system()) at a virtual function call site (*i.e.*, this->FakeVTPtr[idx]()), resulting in a vtable hijacking attack [13, 95] or a COOP attack [72].

VTPtr **Life Cycle.** We observe the following three-phase life cycle of a C++ object with respect to the value of the VTPtr.

1. *Construction:* The value of the VTPtr is assigned when an object is created as each object type has its virtual function table. Once the object type is determined by the constructor, the address of the corresponding virtual function table is determined, and its address is assigned to VTPtr.

2. *In-use:* After construction, the object is now ready for use. The program may invoke virtual functions in the virtual function pointer table via VTPtr.

3. *Destruction:* At the end of an object's life cycle, the object is destructed, making VTPtr

value no longer valid nor legitimate.

Value Invariant Phase of VTPtr**.** VTPtr holds the *value invariant property* throughout in-use phase. Because a C++ object's type will never change during an object's life cycle, neither should its VTPtr value. Thereby, DVI aims to preserve the integrity of the VTPtr during this phase to defeat attacks that attempt to modify it.

3.3 DVI Overview

Here, we give an overview of DVI by demonstrating how to protect VTPtr using DVI.

Overview. DVI protects security-sensitive data from memory corruption attacks by enforcing *value invariant property.* DVI achieves this by making these data immutable (read-only) during their value invariant phase. Therefore, it prevents memory corruption attacks from overwriting the data. To apply DVI to security-sensitive data, identifying their value invariant phase is essential. Hence, DVI requires an analysis of the life cycle of data as we practice in the previous section. After identifying the value invariant phase, DVI asserts the read-only permission to the corresponding memory block that stores protected data.

Fine-grained Memory Permission Control. DVI requires fine-grained permissions control on memory. For instance, to protect VTPtr, we need to make only the 8-byte memory for VTPtr read-only while keeping the rest of memory space storing the object as read-writable. However, commodity hardware cannot support this as current the smallest granularity for memory permissions control is a page, 4KB in size. To this end, DVI relies on a new shadow memory system HyperSpace, a key technique enabling DVI to control memory permissions in 8-byte granularity. In a nutshell, HyperSpace divides a system's memory space in two, a regular and safe region, respectively. The regular region is the general read-writable region, while the safe region is read-only by default and can only be writable via DVI primitives, which we introduce next. We further describe how HyperSpace works and provides fine-grained permissions control with commodity hardware in Chapter 6.

DVI Primitives. Based on the fine-grained memory permissions control backed by HyperSpace, DVI manages the state of a memory location as illustrated in Figure 3.1. When a program starts, all memory is in a *non-sensitive state*, meaning a memory location does not store security-sensitive data. To protect a memory location storing security-sensitive data, DVI first requires the location to be registered upon its allocation (dvi_register). Then, the memory will be in a *sensitive, uninitialized* state. Once the sensitive data is written to the regular memory location, DVI creates a copy of its value in the safe memory region (dvi_write) that DVI manages at runtime. Now, the memory is in a *sensitive, initialized* state. In case that we know a write should be the final one until the deallocation of the memory, we can additionally annotate this (dvi_write_final). This will put the memory into *sensitive, finalized* state and DVI does not allow any further writes to the memory location. VTPtr is a good use-case of this state because it is written only once at object construction and should not be updated until the object's destruction. Before using any sensitive data, the program will check whether its value is corrupted by comparing the value in regular memory with the value in safe memory (dvi_assert). If values do not match or a program attempts to perform an illegal state transition, these anomalies alert the violation of value integrity. Finally, when a sensitive memory location needs to be deallocated, it is unregistered (dvi_unregister) and reverting the memory locations to *non-sensitive state*, allowing it to be reused again in the future.

Applying DVI to VTPtr. To apply DVI, we instrument a program to insert DVI primitives alongside memory access to VTPtr, based on its value invariant life cycle. VTPtr is a case with a clear value invariant phase in the life cycle of a C++ object. First, we instrument

```
1  /** == Example of a control data corruption attack ========== */
2  void X(char *); void Y(char *); void Z(char *);
3
4  typedef void (*FP)(char *);
5  static const FP arr[2] = {&X, &Y};
6
7  void handle_req(int uid, char * input) {
8    FP func; // control data to be corrupted!
9    //   dvi_register(&func, sizeof(func));
10   char buf[20];
11
12   if (uid<0 || uid>1) return; // only allows uid == 0 or 1
13
14   func = arr[uid]; // func pointer assignment, either X or Y.
15   //   dvi_write_final(&func, sizeof(func));
16   strcpy(buf, input); // stack buffer overflow!
17   //   dvi_assert(&func, sizeof(func));
18   (*func)(buf); // func is corrupted!
19   //   dvi_unregister(&func, sizeof(func));
20 }
```

```
21 /** == Example of a non-control data corruption attack ======== *
22 bool authenticate(char *packet);
23
24 void handle_packet(char *input) {
25   int auth = 0; // non-control data to be corrupted!
26   //   dvi_register(&auth, sizeof(auth));
27   //   dvi_write(&auth, sizeof(auth));
28   char buf[1000];
29
30   packet_read(input,buf); // stack buffer overflow!
31   if (authenticate(buf)) {
32     auth = 1;
33     //   dvi_write(&auth, sizeof(auth));
34   }
35   //   dvi_assert(&auth, sizeof(auth));
36   if (auth) { // auth is corrupted!
37     grant_access(buf);
38   }
39   //   dvi_unregister(&auth, sizeof(auth));
40 }
```

Figure 3.2: **Example vulnerable C code.** Attackers can overwrite security-sensitive data by exploiting memory corruption vulnerabilities (strcpy() at Line 16 or packet_read() at Line 30) to subvert control flow (arbitrary code execution at Line 18) or change program behavior (illicitly reach grant_access, Line 37). In Data-Value Integrity (DVI), we directly check if the value of sensitive data is corrupted without tracking control flow or data flow. Sensitive data is first registered (dvi_register) and its value is stored in safe memory (dvi_write, dvi_write_final) upon write. Its integrity is checked before use (dvi_assert). Finally, its memory location is unregistered upon deallocation (dvi_unregister).

dvi_register to register VTPtr inside of a C++ object immediately after object allocation. For the initialization of the VTPtr, there is only one place, in the constructor, that updates the value of the VTPtr. There exist no further legitimate value updates to VTPtr before the destruction of the object. Hence, to protect VTPtr from attacks, right after the constructor assignment of virtual function pointer table address to VTPtr, we invoke dvi_write_final to store the original value into a safe region. Then, the value invariant phase starts. During this phase, read-only permissions are applied to the value of VTPtr in the safe region; this means that no value updates can be made, and thereby, DVI protects VTPtr from any attacks that attempt to overwrite its value. To ensure the integrity of the VTPtr value during this phase, we instrument all uses of VTPtr (*i.e.*, invoking a virtual function call) to call dvi_assert beforehand to compare values in the regular region and its corresponding copy in the safe region. Any mismatch of values is flagged as an attack. Finally, when the object is being destructed, we unregister the memory location from DVI via dvi_unregister to release the integrity protection on the corresponding safe region.

Chapter 4

DVI as a Generic Exploit Mitigation

We further demonstrate how DVI works as a mitigation to memory corruption attacks with two examples–protecting both control and non-control data–in Figure 3.2. Those examples are further extended to provide defense against attacks on control data such as DVI-CPS, DVI-CPI, and virtual function table protection as well as defense against attacks on non-control data, such as protecting inline heap metadata from corruption in Chapter 7. After that, we compare and contrast DVI with existing mitigation such as CFI, CPI, and DFI. We share their security/performance drawbacks and advantages that HyperSpace can leverage by enforcing DVI.

4.1 Protecting Control Data: a Function Pointer

Vulnerability. The left code snippet in Figure 3.2 is an example with a stack buffer overflow vulnerability that allows attackers to alter control data (func at Line 8) via an insecure function call of strcpy() (at Line 16). At Line 14, code assigns the address of either function X or Y to func depending on function argument, uid. Without loss of generality, assume the value of uid at runtime as 0, selecting function X as the call target. Line 18 is supposed to call function X via func. However, attackers may corrupt the value in func by overflowing buf into input (*i.e.*, input size > 20), exploiting a buffer overflow vulnerability to change the value in func from X to some other, Y, Z, or any arbitrary code address (*e.g.*, system() to execute an arbitrary command).

Value Invariant Life Cycle of func. We analyze the life cycle of security-sensitive variable func with respect to its value invariant period as follows.

1. *Assignment:* The first and only assignment to func is on Line 14.
2. *In-use:* After the assignment and before the destruction of the stack, the value of func does not change. Thus, its value invariant phase starts. The phase ends when the variable is destructed.
3. *Destruction:* The stack variable func will become invalid when the function unwinds its stack, *i.e.*, at the function epilogue.

DVI Instrumentation. Our target variable to protect is func; its value invariant phase starts right after assignment to func is done at Line 14. Hence, we instrument Line 14 with dvi_register and dvi_write_final, to register the variable as a sensitive memory block, and write the function address, and lock the memory (Line 9 and 15). Before the program uses the variable, *e.g.*, calling a function via func at Line 18, we need to check if the value stored in func is the same as the value recorded in the safe region. To do this check, we insert dvi_assert on Line 17. When the variable is destructed at function epilogue, we insert dvi_unregister at Line 19 to unregister the variable from protection.

Defeating Attacks with DVI. Overwriting func by exploiting the buffer overflow vulnerability at Line 16 can be mitigated by DVI. Before the program executes Line 16, DVI has already stored the value of func, for example, the address of function X if uid == 0, in the safe region and locked the value with dvi_write_final. Launching an attack may overwrite the value of func in the regular memory region, to some other values such as the address of Y, Z, or an arbitrary function address. However, doing so will raise an alarm when the program uses func because before invoking a function through func, dvi_assert will compare the value in the regular region to that in the safe region, and kill the program process if these values do not match (*e.g.*, X != Y).

4.2 Protecting Non-control Data: Authentication Status

Vulnerability. The code on the right of Figure 3.2 is an example with a stack buffer overflow vulnerability, assuming that receiving more than 1000 bytes in packet_read(input,buf) at Line 30 may overflow the buffer, buf. By exploiting this vulnerability, an attacker may corrupt the auth variable (non-control data at Line 25), placed right next to buf. Without any protection, an attacker can overwrite the value of auth, (*e.g.*, from 0 to 1), to make an unauthenticated session as authenticated one to bypass the check at Line 36.

Value Invariant Life Cycle of auth. There exist two lines that perform assignments to auth, and we can separate its life cycle as the following five-steps.

1. *Assignment #1:* The initial assignment, writing value 0, is made to auth at Line 25.

2. *In-use #1:* Right after assigning initial value 0, the value invariant phase starts. It ends at two locations, Line 32 where new value 1 is assigned after passing authentication check at Line 31, and the function epilogue, when the variable is destructed.

3. *Assignment #2:* After passing an authentication check at Line 31, the program writes value 1 to auth to record the client's authentication status at Line 32.

4. *In-use #2:* After the second assignment, there are no other assignments to auth. The second value invariant phase starts, and ends when the local stack variable is destructed.

5. *Destruction:* The stack variable auth will become invalid when the function destructs its stack, *i.e.*, at the function epilogue.

DVI Instrumentation. Our target variable to protect is auth, and its first value invariant phase starts right after the initial assignment of auth at Line 25. So we instrument that line with dvi_register and dvi_write to register the variable as a sensitive memory block and also write the value 0 to the safe region (Line 26 and 27). There could be an update (*Assignment #2*) during the first value invariant phase. To handle this, we also instrument Line 32 to correctly update the value in the safe region and start the second value invariant phase. Note that across two invariant phases, the value stored in the safe region is kept read-only, and is writable only if being accessed via dvi_write. Before the program uses the variable for checking the authentication status at Line 36, we need to check if the value stored in auth remains the same as the value stored in the safe region. To assert this, we insert dvi_assert at Line 35. When the variable is destructed at the function epilogue, we insert dvi_unregister at Line 39 to unregister the variable from DVI protection.

Defeating Attacks with DVI. The buffer overflow vulnerability at Line 30 is mitigated in the same way as in the previous example in section 4.1. Specifically, dvi_assert will fail on the use of non-control data auth instead of a control data (auth_regular != auth_safe).

4.3 Security Guarantee of DVI

Prerequisites for Security Guarantee. DVI relies on a sound analysis of identifying value invariant phases of security-sensitive data. DVI protection requires a program to be correctly instrumented for all security-sensitive data manipulation. Upon allocation and assignment of such data, the program must place dvi_register and dvi_write accordingly to create a copy in the safe region. To use DVI protected data (*e.g.*, indirect call/jump, reading sensitive data for a branch), the program must check the integrity of values beforehand (*i.e.*, comparing values in the regular and safe regions).

Security Guarantees. DVI guarantees the *value integrity* of security-sensitive data during their value invariant phase. During runtime, the safe memory region is read-only except for legitimate accesses via DVI primitives (*e.g.*, dvi_write or dvi_write_final) to prevent memory corruption attacks in the safe region. For any value discrepancy between the regular and the safe memory region (*i.e.*, the data in regular memory is corrupted) when using the data, DVI raises a security exception in the preceding dvi_assert.

Furthermore, DVI guarantees *spatial memory safety* for security-sensitive data in the safe memory region. This guarantee prevents memory corruption from legitimate uses of DVI primitives (dvi_write and dvi_write_final). In particular, DVI checks the length of data to ensure write does not exceed legitimate bounds.

DVI also guarantees *registration-based temporal memory safety* by keeping track of (un)registering of security sensitive-data via DVI primitives such as dvi_register and dvi_unregister. This can prevent attackers from altering non-sensitive data to confuse the program to refer them as security-sensitive data. In this regard, DVI can prevent the attacks by keeping track of the registration state of security-sensitive data in the safe memory region. Particularly, all security-sensitive data will be registered first (via dvi_register) during allocation and will be deregistered after its life cycle. In contrast, a pointer to a non-sensitive data can point to an deregistered location in the safe memory region. In such a case, DVI detects a deregistered state, and raises an alarm to prevent the attack.

How DVI Mitigates Vulnerability Exploitation. In combination with guaranteeing the memory safety and value invariant property of security-sensitive data, DVI mitigates attacks by cutting off an essential attack step of the exploit. Under the protection of DVI, attackers can still exploit memory corruption vulnerabilities, *e.g.*, a buffer overflow or an arbitrary write vulnerability, to overwrite some data in a program's memory space. However, DVI guarantees that the safe memory region is read-only for all non-DVI primitives access. Thus, the copy of the security-sensitive data in safe memory region under DVI protection, *e.g.*, code pointers, heap metadata, privilege state variables, *etc.*, cannot be overwritten by attackers due to its read-only access permission. Applying read-only permissions blocks essential steps for exploits such as overwriting code pointers to achieve arbitrary code execution, escalating privilege by overwriting privilege status variables, or altering heap metadata to build other exploit primitives.

4.4 Defense Advantages in DVI

We summarize defense advantages of DVI when compared to existing techniques, such as CFI, CPI, and DFI.

vs. CFI. DVI provides better security than CFI in protecting control data. DVI allows only one valid target at each indirect call site, whereas most CFI techniques allow many targets (427 targets for recent work [50]), leaving a wider attack surface for attackers. Additionally, DVI does not require dedicating CPU cores to run background analysis, unlike μCFI [44], and is thereby, more scalable.

vs. CPI. DVI brings better security than CPI [53] by protecting the safe memory region with hardware memory protection (Intel MPK), whereas CPI relies on insecure information hiding [29, 37, 66].

vs. DFI. DVI outperforms DFI on runtime overhead of control and non-control data protection. DVI avoids expensive data-flow analysis while protecting many classes of security-sensitive data.

Chapter 5

Threat Model and Assumptions

DVI and HyperSpace focuses on defending against memory corruption based attacks. Our assumption includes a program that has one or more memory vulnerabilities (*e.g.*, buffer overflow) that allow attackers to read from and write to arbitrary memory. However, the attacker cannot modify or inject code due to Data Execution Prevention (DEP) [47, 57]. The attacker can use arbitrary write capability to perform control data or non-control data attacks. Control data attacks are a subset of data attacks, where attackers exploit memory corruption vulnerabilities to hijack program control flow by overwriting data (*e.g.*, function pointer, virtual function table pointer). Non-control data attacks exploit memory corruption vulnerabilities to overwrite security-critical data (*e.g.*, heap metadata) without hijacking the intended control flow of a program. We assume that all hardware and OS kernel are trusted such that attacks exploiting those vulnerabilities are out of scope.

Chapter 6

HyperSpace Design

The main challenge to realizing DVI is designing a secure and efficient metadata storage mechanism to keep track of the state and value of each sensitive element. In particular, the metadata mechanism cannot be vulnerable to tampering by attackers as well as its access cost and additional memory overhead should be minimal to be practical. To address this problem, we propose HyperSpace, a secure and efficient metadata storage mechanism for DVI. We discuss HyperSpace design details next and how HyperSpace can be leveraged for various security applications in the next chapter.

Parallel Safe Memory Region Layout. To efficiently access the safe memory region, we bisect the virtual address space of a process into a regular memory region and a safe memory region as illustrated in Figure 6.1. When a process is created, HyperSpace kernel bisects the user virtual address space and reserves the upper half of the virtual address space as the safe memory region. Additionally, the %gs register is set to the starting address of the safe memory region. With this parallel memory layout, accessing a safe memory location from a regular memory location is merely adding the original regular memory offset to the start address of the safe memory region; this operation can be encoded with a single instruction in x86 architecture using segmentation (see Figure 6.2). We note that the safe memory region is an anonymous region, managed by the kernel. That is, OS kernel reserves half of the virtual address space, however, a physical page is allocated only on a process's first access to a page in the safe region minimizing runtime memory overhead.

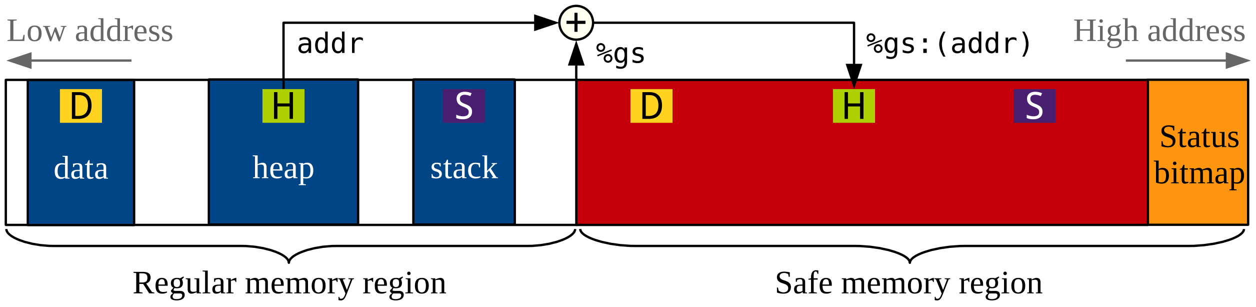

Figure 6.1: **HyperSpace memory layout.** We re-purpose hardware segmentation of x86 architecture for efficient access to HyperSpace. The safe memory region is protected by Intel Memory Protection Keys (MPK).

Protection without Relying on Hiding. The parallel memory layout enables access to the safe memory region to be efficient. However, taking such a large virtual address space makes it infeasible to hide the safe memory region from attackers. Instead, we protect the safe memory region using Intel Memory Protection Keys (MPK), a relatively new hardware feature in x86-64 architecture [45, 51, 70]. By default, the safe memory region is not given write permission (*i.e.*, it is read-only). Only during DVI operations that update the safe memory region (*i.e.*,dvi_register, dvi_unregister, dvi_write, and dvi_write_final), does HyperSpace temporarily grant write permissions (*i.e.*, read-writable) to only the thread executing those DVI operations. The discussion regarding possible misuse of DVI primitives is addressed in Chapter 11. Thus, any write attempt to the safe memory by an unauthorized thread at an unauthorized time will cause a page fault. We use Intel MPK to efficiently change access permissions of the safe memory region for each thread. With MPK, a virtual memory region is assigned to one of the 16 domains under a protection key, which is encoded in a page table entry. Memory access permissions of each domain is independently controlled through an MPK register. Changing memory access permissions is fast as it only takes around 23 CPU cycles using a non-privileged instruction wrpkru. Also, the impact of permission changes is thread-local as the MPK register is per-CPU.

Representing State of Safe Memory. In DVI, each memory location is in one of four states shown in Figure 3.1. HyperSpace manages an additional metadata area at the end of the safe memory regions to represent the state of each memory location. Because HyperSpace manages sensitive data in 8-byte granularity, 2 bits of metadata are assigned for an 8-byte of sensitive data. The state bitmap is updated upon memory state transition. With our state bitmap representation, access to the state is cheap (see Figure 6.2). Also, the maximum memory overhead is bounded to 103.1% of an application's total memory usage in the regular memory region ($T + T * \frac{2\ bits}{64\ bits}$ where T is usage). This is relatively low compared to approaches managing rich metadata (*e.g.*, tag, bounds) such as CPI [53] and SoftBound+CETS [58, 59, 60].

Low Memory Overhead. HyperSpace relies on sparse address space support and lazy loading via the underlying OS for memory management of the safe memory region. Initially, OS kernel reserves the virtual address space without allocating physical memory. When a process accesses the safe memory region, the OS kernel will allocate a physical page for a faulting virtual address. Our evaluation results in subsection 10.3.2 show that additional memory overhead of HyperSpace is marginal even when using 2 MB huge pages to reduce memory access overhead to the safe memory region.

Putting It All Together. With HyperSpace, the design of DVI's API in Figure 3.1 is simple and efficient. Registering/unregistering sensitive data (dvi_register and dvi_unregister) changes the corresponding state bits in the state bitmap. Writing sensitive data (dvi_write and dvi_write_final) copies the sensitive value to the safe memory region and changes state if necessary. HyperSpace temporarily grants write permissions only during running these four DVI operations. DVI checks value integrity by comparing values between regular and safe memory regions (dvi_assert). For all DVI operations, HyperSpace checks if the memory is in a valid state for a given operation. Otherwise, HyperSpace raises a security exception.

```
// Get the safe memory value for a given address
uint64_t dvi_load_safe_memory_8b(void *addr) {
  uint64_t value;
  asm volatile ("mov %%gs:0x0(%[offset]), %[value]"
    :[value] "=r" (value) :[offset] "r" (addr) );
  return value;
}

// Get the first status bit for a given address
uint8_t dvi_get_safe_memory_status_bit0(void *addr) {
  void     *bitmap_addr = (void *)(((uint64_t)addr >> 5) & ~0x3);
  uint64_t bitmap_idx  = ((uint64_t)addr & 0xf8) >> 2;
  uint8_t  bit;
  asm volatile (
    "btq %[bitmap_idx], %%gs:(%[bitmap_addr],%[area_sz])"
    : : [bitmap_idx]  "r" (bitmap_idx),
        [bitmap_addr] "r" (bitmap_addr),
        [area_sz]      "r" (ADDR_SPC_SZ) );
  asm volatile ("setc %[bit]" : [bit] "+rm" (bit) );
  return bit;
}
```

Figure 6.2: **Code for accessing safe memory and its state.**

This enforces memory safety of sensitive data as discussed in section 3.3. With HyperSpace, accessing both the safe memory region and the state bitmap can be efficiently done via %gs segment register as shown in Figure 6.2.

Chapter 7

Security Applications

This chapter demonstrates how DVI can be applied to defeat control and non-control data attacks by enforcing value integrity guarantees. As discussed in section 3.3, registration and deregistration of sensitive data allows DVI to keep track of sensitive data by monitoring their protective status. By saving a copy of sensitive data on valid store instructions, DVI keeps track of legitimately stored value for corruption detection. Most importantly, DVI verifies the integrity of sensitive data before it is used.

Control data protections that HyperSpace implements consist of: Code Pointer Separation (DVI-CPS) (*i.e.*, protecting all code pointers), Code Pointer Isolation (DVI-CPI) (*i.e.*, protecting all sensitive pointers), and virtual function table pointer protection in C++ objects (DVI-VTPtr). We present automatic instrumentation for these three protections. HyperSpace provides coverage of all sensitive global, heap, and stack variables. We use SafeStack [53] to protect return addresses and safe objects–stack objects whose address is not taken–by isolating them from sensitive stack variables that are stored in the regular stack. DVI can be used to protect non-control data as well such as heap metadata. For this, we modified ptmalloc2 [32], which is the default memory allocator in most Linux distributions, manually inlining DVI API into its source code.

7.1 Code Pointer Separation (DVI-CPS)

To guarantee the safety of code pointers, all function pointers must be secured using DVI's register, write, assert and deregister primitives. HyperSpace accomplishes this by accurately identifying and instrumenting all instructions that allocate, write, use, and deallocate function pointers.

This instrumentation feature is a part of the HyperSpace module pass in LLVM. First, we identify function pointers using LLVM type information. Because function pointers can exist inside of structs or arrays, HyperSpace recursively looks through each element of container types as well. For cases where function pointers are recognized as universal pointers (*i.e.*, void* or char*), we look ahead for its typecasting to its *actual* type further down in the program and instrument accordingly.

Registration of function pointers is instrumented immediately after its allocation. Excluding those in the safestack, HyperSpace instruments all heap variables, global variables, and other address-taken function pointers on the regular stack via dvi_register.

To determine when to perform DVI write for function pointers, we look for any unsafe function pointers (*i.e.*, function pointers are not on the safe stack) that are the destination operand of a store instruction. The dvi_write will be instrumented following such store instruction if the variable is not in the safestack.

dvi_assert should be called immediately before using any sensitive data. In the case of function pointers, we look for call and load instructions. Once indirect call instructions are detected, HyperSpace instruments those function pointers that are in the safestack.

For sensitive heap and mmap-ed variables, deregistering is instrumented before free and munmmap calls, respectively. For stack variables, we deregister the entire current stack frame from the last to the first registered variable address in a local frame at once to prevent from having too iteratively deregister.

7.2 VTable Protection in C++ (DVI-VTPtr)

In C++, virtual functions are an essential part of dynamic polymorphism. Hijacking the virtual function table pointer of a live object or an already-freed object (*i.e.*, use-after-free) is a commonly exploited attack [13, 72, 95]. HyperSpace can guarantee the C++ object's VTPtr integrity by storing a copy in the safe memory region during initialization.

To protect virtual function table pointers, we need to first correctly identify VTPtr within C++ objects. This can be detected using HyperSpace's type analysis because we are looking for a pointer to an array that contains function pointers. When recursively dereferenced from all proceeding pointer types, our analysis can identify the code pointers and mark the VTPtr as a sensitive data. The registration of VTPtr is instrumented along with the rest of regular sensitive type registrations during object allocation; no extra registration semantics changes were needed for this support. To guarantee that the VTPtr of an object will never change, HyperSpace instruments the dvi_write_final call right after VTPtr is assigned by the object's constructor. This ensures that the object's VTPtr does not get modified outside of its constructor. A slight extension is made in the pass to recognize when an object's VTPtr is loaded; HyperSpace instruments dvi_assert primitive immediately before the load instruction to guarantee that the VTPtr has not been tampered with. Similar to registration, no changes are necessary for deregistration. The same deregistration semantics as in DVI-CPS is used to deregister the VTPtr along with other sensitive values the object may contain.

7.3 Code Pointer Integrity (DVI-CPI)

Building on Code Pointer Separation (CPS), HyperSpace can be extended to guarantee the integrity of all relevant sensitive pointers. To achieve this, HyperSpace recursively protects all sensitive code pointers and pointers of sensitive objects which are of sensitive type as defined in CPI [53].

In order to detect the additional sensitive pointers required for CPI, the type analysis described in the CPS section is extended to include more cases. Composite type objects that contain a function pointer are recognized as a sensitive type. Hence, pointers to these sensitive types are protected and composite types that contain these pointers are also protected creating a recursive chain of protection.

After detecting protection sets for all of the sensitive types in the LLVM module pass, its instrumentation is similar to CPS. HyperSpace finds and instruments all IR instructions that declare, modify, and use sensitive pointers. When a sensitive variable is declared, HyperSpace looks up its protection sets from the type analysis result and instruments dvi_register accordingly. No changes are made for write instrumentation as HyperSpace simply instruments all the locations where sensitive variables are modified as explained in the CPS instrumentation. When sensitive variables are being used, recursive analysis backtracks to instrument any containing variable that is loaded from a pointer. The process recursively runs to find all load instructions in the path for the primary sensitive variable that is used.

7.4 Heap Metadata Protection

The heap memory allocator is essential in building an efficient and secure program. Ptmalloc2 [32] is one of the most widely adopted heap allocators. Ptmalloc2 and many other heap allocators (*e.g.*, dlmalloc [26] and tcmalloc [38]) adopt inline metadata design. This 32-byte metadata includes previous chunk size, current chunk size and flags, and forward and backward pointers for binlist, 8-byte each. Unfortunately, this inline metadata design suffers a major security flaw. By exploiting heap-based buffer overflow vulnerabilities, an adversary can compromise inlined metadata to perform arbitrary code execution. Several security mechanisms were proposed in an attempt to address this issue, however, they are still able to be bypassed [27, 28, 93].

We manually instrument the ptmalloc2 source code to protect this inlined metadata for each memory chunk. We register 32-byte metadata whenever a new memory chunk is created (*e.g.*, splitting one large chunk into two smaller chunks) and deregister 32-byte metadata whenever a memory chunk is deleted (*e.g.*, merging two small chunks into one big chunk) using DVI APIs (dvi_register, dvi_unregister). For each malloc and free, we first check whether inline metadata is corrupted using dvi_assert. After updating metadata, we keep the newly written metadata to the safe region using dvi_write. This approach protects inline heap metadata against state-of-the-art corruption attacks such as poisoned NULL byte, 1-byte NULL overflow [28], and unsafe unlink [23] by asserting metadata during malloc and free to detect corruption.

Chapter 8

Optimizations

In this chapter, we present various optimization techniques applied to reduce HyperSpace overhead of automatic instrumentation pass and to reduce memory access overhead of safe memory. Early analysis of HyperSpace showed that no single optimization significantly improved performance across all components. Thus, we implemented six major optimization techniques to make HyperSpace versatile and efficient compared to the current state-of-the-art.

8.1 Inlining DVI Functions (INLN)

Due to the overhead of setting up stack frames, the function call overhead of HyperSpace APIs could be significant. To minimize instrumentation overhead and eliminate function call overhead, our instrumentation pass inlines HyperSpace API calls. Furthermore, we designed HyperSpace's API calls specifically for handling and protecting 8-byte data. This is because most sensitive data needing protections are usually various pointer types (VTPtr, function pointer, sensitive object pointer, etc). These 8-byte optimized APIs are inlined using LLVM's Link Time Optimization (LTO).

8.2 Excluding Objects in Safe Stack (SS)

As discussed in Chapter 7, we use the Safestack [83] to protect return addresses and safe objects that are address-not-taken stack objects. The Safestack isolates safe objects from all sensitive stack objects that are on the regular stack. Hence HyperSpace does not need to instrument any objects on the safe stack. This helps to reduce performance overhead especially when a program frequently uses temporary stack variables that belong to sensitive types.

8.3 Runtime Checks to Reduce Permission Changes (RNT)

DVI does not rely on information hiding. Instead, DVI utilizes Intel MPK to control safe memory permissions. In most cases, modifying permissions using MPK is fast. However, it could incur significant overhead if an application requires frequent permission changes. From our observations, reading the current MPK permissions using rdpkru is cheap (∼0.5 CPU cycles) whereas, changing permissions using wrpkru is expensive (∼23.3 CPU cycles). With this, the primary optimization opportunity to reduce usage of wrpkru was within dvi_write, one of the most frequently used DVI functions. For dvi_write, we check if the target safe memory is already in a sensitive, initialized state as well as if the value being written is the same as its safe copy. If so, the write operation is not necessary. This prevents unnecessary writes and MPK permission changes in many applications, where we observed frequent updates of sensitive data with the same value (*e.g.*, 453.povray as described in subsection 10.3.1). Therefore, HyperSpace will only utilize wrpkru in the first dvi_write. Any dvi_write afterward that writes the same data value will be ignored as no update is necessary

```
1   /** == Instrumentation of consecutive writes of sensitive data ==
2    *  - LISTOP is a sensitive type containing a function pointer.
3    *  Thus, its two members, op_last and op_sibling, pointing to
4    *  other LISTOP instances are sensitive data. */
5   OP *Perl_append_list(pTHX_ I32 type, LISTOP *first, LISTOP *last){
6       // ...
7       first->op_last->op_sibling = last->op_first;
8       // dvi_safe_memory_unlock();
9       //   dvi_write(&first->op_last->op_sibling, 8);
10      // dvi_safe_memory_lock();
11      first->op_last = last->op_last;
12      // dvi_safe_memory_unlock();
13      //   dvi_write(&first->op_last, 8);
14      // dvi_safe_memory_lock();
15      first->op_flags |= (last->op_flags & OPf_KIDS);
16      FreeOp(last);
17      return (OP*)first;
18  }
19
20  /** == Coalescing permission changes in a basic block ======== */
21  OP *Perl_append_list(pTHX_ I32 type, LISTOP *first, LISTOP *last){
22      // ...
23      first->op_last->op_sibling = last->op_first;
24      // dvi_safe_memory_unlock();
25      //   dvi_write(&first->op_last->op_sibling, 8);
26      first->op_last = last->op_last;
27      //   dvi_write(&first->op_last, 8);
28      first->op_flags |= (last->op_flags & OPf_KIDS);
29      // dvi_safe_memory_lock();
30      FreeOp(last);
31      return (OP*)first;
32  }
```

Figure 8.1: **An example of before (top) and after (bottom) basic block level coalescing optimization for permission changes in** 400.perlbench **to reduce runtime overhead (Lines 24-29).**

for the safe memory. Overall, this reduces the number of unnecessary wrpkru calls.

8.4 Coalescing Permission Changes within a Basic Block (CBB)

To further optimize unnecessary toggling of safe memory region permissions, we introduce an optimization technique to coalesce a series of HyperSpace protection instrumentation (*i.e.*, dvi_safe_memory_lock and dvi_safe_memory_unlock) within a basic block. Figure 8.1

shows an example instrumented function from 400.perlbench in SPEC CPU2006. The original instrumentation (top), shows the modification of a sensitive object's linked list (Lines 7 and 11), which requires a brief opening of the safe memory.

However, repetitively unlocking and locking is unnecessary if DVI API calls are consecutive in a basic block. In this case, there is neither control flow change nor store instructions capable of corrupting arbitrary memory locations. Therefore, it is safe to place locking instrumentation after the very last DVI API call. To further reduce permissions change overhead, we extend this intuition and introduce a *coalescing-safe basic block*, where we can safely move locking instrumentation to immediately before a terminator instruction of the basic block. All memory writes in a coalescing-safe basic block are guaranteed not to be capable of corrupting arbitrary locations. More specifically, a coalescing-safe basic block contains only store instructions whose destination memory locations point to either the sensitive type which is protected by DVI, the non-sensitive field of a sensitive type of which address is bounded by the sensitive type, and the safe stack. Consequently, the safe memory region can safely remain unlocked until the end of the basic block. Looking at the optimized instrumentation in Figure 8.1 (bottom), intermediary permission changes are removed and a single lock function is placed at the end of its basic block (Line 29). Also, note that we have inlined dvi_register, dvi_write and dvi_unregister to check current permissions of the safe memory region using rdpkru. Thus, this optimization greatly reduces redundant safety guards.

8.5 Coalescing Permission Changes within a Safe Function (CFN)

As we further optimize the number of permission toggles, HyperSpace extends the basic block coalescing optimization. The key idea is for HyperSpace protections in *coalescing-safe functions* to be further coalesced at a *function level* instead of within a basic block. A function is considered to be *coalescing-safe* if it meets three conditions: 1) all basic blocks are coalescing-safe, 2) it does not contain any indirect calls, and 3) all direct call targets are coalescing-safe functions. In other words, all store instructions in the function and all callee functions are guaranteed to not write to arbitrary memory locations. Therefore, unlocking and locking instrumentation can be safely placed at function entry and exit, respectively.

Figure 8.2 shows an example of two variations (with and without this optimization) of the same coalescing-safe function from 400.perlbench. There are four separate basic blocks that each instrument dvi_write for the sensitive linked list pointer value (op_next). The top shows DVI instrumentation before this optimization where each basic block with dvi_write is also fitted with unlocking/locking safe memory. The bottom shows the same function with the optimization enabled, such that all unlocking/locking of safe memory in each basic block is removed and instead a single pair of unlock and lock is placed at function entry and exit (Lines 37,54). This allows HyperSpace to always protect the safe region around indirect calls while avoiding unnecessary locks and unlocks around direct calls.

8.6 Optimizing Safe Memory Access (HGP)

Due to maintaining dual memory regions, HyperSpace experiences more frequent page faults and higher TLB pressure leading to higher overhead in accessing memory. To optimize safe memory access, we utilize huge pages provided by the OS kernel. Compared to the default 4 KB page size, the huge page configuration uses 2 MB pages for the safe memory region to reduce the number of page faults and TLB misses making safe memory access more efficient.

```
 1  /** == Instrumentation of writing sensitive data ================
 2   *  - OP is a sensitive type containing a function pointer.
 3   *  Thus, its member, op__next, pointing to another OP
 4   *  is also sensitive data, which needs to be protected. */
 5  OP * Perl__linklist(pTHX__ OP *o) {
 6    register OP *kid;
 7    // ...
 8    if (cUNOPo->op__first) {
 9      o->op__next = LINKLIST(cUNOPo->op__first);
10      // dvi__safe__memory__unlock();
11      // dvi__write(&o->op__next, 8);
12      // dvi__safe__memory__lock();
13      for (kid = cUNOPo->op__first; kid; kid = kid->op__sibling) {
14        if (kid->op__sibling) {
15          kid->op__next = LINKLIST(kid->op__sibling);
16          // dvi__safe__memory__unlock();
17          // dvi__write(&kid->op__next, 8);
18          // dvi__safe__memory__lock();
19        } else {
20          kid->op__next = o;
21          // dvi__safe__memory__unlock();
22          // dvi__write(&kid->op__next, 8);
23          // dvi__safe__memory__lock();
24    } } }
25    else {
26      o->op__next = o;
27      // dvi__safe__memory__unlock();
28      // dvi__write(&o->op__next, 8);
29      // dvi__safe__memory__lock();
30    }
31    return o->op__next;
32  }
33
34  /** == Coalescing permission changes in a safe function ======= */
35  OP * Perl__linklist(pTHX__ OP *o) {
36    register OP *kid;
37    // dvi__safe__memory__unlock();
38    // ...
39    if (cUNOPo->op__first) {
40      o->op__next = LINKLIST(cUNOPo->op__first);
41      // dvi__write(&o->op__next, 8);
42      for (kid = cUNOPo->op__first; kid; kid = kid->op__sibling) {
43        if (kid->op__sibling) {
44          kid->op__next = LINKLIST(kid->op__sibling);
45          // dvi__write(&kid->op__next, 8);
46        } else {
47          kid->op__next = o;
48          // dvi__write(&kid->op__next, 8);
49    } } }
50    else {
51      o->op__next = o;
52      // dvi__write(&o->op__next, 8);
53    }
54    // dvi__safe__memory__lock();
55    return o->op__next;
56  }
```

Figure 8.2: **An example of before (top) and after (bottom) function level coalescing of permission changes in** 400.perlbench **to reduce runtime overhead (Lines 37 and 54).**

Chapter 9

Implementation

Our HyperSpace prototype is built using Linux kernel 5.0.0 for the x86-64 platform. Code instrumentation is done via module passes on LLVM 9.0.0. Table 9.1 shows the lines of code used to implement HyperSpace each component.

The DVI Library contains and implements all DVI primitives used in automatic instrumentation to secure sensitive data. This includes the APIs shown in Figure 3.1.

The kernel was modified to initialize the virtual address space of a user process of HyperSpace-instrumented executables by splitting into regular and safe memory regions. A few variables were altered to reposition and resize runtime components such as the stack and heap in order to reserve memory space for the safe region. After a safe region is reserved, the segmentation register (%gs) is initialized to be used for fast access to the safe region during runtime. To maintain compatibility with vanilla binaries, none of the kernel changes will apply for non-instrumented executables. This allows non-instrumented executables to run as it would in vanilla Kernel.

HyperSpace's custom LLVM module pass is largely divided into two phases. The first phase includes static analysis of all object types to create a look-up table for lists of sensitive elements within various objects. This look-up table is used in the next phase, the automatic instrumentation. As described in Chapter 7, instrumention phase adds DVI primitives according to where they are needed in program code.

Module	**Lines of Code**		
	Added	**Deleted**	**Total**
DVI Library	505	0	505
Linux Kernel	362	16	378
LLVM	2487	29	2516
ptmalloc2	902	0	902
Total	4256	45	4301

Table 9.1: **Summary of lines of code for HyperSpace components.**

Lastly, we manually instrumented the ptmalloc library to secure heap metadata using DVI primitives. We instrumented malloc() and free() where heap metadata is managed for allocated and freed memory chunks.

Chapter 10

Evaluation

We first evaluate how effectively HyperSpace can prevent real-world attacks by enforcing DVI (section 10.1). Next, we evaluate the efficiency of HyperSpace applications described in Chapter 7 using SPEC CPU 2006 and two real-world applications (section 10.2). Finally, we analyze the impact of our optimization techniques (section 10.3) as well as the memory overhead of HyperSpace.

All applications were run on a 24-core (48-hardware threads) server equipped with two Intel Xeon Silver 4116 processors (2.10 GHz) and 128GB DRAM. This server is running Fedora 28 Server Edition and Linux Kernel v5.0. All benchmarks were compiled with LLVM Safestack [83]. Additionally, GNU gold v2.29.1-23.fc28 is used for linking to enable LLVM LTO.

10.1 Security Experiments

We evaluated all security applications described in Chapter 7, with three real-world exploits and six synthesized attacks.

10.1.1 Real-World Exploits

We first collected three publicly available exploits.

CVE-2016-10190. This is a heap-based buffer overflow in ffmpeg, a popular multimedia framework for encoding and decoding audio and video. This exploit allows remote web servers to execute arbitrary code by overwriting function pointers in an AVIOContext object. DVI-CPS/CPI successfully detects the exploit [61] and halts its execution by asserting the corruption of a function pointer in a victim AVIOContext object.

CVE-2015-8668. This is a heap-based buffer overflow in libtiff, an image file format library. This exploit allows remote attackers to execute arbitrary code. A malicious BMP file causes integer overflow followed by heap overflow and overwriting a function pointer in a TIFF structure. DVI-CPS/CPI successfully detects the exploit [25] by asserting the corrupted function pointer before use.

CVE-2014-1912. This is a buffer overflow in python2.7 caused by a missing buffer size check. An attacker can overwrite a function pointer in PyTypeObject via a crafted string and can execute arbitrary code. DVI-CPS/CPI blocks the exploit [74] by detecting the corruption of the function pointer before use.

10.1.2 Synthesized Exploits

We used synthesized exploits to demonstrate how HyperSpace can defend VTPtr hijacking in C++ objects, COOP attacks [72]–a Turing complete attack via creating fake C++ objects– and heap exploits.

CFIXX C++ Test Suite. We used a C++ test suite [62] released by Burow *et al.* [13]. It provides four VTPtr hijacking exploits (FakeVT, FakeVT-sig, VTxchg, VTxchg-hier), and one COOP exploit. Essentially, the VTPtr hijacking exploits overwrite a VTPtr in a C++ object. In order to make the test suite more similar to real-world memory corruption based attacks, we modified the test suite to corrupt a VTPtr using a heap-based overflow instead of directly overwriting it using memcpy. Our modification is inspired by a synthesized exploit in OS-CFI [50]. DVI-VTPtr detects all four exploits by checking if a VTPtr is corrupted before allowing a call to a virtual function of a given object. The COOP attack creates a fake object without calling the class' constructor and calls a virtual function of the fake object. DVI-VTPtr prevents this exploit by detecting that the VTPtr of the fake object is not initialized as a sensitive data and raises an exception, halting the program before the virtual function call.

Heap Exploit. To evaluate heap metadata protection, we used an exploit from [23], which overwrites inline metadata of an allocated heap memory. HyperSpace thwarts this by detecting the inline heap metadata corruption upon free of a victim memory chunk.

10.2 Performance Evaluation

We evaluate the performance overhead of HyperSpace security mechanisms described in Chapter 7 using SPEC CPU2006 and two real-world applications: NGINX (v1.14.2) and PostgreSQL (REL_12_0). SPEC CPU2006 has realistic compute-intensive applications that are ideal to see the worst-case overhead of HyperSpace. We choose SPEC CPU2006 over SPEC CPU2017 to easily compare HyperSpace with prior work. Figure 10.1 shows performance overhead compared to the unprotected original baseline running on the original kernel.

10.2.1 Performance Overhead of SPEC CPU2006

Heap Metadata Protection. The performance overhead for heap metadata protection

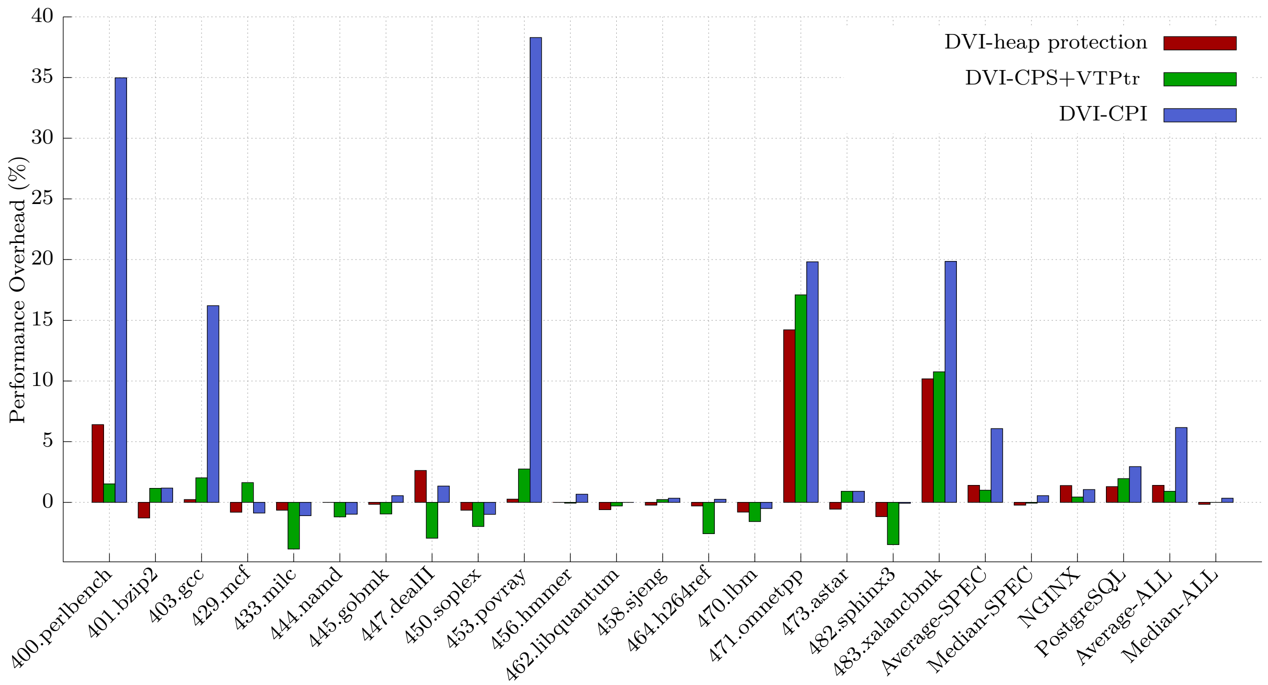

Figure 10.1: **The performance overhead of SPEC CPU2006, NGINX web server, and PostgreSQL database server relative to an unprotected baseline build.** Our three DVI protections are: heap metadata protection, CPS code pointer and C++ VTPtr protection, and CPI protecting all sensitive pointers. HyperSpace imposes negligible performance overhead of 0.9% and 6.2% for DVI-CPS+VTPtr and DVI-CPI, respectively.

with DVI is 1.40% overall as Figure 10.1 shows. Three benchmarks that have more than 5% overhead. These three benchmarks heavily call malloc and free. For example, 471.omnetpp calls malloc and free over 534 million times combined. It continuously opens and closes files during its setup process. Glibc internally allocates heap memory for file open and close so it causes frequent permission changes of the safe region via MPK. In addition to omnetpp, perlbench, dealII, and xalancbmk call malloc and free millions of times as well during their execution, which explains this overhead. This is consistent with results in previous work [20].

DVI-CPS+VTPtr. We then evaluate CPS and C++ VTPtr protection, which together protect *all code pointers* of a program by enforcing DVI. The performance overhead is negligible, 1.02%. A few benchmarks show small performance improvement (1-2%) because the safe stack improves the locality of safe objects by moving large arrays to the regular stack.

In the worst case, only two C++ benchmarks, 471.omnetpp and 483.xalancbmk, exceed 3% overhead. In these two benchmarks, the use of virtual function calls were more frequent compared to other C++ benchmarks, resulting in higher overhead from protecting the integrity of the VTPtr.

DVI-CPI. In addition to CPS, CPI overhead includes instrumentation for recursive sensitive types. HyperSpace's CPI protection performs very well with an average overhead of 6.35%. Two benchmarks, 400.perlbench and 453.povray are exceptions where both exceeds 20%. 400.perlbench accumulates overhead from frequently utilizing sensitive global variables that contain function pointers. For example, a perlbench function, Perl_runops_standard, contains a while loop, where the loop condition contains sensitive indirect call, followed by the return variable being assigned to a sensitive global variable. This causes repetitive permission changes of the safe memory region and collects undesirable, but unavoidable overhead. As for 453.povray, most overheads are from assertions of function pointer in struct Method_Struct. This struct mimics C++'s virtual function table by containing a series of function pointers. Other 453.povray objects use this struct to call function pointers abundantly throughout the runtime. HyperSpace protection recursively extends to pointers of objects that contain struct Method_Struct. These chains of pointers require DVI instrumentation throughout the benchmark resulting in unusual overhead. Recent works, ERIM [87] and IMIX [31], also attempted to utilize MPK for protecting the metadata store of CPI. However, due to lack of proper optimization techniques, they incur significant runtime overhead than DVI-CPI: maximum overhead is 3.2× for ERIM and 28.5× for IMIX, respectively. Moreover, they do not reveal their runtime overhead for 400.perlbench and 453.povray, which are most likely their two highest overhead similar to DVI-CPI.

Summary. The performance overhead of HyperSpace for SPEC CPU2006 is negligible: 1.02% for CPS protection and 6.35% for CPI protection, while the worst-case overhead being

38.3%. In comparison, current state-of-the-art defense techniques Code-Pointer Integrity [53] and μCFI [44], have an average overhead of 8.4% and 7.88% with worst-case overhead of 44% and 49%, respectively. Techniques such as Code-Pointer Integrity require hash table lookup to access metadata for sensitive pointers whereas HyperSpace utilizes a single bit test instruction, btq, using %gs register segmentation. Thus, HyperSpace proves to have better and broader security guarantees while incurring less performance overhead.

10.2.2 Performance Overhead of Real-World Applications

NGINX and PostgreSQL are two widely used web and database servers, respectively. We used the default NGINX configuration, accommodating a max of 1024 connections per processor. Benchmarking is done over a network using a server on the same network switch. Similarly, the default configuration for PostgreSQL was also used with a max of 100 connections and SSL connections disabled.

NGINX. We evaluate the performance of NGINX using an HTTP benchmarking tool wrk [35]. wrk spawns threads that send requests for a 6745-byte static HTML and measures the latency and request throughput (req/sec). We ran wrk with 24 threads with each thread handling 50 HTTP connections. The performance overhead is negligible: heap metadata, CPS and CPI protections impose 1.38%, 0.44% and 1.05% of overhead, respectively.

PostgreSQL. To evaluate the performance of PostgreSQL, we used pgbench [85], which repetitively runs concurrent database sessions that handle a sequence of SQL commands to measure the average transaction rate and latency. We tested PostgreSQL with 24 concurrent database clients. PostgreSQL shows negligible performance impact of 1.30%, 1.96% and 2.04% for heap metadata, CPS, and CPI protections respectively.

10.3 Performance Analysis

We first analyze the impact of our optimization techniques then provide a detailed analysis on memory consumption.

10.3.1 Impact of Performance Optimization

In order to measure the impact of each optimization technique, we turned off one optimization at a time in a fully optimized DVI-CPI. Figure 10.2 shows the impact of each technique for SPEC CPU2006.

+ INLN. Inlining DVI API calls improves performance by 13.7% on average. In particular, 433.milc benefits the most with 60.6% performance improvement due to the frequent use of sensitive stack objects, which need a series of DVI calls.

+ SS. Leveraging SafeStack, HyperSpace does not need to instrument safestack objects. This improves performance by 8.5% on average. The sensitive local variables that are not address-taken safely reside in safestack, separated from the regular stack with variables such as arrays and other address-taken variables that if exploited, could jeopardize the integrity of other variables in the regular stack. SafeStack significantly reduces the number of variables that need to be protected to only those that are in the regular stack. In general, C benchmarks such as 429.mcf (28.4%) and 433.milc (53.5%) benefit from this optimization more than C++ benchmarks since C++ objects are address-taken due to C++ semantics such as constructors, thus do not get included in the safestack.

+ RNT. HyperSpace reduces costly wrpkru instructions by eliminating unnecessary, repetitive modifications of the safe memory region. If an object has already been registered and its value is the same, HyperSpace skips unnecessary, repetitive dvi_register and dvi_write calls

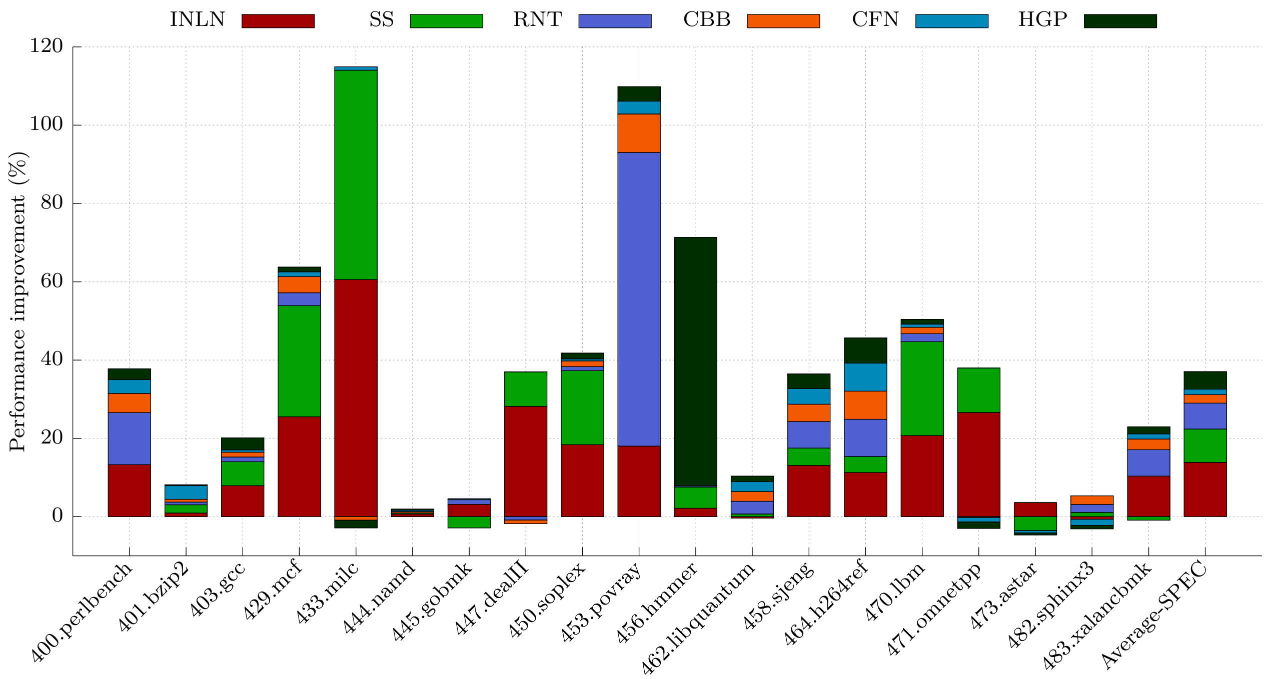

Figure 10.2: **Impact of the performance optimization techniques described in Chapter 8.** (INLN: inlining DVI APIs; SS: safe stack; RNT: runtime permission check; CBB: basic block-level coalescing; CFN: function-level coalescing; HGP: huge page).

to reduce the amount of MPK permission changes. This optimization improves performance by 6.6% on average. In particular, it improves the performance of 453.povray by 75%.

+ **CBB.** Coalescing permission changes within a basic block improves performance 2.17% on average by minimizing the number of permission changes using wrpkru instructions for the safe memory region. 453.povray is one of the most impacted benchmarks with an improvement of 9.8% for having an abundant number of sensitive object pointers that are often updated.

+ **CFN.** Extending coalescing to function scope improves performance 1.4% on average. 464.h264ref, 458.sjeng, and 400.perlbench have higher performance gain of 7.2%, 6.8%, and 3.5%, respectively, due to having commonly used functions such as Perl_linklist recognized as a safe function.

+ **HGP.** Last but not least, using huge pages for the safe memory region improves performance 4.5% on average by reducing the number of page faults and TLB misses. This optimization is effective in the case where sensitive objects are sparsely scattered by accessing larger portions of the safe memory region. In particular, the performance of 456.hmmer improves 63.4% with this change.

10.3.2 Analysis on Memory Consumption

Having a parallel safe region could incur high memory overhead if implemented naively. However, the safe region is an anonymous region that only allocates a physical page if a process writes to the corresponding page in the safe region. Also, its metadata is compact, requiring 2 bits for every 64 bits. We measured the maximum resident set size (RSS) during programming execution. As Figure 10.3 shows, DVI introduces around 15% memory overhead: 14.4% for DVI-CPS+VTPtr and 15.5% for DVI-CPI, respectively. DVI's memory overhead is much smaller than other state-of-the-art defense mechanisms: 105% for the original CPI [53], 50% for DFI [15], and 36% for BOGO [97]. The reason for such memory overhead for state-of-the-art such as CPI is due to having bigger metadata for each sensitive pointers. For CPI, the metadata for each sensitive pointer includes value, upper/lower bounds, and temporal id of the sensitive pointer which require more memory than 2 bits required by HyperSpace.

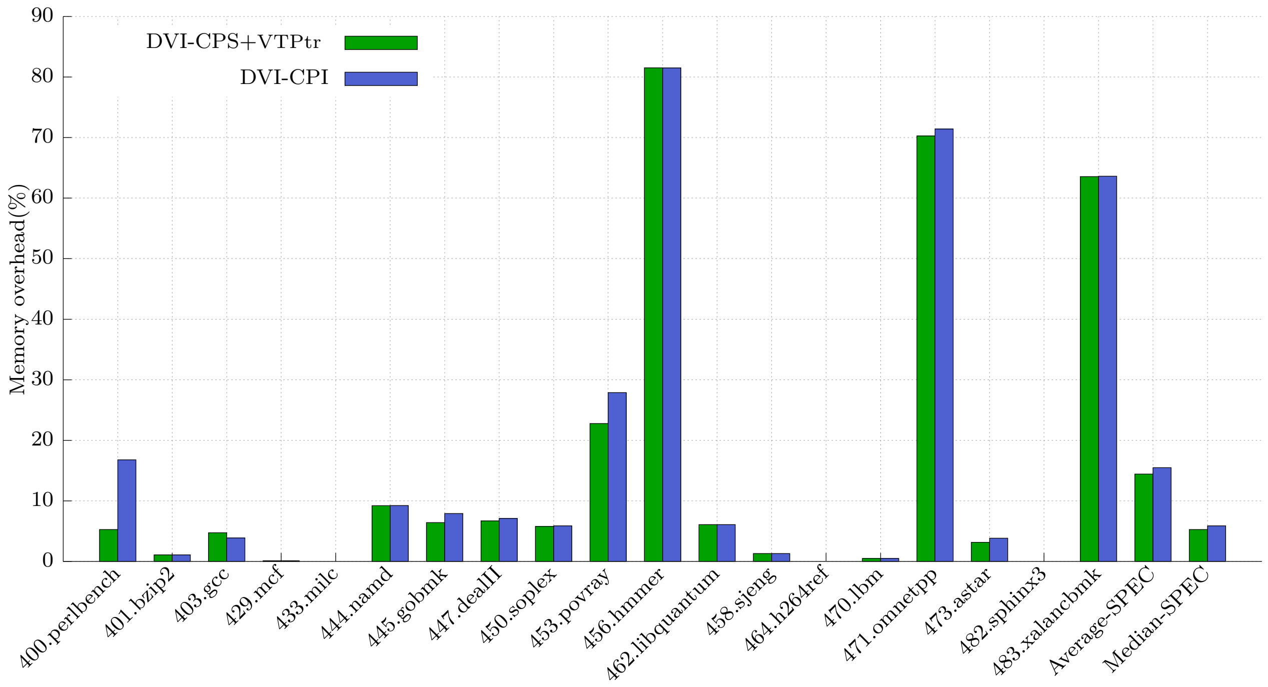

Figure 10.3: **Memory overhead of DVI-CPS+**VTPtr **and DVI-CPI on SPEC CPU2006.** HyperSpace imposes marginal overhead: average and median overhead of DVI-CPI is 15.5% and 5.9%, respectively.

Chapter 11

Discussion & Future Work

Protecting Adversarial Misuse of MPK. Because HyperSpace relies on MPK to protect the safe region, a hypothetical attack that misuses MPK could be possible: if an attacker changes the safe region read-write permissions, by somehow bending a program's control flow, this compromises both regular and safe memory via bypassing value integrity checks. However, such an attack is impossible if a program's control flow is protected by DVI-CPS/CPI protection. In the case that DVI's control flow protection is not deployed (*e.g.*, a program only adopts DVI's VTPtr or heap metadata protection), such an attack could be possible. To eliminate such an attack vector, a program can additionally adopt recently proposed orthogonal defense techniques [41, 87]; ERIM [87] and Hodor [41] are intra-process protection mechanisms using MPK. ERIM uses static binary rewriting to eliminate unintended MPK permission change instructions. Similarly, Hodor monitors usage of such unintended instructions at runtime using hardware watchpoints. For a program's intended use of MPK, they monitor a C library function, pkey_set. HyperSpace can be extended to adopt such techniques to prevent unintended use of MPK permission change instructions.

Alternative Implementation of HyperSpace. The essence of DVI is enforcing value integrity for a given address. HyperSpace is one implementation using MPK and Intel x86 segmentation. In the future, we plan to leverage upcoming hardware features: Intel EPT-based Sub-Page Permissions (SPP) [45, 92] and ARM Pointer Authentication (PA) [4]. DVI can be adopted to use SPP hardware to enforce read-only permissions of sensitive data during value invariant periods. We believe that SPP-based DVI would eliminate the overhead of dvi_assert. Moreover, we believe pointer authentication can be re-purposed to cryptographically enforce value integrity. Using the value as a modifier for a message authentication code (MAC) for sensitive pointers, we believe that PA-based DVI would eliminate additional memory overhead of HyperSpace.

Instrumentation of Security-Sensitive Non-control Data. The foundation of DVI is to correctly identify security-sensitive data along with its value invariant phases. Automatic analysis and instrumentation of control-data using CPS/CPI is well studied. However, such analysis and instrumentation of non-control data have been studied only in a limited context (*e.g.*, security checks in the Linux kernel [79]). We plan to further explore automatic analysis and instrumentation of sensitive non-control data, thereby, a wider range of DVI-based defenses can be automatically applied.

Chapter 12

Conclusion

We proposed Data-Value Integrity (DVI), a new defense policy that enforces the integrity of ”data values” for sensitive C/C++ data. Its key focus is to protect against attacks that corrupt both control and non-control data while having low performance and memory overhead. We then introduced HyperSpace, a prototype that enforces DVI to four security mechanisms that offer diverse policy constraints. Our evaluation of HyperSpace shows the versatility, and efficiency of DVI. HyperSpace incurs an average performance overhead of 1.02% and 6.35% for CPS+VTPtr and CPI, respectively, while incurring only 15% memory overhead for SPEC benchmarks and real-world applications. Additionally, we conducted security experiments using three real-world exploits and six synthesized attacks to show the effectiveness of DVI.

Bibliography

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, November 2005.

[2] One Aleph. Smashing the stack for fun and profit. *http://www. shmoo. com/phrack/Phrack49/p49-14*, 1996.

[3] Autore Anonimo. Once upon a free ().. *Phrack Magazine*, 11(57), 2001.

[4] ARM Limited. ARM Architecture Reference Manual: ARMv8, for ARMv8-A architecture profile, 2017. https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf.

[5] Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[6] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, June 2006.

[7] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, volume 12, pages 291–301, 2003.

[8] Sandeep Bhatkar, Daniel C DuVarney, and R Sekar. Efficient techniques for compre-

hensive protection from memory error exploits. In *USENIX Security Symposium*, pages 17–17, 2005.

[9] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[10] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[11] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, page 30–40, Hong Kong, China, March 2011.

[12] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):16, 2017.

[13] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. CFIXX: Object Type Integrity for C++ Virtual Dispatch. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[14] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, August 2015.

[15] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–160, Seattle, WA, November 2006.

[16] Xi Chen, Herbert Bos, and Cristiano Giuffrida. CodeArmor: Virtualizing The Code Space to Counter Disclosure Attacks. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (Euro S&P)*, Paris, France, April 2017.

[17] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. Ropecker: A generic and practical approach for defending against rop attack. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2014.

[18] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium (Security)*, San Antonio, TX, January 1998.

[19] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[20] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, August 2017.

[21] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection.

In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[22] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[23] Dhaval Kapil. Unlink Exploit, 2019. https://heap-exploitation.dhavalkapil.com/attacks/unlink_exploit.html.

[24] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient protection of path-sensitive control security. In *Proceedings of the 26th USENIX Security Symposium (Security)*, pages 131–148, Vancouver, BC, Canada, August 2017.

[25] Dongliang Mu. CVE-2015-8668, 2018. cve-2015-8668-exploit.

[26] Doug Lea. A Memory Allocator, 2000. http://gee.cs.oswego.edu/dl/html/malloc.html.

[27] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. HeapHopper: Brining Bounded Model Checking to Heap Implementation Security. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[28] Chris Evans. The poisoned NUL byte, 2014 edition, 2014. https://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html.

[29] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point (er): On the effectiveness of code pointer integrity. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[30] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, page 901–913, Denver, Colorado, October 2015.

[31] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. Imix: in-process memory isolation extension. In *Proceedings of the 27th USENIX Conference on Security Symposium*, pages 83–97. USENIX Association, 2018.

[32] Free Software Foundation. MallocInternals - glibc wiki, 2019. https://sourceware.org/glibc/wiki/MallocInternals.

[33] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, Sharad Malik, Mohit Tiwari, and Todd Austin. Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 469–484, Providence, RI, April 2019.

[34] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, April 2017.

[35] Will Glozer. a HTTP benchmarking tool, 2019. https://github.com/wg/wrk.

[36] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of

control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[37] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermining Information Hiding (and What to Do about It). In *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, August 2016.

[38] Google. TCMalloc. https://google.github.io/tcmalloc/.

[39] Jens Grossklags and Claudia Eckert. τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *Proceedings of the 21th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Heraklion, Crete, Greece, September 2018.

[40] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, Scottsdale, AZ, March 2017.

[41] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[42] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. ILR: Where'd My Gadgets Go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[43] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data

attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.

[44] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, October 2018.

[45] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2019. https://software.intel.com/en-us/articles/intel-sdm.

[46] Christopher Jelesnianski, Jinwoo Yom, Changwoo Min, and Yeongjin Jang. Making Code Re-randomization Practical with MARDU, 2019.

[47] Jonathan Corbet. x86 NX support, 2004. https://lwn.net/Articles/87814/.

[48] Michel Kaempf. Vudo malloc tricks. phrack magazine, 57 (8), august 2001.

[49] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. Adaptive call-site sensitive control flow integrity. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 95–110. IEEE, 2019.

[50] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive Control Flow Integrity. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, August 2019.

[51] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, page 437–452, Belgrade, Serbia, April 2017.

[52] Hyungjoon Koo and Michalis Polychronakis. Juggling the gadgets: Binary-level code randomization using instruction displacement. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Xi'an, China, May–June 2016.

[53] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

[54] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Transparent and efficient CFI enforcement with intel processor trace. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, Austin, TX, February 2017.

[55] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[56] Microsoft. Microsoft Docs: /GS (Buffer Security Check), 2019. https://docs.microsoft.com/en-us/cpp/build/reference/gs-buffer-security-check?view=vs-2019.

[57] Microsoft Support. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003, 2017. https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in.

[58] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Soft-Bound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings*

of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Dublin, Ireland, June 2009.

[59] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*, Toronto, Canada, June 2010.

[60] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Everything you want to know about pointer-based checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[61] Nandy Narwhals CTF Team. CVE-2016-10190 Detailed Writeup, 2017. https://nandynarwhals.org/cve-2016-10190/.

[62] Nathan Burow. CFIXX C++ test suite, 2018. https://github.com/HexHive/CFIXX/tree/master/CFIXX-Suite.

[63] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, June 2014.

[64] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[65] Gene Novark and Emery D. Berger. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, page 573–584, Chicago, IL, October 2010.

[66] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida.

Poking holes in information hiding. In *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, August 2016.

[67] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

[68] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[69] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, August 2013.

[70] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 241–254, Renton, WA, July 2019.

[71] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[72] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[73] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov.

AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 309–318, Boston, MA, June 2012.

[74] @sha0coder. Python - 'socket.recvfrom_into()' Remote Buffer Overflow, 2014. URL https://www.exploit-db.com/exploits/31875.

[75] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, October–November 2007.

[76] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Free-Guard: A Faster Secure Heap Allocator. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[77] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. Guarder: A Tunable Secure Allocator. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[78] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[79] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[80] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo

Kim, Wenke Lee, and Yunheung Pack. HDFI: Hardware-Assisted Data-flow Isolation. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[81] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[82] The Clang Team. Clang 10 documentation: CONTROL FLOW INTEGRITY, 2019. https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[83] The Clang Team. Clang 10 documentation: SAFESTACK, 2019. https://clang.llvm.org/docs/SafeStack.html.

[84] The PAX Team. Address Space Layout Randomization, 2003. https://pax.grsecurity.net/docs/aslr.txt.

[85] The PostgreSQL Global Development Group. pgbench: PostgreSQL Client Applications , 2020. https://www.postgresql.org/docs/current/pgbench.html.

[86] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[87] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium (Security)*, pages 1221–1238, Santa Clara, CA, August 2019.

[88] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[89] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[90] Zhe Wang, Chenggang Wu, Jianjun Li, Yuanming Lai, Xiangyu Zhang, Wei-Chung Hsu, and Yueqiang Cheng. Reranz: A Light-weight Virtual Machine to Mitigate Memory Disclosure Attacks. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Xi'an, China, April 2017.

[91] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, November 2016.

[92] Zhang Yi. Intel EPT-Based Sub-page Write Protection Support, 2017. https://lwn.net/Articles/736322/.

[93] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic Techniques to Systematically Discover New Heap Exploitation Primitives. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, August 2020.

[94] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[95] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Protecting Virtual Function Tables' Integrity. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[96] Mingwei Zhang and R Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, August 2013.

[97] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 631–644, Providence, RI, April 2019.