VIP: Safeguard Value Invariant Property for Thwarting Critical Memory Corruption Attacks

Mohammad Ismail†
Virginia Tech, USA
imohammad@vt.edu

Jinwoo Yom††
Virginia Tech, USA
jinwoo7@vt.edu

Christopher Jelesnianski
Virginia Tech, USA
kjski@vt.edu

Yeongjin Jang
Oregon State University, USA
yeongjin.jang@oregonstate.edu

Changwoo Min
Virginia Tech, USA
changwoo@vt.edu

1 Introduction
The foundation of most software stacks is written in unsafe lan-
guages such as C/C++. This jeopardizes not only the security of
programs written in those languages but also the security of pro-
grams written in modern type-safe languages, as the latter often
utilize libraries written in unsafe languages. This causes applica-
tions to be prone to memory corruption vulnerabilities.

Successful memory corruption attacks aim to modify the in-
tended value of security-sensitive data. For example, control-flow
hijacking attacks exploit memory corruption vulnerabilities to over-
write code pointers. In most cases, these targeted code pointers are
return addresses [4, 5, 67, 71], function pointers [9, 16, 24, 31], or
virtual function table pointers in C++ [64, 82]. Overwriting such val-
ues allows attackers to achieve arbitrary code execution. Similarly,
heap overflow attacks exploit memory corruption vulnerabilities to
overwrite heap metadata [3, 21, 22, 56, 69, 70, 81]. Tainting heap
metadata can mislead the memory allocator, allowing attackers to
either have arbitrary write or arbitrary code execution capabilities;
both are capabilities that can be considered critical security threats.

Memory-corruption Defense Landscape. In response, many
defenses have been proposed to thwart memory corruption-based
attacks. However, they suffer from high runtime overhead, or they
are imprecise and thus susceptible to attacks.

Full memory safety enforcement techniques [49, 65, 85] prevent
memory corruption attacks by enforcing spatial and temporal mem-
ory safety. However, these approaches fall short in practicality due
to high runtime performance and memory overhead. For example,
a state-of-the-art system BOGO [85] has 60% runtime overhead and
36% memory overhead.

Control-flow integrity (CFI) techniques [1, 6, 12, 18, 29, 34–
36, 42, 47, 54, 55, 59, 61, 74, 77, 79, 80, 83, 84] provide control data
protection by enforcing the integrity of expected control flows based
on a program’s control-flow graph (CFG). However, CFI techniques
struggle to balance precision and runtime overhead in their control
flow enforcement. Numerous efficient CFI proposals suffer from a
large equivalence class (EC) [41], which is a set of indistinguishable
code targets for each indirect transfer due to the imprecise control
flow analysis. In this case, CFI cannot accurately detect an illegal
lyent control transfer for a given EC [9, 24, 64]. Recent work [36] at-
tempts to address this inherent problem by enforcing a unique code
target (UCT) property (i.e., EC = 1). However, this technique suffers
from scalability, thus inhibiting wide adoption because it requires
background threads to process Intel PT packets, and dedicating a
CPU core for analysis.
Code-pointer integrity (CPI) [45] protects all code pointers (recursively) in a program. Similar to CFI, CPI defends against control data attacks. CPI protects code pointers via isolation relying on information hiding; however, attacks against information hiding [23, 32, 57] can break this security guarantee. Furthermore, CPI has a high memory overhead (105% on average) to keep track of metadata for all sensitive pointers.

Data-flow integrity (DFI) [11, 72, 73] prevents both control and non-control data attacks. 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 for data-flow tracking. It has a 104% performance overhead and a 50% memory overhead on average.

Goal. Under this circumstance, we set our goal to provide a defense mechanism that strikes balance between efficiency and effectiveness. In particular, we focus on thwarting two critical classes of memory corruption attacks, namely control-flow hijacking and heap metadata corruption attacks, both of which let attackers achieve arbitrary code execution. These two attacks are very popular and commonly exploited [13], making them a valuable target to defend against. By focusing on these two critical attacks, we aim to provide a pragmatic memory security defense that is both efficient and effective.

Value Invariant Property. To this end, we propose the Value Invariant Property (VIP), a new security policy that thwarts these two attacks. VIP does this by enforcing the integrity of data values for security-sensitive data (e.g., function pointers for an indirect call, virtual function table pointers in C++ objects, and heap metadata). VIP prevents software from accepting maliciously altered security-sensitive data via memory corruption attacks. VIP achieves this by capitalizing on the life cycle of security-sensitive data; security-sensitive data should only be altered by legitimate updates so it should be immutable between two legitimate updates. We call this its value invariant period. This write-protection approach is similar to WIT [2]. WIT assigns a color to each object and each write instruction so that all objects written by a given instruction are a specific color. However, WIT suffers from information hiding limitations of its color table. VIP makes data immutable during this life cycle period, by having a secure copy of security-sensitive data, which is immutable to memory corruption attacks.

HyperSpace. We realize VIP as HyperSpace, our prototype defense mechanism that applies VIP to thwart critical memory corruption attacks while maintaining low runtime overhead and minimal additional hardware resources. HyperSpace records values of security-sensitive data into a safe memory region and validates values before use to enforce value integrity. The safe memory region is protected by Intel Memory Protection Keys (MPK) [39, 43, 60], an efficient per-thread memory protection mechanism.

HyperSpace enforces VIP to thwart control-flow hijacking and heap metadata corruption attacks. We implement four state-of-the-art security mechanisms with HyperSpace: 1) control-flow integrity (VIP-CFI), 3) code pointer integrity (VIP-CPI), 2) virtual function table pointer protection for C++ objects (VIP-VTPt), and 4) inline heap metadata protection. For the pointer protection mechanisms (1-3), we design a compiler pass that automatically instruments code pointers and sensitive data pointers to protect VIP. This is similar to the protection scope offered by other protection mechanisms such as CPI [45]. However, HyperSpace goes one step further by supporting heap metadata protection, which is an additional source of many sophisticated attacks. Our choice of protecting against both control-flow hijacking and heap metadata corruption is to demonstrate that VIP is capable of protecting not only sensitive code pointers but also other sensitive data types.

In addition, we propose optimization techniques to significantly lower the runtime overhead via reducing: the cost of each VIP protection instrumentation, the number of pointers to protect, and the number of costly permission changes needed for the safe memory region.

We evaluate HyperSpace using standard benchmarks (all C/C++ benchmarks in SPEC CPU2006) and real-world applications (NGINX web server and PostgreSQL database server). In addition, we test HyperSpace against three real-world exploits and six synthesized attacks that include: virtual function pointer table hijacking attacks, a COOP attack [64], and a heap exploit, demonstrating the effectiveness of HyperSpace. We detail how these attacks are successfully detected and blocked by HyperSpace when an attempt of corrupted sensitive data usage is detected. HyperSpace incurs a small performance and memory overhead even when programs are armored with HyperSpace’s strongest defense, VIP-CPI, which guarantees the full integrity of all security-sensitive pointers.

To summarize, our contributions include:

- We propose a new security policy: by protecting the Value Invariant Property (VIP) of security-sensitive data, we can effectively mitigate critical memory corruption attacks.
- We built HyperSpace, a full prototype of the defense mechanism that enforces the integrity of VIP for security-sensitive data in a program. We implemented four state-of-the-art security mechanisms as HyperSpace use-cases to demonstrate how VIP and HyperSpace can be used for protecting sensitive code/data pointers and heap metadata. We also design HyperSpace to prevent attacks originating from malicious MPK use in userspace [14].
- We devised novel compiler optimization techniques that significantly reduce the runtime overhead of HyperSpace instrumentation and make HyperSpace a practical, deployable defense.
- We evaluate HyperSpace and its security mechanisms on benchmarks, real-world applications, and synthesized attacks. Our results show that HyperSpace can defeat control-flow hijacking and heap overflow attacks with an average of 6.18% performance overhead and 13.18% memory overhead in SPEC CPU2006.
- We make our source code of HyperSpace publicly available at https://github.com/cosmoss-vt/vip.

2 Background and Motivation

In this section, we describe our target attack classes that VIP and HyperSpace aim to defend. We particularly focus on two of the most critical memory corruption attacks – 1) control-flow hijacking attacks caused by code/data pointer corruption and 2) heap metadata corruption attacks – because they are the main avenue to achieve arbitrary code execution (ACE) and arbitrary memory write, which let attackers take full control of a system. We note that security vulnerabilities and Common Vulnerability Exposures (CVEs) that allow arbitrary code execution are rated between 7.5 (high)
We identify the Value Invariant Property of the function `handle_req(int, char*)` to subvert a program’s control-flow or change a program’s intended behavior. On the left, an attacker exploits a stack overflow (Figure 1: Two examples of vulnerable C code. Attackers can overwrite security-sensitive data by exploiting memory corruption vulnerabilities to subvert a program’s control-flow or change a program’s intended behavior. On the left, an attacker exploits a stack overflow (Line 15), which overwrites a function pointer (`func`) to subvert control-flow (e.g., arbitrary program execution at Line 17). On the right, an attacker exploits a heap overflow (Line 30), which overwrites the heap metadata to control subsequent memory allocations (Line 37), which can be a primitive for other attacks — e.g., arbitrary memory write at Line 40.

and 9.8 (critical) regarding their security impact [25, 26]. Then, we present our analysis of two example vulnerabilities in Figure 1 and patternize common characteristics of these attack classes. In particular, we define the notion of security-sensitive data, whose value changes are critical to successful memory corruption. Subsequently, we identify the Value Invariant Property as the key characteristic to thwart these attacks, both effectively and efficiently.

2.1 Code Pointer Corruption Attack

Take an example of the vulnerable code on the left side of Figure 1. The function `handle_req()` is our point of interest. It uses a local function pointer `FP func` as a local variable at Line 8 and uses another local buffer variable `char buf[26]` at Line 9. The code assigns the variable `func` by selecting one of the predefined function pointers (Line 11 and Line 13).

Here, `func` can only point either `X()` (if `uid == 0`) or `Y()` (if `uid == 1`). However, since the execution of `strcpy(buf, input)` at Line 15 failed to check the length of input against the buffer size (buf[26]), an attacker may trigger a stack buffer overflow vulnerability by supplying more than 20 characters to the string variable input. This allows the attacker to change `func`, for example, from `X()` (suppose `uid == 0`) to an arbitrary function (e.g., `system()`) to execute an arbitrary command). Thereby, as a result of exploiting the vulnerability at Line 17, the code may call an arbitrary function of the attacker’s choice.

### Implication

**Implication.** To launch a successful control-flow hijacking attack, a popular avenue of attack is to overwrite a code/data pointer to achieve arbitrary code execution.

2.2 Heap Metadata Corruption Attack

The code on the right side of Figure 1 is an example of a heap metadata corruption attack. In particular, it demonstrates how heap metadata corruption changes the behavior of a memory allocator and can be turned into an arbitrary memory write primitive, which allows an attacker to alter arbitrary memory addresses with attacker-assigned values. We assume the example program uses the popular `malloc2` in glibc without loss of generality. The program first allocates three heap objects A, B, and C in order and then frees B (Lines 23–27). After freeing B, the heap will have a free memory block between A and C. The free memory block starts with inline metadata, which stores the size of the memory block and links to the previous/next free memory blocks.

Thus, the `strcpy()` code has a heap overflow vulnerability by missing the length check of the input against the size of object A. Thus, an attacker can trigger the heap overflow by supplying more than 100 bytes to input and consequently will corrupt the heap metadata right next to the object A. Corrupting heap metadata with attacker-chosen data like this allows the attacker to mislead the heap metadata management algorithm (Lines 33–37) [68]. As a result, future allocation will be located at an arbitrary location of the attacker’s choice — e.g., a return address, function pointer, virtual function pointer table pointer of a C++ object, and other security-sensitive data, resulting in arbitrary memory write (Line 40), which again can trigger arbitrary code execution if a code pointer is overwritten.

### Implication

**Implication.** To launch a successful arbitrary memory write attack via exploiting a heap overflow vulnerability, an attacker must corrupt the heap metadata to mislead the heap management algorithm.

2.3 Security-Sensitive Data

Based on the implications of our vulnerability analyses, we define our notion of security-sensitive data to defend against these two target attack classes. In a nutshell, we refer to security-sensitive data as a variable/object in memory that is required to be corrupted to complete a successful control-flow hijacking attack or a heap metadata corruption attack.

From our stack overflow example, we have observed that attacks require `corrupting a code pointer` (e.g., `func`) to achieve arbitrary code execution. In addition, we extend our coverage to all `sensitive pointers` [45], which include all code pointers and all data pointers that can extend to a code pointer, to avoid indirect control-flow hijacking leveraging sensitive data pointers. We follow the same definition of sensitive pointers as Code Pointer Integrity (CPI) [45].

Likewise, in our heap overflow example, the attack requires `corrupting heap metadata` (e.g., a `free()`-ed heap object, object B) to
We also analyze the life cycle of security-sensitive heap metadata. We regard inline heap metadata as security-sensitive data because corrupting such data is essential for launching a successful attack.

In the next section, we discuss how we can mitigate these attacks by safeguarding our new security property, the Value Invariant Property (VIP), on security-sensitive data.

3 Value Invariant Property (VIP)

We introduce the Value Invariant Property (VIP), which is a common property of security-sensitive data in critical memory corruption attacks. Our intuition behind VIP originates from a common pattern in programs: security-sensitive data should never be changed between two legitimate writes so there is a period such that security-sensitive data is immutable. Moreover, in many programs, the value of security-sensitive data does not frequently change. These observations form the basis of our value invariant property. More specifically, during the life cycle of an object, we have observed that the values of security-sensitive data do not change after their legitimate assignments and before the object’s destruction or new legitimate value assignments.

We now re-investigate the two code examples in Figure 1 with respect to our value invariant property in the rest of this section.

3.1 Value Invariant Property of a Code Pointer

We analyze the life cycle of a security-sensitive code pointer func with respect to its value invariant period as follows.

1. Assignment: The first and only assignment to func is on Line 13.
2. In-use (value invariant property holds): After the assignment and before the destruction of the stack, func never changes. Thus, its value invariant period starts right after the value assignment.
3. Destruction: The stack variable func will become invalid when the function unwinds its stack, i.e., at the function epilogue. Thereby, the period ends when the variable is destructed.

**Attack.** Overwriting the value of func during the value invariant period in any manner, e.g., via exploiting a buffer overflow vulnerability on Line 15, may conclude in a successful attack, such as arbitrary code execution.

3.2 Value Invariant Property of Heap Metadata

We also analyze the life cycle of security-sensitive heap metadata with respect to its value invariant period. Unlike the previous example, heap metadata is created internally when malloc() and free() are called. For brevity, we illustrate the life cycle of heap metadata for object B, which is first allocated at Line 24, as follows.

1. Metadata allocation: Calling free() on Line 27 will change the heap metadata for object B, which is between A and C, to a free state.
2. In-use (value invariant property holds): Before running additional heap operations such as malloc() or free(), the heap metadata of B should never change. Thus, its value invariant period starts right after the free().
3. Destruction: When subsequent malloc() is called at Line 33, the memory allocator will allocate new memory on the old object B’s location. Thus, the heap metadata of the old object B is updated to the newly allocated state; the value invariant period ends when the metadata is updated.

**Attack.** Overwriting the allocated heap metadata during the value invariant period in any manner (e.g., via exploiting a heap overflow vulnerability on Line 30) may mislead the heap management algorithm to launch an arbitrary memory write attack. We regard inline heap metadata as security-sensitive data because corrupting such data is essential for launching a successful attack.

4 Threat Model and Assumptions

VIP and HyperSpace focus on thwarting two critical types of memory corruption attacks. Our assumption includes a program that has one or more memory vulnerabilities (e.g., stack/heap buffer overflow) that allow attackers to read from and write to arbitrary memory. The attacker can use an arbitrary write capability to perform code/data pointer corruption attacks and/or heap metadata corruption attacks. However, the attacker cannot modify or inject code due to Data Execution Prevention (DEP) [40, 48]. We assume that all hardware (e.g., Intel MPK) and the OS kernel are trusted such that attacks exploiting those vulnerabilities are out of scope. Regarding the use of Intel MPK for userspace protection keys, Conner et al. discovered an attack that can bypass memory isolation mechanisms based on MPK such as ERIM and Hodor. However, we regard these attacks to be out of scope because HyperSpace does not allow user programs to include or use any wrpkru instructions for userspace protection.

5 HyperSpace Design

The main challenge to realizing VIP is how to efficiently and securely keep track of the value invariant property for sensitive
Figure 2: VIP primitives (left) and the state transition diagram (right) for VIP protected memory. VIP primitives trigger state transitions for a specified memory location. VIP manages the intended value of sensitive data for integrity checking (vip_assert). Mismatching values of sensitive data or an illegal state transition indicates a value integrity violation.

5.1 HyperSpace Primitives

HyperSpace manages the state of a memory location as illustrated in Figure 2. When a program starts, the entire memory space is in a non-sensitive state, meaning that no memory location stores security-sensitive data. To protect a memory location storing security-sensitive data, HyperSpace first requires the location to be registered upon its allocation (vip_register). Then, the memory will be in a sensitive, uninitialized state. Once the security-sensitive data is written to the memory location, HyperSpace creates a copy of its value in the safe memory region (vip_write). Therefore, access to the safe memory location stores security-sensitive data. To protect a memory location storing security-sensitive data, HyperSpace first requires the location to be registered upon its allocation (vip_register). Then, the memory will be in a sensitive, initialized state. Notably, if we know a write should be the final one until the deallocation of the memory, i.e., the data is in value invariant period, then we can annotate this (vip_write_final). This will put the memory into a sensitive, finalized state, and HyperSpace does not allow any further writes to that memory location. VTable pointer in C++ is a use-case of this state because it is written only once at the object construction time and should not be updated until the object is being destructed.

Before loading any sensitive data, the program should check whether its value is changed or not by comparing the value in the regular memory region with the value in the safe memory region (vip_assert). If two values do not match or a program attempts to perform an illegal state transition, it alerts the value integrity violation and stops the program execution. Finally, when a sensitive memory location is deallocated, it is unregistered (vip_unregister) and reverts the memory locations to the default non-sensitive state, allowing it to be reused again in the future.

5.2 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 3. When a process is created, HyperSpace kernel reserves the upper half of the virtual address space as the safe memory region.

Note that dedicating %gs does not harm supporting of TLS in Linux because Linux/glibc uses %gs for TLS. On Windows using %gs for TLS, %gs can be used for HyperSpace.

Figure 3: HyperSpace memory layout. The safe memory region is protected by Intel Memory Protection Keys (MPK) and it is efficiently accessible using hardware segmentation (%gs) in x86 architecture.

Additionally, the %gs register is set to the starting address of the safe memory region.1 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 4).

We note that the safe memory region is an anonymous region, managed by the kernel. That is, the 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.

Even though we are reserving half of the virtual address space, this should not be a problem, since we target the 64-bit x86 architecture. Moreover, note that the address space is reserved without physical memory allocation. In the typical 4-level paging scheme, reducing the user-space virtual address size by half still leaves 64 TB of memory open for addressing. If a 5-level paging scheme is applied, the reduced user-space virtual address size still leaves 32 PB free. These are both still large enough in practice.

5.3 Protecting Safe Memory Using MPK

The parallel memory layout enables efficient access to the safe memory region. 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), which is an efficient per-thread memory protection mechanism in the x86-64 architecture [39, 43, 60].

By default, the safe memory region is read-only. Only during VIP operations that update the safe memory region (i.e., vip_register, vip_unregister, vip_write, and vip_write_final), HyperSpace temporarily grants read-writable permissions to only the thread executing those VIP operations. Thus, any write attempt to safe memory by an unauthorized thread at an unauthorized time will cause a segmentation fault error.

---

1Note that dedicating %gs does not harm supporting of TLS in Linux because Linux/glibc uses %gs for TLS. On Windows using %gs for TLS, %gs can be used for HyperSpace.
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 are independently controlled through an MPK register. Changing memory access permissions is fast as it only takes around 23 CPU cycles on average using a non-privileged instruction \texttt{wptrku} [60]. Also, the impact of permission changes is thread-local as the MPK register is per-CPU. The discussion regarding the possible misuse of VIP primitives is addressed in §10.

5.4 Representing State of Safe Memory

In \textsc{HyperSpace}, each memory location is in one of four states shown in Figure 2. \textsc{HyperSpace} manages an additional area at the end of the safe memory regions to represent the state of each memory location. Because \textsc{HyperSpace} manages sensitive data in 8-byte granularity, 2 bits of metadata are assigned for each 8-byte chunk of sensitive data. The state bitmap is updated upon memory state transition. \textsc{HyperSpace} detects an attempt for illegal state transition (e.g., \texttt{vip\_write} or \texttt{vip\_assert} on non-sensitive data) and prevents illegal/malicious use of \textsc{HyperSpace} primitives. With our bitmap representation, access to the state can be efficiently done using segmentation (see Figure 4).

5.5 Safety of Safe Memory

In \textsc{HyperSpace}, we ensure the spatial and temporal safety of security-sensitive data with \textsc{HyperSpace} primitives and the memory state management model. For temporal safety, \textsc{HyperSpace} keeps track of the registration status of the protected data. In this regard, the \texttt{vip\_register} operation changes the status of safe memory from not-in-use to in-use. Based on this temporal status tracking, \textsc{HyperSpace} guarantees spatial safety of safe memory by managing the safe memory region in an 8-byte block granularity. \textsc{HyperSpace} allows access only to the registered, valid blocks and faults on any unregistered access, such as out-of-bound unregistered access of the safe memory. Although \textsc{HyperSpace} does not guarantee full temporal safety, it significantly raises the bar. By providing registration-based checking, \textsc{HyperSpace} denies access to unregistered sensitive data. However, it cannot distinguish if a sensitive data is freed then reallocated. This could open the possibility of a temporal attack, albeit it would be significantly harder to achieve.

5.6 Low Memory Overhead

The maximum memory overhead is bounded to 103.1% of an application’s total memory usage in the regular memory region ($T + T \times 32$ bytes). This is relatively low compared to approaches managing rich metadata (e.g., tag, bounds) such as CPI [45] and SoftBound+CETS [49–51]. Actual memory overhead is much lower than the maximum overhead because \textsc{HyperSpace} relies on sparse address space support of the underlying OS for the safe memory region. Initially, the 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. This also applies to the page table entries, hence, \textsc{HyperSpace} will allocate memory pages only for the corresponding regular memory page that stores safe data. Our evaluation results in §9.3.2 show that the additional

1 // Get the safe memory value for a given address
2 uint64_t vip_load_safe_memory(uint32_t addr) {
3     uint64_t value;
4
5     asm volatile (
6         "mov %%gs:0x0(%[offset]), %[value]"
7         : [value] "r" ([value])
8         : [offset] "r" ([offset])
9         : "r"
10     );
11
12 return value;
13 }

Figure 4: Code for accessing safe memory and its state.

memory overhead of \textsc{HyperSpace} is marginal (13.18% on average with SPEC 2006).

5.7 Putting It All Together

With \textsc{HyperSpace}, the design of VIP’s API in Figure 2 is simple and efficient. Registering/unregistering sensitive data (\texttt{vip\_register} and \texttt{vip\_ unregister}) changes the corresponding bits in the state bitmap. Writing sensitive data (\texttt{vip\_write} and \texttt{vip\_write\_final}) copies the sensitive value to the safe memory region and changes the state bits if necessary. \textsc{HyperSpace} temporarily grants write permissions to the safe memory region only for these four VIP operations and only to the calling thread. \textsc{HyperSpace} checks value integrity by comparing values between the regular and safe memory regions (\texttt{vip\_assert}). For all VIP operations, \textsc{HyperSpace} checks if the memory is in a valid state for a given operation. Otherwise, \textsc{HyperSpace} raises a security exception against any attempt of illegal state transition.

6 \textsc{HyperSpace} Defenses

In this section, we present four defense mechanisms based on \textsc{HyperSpace} to defeat control-flow hijacking and heap metadata corruption attacks by enforcing the value invariant property. To defeat control-flow hijacking attacks, \textsc{HyperSpace} implements: (1) Control Flow Integrity (VIP-CFI) – protecting all code pointers, (2) Code Pointer Isolation (VIP-CPI) – protecting all sensitive code/data pointers, and (3) virtual function table pointer protection in C++ objects (VIP-VTPtr). We present automatic instrumentation for these three protections. \textsc{HyperSpace} covers all sensitive global, heap, and stack variables. To prevent heap metadata corruption attacks, \textsc{HyperSpace} (4) extends \texttt{ptrmalloc2} [28], which is the default memory allocator in most Linux distributions, manually inlining VIP API into its source code.

6.1 Control Flow Integrity (VIP-CFI)

We enforce the integrity of control flow by guaranteeing the safety of all code pointers. VIP-CFI allows indirect control-flow transfer only when a target code pointer matches with its legitimate copy in the shadow memory region (i.e., the code pointer does not violate its value invariant property). Thus, all function pointers must be secured using VIP’s register, write, assert and deregister primitives. \textsc{HyperSpace} accomplishes this by accurately identifying and instrumenting all instructions that allocate, write, use, and deallocate code pointers. We note that VIP-CFI provides the unique code target (UCT) property [36] as it allows only a single target for
each indirect control-flow transfer, the pointer that HyperSpace made invariant. Thereby, VIP-CFI does not suffer from the attack that ConFIRM [46] launches, which replaces an indirect call/jump target to another allowed target in an equivalence class of multiple allowed code addresses.

We identify code pointers using LLVM type information. Because code pointers can exist inside of structs or arrays (i.e., a composite data type), HyperSpace recursively looks through each element of container types as well. For cases where code 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. Specifically, if a pointer is ever cast into a sensitive type – a code pointer or a composite data type that is reachable to a code pointer - within a function, or returned to another function where it is cast to a sensitive pointer, HyperSpace considers this to be sensitive as well.

We handle memcpy and mmap as special cases separately with their own intrinsics. Since memcpy and mmap take void* arguments, HyperSpace gets the actual operand types before being cast to void*, and instrumentation for memcpy and mmap operands is done separately. In the case of an array of code pointers that is memcpy’d to another location, HyperSpace registers/writes each array element to safe memory.

Registration of code pointers is instrumented immediately after its allocation. We use a shadow stack (specifically, SafeStack [45]) 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. HyperSpace instruments all other heap variables, global variables, and other address-taken code pointers on the regular stack via vip_register.

To determine when to perform vip_write for code pointers, we look for any unsafe code pointers (i.e., code pointers not on the SafeStack) that are the destination operand of a store instruction. vip_write will be instrumented following such store instructions if the variable is not in the SafeStack. vip_assert should be called immediately before using any code pointer. Specifically, we look for call and load instructions for instrumentation.

For sensitive heap and mmap-ed variables, deregistering is instrumented before free and mmap 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 having iterative deregisters. Note that we did not handle C unions because we did not encounter any in our evaluation.

6.2 Code Pointer Integrity (VIP-CPI)

In addition to code pointer protection in VIP-CFI, HyperSpace can be used to guarantee the integrity of all sensitive code and data pointers. HyperSpace recursively protects all sensitive pointers as defined in CPI [45] - all code pointers and all data pointer types that can reach a code pointer.

In order to detect the additional sensitive pointers required for VIP-CPI, the type analysis of VIP-CFI is extended to include more cases. Composite type objects that contain a function pointer are recognized as 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 the protection sets for all the sensitive types in the LLVM pass, its instrumentation is similar to VIP-CFI. 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 set from the type analysis and instruments vip_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 VIP-CFI instrumentation. When sensitive variables are being used, all load instructions of the sensitive variables are instrumented.

VIP-CFI leverages static analysis for determining sensitive-data. Since static analysis is known to be imprecise, we over-approximate when detecting security-sensitive pointers to guarantee full coverage. That is, HyperSpace regards a pointer as security-sensitive if it cannot determine a pointer as non-security-sensitive at compile time. One such example is that C/C++ allows char* pointers to point to objects that are of any type. This conservative approach may induce false positives (i.e., unnecessary protection), however, such false positives will not compromise VIP’s security guarantees.

6.3 VTable Protection in C++ (VIP-VTPtr)

Hijacking the virtual function table pointer of an object is a commonly exploited attack [7, 64, 82]. In C++, virtual functions are an essential part of dynamic polymorphism. At each virtual function call, an appropriate function is chosen according to the object type. The object type mapping to a virtual function is through the use of a virtual function table pointer (VTPtr). The VTPtr is an array pointer that includes virtual function pointers available for a specific object class type. A VTPtr is located in the header of an object and is initialized in an object’s constructor. After initialization, this pointer variable should not be altered during the entirety of its lifetime.

To protect virtual function table pointers, we need to first correctly identify the VTPtr within C++ objects. This can be detected using HyperSpace’s type analysis. When recursively dereferenced from all proceeding pointer types, our analysis can identify the code pointers and mark the VTPtr as a security-sensitive pointer.

The registration of VTPtr is instrumented along with the rest of regular sensitive type registrations during object allocation. Compared to VIP-CFI, no extra registration semantic changes were needed for this support. To guarantee that the VTPtr of an object will never change, HyperSpace instruments the vip_write_final call right after the VTPtr is assigned by the object’s constructor. This ensures that the object’s VTPtr does not get modified outside of its constructor.

HyperSpace instruments vip_assert primitive immediately before the load instruction to guarantee that the VTPtr has not been tampered with. Similar to registration, the same deregistration semantics as in VIP-CFI are used to deregister the VTPtr along with other sensitive values the object may contain.

6.4 Heap Metadata Protection

The heap memory allocator is essential in building an efficient and secure program. tmalloc [28] is one of the most widely adopted heap allocators. tmalloc and many other heap allocators (e.g., dmalloc [20] and tcmalloc [33]) adopt an inline metadata design for performance reasons. Unfortunately, this inline metadata design
suffers a major security flaw. As shown in §2.2, an adversary can compromise inlined metadata to perform arbitrary code execution by exploiting heap-based buffer overflow vulnerabilities. Although several security mechanisms were proposed in an attempt to address this issue, they are still able to be bypassed [21, 22, 81].

To defend against this inline heap metadata corruption attack, we instrument the partalloc source code manually. We register 32-byte metadata whenever a new memory chunk is created (e.g., splitting one large chunk into two smaller chunks) and deregister the 32-byte metadata whenever a memory chunk is deleted (e.g., merging two small chunks into one big chunk) using vip_register and vip_unregister. For each malloc and free, we first check whether the inline metadata is corrupted using vip_assert. After updating metadata, we copy the newly written metadata to the safe region using vip_write. This approach protects inline heap metadata against state-of-the-art corruption attacks such as poisoned NULL byte, 1-byte NULL overflow [22], and unsafe unlink [17] by asserting metadata during malloc and free to detect corruption.

7 HyperSpace Optimizations

We present optimization techniques applied to reduce HyperSpace’s overhead of instrumentation and to reduce the memory access overhead for our safe memory region.

7.1 Runtime Silent Store Elimination (SLNT)

VIP utilizes Intel MPK to efficiently control safe memory permissions. In most cases, changing the permissions of the safe memory region using MPK is fast enough. However, it could incur significant overhead if an application requires frequent permission changes. We have observed that some applications keep updating sensitive data with the same value – also known as a silent store. Such frequent silent stores to the safe memory region are detrimental because it requires frequent MPK permission changes.

With this, we eliminate silent stores to the safe memory region using runtime checking. For vip_write, we check if the value being written is the same as its safe copy (i.e., silent store) as well as if the target safe memory is already in a sensitive, initialized state. If so, the write operation is not necessary, allowing HyperSpace runtime to skip the vip_write. Therefore, HyperSpace only utilizes wrpkru in the first vip_write. Any subsequent vip_write that writes the same data value will be ignored as no update is necessary to safe memory. This reduces the number of unnecessary wrpkru calls.

Our optimization is effective because vip_write is one of the most frequently used VIP functions and changing permissions using wrpkru is more expensive (~23.3 CPU cycles) compared to reading the current MPK permissions using rdpkru (~0.5 CPU cycles). This prevents unnecessary writes and MPK permission changes in many applications (e.g., 453.povray as described in §9.3.1).

7.2 Coalescing Permission Changes within a Basic Block (CBB)

To further reduce the unnecessary toggling of safe memory region permissions, we introduce an optimization technique to coalesce a series of HyperSpace protection instrumentation (i.e., vip_safe_memory_unlock and vip_safe_memory_lock) within a basic block. vip_safe_memory_unlock and vip_safe_memory_lock refer to the opening and closing of the safe memory region with the MPK instruction, wrpkru.

Figure 5: Before (top) and after (bottom) basic block level coalescing optimization for permission changes in 400.perlbench (Lines 23-28).

Figure 5 shows an example of an instrumented function from 400.perlbench in SPEC CPU2006, where LISTOP is a sensitive type. The original instrumentation (top), shows the modification of a sensitive object’s linked list (Lines 7 and 11), which requires opening of the safe memory region. However, repetitively unlocking and locking is unnecessary if VIP 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 the locking instrumentation (vip_safe_memory_lock), which reverts the safe memory permission to read-only, after the very last VIP API call as shown in Figure 5 (bottom).

Based on this intuition, we introduce a coalescing-safe basic block, where we can safely coalesce all write instrumentations in a basic block. All memory writes in a coalescing-safe basic block are guaranteed to not be capable of corrupting arbitrary memory locations. Therefore, a store target address should be limited to one of the following: (1) a sensitive type that is protected by VIP, (2) a non-sensitive field of a sensitive type whose address is bounded by the sensitive type, or (3) a local variable in a 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 5 (bottom), all intermediary permission changes are removed and a single lock function is placed at the end of its basic block (Line 28). This greatly reduces unnecessary permission changes.

7.3 Coalescing Permission Changes within a Function (CFN)

We introduce a coalescing-safe function by extending the notion of a coalescing-safe basic block to further reduce the MPK permission change overhead. A function is considered to be coalescing-safe – i.e., not capable of corrupting arbitrary memory locations – if it meets three conditions: (1) all basic blocks in the function 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...
To minimize instrumentation overhead and eliminate function call overhead, our instrumentation pass inlines HyperSpaceAPI calls. Furthermore, we optimized HyperSpace’sAPI calls specifically for handling and protecting 8-byte data. This is because most sensitive data needing protection are usually various pointer types. These 8-byte optimized APIs are inlined using LLVM’sLink Time Optimization (LTO).

7.4 Inlining VIP Functions (INLN)

To optimize overhead and eliminate function call overhead, our instrumentation pass inlines HyperSpaceAPI calls. Furthermore, we optimized HyperSpace’sAPI calls specifically for handling and protecting 8-byte data. This is because most sensitive data needing protection are usually various pointer types. These 8-byte optimized APIs are inlined using LLVM’s Link Time Optimization (LTO).

7.5 Excluding Objects in Safe Stack (SS)

As discussed in §6, we use SafeStack [75] to protect return addresses and safe objects that are address-not-taken stack objects. 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 SafeStack. This helps to reduce performance overhead especially when a program frequently uses temporary stack variables that belong to sensitive types.

7.6 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 4KB page size, the huge page configuration uses 2MB pages for the safe memory region to reduce the number of page faults and TLB misses making safe memory access more efficient.

8 Implementation

Our HyperSpace prototype consists of 4300 lines of code in Linux Kernel, LLVM, ptmalloc2, and a library to implement the VIP API. Code instrumentation is done via a Module pass on LLVM 9.0.0 (2516 lines of code). Linux kernel 5.0.0 was modified (378 lines of code) to initialize the virtual address space of a user process for VIP by splitting the userspace virtual address into regular and safe memory regions. Also, we manually instrumented ptmalloc2 (902 lines of code) with VIP primitives to guarantee the integrity of heap metadata. The VIP library consists of 305 lines of code.

We note that HyperSpace executables are compatible with uninstrumented shared libraries to a certain extent. Uninstrumented libraries with read-only operations are fully compatible; accessing sensitive pointers from un-instrumented libraries will only access data from the regular memory because there are no VIP primitives. Un-instrumented libraries that perform write operations will prompt a crash by vip_assert before instrumented code uses the sensitive pointer as its safe memory counter part cannot be updated without VIP primitives. If such a use-case is required (i.e., library code needs to modify a sensitive pointer), the library should be recompiled with HyperSpace such that it is also instrumented.

9 Evaluation

We first evaluate how effectively HyperSpace can prevent real-world attacks by enforcing the value invariant property (§9.1). Next, we evaluate the efficiency of HyperSpace applications described in §6 using SPEC CPU 2006 and two real-world applications (§9.2). Lastly, we analyze the impact of our optimization techniques (§9.3) as well as the memory overhead of HyperSpace.

All applications were run on a 24-core server equipped with two Intel Xeon Silver 4116 processors (2.10GHz) and 128GB DRAM. All benchmarks were compiled with LLVM SafeStack [75]. Additionally, GNU gold v2.29.1-23.fc28 is used for linking to enable LLVM LTO.

9.1 Security Experiments

We evaluated all security applications described in §6, with three real-world exploits and six synthesized attacks.

9.1.1 Real-World Exploits

We first collected three publicly available exploits against three vulnerable programs.
We used a C++ test suite [53] released (1) CFIXX C++ Test Suite. This is a buffer overflow in (3) CVE-2014-1912. This is a heap-based buffer overflow in (2) CVE-2015-8668. This is a heap-based buffer overflow in llibtiff, an image file format library. This exploit allows remote attackers to execute arbitrary code. A malicious BMP file causes an integer overflow followed by a heap overflow that overwrites a function pointer in a TIFF structure. VIP-CFI/CPI successfully detects this exploit [19] by asserting the corrupted function pointer before use.

(3) 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. VIP-CFI/CPI blocks this exploit [66] by detecting the corruption of the function pointer before use.

9.1.2 Synthesized Exploits We used synthesized exploits to demonstrate how HyperSpace can defend VTPtr hijacking in C++ objects, COOP attacks [64] – a Turing complete attack via creating fake C++ objects – and heap exploits.

(1) CFIXX C++ Test Suite. We used a C++ test suite [53] released by Burow et al. [7]. 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 the VTPtr using memcpy. This modification is required for the attack to be accurately replicated and does not affect the success or failure of the attack. Without this modification, HyperSpace would determine this memcpy to be legitimate since it would be interpreted as the programmer’s intention. Our modification is inspired by a synthesized exploit in OS-CFI [42]. VIP-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. VIP-VTPtr prevents this exploit by detecting that the VTPtr of the fake object is not initialized as sensitive data and raises an exception before the virtual function call.

(2) Heap Exploit. To evaluate heap metadata protection, we used an exploit from [17], which overwrites the inline metadata of an allocated heap memory. HyperSpace thwarts this attack by detecting the metadata corruption upon free of a victim memory chunk.

9.2 Performance Evaluation We evaluate the performance overhead of HyperSpace security mechanisms described in §6 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 7 shows performance overhead compared to an unprotected original baseline build running on the original kernel. The average numbers reported here are geometric means.

9.2.1 Performance Overhead of SPEC CPU2006

(1) Heap Metadata Protection. The performance overhead for heap metadata protection with HyperSpace is 1.33% overall as Figure 7 shows. Three benchmarks have more than a 5% overhead. These three benchmarks heavily call malloc and free. For example, 471.omnetpp calls malloc and free over 534 million times combined. In addition to omnetpp, perlbench, dealII, and xalancbmk call malloc and free millions of times as well, which explains this overhead. This is consistent with results in previous work [15].

(2) VIP-CFI+VTPtr. We then evaluate CFI and C++ VTPtr protection, which together protect all code pointers of a program by enforcing VIP. The performance overhead is negligible, 0.88%. A few benchmarks show small performance improvement (1-2%) because SafeStack 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 was more frequent compared to other C++ benchmarks, resulting in higher overhead from protecting the integrity of the VTPtr.

(3) VIP-CPI. HyperSpace’s CPI protection performs well, with an average overhead of 6.18%. Two benchmarks, 400.perlbench and 453.povray, show the highest overhead. 400.perlbench accumulates overhead from frequently utilizing sensitive global variables that contain function pointers. For example, a perlbench function, Perl1_unops_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 same memory region and collects undesirable, but unavoidable overhead. As for 453.povray, most overheads are from assertions of function pointers in the 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 its runtime. HyperSpace protection recursively extends to
 pointers of objects that contain the struct Method_Struct. These chains of pointers require VIP instrumentation throughout the benchmark resulting in unusual overhead. Recent works, ERIM [78] and IMIX [27], also attempted to utilize MPK for protecting the metadata store of CPI. However, they incur much more runtime overhead than VIP-CPI: the maximum overhead is 3.2x higher for ERIM and 28.5x for IMIX than VIP-CPI. Moreover, they do not reveal their runtime overhead for 400.per1bench and 453.povray, which are most likely their two highest overheads similar to VIP-CPI. This shows that our optimization techniques (especially, to reduce the MPK permission change overhead) described in §7 are effective.

(4) Summary. The performance overhead of HyperSpace for SPEC CPU2006 is negligible: 0.88% for VIP-CFI protection and 6.18% for VIP-CPI protection. In comparison, current state-of-the-art defense techniques like Code-Pointer Integrity [45] and µCFI [36], have an average overhead of 8.4% and 7.88% with the worst-case overhead of 44% and 49%, respectively.

9.2.2 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. The numbers reported here are latency at 24-cores. We configured benchmark clients – wrk and pgbench – to fully stress the server.

(1) NGINX. We evaluate the performance of NGINX using an HTTP benchmarking tool wrk [30]. wrk spawns threads that send requests for a 6,745-byte static HTML page and measures the latency and request throughput (req/sec). We ran wrk with 24 threads with each thread handling 50 HTTP connections to fully stress the server. We ran the host and server on two separate machines with a 100Gbps Ethernet connection. The performance overhead is negligible: heap metadata, CPI and CPI protections impose 1.38%, 0.44% and 1.05% of overhead, respectively. The request throughput was 2.32K, 2.33K and 2.32K req/sec, respectively, compared to a 2.35K req/sec baseline. In addition, we evaluated NGINX with SSL enabled. The results confirm that the performance overhead with SSL is still negligible: 1.42%, 0.47% and 1.09% for heap metadata, CPI and CPI, respectively. That is because while SSL is CPU-intensive, it does not have many sensitive pointers. The request throughput was 2.06K, 2.08K and 2.07K req/sec, respectively, compared to a 2.09K req/sec baseline.

(2) PostgreSQL. To evaluate the performance of PostgreSQL, we used pgbench [76], which repetitively runs concurrent database sessions that handle a sequence of SQL commands to measure the average transaction rate and latency. We ran the host and server on the same machine. We tested PostgreSQL with 24 concurrent database clients. PostgreSQL shows negligible performance impact of 1.30%, 1.96% and 2.04% for heap metadata, CPI, and CPI protections, respectively. Concretely, PostgreSQL showed 1696, 1685, and 1683 transactions-per-second, respectively, compared to a 1719 baseline.

9.3 Performance Analysis

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

Figure 8: Impact of the performance optimization techniques described in §7. (SLNT: runtime silent store elimination; CBB: basic block-level coalescing; CFN: function-level coalescing; INLN: inlining VIP APIs; SS: safe stack; HGP: huge page).

9.3.1 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 VIP-CPI. Figure 8 shows the impact of each technique for SPEC CPU2006.

(1) Runtime Silent Store Elimination (SLNT). HyperSpace reduces costly wpkru instructions by eliminating unnecessary, repetitive modifications of the safe memory region. If an object has already been registered and its value is the same (i.e., silent store), HyperSpace skips unnecessary, repetitive vip_register and vip_write calls to reduce the costly MPK permission changes. This optimization improves performance by 6.6% on average. In particular, it improves the performance of 453.povray by 75%.

(2) Coalescing Permission Changes within a Basic Block (CBB). Coalescing permission changes within a basic block improves performance 2.17% on average by minimizing the number of permission changes. 453.povray is improved 9.8% for having an abundant number of sensitive object pointers that are often updated.

(3) Coalescing Permission Changes within a Function (CFN). Extending coalescing to function scope improves performance 1.4% on average. 464.h264ref, 458.sjeng, and 460.per1bench have higher performance gain of 7.2%, 6.8%, and 3.5%, respectively, due to having commonly used functions such as Perl1_lnklist recognized as a safe function.

(4) Inlining VIP Functions (INLN). Inlining 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 VIP calls.

(5) Excluding Objects in Safe Stack (SS). Leveraging SafeStack, HyperSpace does not need to instrument safe objects. This improves performance by 8.5% on average. SafeStack 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.nm (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.

(6) Optimizing Safe Memory Access (HGP). Last but not least, using huge pages for the safe memory region improves performance by 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.hammer improves 63.4% with this change.
Also, Intel CET [38] – a hardware-based secure shadow stack – is available in mobile processors [37] and shows lower overhead. We expect that hardening the current software-based Safe Stack using OS-CFI+ [8] shows that a stronger isolation guarantee for shadow stack is possible using Intel MPX (Memory Protection eXtension) [58] with a moderate performance overhead. Also, Intel CET [38] – a hardware-based secure shadow stack – is available in mobile processors [37] and shows lower overhead. We expect that hardening the current software-based Safe Stack using OS-CFI+.

9.3.2 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 9 shows, VIP introduces around 13% memory overhead: with heap metadata protection enabled, VIP required 12.16% more memory for VIP-CFI-VTPtr and 13.18% for VIP-CPI on average, respectively. VIP’s memory overhead is much smaller than other state-of-the-art defense mechanisms (e.g., 105% for the original CPI [45]). The reason for such memory overhead in state-of-the-art techniques such as CPI is due to them having bigger metadata for each sensitive pointer. A few benchmarks have relatively high memory overhead because these benchmarks have many sensitive types, so they tend to generate and access more sensitive pointers. For example, in omnetpp, many data pointers are sensitive pointers because they can eventually reach a code pointer. However, we found that this is uncommon.

10 Discussions & Limitations

Protecting Adversarial Misuse of MPK. A hypothetical attack against HyperSpace is that MPK instructions could be adversarially misused. Because all MPK instructions, including wrpkru, are unprivileged instructions, if an attacker could subvert the control flow and change the MPK permission of the safe region to read-writable, then she is able to bypass HyperSpace defenses. However, such an attack is unfeasible if the control flow is protected by HyperSpace’s control flow hijacking defenses (e.g., VIP-CFI/CPI).

Limitation of Safe Stack. Safe Stack is one of our six optimization techniques for VIP-CFI/CPI. Our current implementation relies on software-based randomization, so it is susceptible to information leakage attacks [23, 32, 57, 86]. However, it can be further hardened by using a more secure shadow stack implementation. For instance, Burow et al. [8] shows that a stronger isolation guarantee for shadow stack is possible using Intel MPX (Memory Protection eXtension) [58] with a moderate performance overhead. Also, Intel CET [38] – a hardware-based secure shadow stack – is available in mobile processors [37] and shows lower overhead. We expect that hardening the current software-based Safe Stack using those new hardware features will guarantee stronger protection in HyperSpace with lower performance overhead.

Concurrent access and TOCTOU. Accessing the safe memory region in HyperSpace needs to be protected against race conditions. On one hand, changing memory permissions of the safe region using MPK does not suffer from concurrent access issues because MPK registers are assigned per-CPU (and thereby per-thread). On the other hand, changing of data values in the safe region may suffer from data race issues; however, this can be protected by making VIP primitives atomic for update operations (vip_register, vip_unregister, vip_write, vip_write_final).

Extending VIP Protection to Other Security-sensitive Data. We believe that VIP can be effectively extended to protect other security-sensitive data beyond code/data pointers and heap metadata, which we focused on in this paper. The foundation of VIP is to correctly identify security-sensitive data along with its value invariant period. Once such analysis is available, HyperSpace can work as a framework, i.e., we can plug in such analysis to enable VIP protection of new security-sensitive data. We plan to further explore the automatic analysis of sensitive non-control data, thereby, a wider range of VIP-based defenses can be automatically applied.

11 Related Work

VIP is a framework and encompasses several policies as we discussed. In the following, we discuss previously proposed mechanisms and how VIP policies compare with them.

11.1 CFI

Protection Target/Method. Control-flow Integrity relies on a program’s legitimate control-flow information. For most methods, this is done by constructing a control-flow graph (CFG) which showcases the control-flow information. Thus, by conforming to the CFG, CFI allows only legitimate control-flow transition at all indirect call/jump and return sites of the program.

Technique. Different CFI methods have different ways for determining the control-flow information and constructing the CFG. Although there is only one legitimate target that should be allowed at each call/jump site at a specific runtime, these methods inherently have a large equivalence class (EC) size (i.e., number of allowed legitimate targets at one call site). A state-of-the-art implementation of CFI, OS-CFI [42], has an indirect call site that allows 427 legitimate call targets in SPEC CPU2006. For such a call site, attackers may exploit existing attacks such as CFB [9] and COOP [64]. In contrast, VIP can always guarantee the EC size to be 1 because the legitimate pointer that can be used is immutable after its assignment.

Runtime checking. Approaches that check runtime program data [29, 35, 36, 47] can further restrict the EC size, as small as 1 (μCFI [36]), which allows only the legitimate call target at a specific runtime. However, these approaches require running additional threads to parse the data from Intel Processor Trace (PT) and apply runtime analysis, limiting scalability. In contrast, HyperSpace does not require running additional threads for protection.

Overhead. Recent CFI implementations incur a little runtime performance overhead. OS-CFI incurs 7.1% and μCFI incurs 9.9% (and dedicating 1 CPU core for trace analysis) of performance overhead while running the SPEC CPU2006 benchmark. VIP-CPI incurs less overhead (6.18%) while guaranteeing better security.

11.2 OTI

Protection Target. Object Type Integrity (OTI) [7] aims to provide protection to the virtual function table pointer of a C++ object.

Technique. By storing metadata on object construction and checking the metadata at a virtual function call site, OTI can enforce the
type integrity to C++ objects at runtime. To protect its metadata, the metadata storage of OTI is protected by Intel Memory Protection eXtension (MPX). Similarly, VIP protects its metadata with HyperSpace via Intel MPK.

**Overhead.** OTI incurs 4.98% of performance overhead in the SPEC CPU2006 benchmark. VIP offers better performance. Our evaluation of VIP-CPS+VTptr incurs only 0.88% of performance overhead.

### 11.3 CPI

**Protection Target.** Code Pointer Integrity (CPI) [45] aims to enforce the integrity of sensitive pointers.

**Technique.** By defining sensitive pointers as code pointers and pointers that refer to sensitive pointers recursively, CPI protects the program by isolating all such sensitive pointers from attackers. In isolating memory space for storing sensitive pointers, CPI randomly allocates an address space and hides it from attackers. Such a technique, information hiding, is proven to be susceptible to recent attacks [23, 32, 57]. In contrast, VIP/HyperSpace provides memory space isolation protection and tightened security of the metadata based on hardware permission control via Intel MPK.

**Overhead.** CPI incurs 8.4% performance overhead, while its lighter-weight alternative, Code Pointer Separator (CPS), incurs 1.9% overhead in the SPEC CPU2006 benchmark. VIP incurs lesser overhead than CPI/CPS for their counterpart implementation; VIP-CFI incurs 0.88% and VIP-CPI incurs 6.18% of performance overhead.

**Comparison with VIP.** As mentioned in §2.3, we follow the same definition of a sensitive pointer as in CPI. However, VIP has several advantages that overcome the critical limitations in CPI and other recent efforts that use Intel MPK to enhance the security of CPI [27, 78]. The main limitation in CPI is the reliance on information hiding to protect its safe region [23, 32, 57]. Our optimized use of MPK resolves this. The naive use of MPK results in huge performance overheads, as demonstrated by ERIM [78] and IMIX [27] in their CPI+MPK evaluation (up to 320% for ERIM and up to 2856% for IMIX). We have several optimizations (§7) that reduce the performance overhead, and guarantee better security. Beyond defeating control-flow hijacking attacks, we go a step further and protect heap metadata which is not considered in CPI. ARM’s Pointer Authenticaton Code (PAC) [63] enforces its protection by encoding the unused bits of the pointer with a cryptographic hash. PAC only exists on the ARM architecture. We designed VIP for x86_64 architecture, which does not have a hardware mechanism like PAC.

### 11.4 Heap-metadata protection

**Protection Target/Method.** One of the ways to secure the heap metadata is through secure memory allocators. FreeGuard [69] combines techniques from free-list allocators and BIBOP (Big Bag of Pages) by acquiring a large block and dividing it into multiple sub-heaps. Guarder [70] is similar in design to FreeGuard but focuses on tunable security guarantees and enhancing randomization entropy.

**Heap metadata hardening.** In spite of the fact that the metadata could be fully isolated, the metadata is not protected. Thus, it relies on randomness and information hiding. Even though the attacker would have more difficulty carrying out the attack, it is still possible. This is due to the fact that the relationship between heap objects and its metadata is deterministic. VIP-heap protection can be used as an extension to harden allocators, to protect the metadata.

### 11.5 Memory Safety

**Protection Target/Method.** Memory safety techniques stop memory corruption attacks by enforcing spatial and temporal safety. SoftBound [50] protects against spatial memory attacks by storing the bounds of every pointer as disjoint metadata. CETS [51] protects against temporal memory attacks by storing a unique identifier with each object. BOGO [85] reuses the bounds stored by Intel MPX [58] to achieve temporal safety by scanning the MPX bound tables and invalidating the bounds of dangling pointers.

**VIP scope.** Full memory safety solutions incur significantly high overhead (≈116% for SoftBound+CETS and ≈60% for BOGO). VIP strikes the balance between practicality and security by offering a specialized scope of memory safety. By protecting security-sensitive data, critical memory corruption attacks, particularly control-flow hijacking and heap metadata attacks, can be thwarted whilst maintaining relatively low overhead. This is more feasible than full memory safety while providing strong security guarantees.

### 11.6 Non-Control Defenses and Protecting the Safe Region

Several defense mechanisms have been proposed that protect non-control data or the metadata of other defense mechanisms (referred to as the safe region). xMP [62] isolates sensitive data into domains, leveraging Intel virtualization extensions. However, xMP [62] requires heavy kernel modifications for its domains. DCI [10] separates the memory into two regions. It does bound-checking on sensitive data and prevents pointers of non-sensitive data from being dereferenced if they point to sensitive data. DCI [10] relies on the programmer’s annotations to identify sensitive data. MemSentry [44] is a framework to enhance safe region isolation and harden modern defense mechanisms, but does not utilize MPK. Overall, MemSentry has a higher overhead than HyperSpace and does not protect VTable pointers or heap metadata. In contrast, VIP-CPI is fully automatic without requiring manual annotation and HyperSpace requires minimal kernel modifications.

### 12 Conclusion

We have introduced the Value Invariant Property (VIP), which is a common property of security-sensitive data for critical memory corruption attacks. Our main focus is defending against two of the most critical memory corruption attacks by securing security-sensitive data – code/data pointers and heap metadata – to thwart control-flow hijacking and heap metadata corruption attacks with a defense offering low performance and memory overhead. We then introduced HyperSpace, a prototype that protects VIP, and implemented four security mechanisms. Our evaluation results show that HyperSpace incurs low performance and memory overhead: an average performance overhead of 0.88% and 6.18% for CPI+VTptr and CPI, respectively, and 13.18% memory overhead for SPEC benchmarks and real-world applications. Our security experiments using three real-world exploits and six synthesized attacks show the effectiveness of VIP and HyperSpace.

### Acknowledgment

We thank the anonymous reviewers and our shepherd, Georgios Portokalidis, for their insightful comments and feedback. This work is supported in part by the US Office of Naval Research under grants N00014-18-1-2022.
Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict Protection for


