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

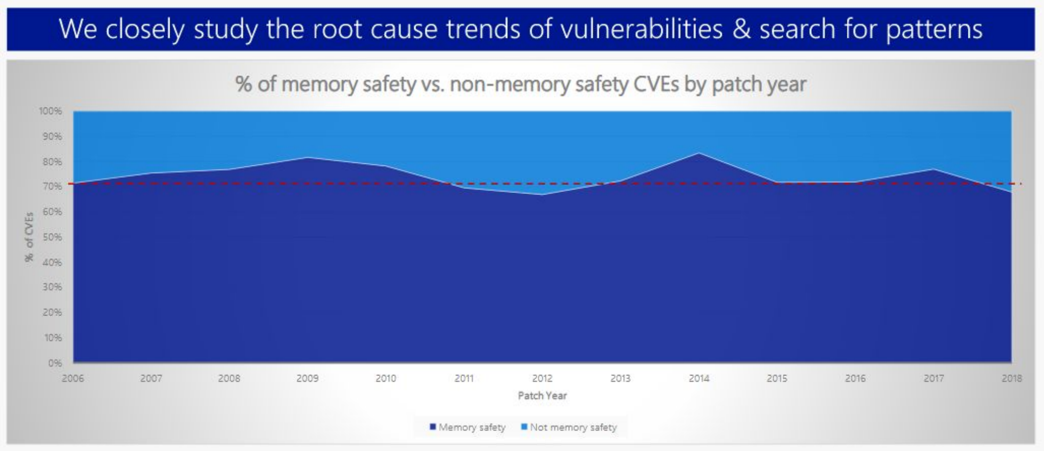
Mohannad Ismail (Virginia Tech), Jinwoo Yom (Virginia Tech), Christopher Jelesnianski (Virginia Tech), Yeongjin Jang (Oregon State University), Changwoo Min (Virginia Tech)

CCS '21: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security

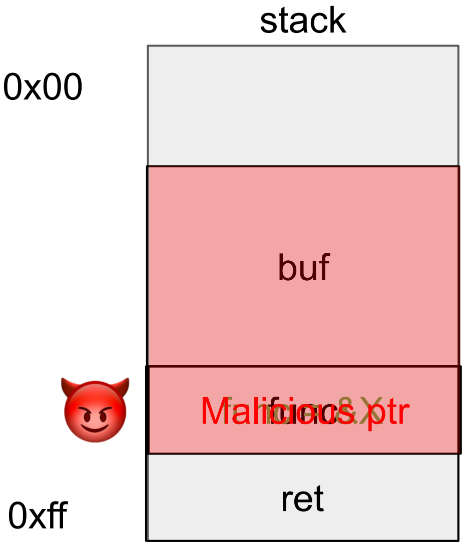
Memory corruption vulnerability is root of all EVIL

- Microsoft reported that **70%** of all security bugs are due to various **memory safety** issues.

- Top three memory corruption attacks:
    - Heap out-of-bounds.
    - Use-after-free.
    - Type confusion.

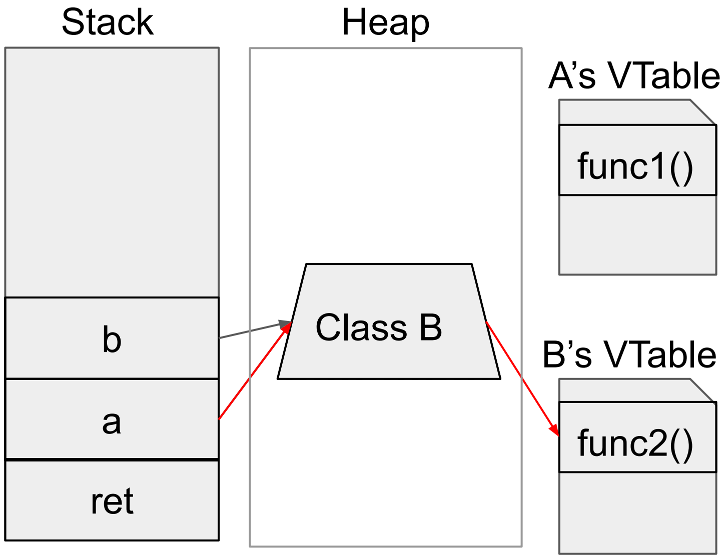


Vulnerable Control Data Example




```c
/** == An example of a code pointer corruption attack ========= */
void X(char *); void Y(char *); void Z(char *);

typedef void (*FP)(char *);
static const FP arr[2] = {&X, &Y};

void handle_req(int uid, char * input) {
  FP func; // control data to be corrupted!
  char buf[20]; // buffer that may overflow

  if (uid<0 || uid>1) return; // only allows uid == 0 or 1

  func = arr[uid]; // func pointer assignment, either X or Y.

  strcpy(buf, input); // stack overflow corrupting a code pointer!!!

  (*func)(buf); // func is corrupted!

}
// END
```

C++ VTable pointer Use-after-free example




```cpp
class A {
public:
    virtual void func1() {};
};
class B {
public:
    virtual void func2() {};
};

int main() {
    A *a = new A();

    delete a;

    B *b = new B();

    a->func1();
}
```

Problems with state-of-the-art techniques

- Incur high overhead (e.g., DFI [OSDI06]):
  - Frequent metadata lookup.
  - Excessive instrumentation.

- Require additional resources (e.g., uCFI [CCS18]):
  - Dedicated CPU cores for background analysis.

- Are narrow-scoped defense (e.g., OTI [NDSS18]):
  - Needs to be used orthogonally with other defense techniques to provide stronger security.

Breaking an essential step in memory corruption attacks

Solution

STEP 1 → STEP 2 → STEP 3 → STEP 4

Read program layout.

Locate sensitive data (e.g., code pointers).

Corrupt sensitive data.

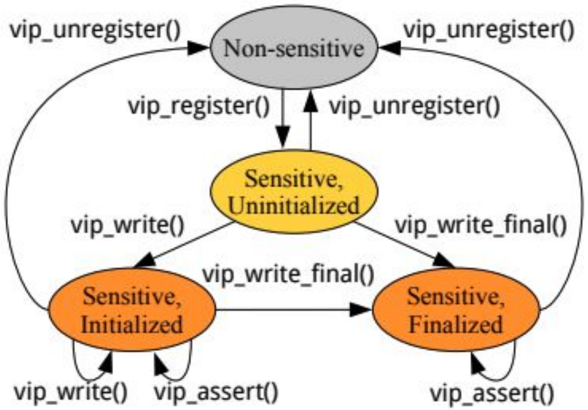
~~Use corrupted data~~
Do NOT allow attackers to use the corrupted data.

Outline

1. Introduction
2. **Value Invariant Property (VIP)**
3. HyperSpace
4. Implementation
5. Evaluation
6. Discussion
7. Conclusion

Overview of Value Invariant Property (VIP)

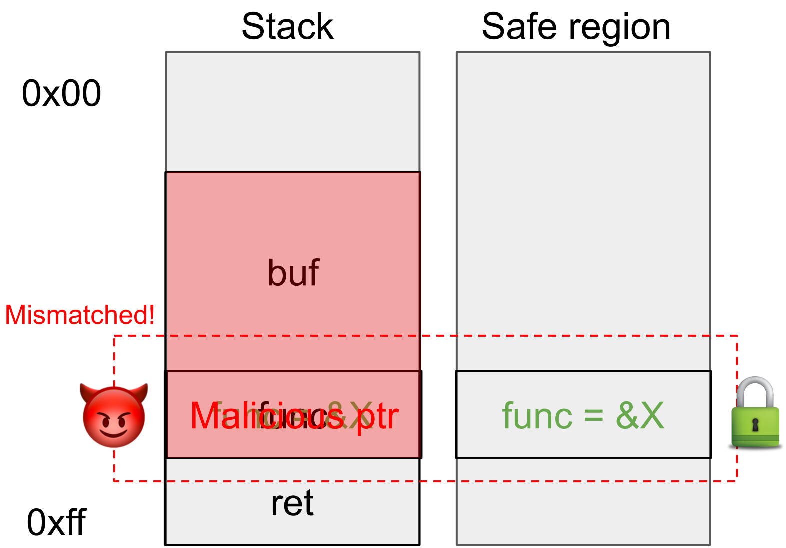
- Our intuition behind VIP originates from a common pattern in programs:
  - *Security-sensitive data should never be changed between two legitimate writes so there is a period such that security-sensitive data is immutable.*
- This period is represented by the state transition diagram, that relies on VIP primitives to enforce value integrity of security-sensitive data.



Overview of Value Invariant Property (VIP)

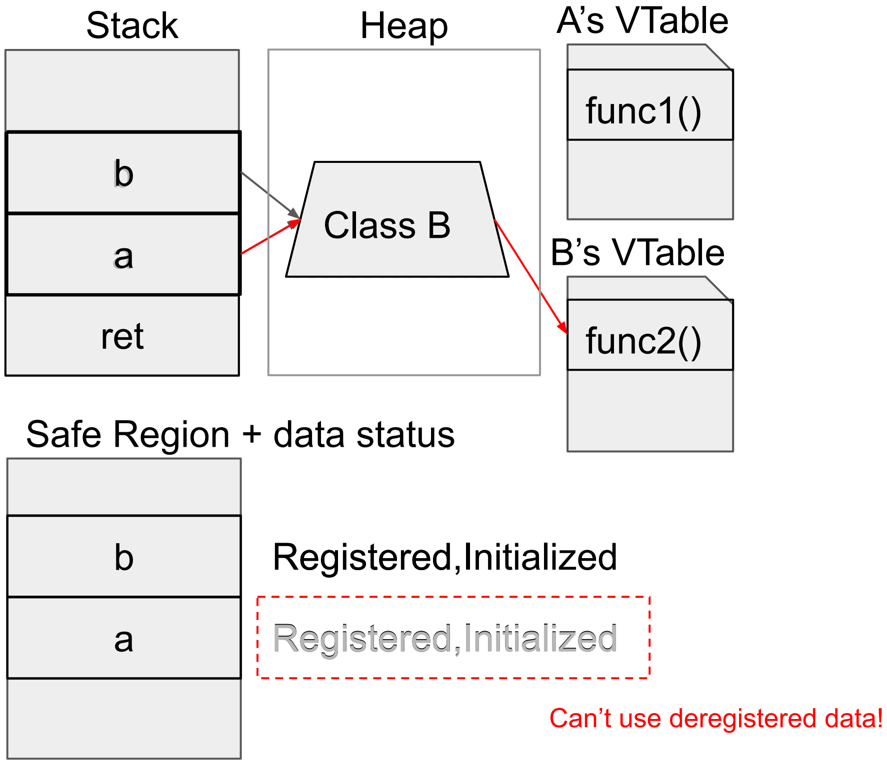
- VIP maintains a shadow copy of sensitive data in an isolated “safe” region.
- VIP checks and verifies value integrity instead of tracking control-flow.
- To provide value integrity, VIP checks if the “value” of sensitive data is corrupted or not.
- If corruption is detected, VIP will raise a security exception.
- Compromised application is halted and prevented from executing further.

Main concept of VIP for Control data




```
/** Example of a control data corruption attack */
void X(char *); void Y(char *); void Z(char *);

typedef void (*FP)(char *);
static const FP arr[2] = {&X, &Y};

void handle_req(int uid, char * input) {
  FP func; // control data to be corrupted!

  char buf[20];

  if (uid<0 || uid>1) return; // only allows uid == 0 or 1

  func = arr[uid]; // func pointer assignment, either X or Y.

  strcpy(buf, input); // stack buffer overflow!

  (*func)(buf) // validation failed, program aborted!

}
```

Main concept of VIP for C++ VTable use-after-free

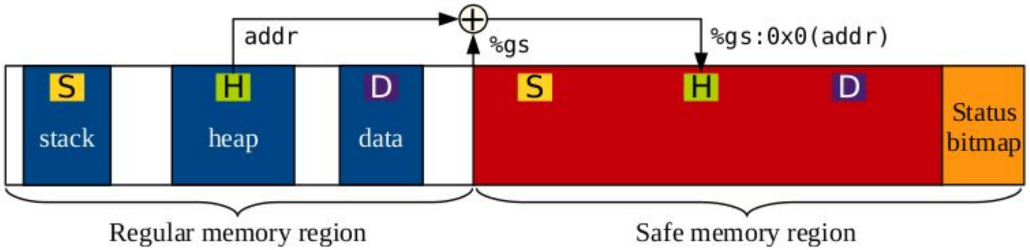



```cpp
class A {
public:
    virtual void func1() {};
};
class B {
public:
    virtual void func2() {};
};

int main() {
    A *a = new A();

    delete a;

    B *b = new B();

    a->func1();
}
```

VIP safe memory region

The safe memory region is created by VIP’s modified Kernel.

- Application’s memory space is bisected into regular and safe memory regions.
- Status bitmap region holds the status bits for data that exists in the safe memory region.
- 8 byte in safe memory => 2 bits in status bitmap.
- Maximum memory overhead is bounded to 103.1%.
- `%gs` register holds the start address of safe region and used for fast safe region access.



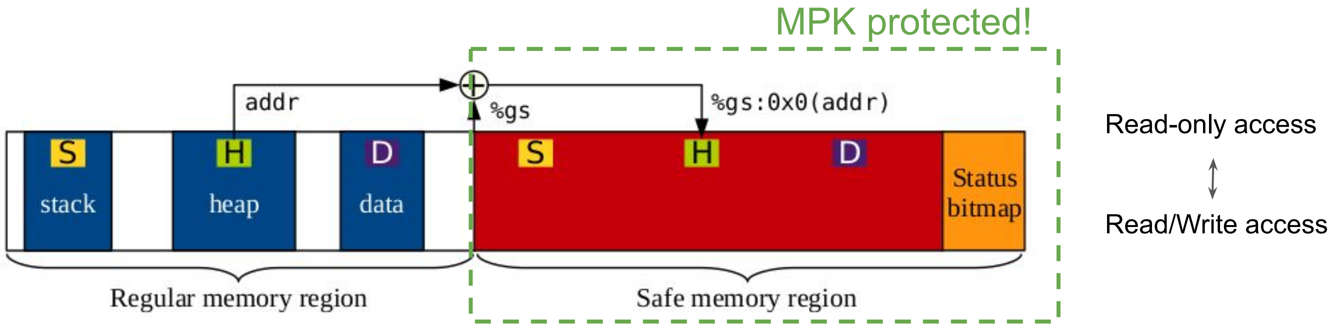
Security of the safe memory region

- The main issue with the use of the safe region is how to secure it from being maliciously corrupted.
- Previous works, such as CPI [1], relied on information hiding.
- The main challenge is ensuring that the only way to access the safe region is through the legitimate program logic.
- In order to overcome this challenge, we rely on Intel's Memory Protection Keys (MPK).
- By leveraging MPK, we can unlock the safe region when access is needed by the program, and then lock it to prevent illegitimate access.

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

Memory Protection Keys (MPK)

- Intel's new hardware primitive.
- Utilizes previously unused 4 bits in each page table for upto 16 different fine-grained access control keys.
- New user-accessible register (PKRU) with Access/Write disable bits for each key.
- PKRU is a CPU register; thus, is thread-local.
- Two new instructions:
  - `rdpkru` - for reading page permission.
  - `wrpkru` - for modifying page permission.

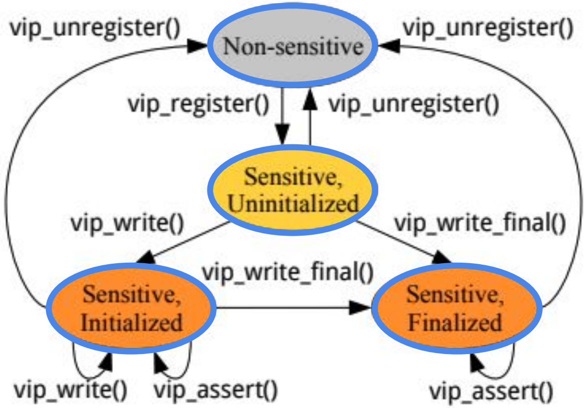


Outline

1. Introduction
2. Value Invariant Property (VIP)
3. **HyperSpace**
4. Implementation
5. Evaluation
6. Discussion
7. Conclusion

HyperSpace

- HyperSpace manages the state of a memory location.
- When a program starts, the entire memory space is in a **non-sensitive state**, meaning that no memory location stores security-sensitive data.
- HyperSpace first requires the location to be registered upon its allocation. Then, the memory will be in a **sensitive, uninitialized state.**
- Once the security-sensitive data is written to the memory location, the memory will be in a **sensitive, initialized state.**
- If we know a write should be the final one until the deallocation of the memory, then we can put the memory into a **sensitive, finalized state**, and HyperSpace does not allow any further writes to that memory location.
- These restrictions are enforced by HyperSpace with MPK.



HyperSpace

- Fully implemented prototype that enforces VIP.
- Four Defense Mechanisms:
  - Control flow integrity (VIP-CFI)
    - Protects ***all*** code pointers.
  - C++ VTable pointers protection (VIP-VTPtr)
    - Protects ***all*** virtual function table pointers (VTPtrs).
  - Code pointer integrity (VIP-CPI)
    - VIP-CFI and VIP-VTPtr protection + ***all*** sensitive object pointer protections.
  - HyperSpace heap metadata protection
    - Protect against inline heap metadata corruption attacks.

What is sensitive data?

- Sensitive data varies for each security mechanism:
  - All function pointers. <- VIP-CFI & VIP-CPI

```
void (*func_ptr)(int);

struct Foo {
  void (*func_ptr)(int);
};

void (*func_ptr2[3])(int);
```

  - VTPtrs in C++ objects. <- VIP-VTPtr & VIP-CPI
  - Sensitive Object pointers. <- VIP-CPI

```
struct Bar {
  struct Foo *foo;
};
```

```
struct Baz {
  struct Bar *bar;
};
```

- How can we detect these sensitive data?
  - Recursive static analysis/Instrumentation.

Optimization

- MPK is fast, but it's not fast enough:
  - `rdpkru` ~0.5 CPU cycles
  - `wrpkru` ~23.3 CPU cycles
- Frequent usage of `wrpkru` to modify safe region can incur significant overhead.
- There is no single optimization that significantly improved performance across all components.

- **Six major optimizations:**
  - Inlining DVI functions.
  - Not instrumenting objects in the SafeStack.
  - Runtime checks to reduce permission changes.
  - Coalescing permission changes within a Basic Block.
  - Coalescing permission changes within a safe function.
  - Huge Page enabled.

Reduces `wrpkru` usage!

Optimization: Basic-block level coalescing

- To reduce the unnecessary toggling of safe memory region permissions, we introduce an optimization technique to coalesce a series of HyperSpace protection instrumentation within a basic block.
- All memory writes in a coalescing-safe basic block are guaranteed to not be capable of corrupting arbitrary memory locations.

```c
/** == Instrumentation of consecutive writes of sensitive data ==
 *  - LISTOP is a sensitive type containing a function pointer.
 *  Thus, its two members, op_last and op_sibling, pointing to
 *  other LISTOP instances are also sensitive data. */
OP *Perl_append_list(pTHX_ I32 type, LISTOP *first, LISTOP *last){
    // ...
    first->op_last->op_sibling = last->op_first;
    // vip_safe_memory_unlock();
    //   vip_write(&first->op_last->op_sibling, 8);
    // vip_safe_memory_lock();
    first->op_last = last->op_last;
    // vip_safe_memory_unlock();
    //   vip_write(&first->op_last, 8);
    // vip_safe_memory_lock();
    first->op_flags |= (last->op_flags & OPf_KIDS);
    FreeOp(last);
    return (OP*)first;
}
```

Before

```c
/** == Coalescing permission changes in a basic block ======== */
OP *Perl_append_list(pTHX_ I32 type, LISTOP *first, LISTOP *last){
    // ...
    first->op_last->op_sibling = last->op_first;
    // vip_safe_memory_unlock();
    //   vip_write(&first->op_last->op_sibling, 8);
    first->op_last = last->op_last;
    //   vip_write(&first->op_last, 8);
    first->op_flags |= (last->op_flags & OPf_KIDS);
    // vip_safe_memory_lock();
    FreeOp(last);
    return (OP*)first;
}
```

After

Optimization: Function-level coalescing

1. All basic blocks in the function are coalescing-safe.
2. It does not contain any indirect calls.
3. All direct call targets are coalescing-safe functions.

```
/** == Instrumentation of writing sensitive data ================
 *  - OP is a sensitive type containing a function pointer.
 *  Thus, its member, op_next, pointing to another OP
 *  is also sensitive data, which needs to be protected. */
OP * Perl_linklist(pTHX_ OP *o) {
  register OP *kid;
  // ...
  if (cUNOPo->op_first) {
    o->op_next = LINKLIST(cUNOPo->op_first);
    // vip_safe_memory_unlock();
    // vip_write(&o->op_next, 8);
    // vip_safe_memory_lock();
    for (kid = cUNOPo->op_first; kid; kid = kid->op_sibling) {
      if (kid->op_sibling) {
        kid->op_next = LINKLIST(kid->op_sibling);
        // vip_safe_memory_unlock();
        // vip_write(&kid->op_next, 8);
        // vip_safe_memory_lock();
      } else {
        kid->op_next = o;
        // vip_safe_memory_unlock();
        // vip_write(&kid->op_next, 8);
        // vip_safe_memory_lock();
  } } }
  else {
    o->op_next = o;
    // vip_safe_memory_unlock();
    // vip_write(&o->op_next, 8);
    // vip_safe_memory_lock();
  }
  return o->op_next;
}
```

Before

```
/** == Coalescing permission changes in a safe function ======= */
OP * Perl_linklist(pTHX_ OP *o) {
  register OP *kid;
  // vip_safe_memory_unlock();
  // ...
  if (cUNOPo->op_first) {
    o->op_next = LINKLIST(cUNOPo->op_first);
    // vip_write(&o->op_next, 8);
    for (kid = cUNOPo->op_first; kid; kid = kid->op_sibling) {
      if (kid->op_sibling) {
        kid->op_next = LINKLIST(kid->op_sibling);
        // vip_write(&kid->op_next, 8);
      } else {
        kid->op_next = o;
        // vip_write(&kid->op_next, 8);
  } } }
  else {
    o->op_next = o;
    // vip_write(&o->op_next, 8);
  }
  // vip_safe_memory_lock();
  return o->op_next;
}
```

After

Outline

1. Introduction
2. Value Invariant Property (VIP)
3. HyperSpace
4. **Implementation**
5. Evaluation
6. Discussion
7. Conclusion

HyperSpace Implementation

LLVM Compiler pass

Custom compiler pass
for static analysis and
instrumentation
(2516 LoC)

Linux Kernel modification

Bisect the memory
layout of VIP
enabled executables
(378 LoC)

VIP API library

Library for VIP
primitives
(505 LoC)

+Additional support

ptmalloc inlining for
heap metadata
protection
(902 LoC)

Outline

1. Introduction
2. Value Invariant Property (VIP)
3. HyperSpace
4. Implementation
5. **Evaluation**
6. Discussion
7. Conclusion

Evaluation setup

- 2 x Intel Xeon Silver 4116 processor (2.10 GHz)
- 128GB DRAM
- 12 Cores
- Fedora 28 Server Edition + Linux Kernel v5.0
- Compiled using LLVM SafeStack
- Linked with GNU gold v2.29.1-23.fc28

Security evaluation with:

- 3 real-world exploits (CVEs)
- 6 synthesized attacks

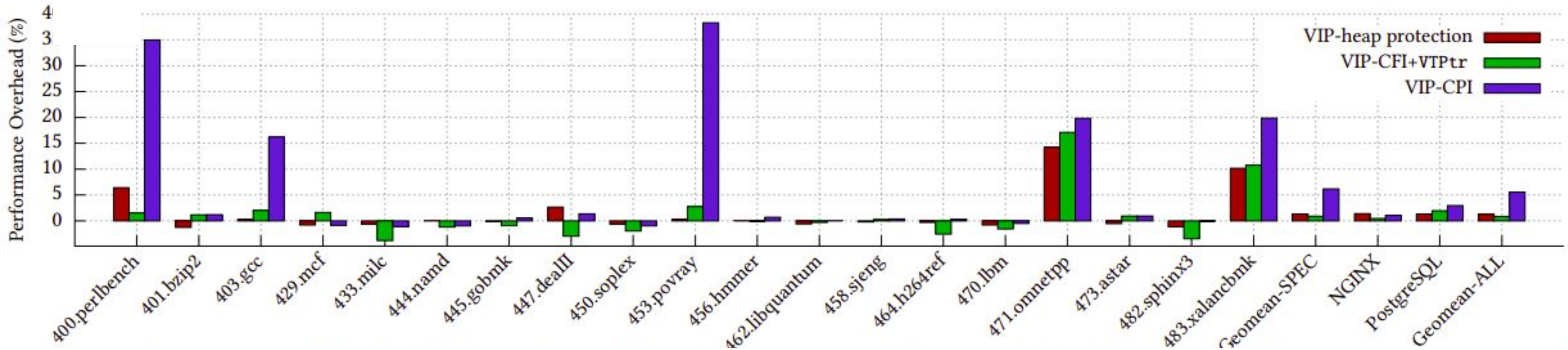
Performance/Memory evaluation with:

- SPEC CPU 2006
- NGINX (v1.14.2)
- PostgreSQL

Security Evaluation

- We evaluated HyperSpace with **three real-world exploits** and **six synthesized attacks**. These attacks demonstrate the effectiveness and versatility of HyperSpace.
- Real-world exploits:
  - CVE-2016-10190 : Heap-based buffer overflow in ffmpeg.
  - CVE-2015-8668 : Heap-based buffer overflow in libtiff.
  - CVE-2014-1912 : Buffer overflow in Python2.7.
  - **All prevented when using VIP-CFI/CPI, since exploit occurs on sensitive pointers.**
- C++ VTPtr synthesized attacks:
  - CFIXX C++ test suite released by Burow et al (VTPtr hijacking attacks).
  - COOP attack.
  - **All prevented using VIP-CFI/CPI protection since we protect the VTPtrs**
- Synthesized heap attack:
  - Overwrite inline metadata of an allocated heap memory during "unlink"; while removing a memory chunk.
  - **HyperSpace's heap metadata protection can defend this, since we write/assert the metadata during all malloc/free functions.**

Runtime overhead of HyperSpace



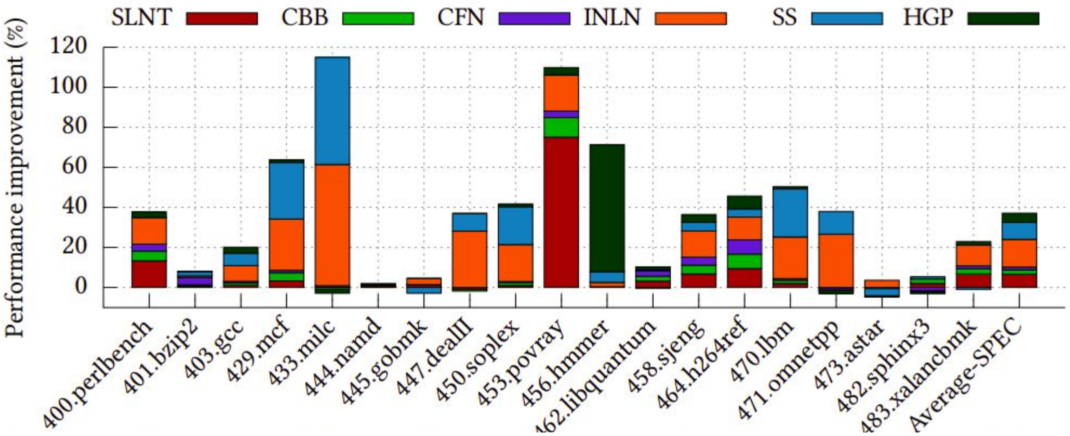
HyperSpace

Heap Metadata Protection
- Average: 1.40%
- Median: -0.23%

VIP-CFI + VTPtr
- Average: 1.02%
- Median: 0.23%

VIP-CPI
- Average: 6.35%
- Median: 0.67%

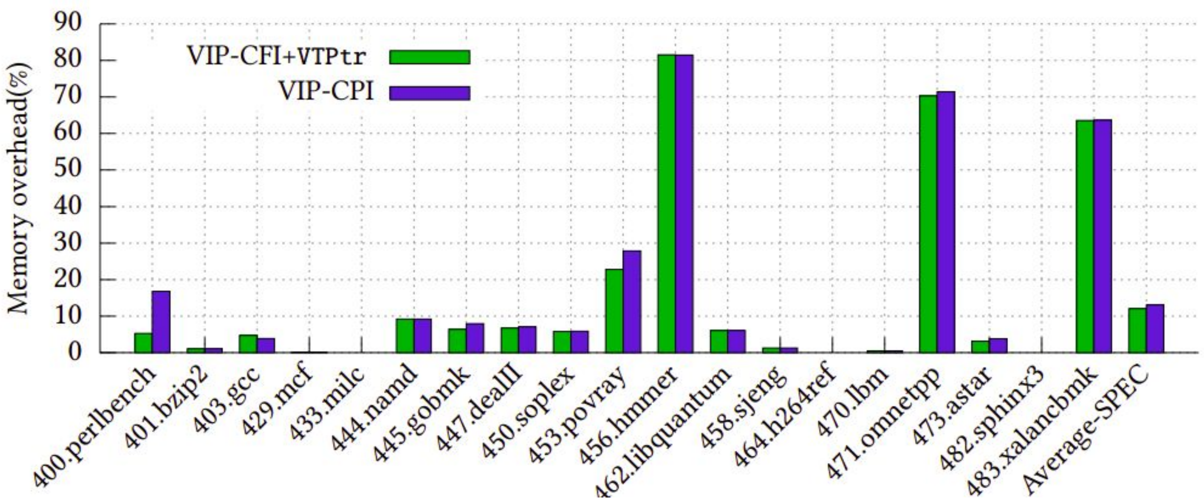
Impact of the optimization techniques



Optimizations

- INLN = DVI API inlining
- SS = SafeStack
- RNT = Runtime permission check
- CBB = Basic Block-level coalescing
- CFN = Function-level coalescing
- HGP = Huge Page

Memory overhead of HyperSpace



HyperSpace

VIP-CPI
- Average: 15.47%
- Median: 5.88%

VIP-CFI+VTPtr
- Average: 14.42%
- Median: 5.27%

Outline

1. Introduction
2. Value Invariant Property (VIP)
3. HyperSpace
4. Implementation
5. Evaluation
6. **Discussion**
7. Conclusion

Discussion

- **Could MPK be 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).

Comparison with the state-of-the-art

- **Control Flow Integrity (CFI):**
  - OS-CFI [1] incurs 7.1% and μCFI [2] incurs 9.9% performance overhead running SPEC CPU2006 Benchmark suite (while also requiring one dedicated CPU core for trace analysis).
  - VIP-CPI incurs less overhead (6.18%) while guaranteeing better security and does not require running additional threads for protection.
- **Code Pointer Integrity (CPI):**
  - CPI [3] suffers from reliance on information hiding to protect its safe region.
  - VIP's optimized use of MPK solves this. We have several optimizations that reduce the performance overhead, and guarantee better security.
  - VIP also goes a step further by protecting heap metadata which is not considered in CPI.
- **Object Type Integrity (OTI):**
  - OTI [4] incurs 4.98% of performance overhead in the SPEC CPU2006 benchmark.
  - VIP offers better performance as well as greater protection coverage. Our evaluation of VIP-CFI+VTPtr incurs only 0.88% of performance overhead.

[1] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. 2019. Origin-sensitive Control Flow Integrity. In Proceedings of the 28th USENIX Security Symposium (Security).
[2] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS).
[3] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI).
[4] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. 2018. CFIXX: Object Type Integrity for C++ Virtual Dispatch. In Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS).

Outline

1. Introduction
2. Value Invariant Property (VIP)
3. HyperSpace
4. Implementation
5. Evaluation
6. Discussion
7. **Conclusion**

Conclusion

- Value Invariant Property (VIP) is a new defense policy that provides a versatile and elegant solution to thwarting memory corruption exploits.
- Our prototype, HyperSpace, enforces VIP to provide various security mechanisms with the strongest guarantee (VIP-CPI) having 6.35% runtime overhead and 15.47% memory overhead.
- Contributions:
  - VIP.
  - HyperSpace.
  - Optimization of HyperSpace.
  - Thorough evaluation of HyperSpace.

Thank You !