HyperSpace: Data-Value Integrity for Securing Software

Jinwoo Yom

Master's Thesis Defense

~~Address Space Layout Randomization (ASLR)~~

Return-to-libc

~~Data-Execution Prevention (DEP)~~

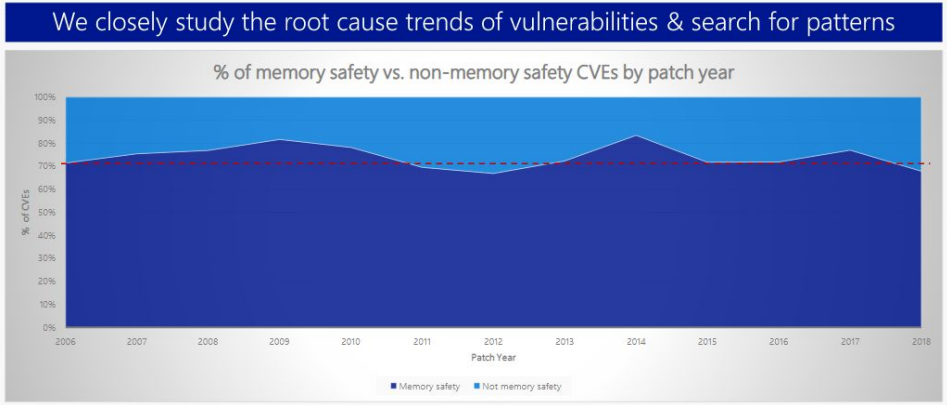
Just-In-Time Return Oriented Programming (JIT-ROP)

~~Stack Canary~~

Blind Return Oriented Programming (BROP)

Memory corruption vulnerability is root of all EVIL

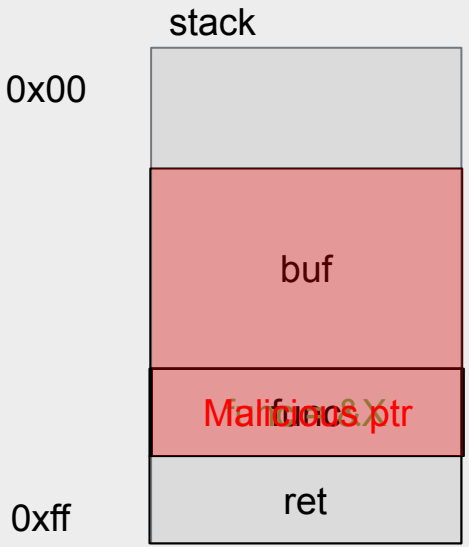
- In 2019, Microsoft reported that 70% of all security bugs are due to various memory safety issues
- Top three memory corruption attacks:
  - Heap out-of-bounds
  - Use-after-free
  - Type confusion



Code Examples

- Control data corruption
  - Control data directly impact the flow of the code (ex, code pointers)
- Non-control data corruption
- Use-after-free

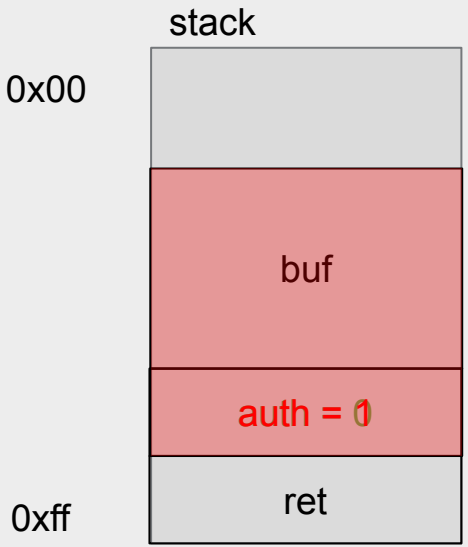
Vulnerable Control Data Example



uid = 0
input=<longer than 20 characters>

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

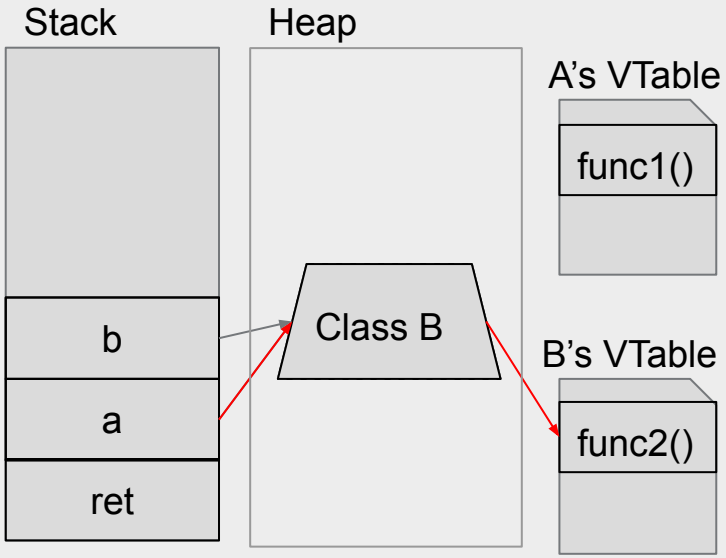
Vulnerable Non-Control Data Example



Input = <invalid packet w/ more than 1000 characters>

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

Use-after-free example




```
 5  class A {
 6  public:
 7      virtual void func1() {};
 8  };
 9  class B {
10  public:
11      virtual void func2() {};
12  };
13
14  int main() {
15      A *a = new A();
16
17      delete a;
18
19      B *b = new B();
20
21      a->func1();
22  }
```

Related work

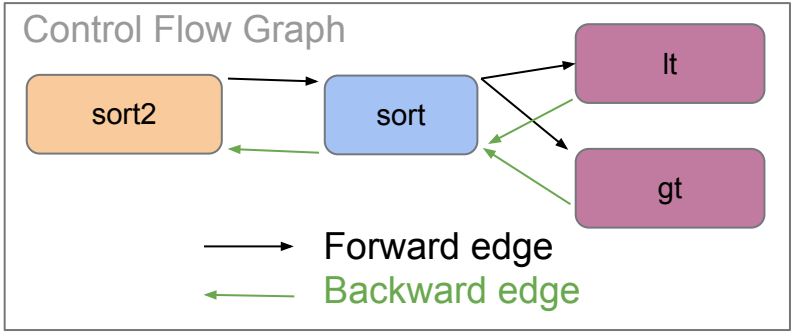
Control Flow Integrity (CFI) [CCS05]

Computes the call/return Control Flow Graph (CFG)

- Builds Equivalence Classes (EC)
- Monitors indirect calls (jmp, call, ret)

- Fairly low-overhead
- Does not defend non-control data attack
- Suffers from over-conservative precision due to large equivalence classes

```
3   bool lt(int x, int y) {
4       return x < y;
5   }
6   bool gt(int x, int y) {
7       return x > y;
8   }
9   sort2(int a[ ], int b[ ], int len)
10  {
11      sort( a, len, lt );
12      sort( b, len, gt );
13  }
```



Related works (cont.)

Data Flow Integrity (DFI) [OSDI06]

- Instruments every store operations with an ID
- Instruments every load instruction to validate the last writer of the data to be within allowed sets

- Can protect both control & non-control data
- High overhead (~104%)

Code Pointer Integrity (CPI) [OSDI14]

- Isolates code pointers into a safe region
- Safe region is protected via information hiding

- Efficiently enforce memory safety of code pointers
- Safe region can be discovered & corrupted
- Does not protect non-control data

Object Type Integrity (OTI) [NDSS18]

- Protects object types by recording it into a separate metadata table during compile time.

- Narrowly scoped. (eg. only protect VTPtr in C++ objects)

Problems with state-of-the-art techniques

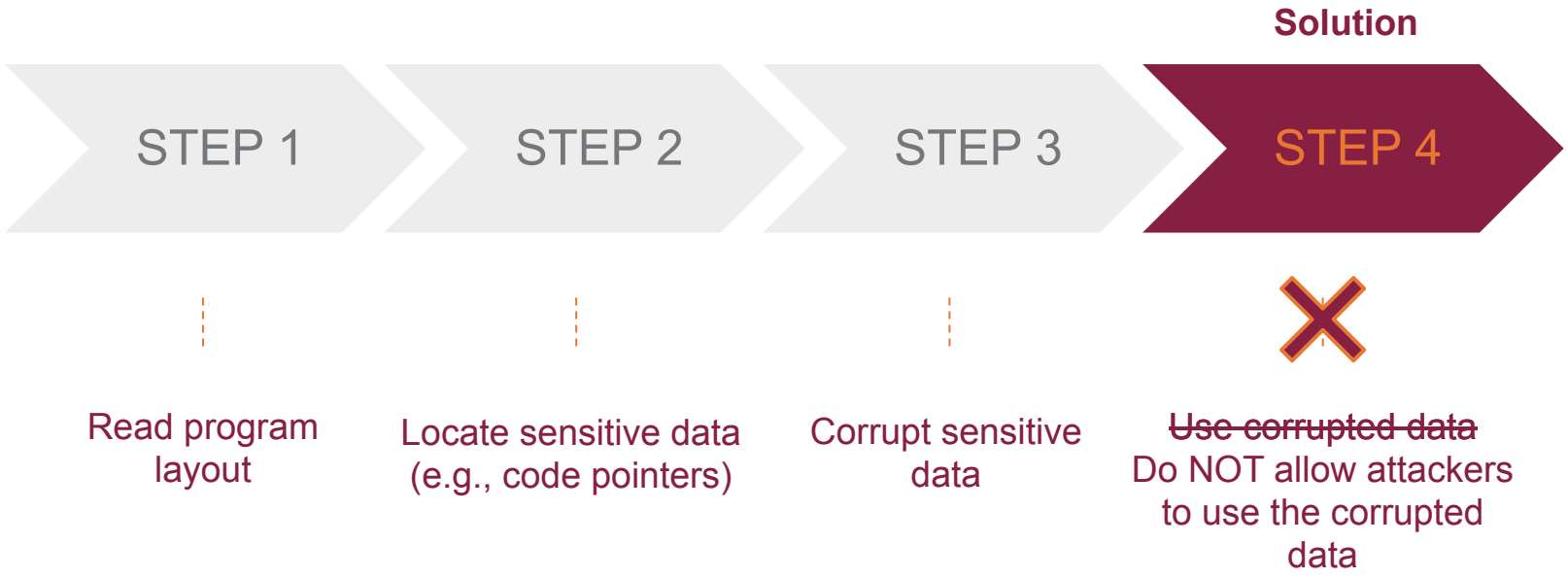
- Incur high overhead (e.g., DFI[OSDI06])
  - Metadata lookup
  - Frequent instrumentation
- Requires additional resources (e.g., uCFI[CCS18])
  - Dedicated CPU cores for background analysis
- Have narrow-scoped defense (e.g., OTI[NDSS18])
  - Still vulnerable
  - Requires to be used orthogonally with other defense techniques to provide stronger security

Too Slow !
Unscalable !
Too specific !

Outline

1. Background
2. **Introduction**
3. Data-Value Integrity (DVI)
4. HyperSpace
5. Implementation
6. Evaluation
7. Conclusion

Key idea: breaking an essential step of a memory corruption attack



Contributions

- Data-Value Integrity (DVI)
  - A new fast, scalable, & generic defense policy
  - Able to mitigate both control & non-control data attack
- HyperSpace
  - Prototype defense mechanism that enforces Data Value Integrity
  - Can enforce three different types of security
    - Heap metadata protection
    - DVI-Code Pointer Separation (DVI-CPS)
    - DVI-Code Pointer Integrity (DVI-CPI)
- Evaluations of HyperSpace

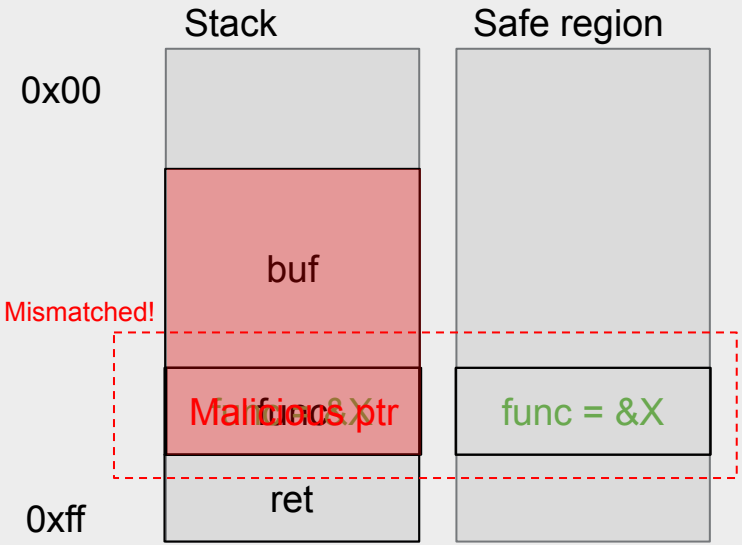
Outline

1. Introduction
2. **Data-Value Integrity (DVI)**
3. HyperSpace
4. Implementation
5. Evaluation
6. Conclusion

Overview of Data-Value Integrity

- DVI maintains shadow copy of sensitive data in an isolated “safe” region.
- DVI checks for the value integrity instead of tracking control-flow or data-flow
- To provide value integrity, DVI checks if the “value” of sensitive data is corrupted or not
- If the corruption is detected, DVI will raise a security exception

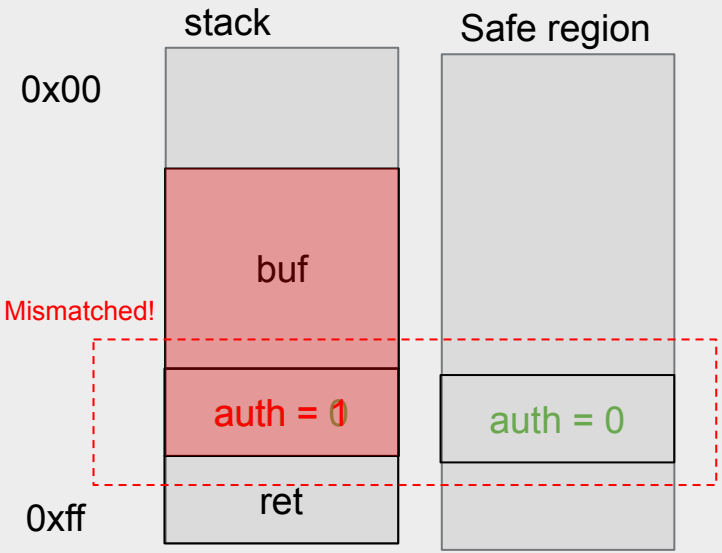
Main concept of DVI for Control data



uid = 0
input=<longer than 20 characters>

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

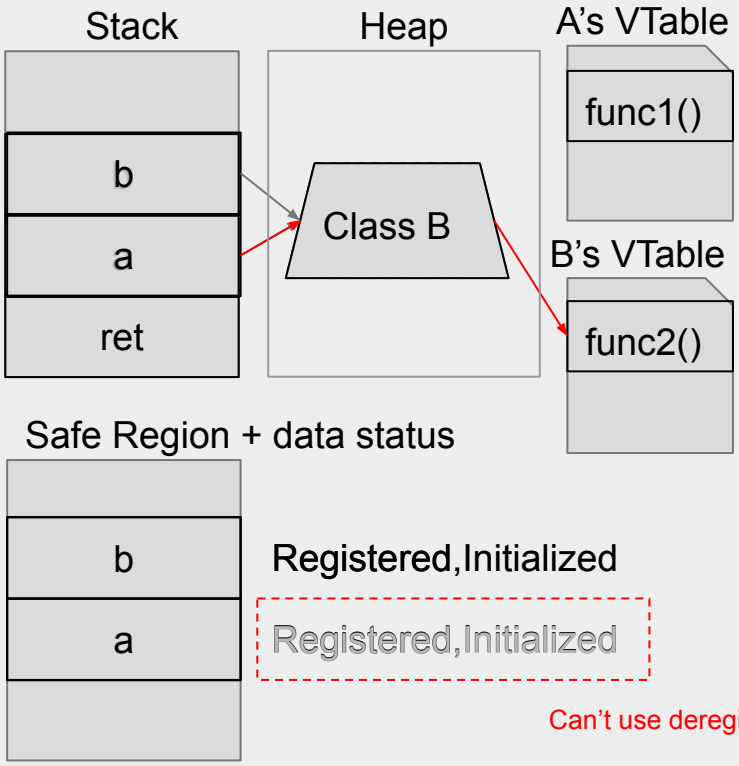
Main concept of DVI for Non-Control data



packet= <invalid packet w/ more than 1000 characters>

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

Main concept of DVI for use-after-free



Can't use deregistered data!

```
 5  class A {
 6  public:
 7      virtual void func1() {};
 8  };
 9  class B {
10  public:
11      virtual void func2() {};
12  };
13
14  int main() {
15      A *a = new A();
16
17      delete a;
18
19      B *b = new B();
20
21      a->func1();
22  }
```

Requirements for DVI

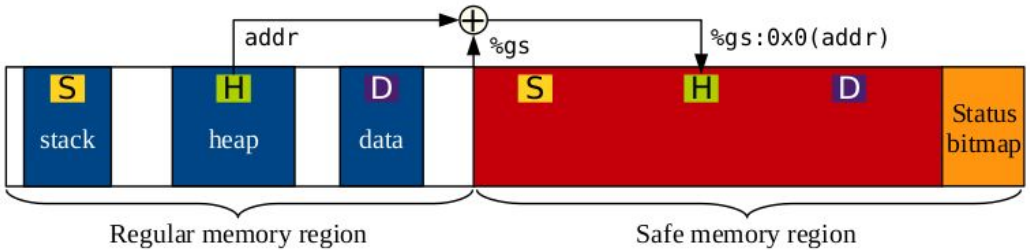
In order to provide DVI protection, we need…

- **Isolated "safe" memory region** for copy of data to reside
- Fast access to the safe memory region for **low performance overhead**
- **Secure** the safe memory region
  - Cannot be corrupted to hold data integrity
- A way to **keep track of the status** of data copy in safe memory region
  - To provide temporal safety
  - This also needs to be fast and efficient as possible

Safe region of DVI

Safe region is created by the DVI modified Kernel

- Application's memory space is bisected into regular and safe memory regions
- Status bitmap region holds the status bits for data exist in 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!

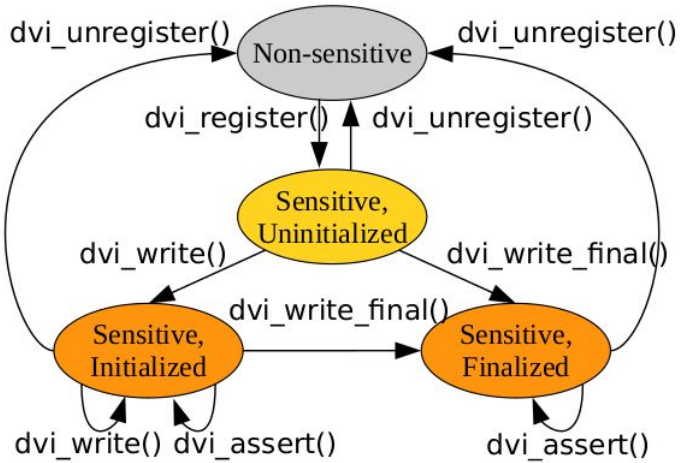


Data-Value Integrity (DVI)

API for DVI

```
1  // Register a sensitive memory region
2  // starting at addr with size
3  void dvi_register(void *addr, int size);
4  // Unregister a sensitive memory region
5  void dvi_unregister(void *addr, int size);
6  // Write the current value in a sensitive memory
7  // region to the corresponding safe memory region
8  void dvi_write(void *addr, int size);
9  // Same as dvi_write() but do not allow further writes
10 void dvi_write_final(void *addr, int size);
11 // Check if the sensitive memory value is the same
12 // as the safe memory value
13 void dvi_assert(void *addr, int size);
```

DVI State transitions



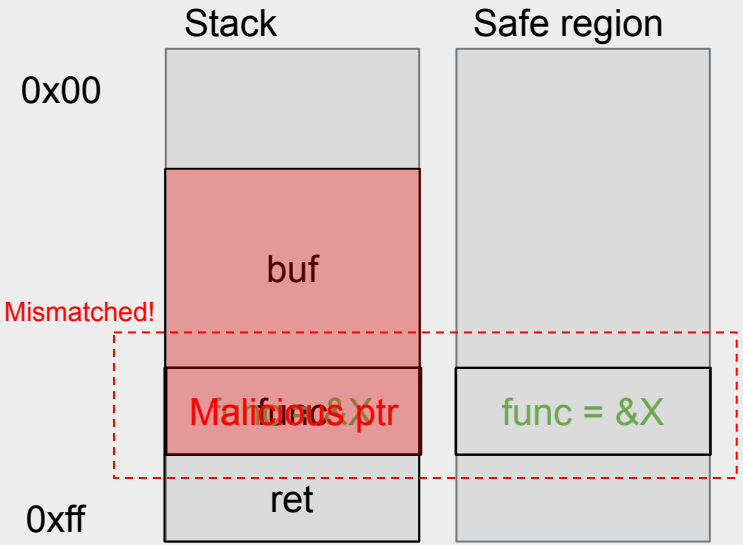
- Four different data status using 2 status bits (00, 01, 10, 11)
- DVI state transitions are managed and validated using the status bits
- Illegal transition will raise a flag

Safe region access is FAST!

- For status bits, DVI uses Bit Test (bt) instructions in x86 instruction set
- Accessing safe memory can be done with single instruction using %gs segmentation register

```
1   // Get the safe memory value for a given address
2   uint64_t dvi_load_safe_memory_8b(void *addr) {
3     uint64_t value;
4     asm volatile ("mov %%gs:0x0(%[offset]), %[value]"
5       :[value] "=r" (value) :[offset] "r" (addr) );
6     return value;
7   }
8
9   // Get the first status bit for a given address
10  uint8_t dvi_get_safe_memory_status_bit0(void *addr) {
11    void    *bitmap_addr = (void *)(((uint64_t)addr >> 5) & ~0x3);
12    uint64_t bitmap_idx  = ((uint64_t)addr & 0xf8) >> 2;
13    uint8_t  bit;
14    asm volatile (
15      "btq %[bitmap_idx], %%gs:(%[bitmap_addr],%[area_sz])"
16      : : [bitmap_idx]  "r" (bitmap_idx),
17          [bitmap_addr] "r" (bitmap_addr),
18          [area_sz]     "r" (ADDR_SPC_SZ) );
19    asm volatile ("setc %[bit]" : [bit] "+rm" (bit) );
20    return bit;
21  }
```

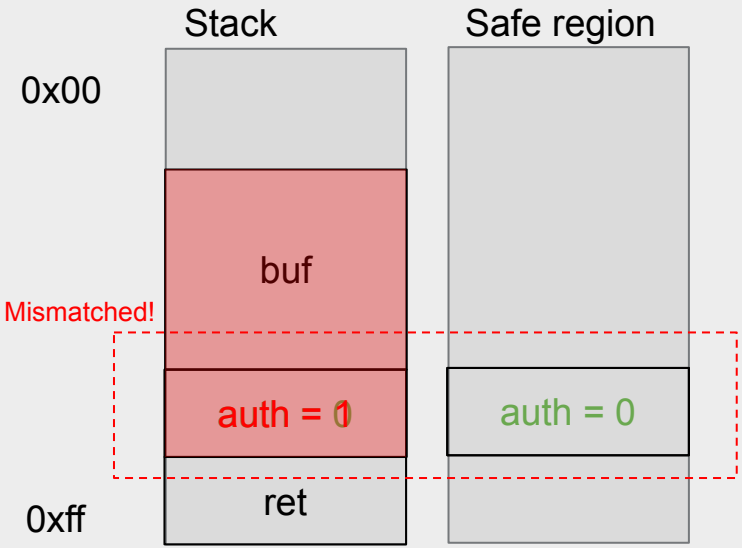
Protecting control data w/ DVI



uid = 0
input=<longer than 20 characters>

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

Protecting non-control data w/ DVI



packet= <invalid packet w/ more than 1000 characters>

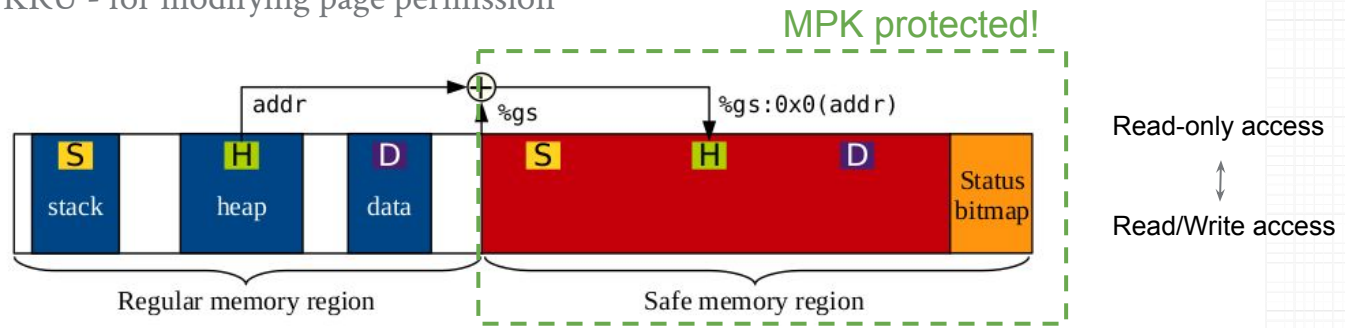
```
21  /** Example of a non-control data corruption attack */
22  bool authenticate(char *packet);
23
24  void handle_packet(char *input) {
25    int auth = 0; // non-control data to be corrupted!
26    //   dvi_register(&auth, sizeof(auth));
27    //   dvi_write(&auth, sizeof(auth));
28    char buf[1000];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

How can we guarantee the security of the safe region?

By tightly managing the safe region's memory access using Intel's Memory Protection Keys (MPK)

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. Data-Value Integrity (DVI)
3. **HyperSpace**
4. Implementation
5. Evaluation
6. Conclusion

HyperSpace

- Fully implemented prototype that enforces DVI
- Defense Mechanisms
  - Code pointer separation (DVI-CPS & C++ VTPtr Protection)
    - Protects ***all*** code pointers & virtual function table pointers (VTPtrs)
  - Code pointer integrity (DVI-CPI)
    - DVI-CPS protections + ***all*** sensitive object pointer protections
  - Heap Metadata Protection

What are sensitive data?

- Sensitive data varies for security mechanism:
  - All function pointers <- DVI-CPS & DVI-CPI
  - VTPtrs in C++ objects <- DVI-CPS & DVI-CPI
  - <- DVI-CPI
- How can we detect these sensitive data?
  - Recursive static analysis/Instrumentation

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

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

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

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

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

Outline

1. Introduction
2. Data-Value Integrity (DVI)
3. HyperSpace
4. **Implementation**
5. Evaluation
6. Conclusion

HyperSpace Implementation

LLVM Compiler pass

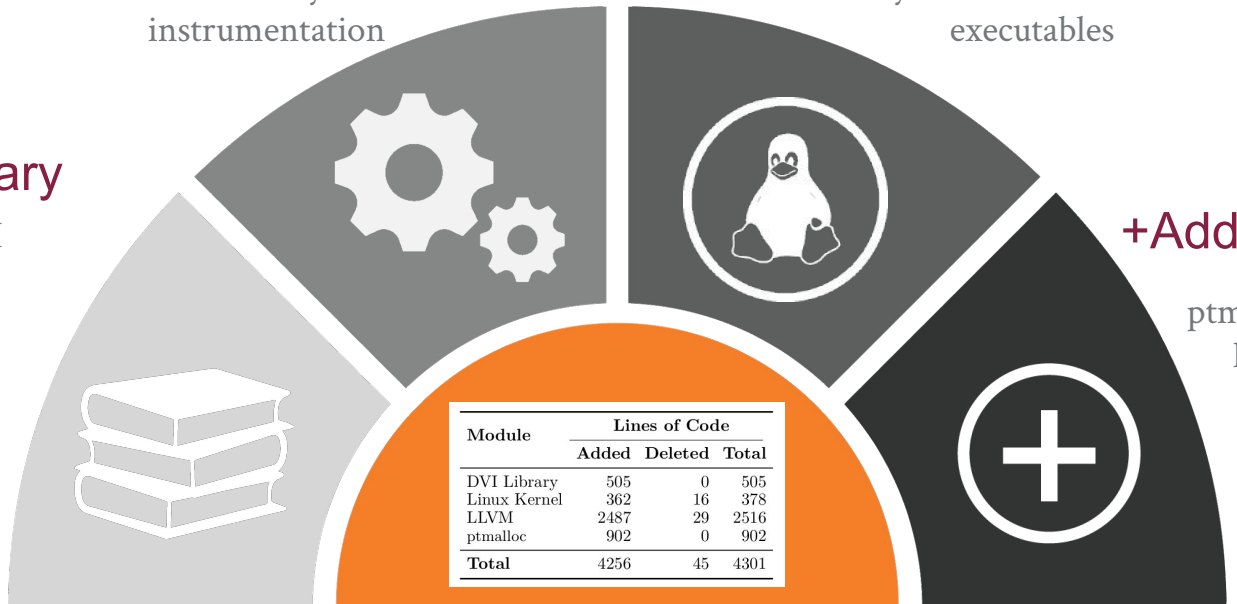
Custom compiler pass for static analysis and instrumentation

Linux Kernel modification

Bisect the memory layout of DVI enabled executables

DVI API library

Library for DVI primitives



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

+Additional support

ptmalloc inlining for heap metadata protection

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 optimization:
  - Inlining DVI functions
  - Not instrumenting objects in 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 example: Basic block coalescing

```
/** 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 sensitive data. */
OP *Perl_append_list(pTHX_ I32 type, LISTOP *first, LISTOP *last){
    // ...
    first->op_last->op_sibling = last->op_first;
    // dvi_safe_memory_unlock();
    //   dvi_write(&first->op_last->op_sibling, 8);
    // dvi_safe_memory_lock();
    first->op_last = last->op_last;
    // dvi_safe_memory_unlock();
    //   dvi_write(&first->op_last, 8);
    // dvi_safe_memory_lock();

    first->op_flags |= (last->op_flags & OPf_KIDS);
    FreeOp(last);
    return (OP*)first;
}
```

Before

```
/** 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;
    // dvi_safe_memory_unlock();
    //   dvi_write(&first->op_last->op_sibling, 8);
    first->op_last = last->op_last;
    //   dvi_write(&first->op_last, 8);

    first->op_flags |= (last->op_flags & OPf_KIDS);
    // dvi_safe_memory_lock();
    FreeOp(last);
    return (OP*)first;
}
```

After

Outline

1. Introduction
2. Data-Value Integrity (DVI)
3. HyperSpace
4. Implementation
5. **Evaluation**
6. 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 SafeStack
- Linked with GNU gold v2.29.1-23.fc28

Performance/Memory evaluation with:

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

Security evaluation with:

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

Runtime overhead of HyperSpace

CFI [CCS'05]

- Average: 8.4%

uCFI [CCS'18]

- Average: 7.88%

DFI [OSDI'06]

- Average: 104%

CPI [OSDI'14]

CPS
- Average: 1.9%

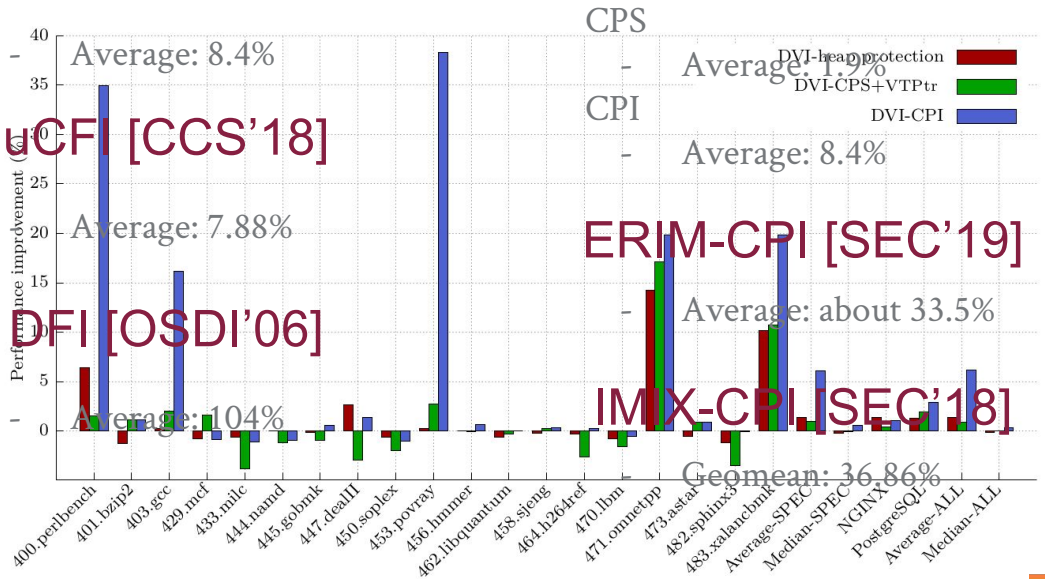
CPI
- Average: 8.4%

ERIM-CPI [SEC'19]

- Average: about 33.5%

IMIX-CPI [SEC'18]

- Geomean: 36.86%



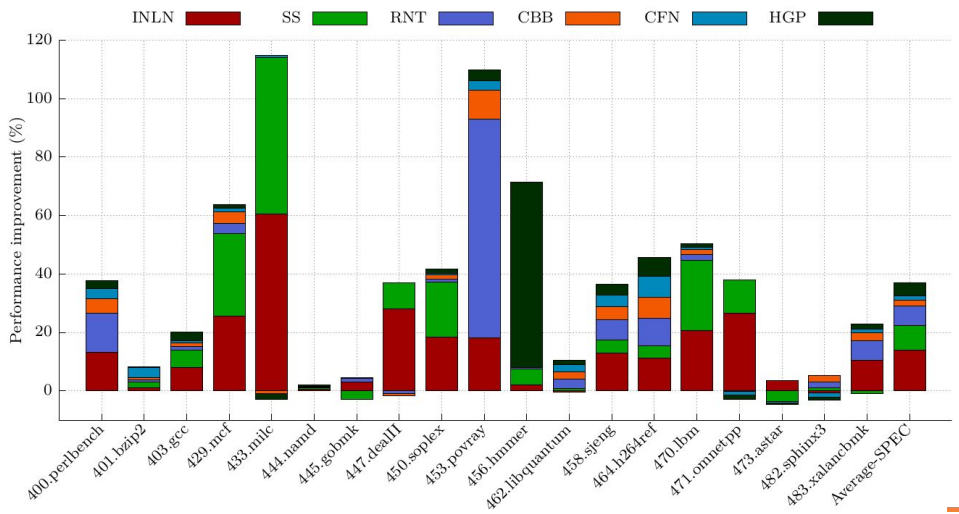
HyperSpace

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

DVI-CPS + VTPtr
- Average: 1.02%
- Median: 0.23%

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

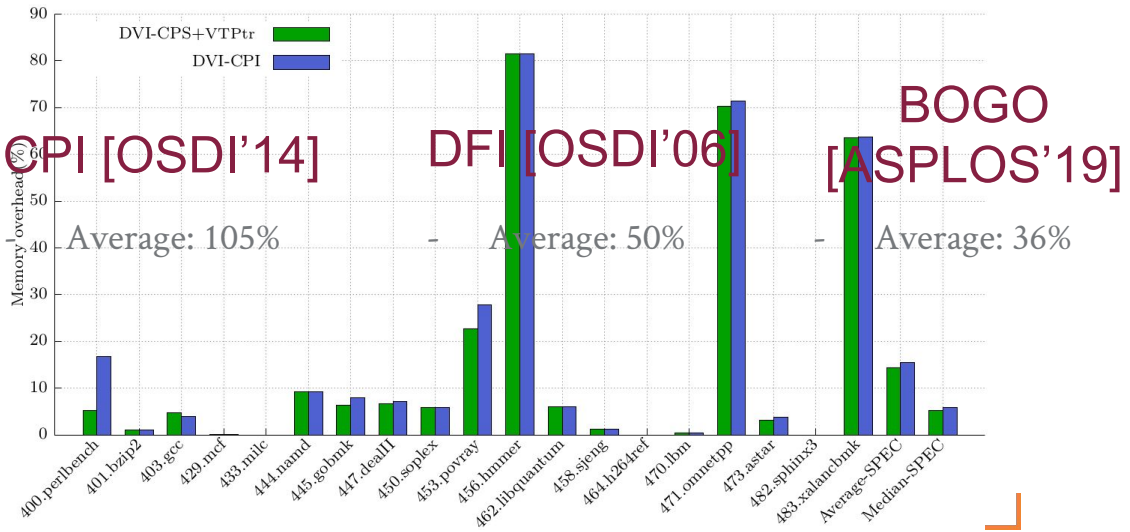
Impact of the optimization techniques



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



CPI [OSDI'14]
- Average: 105%

DFI [OSDI'06]
- Average: 50%

BOGO [ASPLOS'19]
- Average: 36%

HyperSpace

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

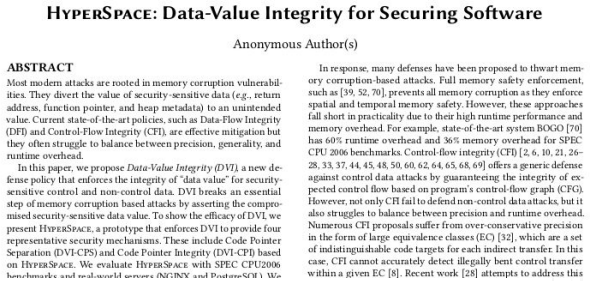
DVI-CPS+VTPtr
- Average: 14.42%
- Median: 5.27%

Outline

1. Introduction
2. Data-Value Integrity (DVI)
3. HyperSpace
4. Implementation
5. Evaluation
6. **Conclusion**

Conclusion

- Data-Value Integrity is a new defense policy that provides versatile and elegant solution to thwarting memory corruption exploits
- Our prototype, HyperSpace, enforces DVI to provide various security mechanisms with the strongest guarantee having 6.35% runtime overhead and 15.47% memory overhead.
- Contributions:
  - DVI
  - HyperSpace
  - Thorough evaluation of HyperSpace
- Projected for CCS ‘20 submission



HyperSpace: Data-Value Integrity for Securing Software

Anonymous Author(s)

ABSTRACT

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

In this paper, we propose *Data-Value Integrity (DVI)*, a new defense policy that enforces the integrity of "data value" for security-sensitive control and non-control data. DVI breaks an essential step of memory corruption based attacks by asserting the compromised security-sensitive data value. To show the efficacy of DVI, we present HyperSpace, a prototype that enforces DVI to provide four representative security mechanisms. These include Code Pointer Separation (DVI-CPS) and Code Pointer Integrity (DVI-CPI) based on HyperSpace. We evaluate HyperSpace with SPEC CPU2006 benchmarks and real-world servers (NGINX and PostgreSQL). We

In response, many defenses have been proposed to thwart memory corruption-based attacks. Full memory safety enforcement, such as [39, 52, 70], prevents all memory corruption as they enforce spatial and temporal memory safety. However, these approaches fall short in practicality due to their high runtime performance and memory overhead. For example, state-of-the-art system BOGO [70] has 60% runtime overhead and 36% memory overhead for SPEC CPU 2006 benchmarks. Control-flow integrity (CFI) [2, 6, 10, 21, 26–28, 33, 37, 44, 45, 48, 50, 60, 62, 64, 65, 68, 69] offers a generic defense against control data attacks by guaranteeing the integrity of expected control flow based on program's control-flow graph (CFG). However, not only CFI fail to defend non-control data attacks, but it also struggles to balance between precision and runtime overhead. Numerous CFI proposals suffer from over-conservative precision in the form of large equivalence classes (EC) [32], which are a set of indistinguishable code targets for each indirect transfer. In this case, CFI cannot accurately detect illegally bent control transfer within a given EC [8]. Recent work [28] attempts to address this


VIRGINIA TECH™

Threat Model

- Assumption:
  - A program may have one or more memory vulnerabilities
- Attackers can use memory vulnerabilities to read from/write to arbitrary memory
- Attacker can not modify or inject code due to DEP
- All hardware is trusted
- Attacks exploiting hardware vulnerabilities are out-of-scope

Detecting HyperSpace Executables

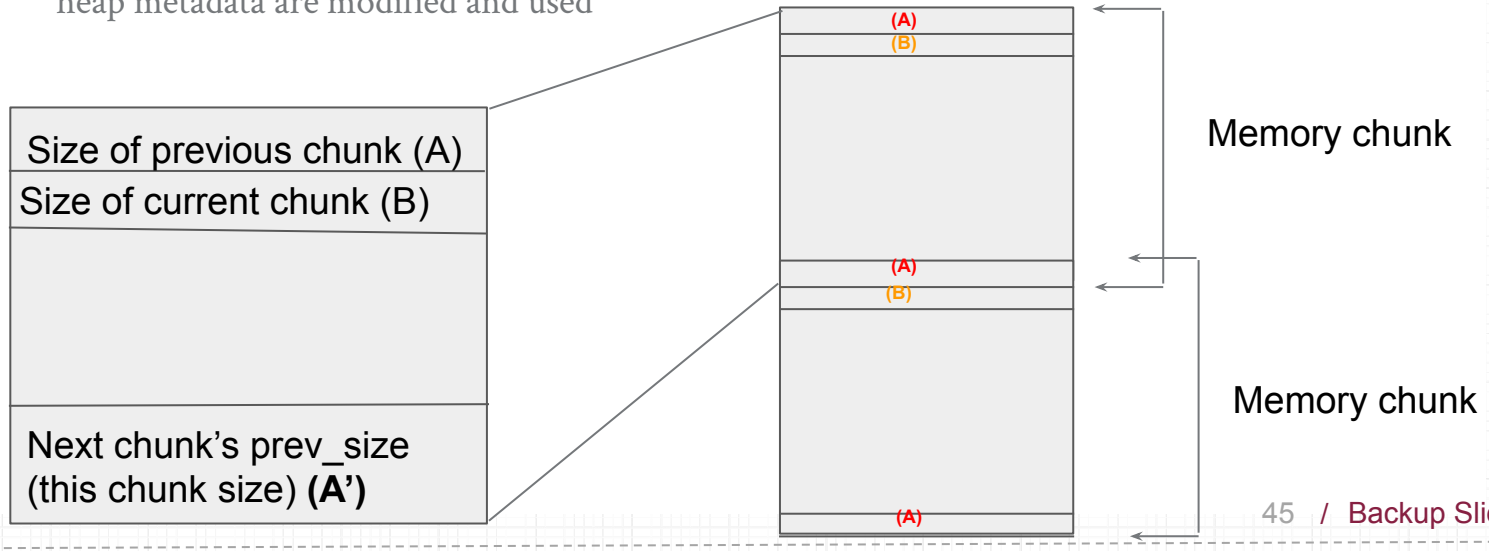
- To mark an executable to be HyperSpace enabled, we wrote a python tool to modify the ***e_flags*** field in the executable's ELF header.
- ***e_flags***, also known as ***EF_machine_flag***, is an unused 4-byte long field in x86 architecture and are usually set to 0.
- We mark this field with HyperSpace's unique signature value "***0xBE***"
- This marking is checked during Kernel's program initialization process to determine whether or not it should create safe region for DVI protection.

Discussion

- Could MPK be misused?
  - Only if the program only has non-control flow protection (heap metadata protection), then the attack could be possible
    - Such attack can be mitigated by additionally adopting recently proposed orthogonal defense techniques such as ERIM and HODOR

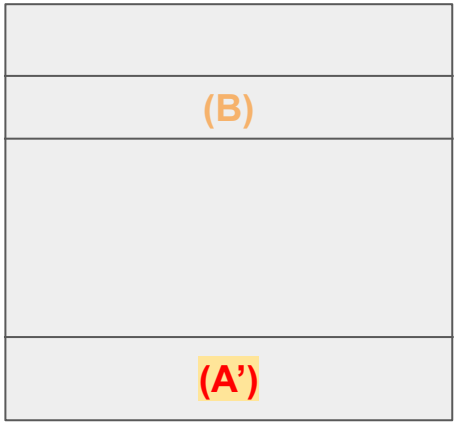
Ptmalloc inlining

- The function involved in malloc and free in Ptmalloc such as: mspace_malloc and mspace_free are instrumented with DVI
- These APIs are directly inlined into corresponding locations where heap metadata are modified and used

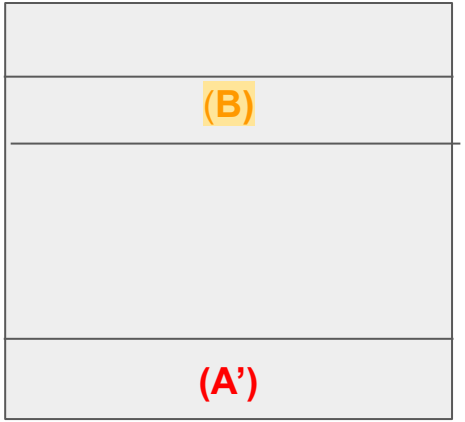


Ptmalloc inlining (Allocation)

1. Memory gets allocated

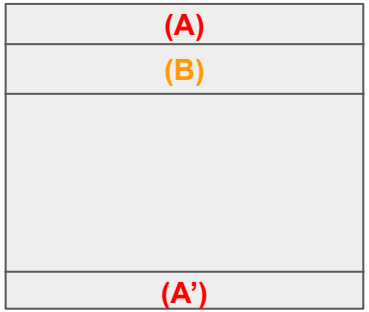


2. Using the existing (B), calculate the location of next (A') and update the size accordingly

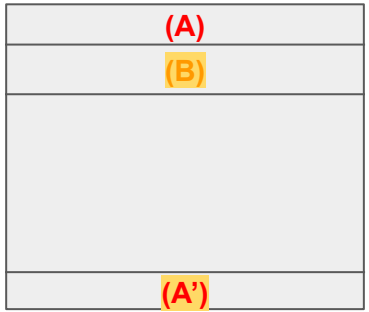


3. Update the current chunk size

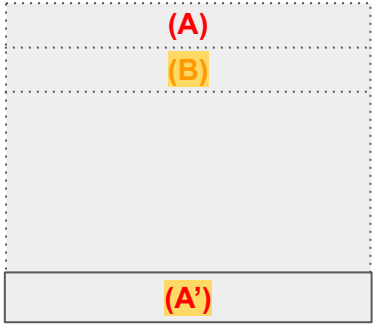
Ptmalloc inlining (Deallocation)



1. Uses current chunk's (A) to determine the location of previous chunk location



2. Check to see if current chunk can be merged into other chunks when it frees itself
3. If it can, it performs the merge and updates the new (B) and Next chunk's (A)



4. Current chuck is deallocated

Security Evaluation 1

- CVE-2016-10190 : Heap-based BoF in ffmpeg
- Ffmpeg is a popular multimedia framework for encoding and decoding audio and video.
- Vulnerability is in its HTTP parsing functionality
- The overflow happens due to ffmpeg allowing signed integers to be allocated as sizes and then converting them to unsigned, causing a large buffer to be read.
- This allows attackers to overwrite function pointers in an AVIOContext object.

Solution:

- Using DVI-CPS/CPI we can protect this attack by asserting the corruption of a function pointer in a victim AVIOContext object.

Security Evaluation 2

- CVE-2015-8668 : Heap-based BoF in libtiff
- Libtiff is an image file format library and the BoF is in the PackBitsPreEncode function in tif_packbits.c
- A malicious BMP file causes integer overflow followed by heap overflow and overwriting a function pointer in a TIFF structure.

Solution:

- Using DVI-CPS/CPI we can protect this attack by asserting the corruption of a function pointer in a victim TIFF structure.

Security Evaluation 3

- CVE-2014-1912: BoF in python2.7
- This is caused by missing buffer size check
- Due to this, a function pointer in PyTypeObject is corrupted.

Solution:

- Using DVI-CPS/CPI we can protect this attack by asserting the corruption of a function pointer in PyTypeObject.

Security Evaluation 4

- Synthesized exploits:
  - C++ test suite released by Burow et al (VTPtr hijacking exploits)
    - FakeVT, FakeVT-sig, VTxcg, VTxchg-hier
  - COOP attack
  - All protected using DVI-CPS/CPI protection since we protect the VTPtrs

  - Heap exploit
    - 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