Practical Exploit Mitigation Design Against Code Re-Use and System Call Abuse Attacks

Christopher Jelesnianski
Ph.D. Dissertation Defense

December 2, 2022

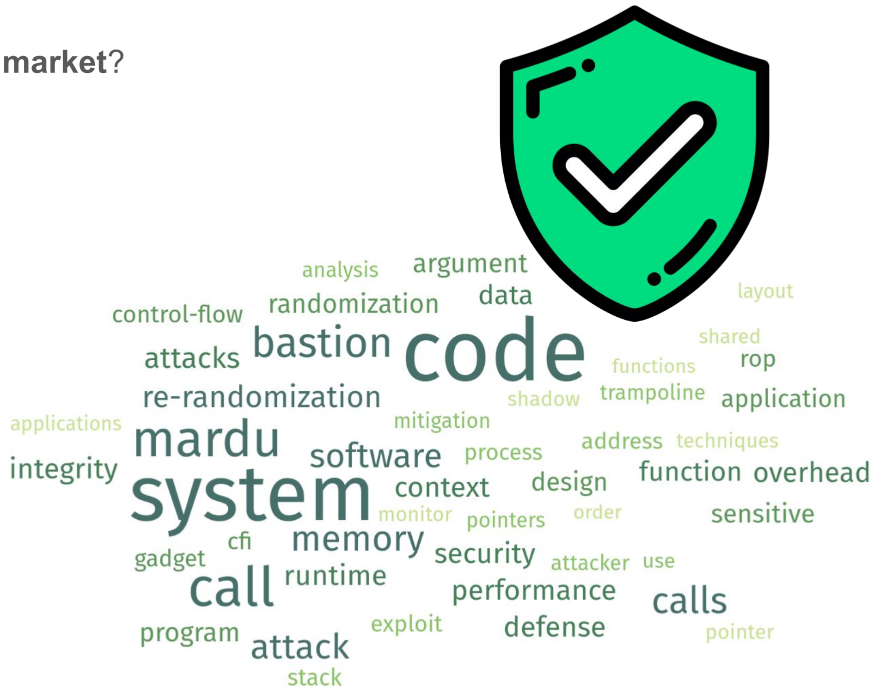
Committee:
Changwoo Min (Chair)
Yeongjin Jang, Wenjie Xiong, Danfeng Yao, & Haibo Zeng

VIRGINIA TECH™

Executive Summary

Security Domain:

- Why are software exploit mitigation designs **not making it to market**?
    - **Core challenges** to build **practical** exploit mitigations
- How to make **randomization** effective **against code re-use**?
- How to **protect system calls**?
    - **System call usage** is **common theme** among attacks
    - Gap exists in current security coverage of system calls



My Research:

- Designing (& proof of concept of) practical exploit mitigations
- Enabling randomization to be scalable and performant
- Strengthening system call protection

Outline

- Introduction
- Background
- MARDU: Practical Randomization
- BASTION: Securing System Calls
- Future Work & Conclusion

Outline

- Introduction
  - Motivation & Problem Statement
- Background
- MARDU: Practical Randomization
- BASTION: Securing System Calls
- Future Work & Conclusion

Mainstream Adoption of Software Mitigation Techniques

- PaX Linux Address Space Layout Randomization (**ASLR**) (2001)
- Linux **Kernel Stack ASLR** (2002)

- No-Execute Bit (**NX-bit**)
  - Linux 2.6.8 (2004)
  - Windows XP SP2 (2004) -- Data Execution Prevention (**DEP**)

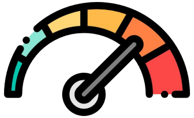
- System Call Whitelisting (**seccomp**)
  - Linux 2.6.12 - (2005)
  - seccomp mode 2 - Linux 3.5 (2012)
  - seccomp eBPF - Linux 3.8 (2013)

- GCC & Clang **Control-Flow Integrity & SafeStack** (USENIX Security 2014)

Adoption is limited & slow!

Challenges in Exploit Mitigation Design

Why is Practical Exploit Mitigation Design Difficult to Achieve?

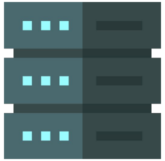


Non-Negligible Performance

- **Complex** or **frequent** verification is hard to make **fast**

Attack Diversity & Complexity

- Modern attacks require **more code components** to be **guarded**

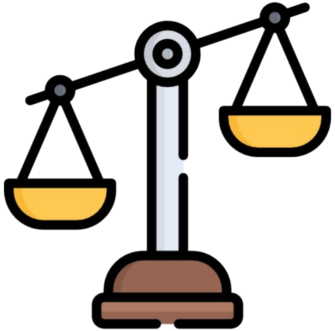


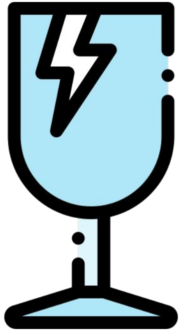
Limited Scalability

- Designing mechanisms with **negligible system impact** is hard

Fragility

- **Perfect analysis** is challenging to achieve in practice

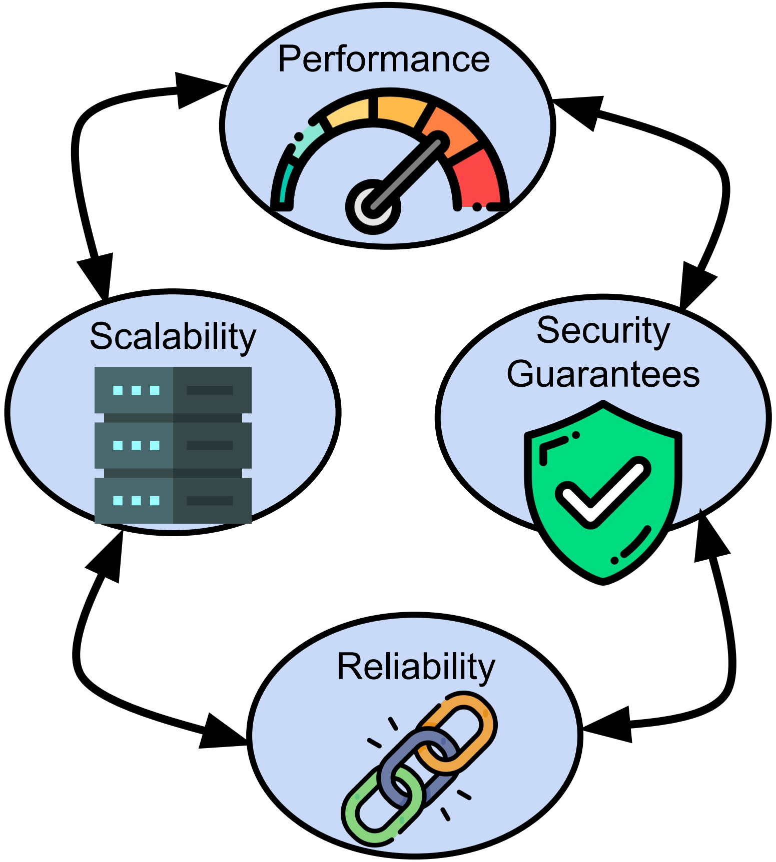




Blueprint For Success in Practical Exploit Mitigation Design

Practical Design Properties

- Low Performance Impact
  - Defenses should achieve **low** performance
- Strong Security Guarantees
  - Defenses should provide **strong** security guarantees
- Scalable Framework
  - Defenses should **minimize** use of additional CPU or memory resources
- Reliable Defense
  - Defenses should **not** break the application runtime



Problem Statement, Dissertation & Contributions

Problem Statement:

How can one create strong, yet practical exploit mitigation designs?

Dissertation Aims & Objectives:

- *Understand & identify critical attack/defense aspects to devise practical exploit mitigation designs*
- *Show through prototyping that practical designs are achievable*

Dissertation Contributions:

MARDU [SYSTOR'20 / DTRAP'22]

- On-demand, reactive re-randomization to minimize system performance impact
- Ability of sharing randomized code system-wide
- Re-randomizing without pausing execution

BASTION [ASPLOS'23 - Major Revision]

- Fine-grained system call filtering
- 3 new contexts to enforce System Call Integrity
  - Call-Type Context
  - Execution Path Context
  - Argument Integrity Context
- Block attack classes that rely on system calls

Outline

- Introduction
  - Motivation & Problem Statement
- Background
  - Attacks: Code Re-use & System Call Use
  - Scope: Current Security Landscape
  - Defenses: Randomization & System Call Filtering
- MARDU: Practical Randomization
- BASTION: Securing System Calls
- Conclusion & Future Work

How Code Re-Use Attacks Work

Assumptions / Threat Model

- Memory Vulnerability Exists
  - Stack-based - Buffer Overflow
  - Heap-based - Dangling Pointer (e.g., use-after-free, double-free)

Generalized Code Re-Use Attack Procedure

1 → 2 → 3 → 4 → 5 =

1. Identify Memory Vulnerability
2. Perform Memory Disclosure
3. Identify Code Components
4. Sensitive Data and/or Control-Flow Hijacking
5. Illegitimate System Call Usage

Code Re-Use Variants

● Return Oriented Programming (ROP)	*Shacham et al. (CCS 2007)*
● Just-In-Time ROP (JIT-ROP)	*Snow et al. (S&P 2013)*
● Blind ROP (BROP)	*Bittau et al. (S&P 2014)*
● Control Data & Non-Control Data Attacks	*van der Veen et al. (CCS 2017)*

Modern Defense Archetypes

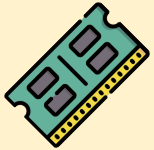
Generalized Code Re-Use Attack Procedure

0. Presence of Sensitive Code Components
1. Identify Memory Vulnerability
2. Perform Memory Disclosure
3. Identify Viable Code Components
4. Sensitive Data and/or Control-Flow Hijacking
5. Illegitimate System Call Usage

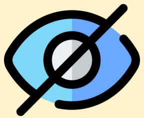
Generalized Defense Archetype Created

Debloating

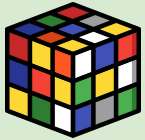
Memory Safety



Information Hiding

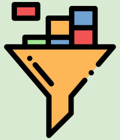


Re-Randomization



Data Integrity & Control Flow Integrity

System Call Filtering



Why **Re-Randomization** & **System Call Filtering**?

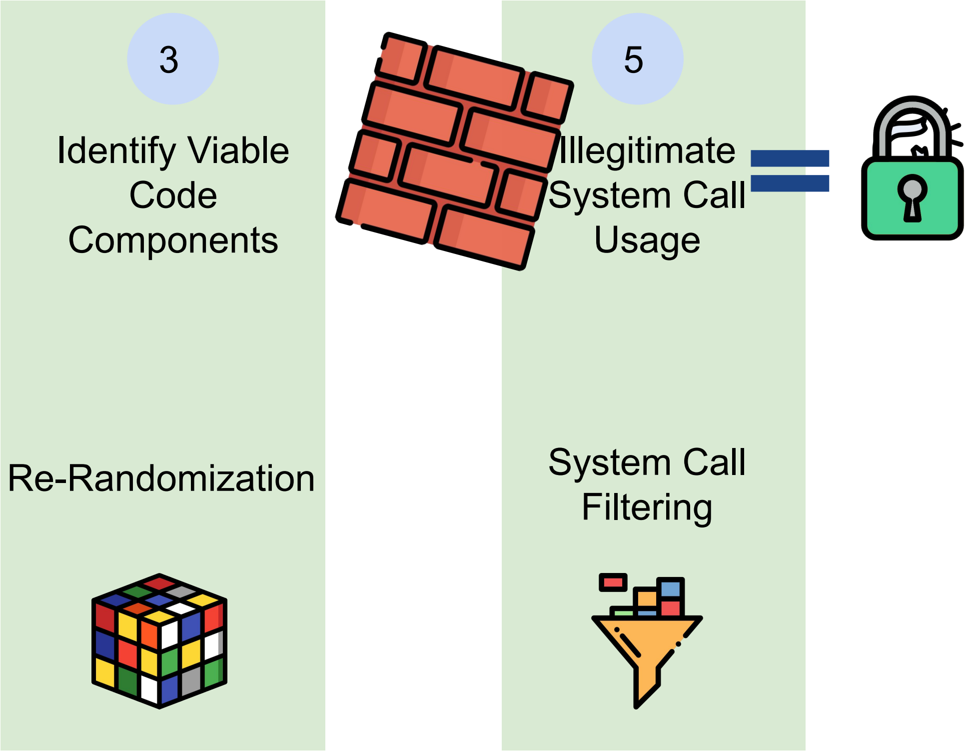
Both great archetypes to refine design and make practical

Re-Randomization

- **Performance**: Randomization imposes significant performance impact
- **Scalability**: No code sharing support

System Call Specialization

- **Security**: Coarse-grained, could be better
- **Scalability:** Not able to support protecting dynamic arguments



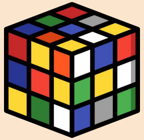
Outline

- Introduction
- Background
- MARDU: Practical Randomization (Brief Review)
- BASTION: Securing System Calls
- Conclusion & Future Work

3

Identify Viable Code Components

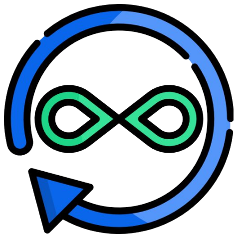
Re-Randomization



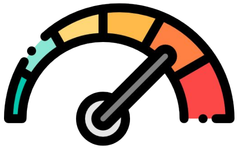
MARDU: Background

- Attacks rely on ***known locations*** within the binary
  - Fundamental assumption to launch attacks is knowing process layout
  - Addresses of code gadgets & stack elements

- Continuous randomization ***breaks attacker assumptions***
  - Continuous churn makes code and data harder to find
    - Process layout is no longer deterministic



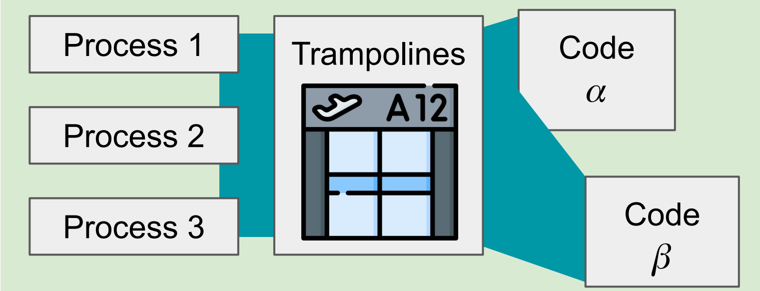
- However, continuous randomization is ***not practical*** to be deployed
  - High performance impact & memory overhead
  - Code-sharing system-wide not possible



MARDU: Key Ideas

Stationary Trampolines

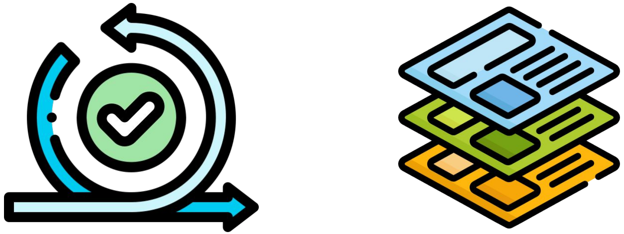
- Simplify randomization tracking
- Connect function entry to function body, but hide code gadgets
- Memory Protection Keys (MPK) protect the code region by hiding it



Trampolines act as:
entry gateway to each function call
exit gateway for each function return

What Trampolines Enable

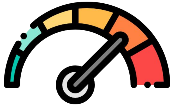
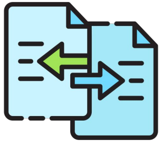
- MARDU shares randomized code in **system-wide manner**
- Code sharing unlocks **memory deduplication**
- Facilitate seamless migration to newly randomized code
- Re-randomization ***without stopping-the-world***



MARDU: Summary

MARDU Contributions

- Simple runtime tracking & reactive ***on-demand*** runtime re-randomization
- Runtime re-randomization does not pause execution or stop-the-world
- First randomization scheme capable of runtime re-randomization ***with*** code sharing
- Trampolines ***separate and hide code gadgets*** from attackers


Advantages

- Performance: **5.5% (AVG)**
- Significant Memory Savings (**~7.5 GB**)
- Blocks all ROP attack variants

Limitations

- Re-Randomization is **probabilistic**
- Worst-case performance: **18.3%**
- **Non-Control Data Attacks** not covered

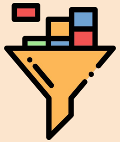
Outline

- Introduction
- Background
- MARDU: Practical Randomization
- BASTION: Securing System Calls
  - Motivation
  - Design
  - Implementation
  - Evaluation
- Conclusion & Future Work

5

Illegitimate System Call Usage

System Call Filtering



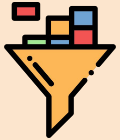
Outline

- Introduction
- Background
- MARDU: Practical Randomization
- BASTION: Securing System Calls
  - Motivation
  - Design
  - Implementation
  - Evaluation
- Conclusion & Future Work

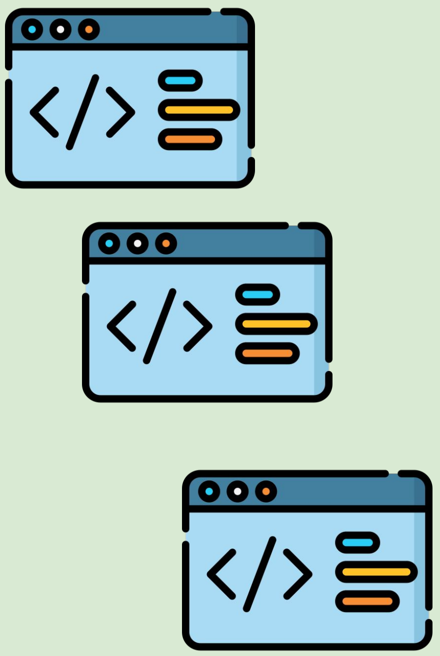
5

Illegitimate System Call Usage

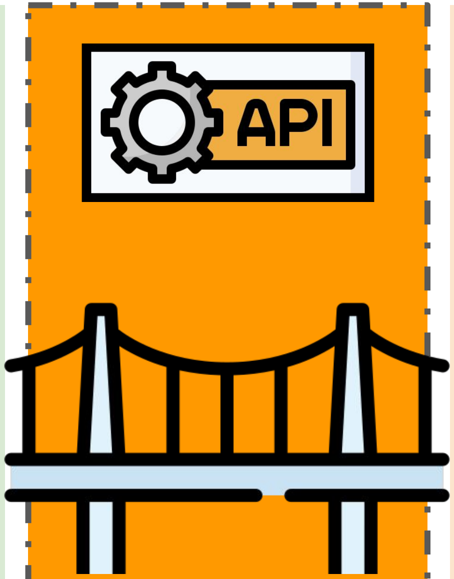
System Call Filtering



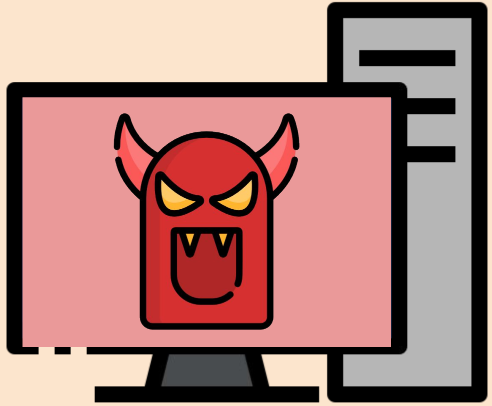
Why Do System Calls Matter?



< userspace >



System Call Interface



Process

Arbitrary Code Execution

Device

Privilege Escalation

Network Reconfiguration

Memory Permission Changes

Maintenance

Core attack goals: invoke one or more system calls

Out of control: Overcoming control-flow integrity [S&P'14]

< kernelspace >

System Calls: Use to Abuse

NGINX Web Server

Legitimate Use

- `execve()` used to update server in place during runtime

Example 1

```
// nginx/src/os/unix/ngx_process.c
static void ngx_execute_proc(ngx_cycle_t *cycle,void *data){

   ngx_exec_ctx_t *ctx = data;

   if( execve( ctx->path, ctx->argv, ctx->envp ) == -1){

      ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
            "execve() failed");
```

System Calls: Use to Abuse

NGINX Web Server

Attacker Abuse

- `execve()` can launch an attacker binary or start shell ("`/bin/sh`")

Example 1

```
// nginx/src/os/unix/ngx_process.c
static void ngx_execute_proc(ngx_cycle_t *cycle,void *data){

  ngx_exec_ctx_t *ctx = data;

  if( execve( ctx->path, ctx->argv, ctx->envp ) == -1){

    ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
          "execve() failed");
```

System Calls: Use to Abuse

NGINX Web Server

Attacker Abuse

- Sometimes, system calls are never needed/used in an application
- `mprotect()` is never used in NGINX.
- Reaching `mprotect()` can change attacker controlled memory region to have executable permissions

Attacker Pattern Insight:

- How are system calls invoked?
- How are system calls reached?
- What is passed to system calls?

Example 2

```
// nginx/src/http/ngx_http_variables.c
ngx_http_variable_value_t *ngx_http_get_indexed_variable(
                    ngx_http_request_t *r,   index = 4321   ){
    ...
    if(         &mprotect          (r, &r->variables[index],
                    v[index].data) == NGX_OK){
          ...
          if( v[index].flags & NGX_HTTP_VAR_NOCACHEABLE){
                    r->variables[index].no_cacheable = 1;
          }
          return &r->variables[index];
    }
    ...
    return NULL;
```

Related Work: System Call Protection

`seccomp` Introduced [2005]

Coarse-grained call whitelisting

Manual configuration required.
Configuration is challenging!

Automation Added

`sysfilter`: *Automated system call filtering for commodity software* [RAID'20]

Automating Seccomp Filter Generation for Linux Applications [CCSW'21]

Does not strengthen system call security!

Refined Whitelisting

Temporal System Call Specialization [USENIX Sec'20]

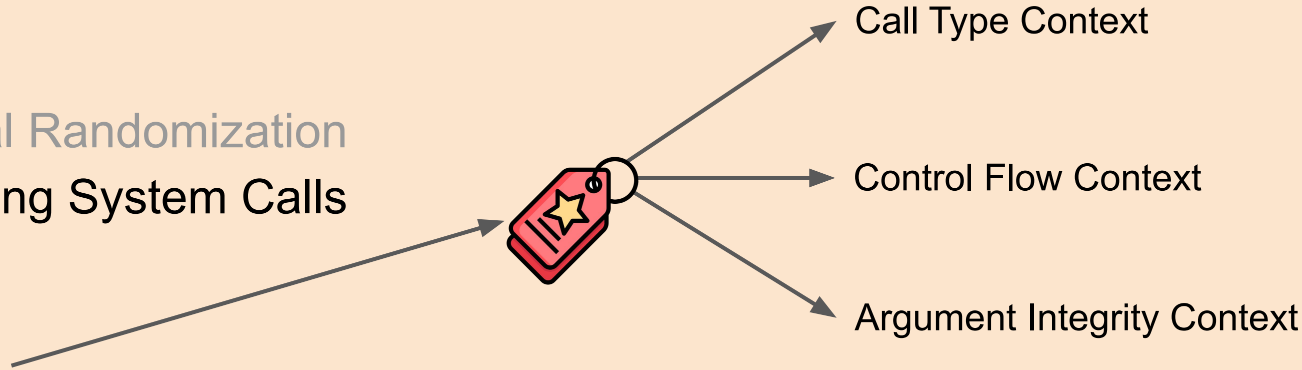
Does not resolve attack angles in previous examples!

Core Research Question:

What **information** should be considered **to check legitimacy** of system call invocations?

Outline

- Introduction
- Background
- MARDU: Practical Randomization
- BASTION: Securing System Calls
  - Motivation
  - Design
    - Contexts
    - Overview
    - Compiler Component
    - Monitor Component
  - Implementation
  - Evaluation
- Conclusion & Future Work

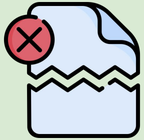


BASTION Contexts - Call Type Context

Guarantee: Only permitted system calls are allowed to be called in their expected manner

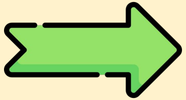
- Assigned Per-System-Call
- Three (3) Types

Applicable to All System Calls

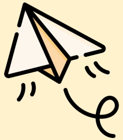


Not-Callable
(Never used by Application)

Sensitive System Calls Only



Directly-Callable
(Traditional Direct Call)



Indirectly-Callable
(e.g., code pointers)

Example 3

```
void foo ( int f0, char* f1 ){
    f2 = getConfigString(f0, f1);
    int flags = MAP_ANON|MAP_SHARED;
    bar( x1, flags );
    ...
}
void bar ( char*  b1, int b2 ){
    int prots = PROT_READ|PROT_WRITE;
    mmap( NULL, gshm->size, prots, b2,
           -1, 0 );
    ...
}
```

System Call	Call Type
mmap	Directly-Callable
mprotect	Not-Callable

BASTION Contexts - Control Flow Context

Guarantee: A sensitive system call is reached and invoked only through legitimate control-flow paths during runtime

Example 3

```
void foo ( int f0, char* f1 ){
   f2 = getConfigString(f0, f1);
   int flags = MAP_ANON|MAP_SHARED;
   bar( x1, flags );
   ...
}
void bar ( char*  b1, int b2 ){
   int prots = PROT_READ|PROT_WRITE;
   mmap( NULL, gshm->size, prots, b2,
        -1, 0 );
   ...
}
```

Valid Control Flow
`bar < foo`
`mmap < bar`
...

BASTION Contexts - Argument Integrity Context

Guarantee: A sensitive system call can only use valid arguments when being invoked

- ***Even if*** attackers have access to memory corruption vulnerabilities

Argument Type Coverage

- Constants
- Global Variables
- Local Variables
- Caller Parameters

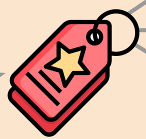
constant | global variable | local variable | caller parameter

Example 3

```
void foo ( int f0, char* f1 ){
   f2 = getConfigString(f0, f1);
   int flags = MAP_ANON|MAP_SHARED;
   bar( x1, flags );
   ...
}
void bar ( char*  b1, int b2 ){
   int prots = PROT_READ|PROT_WRITE;
   mmap( NULL, gshm->size, prots, b2,
         -1, 0 );
   ...
}
```

Outline

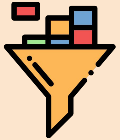
- Introduction
- Background
- MARDU: Practical Randomization
- BASTION: Securing System Calls
  - Motivation
  - Design
    - Contexts
    - Overview
    - Compiler Component
    - Monitor Component
  - Implementation
  - Evaluation
- Conclusion & Future Work

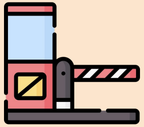


Call Type Context

Control Flow Context

Argument Integrity Context





BASTION Criteria & Coverage

Defense Framework Criteria

1. Disable unnecessary system calls
2. Ensure system calls are reached legitimately
3. Ensure legitimate system call arguments
4. Easy deployment
5. Ensure practical performance

System Call Integrity

Attackers must use one of these ***sensitive system calls*** to achieve code re-use!

Security-Sensitive System Calls (20)

Arbitrary Code Execution

`execve, execveat, fork, vfork, clone, ptrace`

Memory Permission Changes

`mprotect, mmap, mremap, remap_file_pages`

Privilege Escalation

`chmod, setuid, setgid, setreuid`

Networking Reconfiguration

`socket, bind, connect, listen, accept, accept4`

BASTION THREAT MODEL

We make the following assumptions:

- Assume powerful adversary with arbitrary `read/write` capabilities
  - Via exploiting present memory vulnerabilities in code base
- Assume common security defenses are deployed:
  - ASLR
  - DEP
  - Shadow Stack
- Assume the host operating system (OS) and hardware are trusted
  - Attacks targeting hardware and side-channels are out of scope:
    - Meltdown / Spectre
    - RowHammer

BASTION Design - Overview

BASTION Compiler

- Call Type Analysis
- Control Flow Analysis
- Argument Integrity Analysis
- Sensitive Variable Instrumentation

BASTION Runtime Monitor Process

BASTION Context Metadata

User Application

Process State Information

Context Checking

Call Type

Control Flow

Argument Integrity

Operating System

Every ***Sensitive System Call*** is intercepted by BASTION

Safe System Call Invocation

BASTION Compiler - Call Type Context

Goal: Provide more fine-grained calling constraints!
Enforce not only if system calls are allowed to be invoked, but **how**!

Analysis Procedure:

- Initially assume all system calls: **Not-Callable**
- Inspect **all call instructions** in LLVM IR
- Non-sensitive system calls invoked: **Whitelisted**
- Sensitive system calls invoked directly: **Directly-Callable**
- **Handling indirect call sites:**
  a. Record all code pointers
  b. Scan for assignment of sensitive system calls: **Indirectly-Callable**

Sensitive system calls are sparsely used & rarely invoked indirectly.

read
mmap
gettimeofday
nanosleep
accept4, 0x551234
mlock
munlock
socket
connect

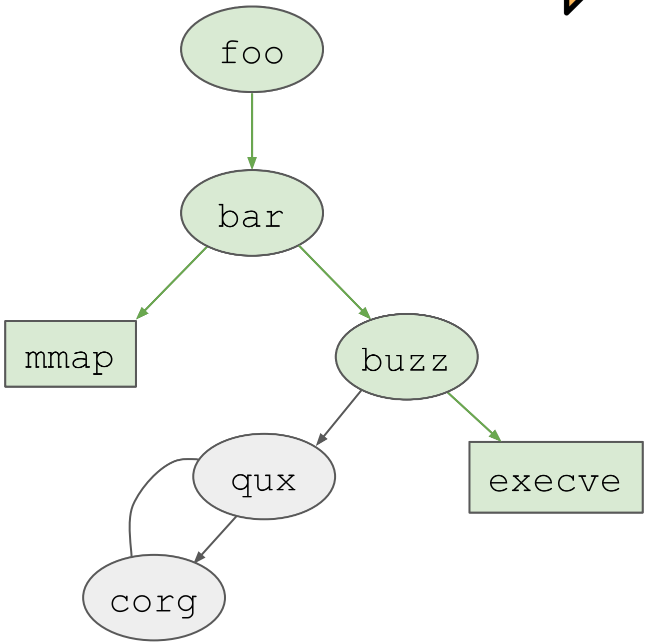
Call Type Context Metadata

BASTION Compiler - Control Flow Context

Goal: Prevent control-flow hijacking from reaching sensitive system calls!

Analysis Procedure:

- Derive application **Control Flow Graph (CFG)**
- Start at each sensitive system call callsite
- Recursively record each ***callee→caller*** association
- Metadata contains all valid CFG paths for each sensitive system call



Example CFG

Call chains of sensitive system calls are usually short!

BASTION Compiler - Argument Integrity Context

Goal: Ensure variables passed as arguments are never corrupted! (Data value integrity)

Protection Scope:

Sensitive system call arguments can be two (2) types:

1) **Direct Arguments** (non-pointer variables)
   - Only need to check value

⭐ Most arguments are constant!
Constant arguments cannot be altered.

Example 3

```
void foo ( int f0, char* f1 ){
   f2 = getConfigString(f0, f1);
   int flags = MAP_ANON|MAP_SHARED;
   bar( x1, flags );
   ...
}
void bar ( char*  b1, int b2 ){
   int prots = PROT_READ|PROT_WRITE;
   mmap( NULL, gshm->size, prots, b2,-1, 0 );
   ...
}
```

Arg1 ?= NULL
Arg2 ?= 4096
Arg3 ?= PROT_READ | PROT_WRITE
Arg4 ?= MAP_ANON | MAP_SHARED
Arg5 ?= -1
Arg6 ?= 0

BASTION Compiler - Argument Integrity Context

Goal: Ensure variables passed as arguments are never corrupted! (Data value integrity)

Protection Scope:

Sensitive system call arguments can be two (2) types:

2) **Extended Arguments** (pointers/pointer fields)
   - Need to check pointer address &
   - Need to check value

Example 1

```
// nginx/src/os/unix/ngx_process.c
ngx_execute_proc(ngx_cycle_t *cycle, void *data){

 ngx_exec_ctx_t *ctx = data;
 ...

if(execve(ctx->path,ctx->argv,ctx->envp)==-1)
    ngx_log_error(NGX_LOG_ALERT, cycle->log,
        ngx_errno, "execve() failed");
```

Arg1 Adr ?= `0x5efe9000`
Arg1 Val ?= `"/usr/local/nginx"`

Arg1 Adr ?= `0x5efe9004`
Arg1 Val ?= `"config=/usr/local/def.conf"`

Arg3 Adr ?= `0x00000000`
Arg3 Val ?= `NULL`

BASTION Compiler - Argument Integrity Context

Instrumentation

`ctx_write_mem()`

- Added at each `write` operation
- Update sensitive variable values

`ctx_bind_mem()/ctx_bind_const()`

- Added at each associated callsite
- Provide staging for performing runtime checking

Procedure:

- Work backwards from each callsite
- **Use-Def chains** derived from LLVM IR
- Internals automatically handle:
  - Direct vs Extended Arguments

Call depth to set system call arguments is fairly shallow – within the same function or only a few functions away.

Example 3

```
void foo ( int f0, char* f1 ){
    f2 = getConfigString(f0, f1);

    int flags = MAP_ANON|MAP_SHARED;
    ctx_write_mem(&flags, sizeof(int));
    ctx_bind_mem_2(&flags);
    bar( x1, flags );
    ...
}
void bar ( char*  b1, int b2 ){
    ctx_write_mem(&b2, sizeof(int));
    int prots = PROT_READ|PROT_WRITE;
    ctx_write_mem(&prots, sizeof(int));

    ctx_bind_const_1(NULL);
    ctx_bind_mem_2(&gshm->size);
    ctx_bind_mem_3(&prots);
    ctx_bind_mem_4(&b2);
    ctx_bind_const_5(-1);
    ctx_bind_const_6(0);
    mmap( NULL, gshm->size, prots, b2, -1, 0 );
    ...
```

constant | global variable | local variable | caller parameter

BASTION Compiler - Context Metadata Summary

Context MetaData

- Each context extracts different metadata to enforce what is given to monitor

Call Type Context

Non-sensitive System Call List

- **Whitelisted** syscall numbers

Sensitive System Call List

- **Directly-Callable** syscall numbers
- **Indirectly-Callable** syscall numbers
- File offsets of all indirect callsites

Whitelisted	`read,gettimeofday,...`
Directly-Callable	`mmap,socket,connect,...`
Indirectly-Callable	`accept4`
Indirect Callsite Binary Offsets	`343,9238,2341,1192, ...`

Control Flow Context

Association list of valid **callee → caller**

Valid Control Flow Paths
`mmap < bar`
`bar < foo`
`execve < buzz`
`buzz < bar`

Argument Integrity Context

For *sensitive system call call-sites*:

- Argument Type
  - Constant
  - Variables
  - Caller Parameters

For *caller parameter call-sites:*

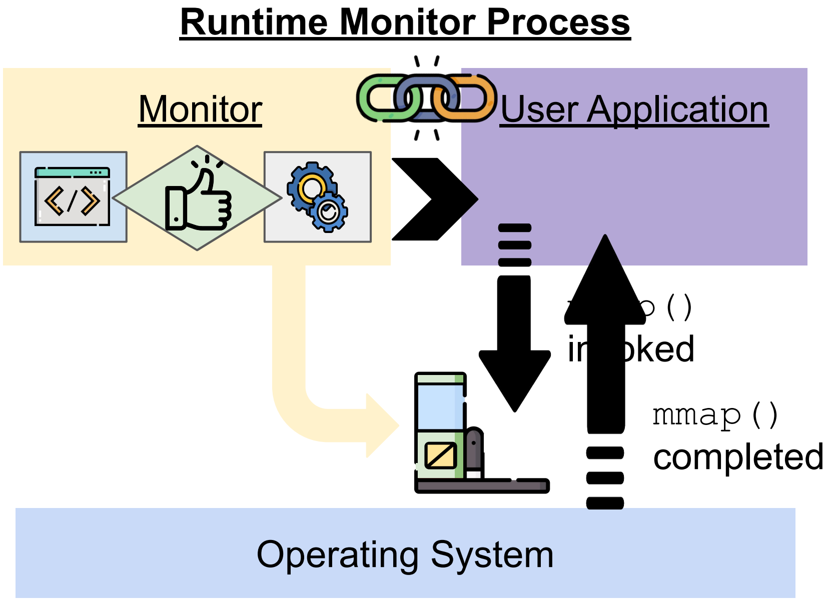
- Argument # that be must checked
- Argument Type

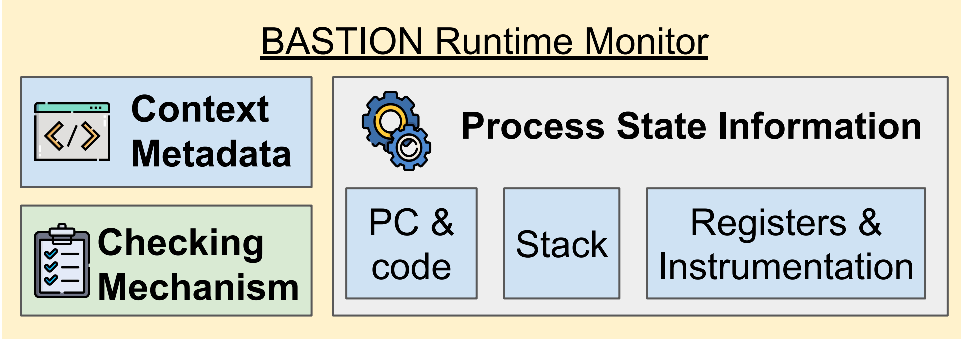
Callsites (`D` = `Direct Arg`, `Ext` = `Extended Arg`)
`396: bar(IGNORE, D)`
`441: mmap(D, D, D, D, D, D)`
`983: execve(Ext, Ext, Ext)`

BASTION Design - Monitor Component

Monitor Goals:

- Act as liaison between application and OS
    - Safeguard system calls from arbitrary use!
- Separate process
    - Isolates BASTION from untrusted application!
    - Attacker cannot bypass/disable BASTION hooks
- Only check contexts when system call invoked
    - Minimize interference for max performance!





BASTION Design - Checking Call Type Context

- Every callsite occurs at specific address
- Use Program Counter (PC) to get current binary offset
- Inspect OPCODE for call type (**Directly Callable** vs **Indirectly-Callable**)

intel®

INSTRUCTION SET REFERENCE

CALL—Call Procedure

Opcode	Instruction	Description
E8 *cw*	CALL *rel16*	Call near, relative, displacement relative to next instruction
E8 *cd*	CALL *rel32*	Call near, relative, displacement relative to next instruction
FF /2	CALL *r/m16*	Call near, absolute indirect, address given in *r/m16*
FF /2	CALL *r/m32*	Call near, absolute indirect, address given in *r/m32*
9A *cd*	CALL *ptr16:16*	Call far, absolute, address given in operand
9A *cp*	CALL *ptr16:32*	Call far, absolute, address given in operand
FF /3	CALL *m16:16*	Call far, absolute indirect, address given in *m16:16*
FF /3	CALL *m16:32*	Call far, absolute indirect, address given in *m16:32*

BASTION Design - Checking Control Flow Context

- Leverage current process stack against **callee → caller Association List**

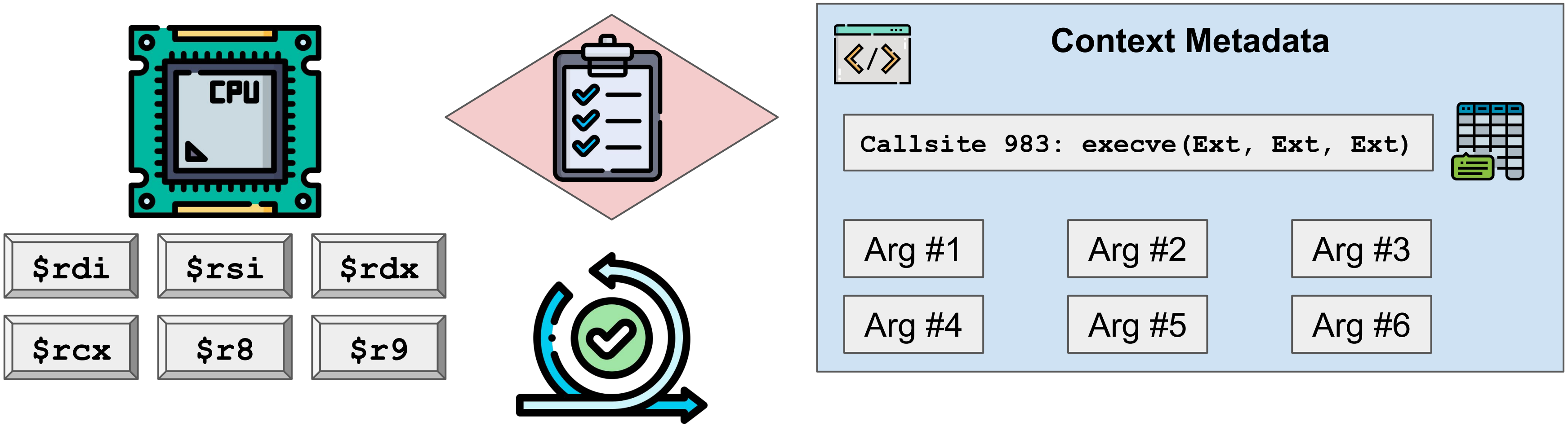
Valid Control Flow Paths
`mmap < bar`
`bar < foo`
`execve < buzz`
`buzz < bar`

Current Process Stack

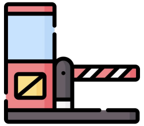
Frame 3:	`foo`
Frame 2:	`bar`
Frame 1:	`buzz`
Frame 0:	`execve`

BASTION Design - Checking Argument Integrity

1. Fetch register values
2. Fetch Context Metadata
   - Callsite Information
   - Expected Values
3. Perform check for each callsite
4. Unpause application



BASTION Runtime Checking



Application HALTED

How System Call Checking Works:

1. `seccomp` catches each sensitive system call for BASTION

2. `ptrace` fetches application process state for BASTION

3. **Checking Procedure**
   - Confirm valid Call Type
   - Confirm valid Control Flow
   - Confirm Argument Integrity (for all parameters)
     - Fetch expected values from hashtable
   - Any inconsistency triggers program HALT

Example 1

```
// nginx/src/os/unix/ngx_process.c
static void ngx_execute_proc(ngx_cycle_t *cycle,void *data){

   ngx_exec_ctx_t *ctx = data;

   if( execve("/bin/bash",ctx->argv, ctx->envp ) == -1){

     ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
           "execve() failed");
```

Call Type

Control Flow

Argument Integrity

Outline

- Introduction
- Background
- MARDU: Practical Randomization
- BASTION: Securing System Calls
  - Motivation
  - Design
  - Implementation
  - Evaluation
- Future Work & Conclusion

5

Illegitimate System Call Usage

System Call Filtering

Implementation

- Working Prototype
- BASTION Compiler
  - LLVM 10.0.0
  - ~4K LoC
- BASTION Library API
  - ~700 LoC
- BASTION Monitor
  - ~8K LoC
  - `seccomp-BPF`
  - `ptrace`
- Kernel
  - X86-64 Linux 5.19.14

LLVM
Compiler Infrastructure

Outline

- Introduction
- Background
- MARDU: Practical Randomization
- BASTION: Securing System Calls
  - Motivation
  - Design
  - Implementation
  - Evaluation
- Future Work & Conclusion

5

Illegitimate System Call Usage

System Call Filtering

BASTION Evaluation

Evaluation Summary

- Performance: System-call & I/O Intensive Applications
- Security: 32 ROP payloads, real-world CVEs, & synthesized attacks

Evaluation Questions

Security

1) How secure is BASTION?
2) How does BASTION defend against different attack strategies?
3) How does BASTION compare to other security archetypes?

Performance

4) What is each context's performance impact?
5) How much overall performance overhead does BASTION impose?

BASTION Evaluation

Experimental Setup

- All programs compiled via BASTION LLVM compiler (`-fPIC` optimization flags)
- Platform:
  - 8-core (16-hardware thread) machine featuring AMD Ryzen 7 PRO 5850U CPU
  - 16 GB DRAM

Applications

- NGINX Most widely deployed web server
- SQLite Database Engine
- vsftpd FTP Server

How BASTION Defends

32-Attack Case Study

ROP Payloads (18)

- Stack pivot gives away ROP chain

Direct System Call Manipulation (9)

- Naive attacks corrupting function pointers

Indirect System Call Manipulation (5)

- Advanced attacks mimic valid program behavior to varying degrees
- All attacks attempt to corrupt arguments

Core Summary

- BASTION foremost protects the system to not be compromised
- BASTION protects against attacks other fine-grained defenses cannot

	Violated Context		
Attack Category	Call Type	Control Flow	Argument Integrity
Return-Oriented Programming (18)	✗	✓	✓
Direct System Call Manipulation (9)	✓	✓	✓
Indirect System Call Manipulation (5)			
NEWTON CPI Attack *[SIGSAC'17]*	✓	✓	✓
AOCR Apache Attack *[NDSS'17]*	✗	✓	✓
AOCR NGINX Attack 2 *[NDSS'17]*	✗	✗	✓
COOP *[S&P'15]*	✗	✗	✓
Control Jujutsu *[CCS'15]*	✗	✗	✓

Advanced defenses:
Code Pointer Integrity *[OSDI'14]*
Context Sensitive CFI e.g., **GRIFFIN *[ASPLOS'17]***
cannot defend against these advanced attacks!

Defending Indirect System Call Manipulation

Example: AOCR NGINX Attack 2

Main Loop

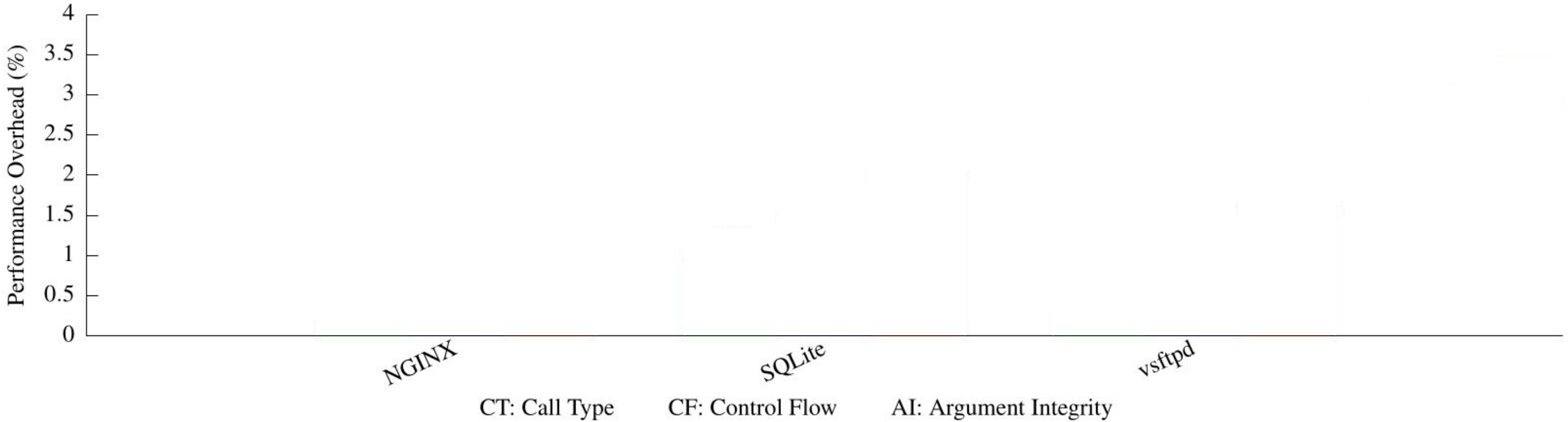
```
void ngx_master_process_cycle(){
    ...
    if (ngx_change_binary) {
        ...
        ngx_new_binary = ngx_exec_new_binary(cycle, ngx_argv);
    }
```

Leverage memory corruption vulnerability to:

1. Change parameters (`ngx_cycle_t cycle, char **ngx_argv`) to attacker-controlled values
2. Trigger `ngx_change_binary` signal to fall into `nginx_exec_new_binary()`

BASTION Argument Integrity Context detects argument corruption in `ngx_argv` and halts execution!

How BASTION Performs



- **Argument Integrity** Context is BASTION’s **most expensive** context to deploy
- BASTION **overall performance overhead** is **low** (<2.01%)

BASTION Summary

- BASTION is practical
- **System-call-specialized coverage** minimizes defense interference
- Security **blocks all ROP & attack classes** that rely on **leveraging system calls**
- BASTION hardens system calls using **three new contexts** to accomplish System Call Integrity
- **2.01%** worst-case performance overhead
- BASTION can be used as starting framework to protect against other system call threats

Outline

- Introduction
- Background
- MARDU: Practical Randomization
- BASTION: Securing System Calls
- Future Work & Conclusion

Future Work

BASTION

- **Extend BASTION** to be a generic defense framework
  - Implementing as **Kernel Module** would ensure maintained fast performance as more features added:
    - More system call classes support
    - More fine-grained contexts
  - Protect beyond ROP-originating attacks

- Augment BASTION protection to cover **Information Disclosure**
  - `read, write, open, sendfile64, sendto, sendmsg, sendmmsg`

- Perform systematic research to **classify and quantify the security threat** each available system call imposes on the host OS
  - Opportunity to:
    - Better map system call abilities to attacker intentions
    - Uncover new system call attack classes

Operating System Enhancements to Prevent the Misuse of System Calls [CCS'00]

Conclusion

Problem Statement:

How can one create strong, yet practical exploit mitigation designs?

Dissertation Aims & Objectives:

- *Understand & identify critical attack/defense aspects to devise practical exploit mitigation designs*
- *Show through prototyping that practical designs are achievable*

Dissertation Contributions:

MARDU [SYSTOR'20 / DTRAP'22]

- On-demand, reactive re-randomization
- Capable of code sharing randomized code system-wide
- Capable of randomizing without pausing execution

BASTION [ASPLOS'23 - Major Revision]

- Fine-grained system call filtering
- 3 contexts to enforce System Call Integrity
- Can block all attack classes that rely on system calls to carry out attack
- *Notification - Jan. 19, 2023*

Publications

BASTION

Protect the System Call, Protect (most of) the World with BASTION (*ASPLOS '23* - Major Revision - Notification Jan 19, 2023)
Christopher Jelesnianski, Mohannad Ismail, Yeongjin Jang, Dan Williams, Changwoo Min

MARDU

Securely Sharing Randomized Code that Flies
Christopher Jelesnianski, Jinwoo Yom, Changwoo Min, and Yeongjin Jang **(*DTRAP '22*)**

MARDU: Efficient and Scalable Code Re-randomization
Christopher Jelesnianski, Jinwoo Yom, Changwoo Min, and Yeongjin Jang **(*SYSTOR '20*)**

Other Security

Tightly Seal Your Sensitive Pointers with PACTight
Mohannad Ismail, Andrew Quach, **Christopher Jelesnianski**, Yeongjin Jang and Changwoo Min **(*USENIX Security '22*)**

- Data protection utilizing ARM Pointer Authentication (PA) security primitive

VIP: Safeguard Value Invariant Property for Thwarting Critical Memory Corruption Attacks
Mohannad Ismail, Jinwoo Yom, **Christopher Jelesnianski**, Yeongjin Jang and Changwoo Min **(*CCS '21*)**

- Value Integrity for security-relevant data types

Publications

Compilers & System Software

Breaking the Boundaries in Heterogeneous-ISA Datacenters
Antonio Barbalace, Robert Lyerly, **Christopher Jelesnianski**, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran **(*ASPLOS'17*)**

Operating system process and thread migration in heterogeneous platforms
Robert Lyerly, Antonio Barbalace, **Christopher Jelesnianski**, Vincent Legout, Anthony Carno, and Binoy Ravindran **(*MaRS'16*)**

Popcorn: Bridging the programmability gap in heterogeneous-isa platforms
Antonio Barbalace, Marina Sadini, Saif Ansary, **Christopher Jelesnianski**, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran **(*Eurosys'15*)**

Acknowledgements

Ph.D. Committee:

- Dr. Changwoo Min
- Dr. Yeongjin Jang
- Dr. Wenjie Xiong
- Dr. Danfeng Yao
- Dr. Haibo Zeng

Achieving Harmony in Practical Exploit Mitigation Design Against Code Re-Use Attacks and System Call Abuse

Christopher Jelesnianski

Thank you!

Questions?

VIRGINIA TECH™

I’m Headed to Apogee Research

https://apogee-research.com/

Arlington, Virginia

Timing

12 min (6 + 4 + 4)

Introduction = Done (6 Min)
Background = Done (4 Min)
MARDU = Done (4 Min)

BASTION = (25 Min)
    Motivation = 10 Min >> 6
    Design = 10 Min >> 14
    Implementation = Done (0 Min)
    Evaluation = (5 Min)

Future Work + Conclusion = Done (5 Min)

Target = 43 minutes (13+25+5)

Back-up Slides

VIRGINIA TECH™

BASTION Insights

- Some system calls are called more than others (e.g., `accept4` vs `connect`)
- System calls are **sparsely** used
- System calls are **called indirectly very rarely**
- **Constant arguments** are **common**

Application	NGINX	SQLite	vsftpd
Total # application callsites	7,017	12,253	4,695
Total # arbitrary direct callsites	6,692	12,026	4,688
Total # arbitrary in-direct callsites	325	227	7
Total # sensitive callsites	26	13	12
Total # sensitive system calls called indirectly	0	0	0
`ctx_write_mem()`	5,226	1,337	204
`ctx_bind_mem()`	43	18	33
`ctx_bind_const()`	18	13	9
Total instrumentation sites	5,287	1,368	246

Application	NGINX (32 workers)	SQLite	vsFTPd
`execve`	0	0	0
`execveat`	0	0	0
`fork`	0	0	0
`vfork`	0	0	0
`clone`	96	48	36
`ptrace`	0	0	0
`mprotect`	334	501	7
`mmap`	534	42	33
`mremap`	0	0	0
`remap_file_pages`	0	0	0
`chmod`	0	0	0
`setuid`	32	0	12
`setgid`	32	0	12
`setreuid`	0	0	0
`socket`	32	1	85
`connect`	32	0	8
`bind`	1	1	77
`listen`	2	1	77
`accept`	0	11	87
`accept4`	5,665	0	0
Total BASTION monitor hook	6,713	557	433

BASTION Contexts - Call Type Context (Indirect Call)

Indirectly Callable Type

- BASTION checks all right hand assignments for sensitive system calls

System Call	Call Type
execve	Directly-Callable, Indirectly-Callable

Running Example 2

```
int(*cmd_fp)(char*, char*, char*);

void foo ( int f0, char* f1 ){
    if(SETUP)
        cmd_fp = &execve;
    else
        cmd_fp = &custom_exec;
}
void custom_exec ( char*  e1, char*
e2, char* e3 ){
    ...
    execve( ... );
}
```