Protect the System Call, Protect (most of) the World with BASTION

Christopher Jelesnianski, Mohannad Ismail, Yeongjin Jang*, Dan Williams, Changwoo Min

VIRGINIA TECH

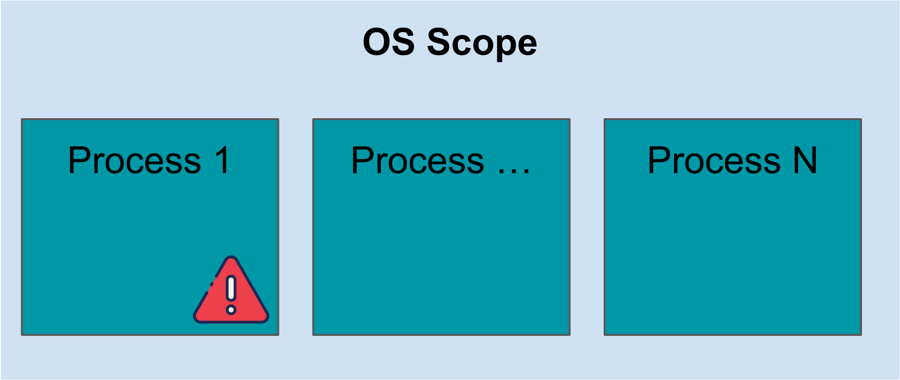
* Oregon State University

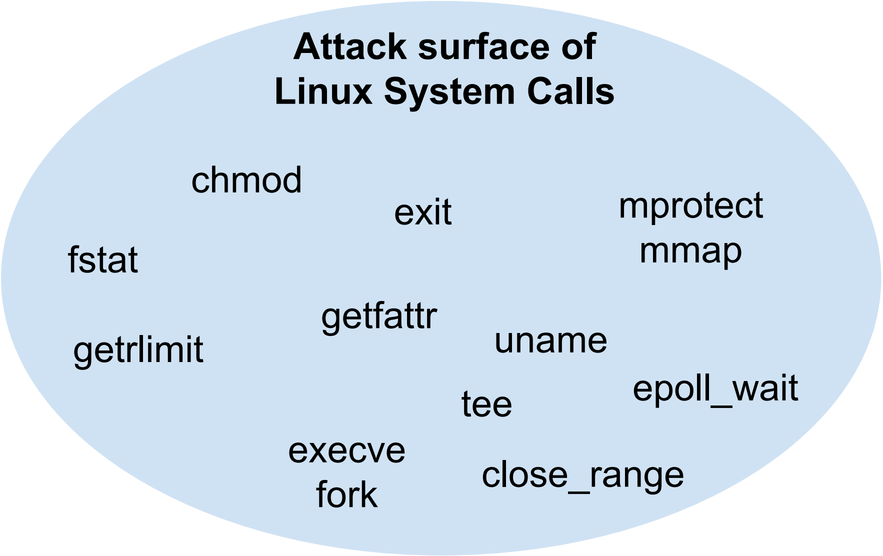
Takeaway

- System Calls are important
  - **Core API interface** between *processes* and the *Operating System*
  - **Prevalent medium** for code reuse to compromise entire system from a vulnerable application
- Minimal guarding of System Calls
  - Linux `seccomp`
  - Eliminating surface area instead of eliminating abuse
  - **Coarse-grained** defenses
- **System Call Integrity**: A targeted methodology to shore up system call defenses
  - **Protection of the system**, *not protection of the application*
  - Fine-grained & specialized protection that is efficient *and* strong

Medium for Critical Attacks

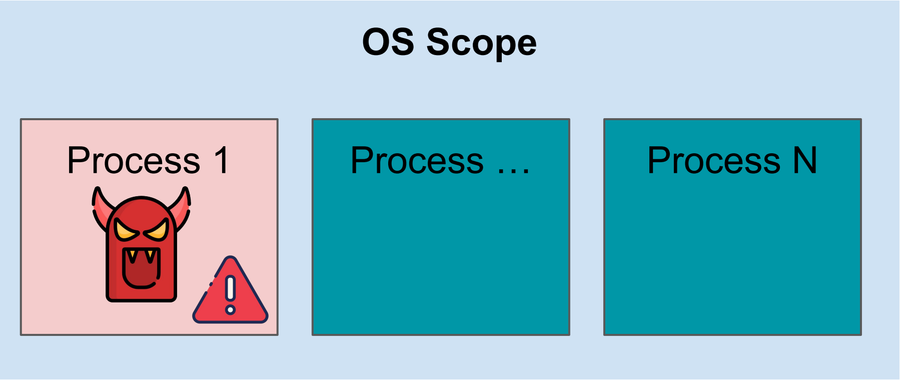
- Many **code re-use attacks end-goal** require leveraging a system call
  - Memory vulnerabilities continue to persist
  - Attacker *intermediate* steps may cause undefined behavior in application
  - But, cannot leave application process scope **without system call**
- Majority system calls are **non-security sensitive**

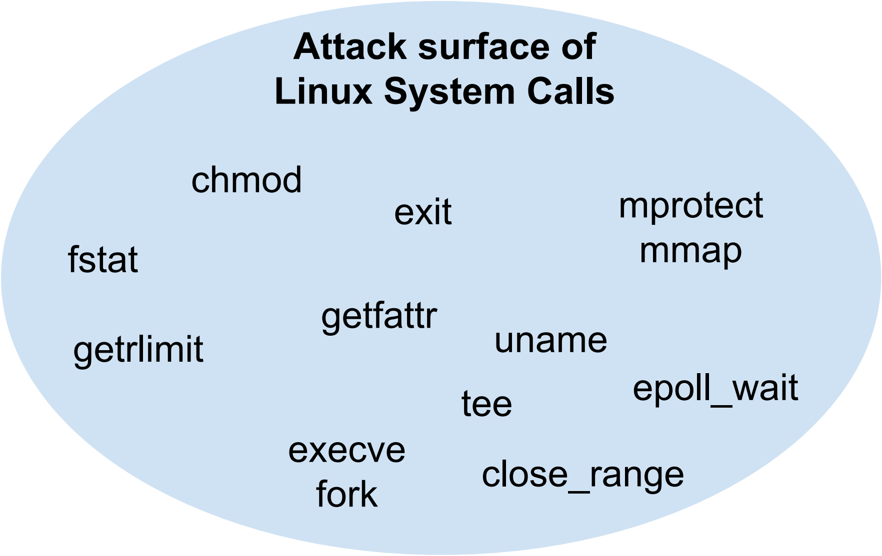




Medium for Critical Attacks

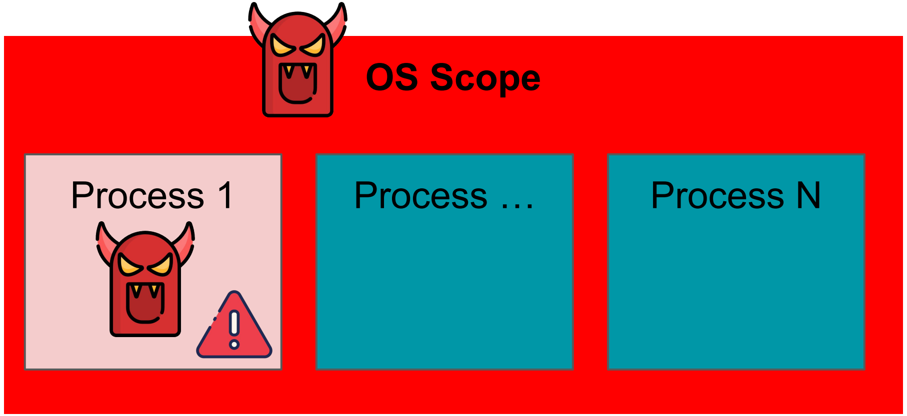
- Many **code re-use attacks end-goal** require leveraging a system call
  - Memory vulnerabilities continue to persist
  - Attacker *intermediate* steps may cause undefined behavior in application
  - But, cannot leave application process scope **without system call**
- Majority system calls are **non-security sensitive**

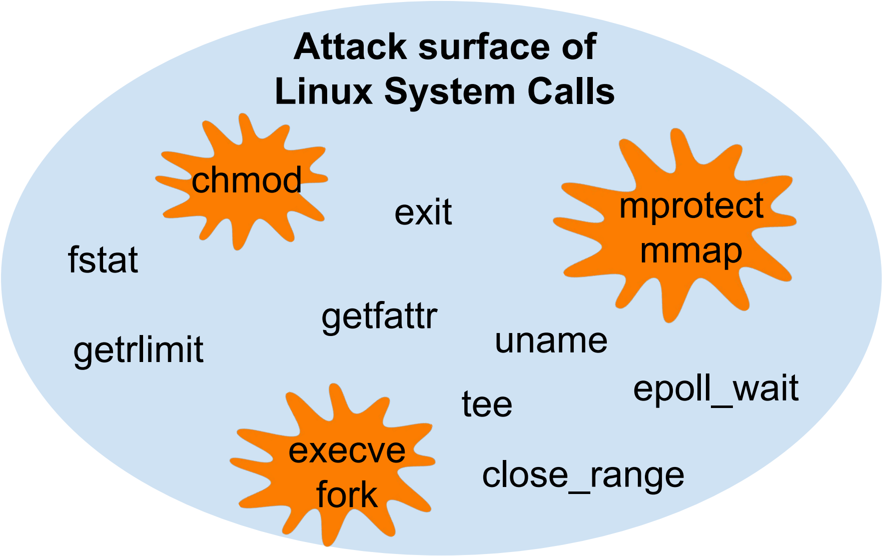




Medium for Critical Attacks

- Many **code re-use attacks end-goal** require **leveraging a system call**
  - Memory vulnerabilities continue to persist
  - Attacker *intermediate* steps may cause undefined behavior in application
  - But, cannot leave application process scope **without system call**
- Majority system calls are **non-security sensitive**





System Call Defenses *(and why they don't do enough)*

Defenses

- Linux `seccomp`
  - *Linux deployed coarse-grained allowlist/denylist*
- Automated System Call Filtering
  - *`sysfilter`: Automated system call filtering for commodity software* [RAID'20]
- Refined Whitelisting
  - *Temporal System Call Specialization* [USENIX Sec'20]

Bottom Line

- Coarse-grained filtering is not sufficient
- System calls cannot be disabled because of **core process necessity**
  - Coincidently are **targeted for attacker abuse**
  - e.g., **`execve`**, **`mmap`**, **`mprotect`**
- Instead of finding system call minimal set, ***find meaningful context surrounding system calls***!

Our Work: Introduction of **System Call Integrity**

```
execve( ctx->path, ctx->argv, ctx->envp );
```

Our Work: Introduction of **System Call Integrity**

- System Call Integrity
  - Comprised of **three contexts** 
  - Based on attacker pattern insight

<u>Attacker Pattern Insight:</u>

1. How are system calls invoked?
2. How are system calls reached?
3. What is passed to system calls?

```
execve( ctx->path, ctx->argv, ctx->envp );
```

Our Work: Introduction of **System Call Integrity**

- System Call Integrity
  - Comprised of **three contexts** 
  - Based on attacker pattern insight

Call-Type Context

Is this system call allowed to be called indirectly?

Attacker Pattern Insight:

1. How are system calls **invoked**?
2. How are system calls reached?
3. What is passed to system calls?

1

```
execve( ctx->path, ctx->argv, ctx->envp );
```

Our Work: Introduction of **System Call Integrity**

- System Call Integrity
    - Comprised of **three contexts** 
    - Based on attacker pattern insight

Call-Type Context
Is this system call allowed to be called indirectly or at all?

Control-Flow Context
Does the live stack trace match expected program control-flow?

Attacker Pattern Insight:

1. How are system calls **invoked**?
2. How are system calls **reached**?
3. What is passed to system calls?

2

1

```
execve( ctx->path, ctx->argv, ctx->envp );
```

Our Work: Introduction of **System Call Integrity**

- System Call Integrity
  - Comprised of **three contexts** 
  - Based on attacker pattern insight

Attacker Pattern Insight:

1. How are system calls **invoked**?
2. How are system calls **reached**?
3. **What is passed** to system calls?

Call-Type Context
Is this system call allowed to be called indirectly?

Control-Flow Context
Does the live stack trace match expected program control-flow?

Argument Integrity Context
Are any arguments corrupted?

2

1 3

```
execve( ctx->path, ctx->argv, ctx->envp );
```

System Call Integrity - 1 - **Call Type Context**

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

- Assigned Per-System-Call
- 3 Types

Example

```
void foo ( int f0 ){

  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 Integrity - 1 - **Call Type Context**

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

- Assigned Per-System-Call
- 3 Types

Not-Callable

Example

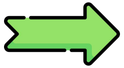
```
void foo ( int f0 ){

  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 Integrity - 1 - **Call Type Context**

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

- Assigned Per-System-Call
- 3 Types

Not-Callable



Directly Callable
traditional direct call

Example

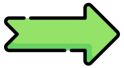
```
void foo ( int f0 ){

   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 Integrity - 1 - **Call Type Context**

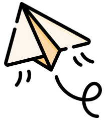
Guarantee: Only permitted system calls are allowed to be called in their expected manner

- Assigned Per-System-Call
- 3 Types

Not-Callable



Directly Callable
traditional direct call



Indirectly-Callable
code pointers

Example

```
void foo ( int f0 ){

  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 );
  ...
}
```

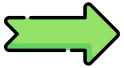
System Call Integrity - 1 - **Call Type Context**

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

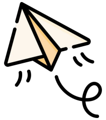
- Assigned Per-System-Call
- 3 Types

Applicable to All System Calls

Not-Callable



Directly Callable
traditional direct call



Indirectly-Callable
code pointers

Example

```
void foo ( int f0 ){

  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 Integrity - 1 - **Call Type Context**

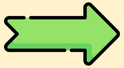
Guarantee: Only permitted system calls are allowed to be called in their expected manner

- Assigned Per-System-Call
- 3 Types

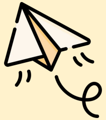
Applicable to All System Calls

Not-Callable

Sensitive System Calls Only



Directly Callable
traditional direct call



Indirectly-Callable
code pointers

Example

```
void foo ( int f0 ){

   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 Integrity - 1 - **Call Type Context**

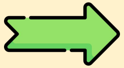
Guarantee: Only permitted system calls are allowed to be called in their expected manner

- Assigned Per-System-Call
- 3 Types

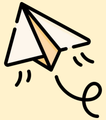
Applicable to All System Calls

Not-Callable

Sensitive System Calls Only



Directly Callable
traditional direct call



Indirectly-Callable
code pointers

Example

```
void foo ( int f0 ){

   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 Integrity - 1 - **Call Type Context**

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

- Assigned Per-System-Call
- 3 Types

Applicable to All System Calls

Not-Callable

Sensitive System Calls Only

Directly Callable
traditional direct call

Indirectly-Callable
code pointers

Sensitive system call use is **sparse** & rarely invoked **indirectly**.

Example

```
void foo ( int f0 ){

  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

System Call Integrity - 2 - **Control Flow Context**

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

Example

```
void foo ( int f0 ){

   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 Integrity - 2 - **Control Flow Context**

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

Call chains of sensitive system calls are usually **short**!

Example

```
void foo ( int f0 ){

   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
...

System Call Integrity - 3 - **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

```
void foo ( int f0 ){

    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 );
    ...
}
```

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

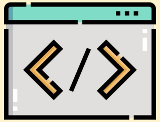
BASTION Overview - System Call Integrity in Practice

<u>BASTION Compiler</u>

- Static analysis
- Record metadata
- Sensitive variable instrumentation

<u>BASTION Runtime Monitor</u>

- Separate process
- Leverage context metadata
- Dynamic context checking



<u>User Application</u>

<u>Operating System</u>

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

BASTION Compiler - Argument Integrity Context

Procedure

- Instrumented as inline assembly
- Use variable **use-def chains** derived from LLVM IR
- **Static and dynamic** variable support

Instrumentation

```
void foo ( int f0 ){
    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 );
    ...
```

constant | global variable | local variable | caller parameter

BASTION Compiler - Argument Integrity Context

Procedure

- Instrumented as inline assembly
- Use variable **use-def chains** derived from LLVM IR
- **Static and dynamic** variable support

Instrumentation

`ctx_write_mem()`

- Added at each argument `write` operation

```
void foo ( int f0 ){
    int flags = MAP_ANON|MAP_SHARED;
    ctx_write_mem(&flags, sizeof(int));
    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));
    mmap( NULL, gshm->size, prots, b2, -1, 0 );
    ...
```

constant | global variable | local variable | caller parameter

BASTION Compiler - Argument Integrity Context

Procedure

- Instrumented as inline assembly
- Use variable **use-def chains** derived from LLVM IR
- **Static and dynamic** variable support

Instrumentation

`ctx_write_mem()`

- Added at each argument `write` operation

`ctx_bind_mem()/ctx_bind_const()`

- Stages expected values for performing runtime checking

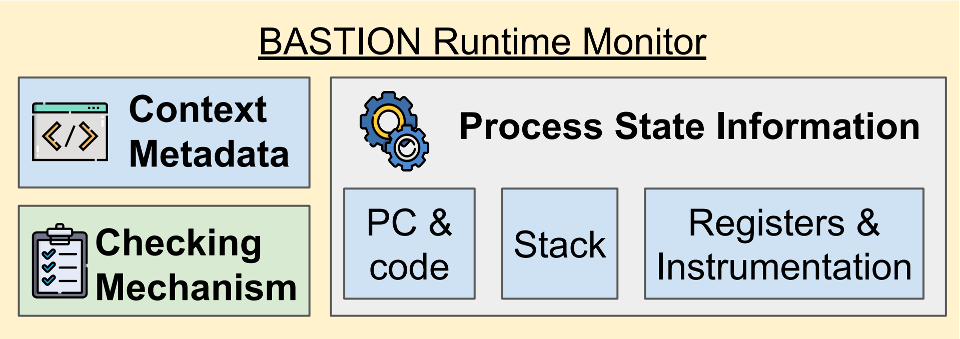
```
void foo ( int f0 ){


    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

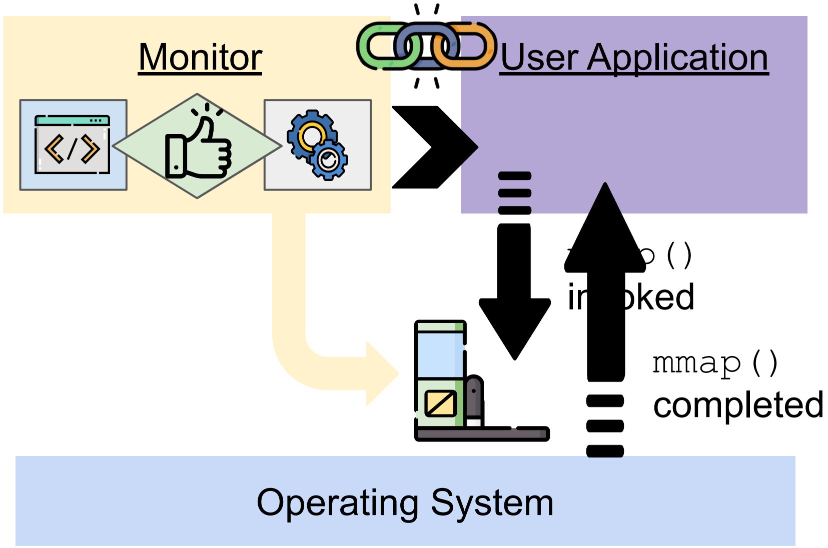
BASTION Design - Monitor Component

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!



Runtime Monitor Procedure



BASTION Prototype Implementation

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

LLVM
Compiler Infrastructure

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

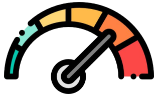
`socket, bind, connect, listen, accept, accept4`

BASTION Evaluation

Evaluation Summary

- Performance: System-call & I/O Intensive Applications
    - **NGINX** - Most widely deployed web server
    - **SQLite** - Database Engine
    - **vsFTPd** - FTP server
- Security: **32 Attack Study**: ROP payloads, real-world CVEs, & synthesized attacks

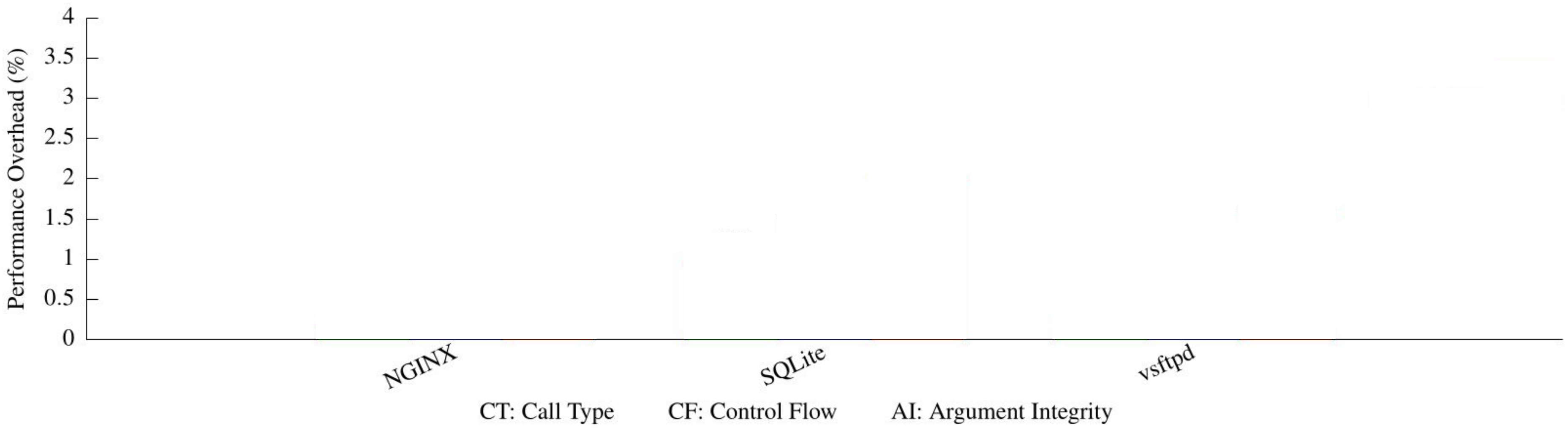
Evaluation Questions



Performance
1) What is each context's performance impact?
2) How much overall performance overhead does BASTION impose?

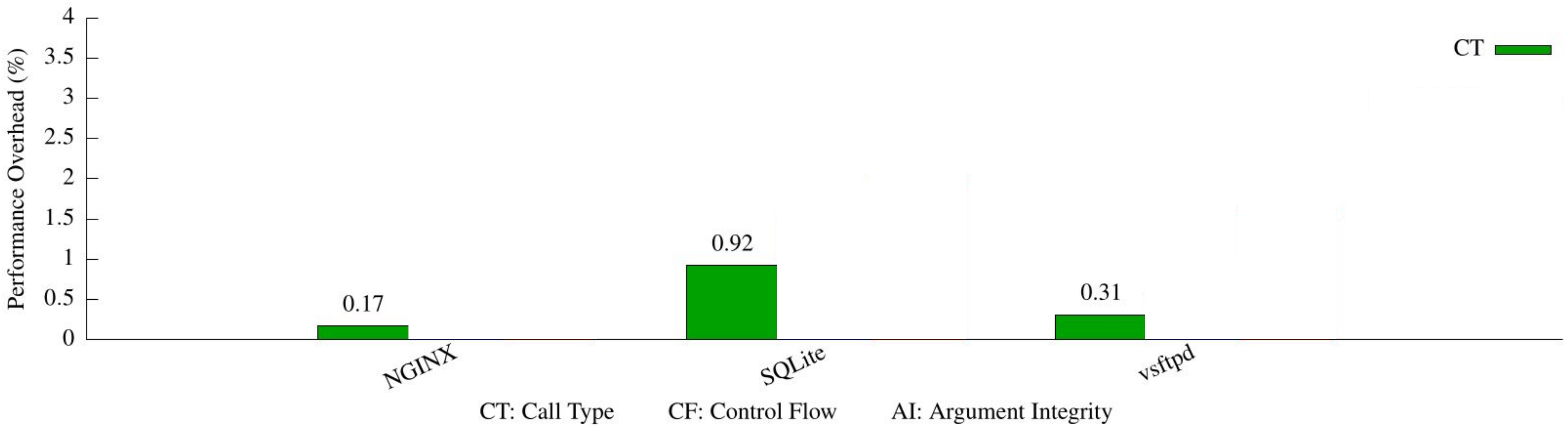
Security
1) How secure is BASTION?
2) How does BASTION defend against different attack strategies?
3) How does BASTION compare to other security archetypes?

BASTION Performance



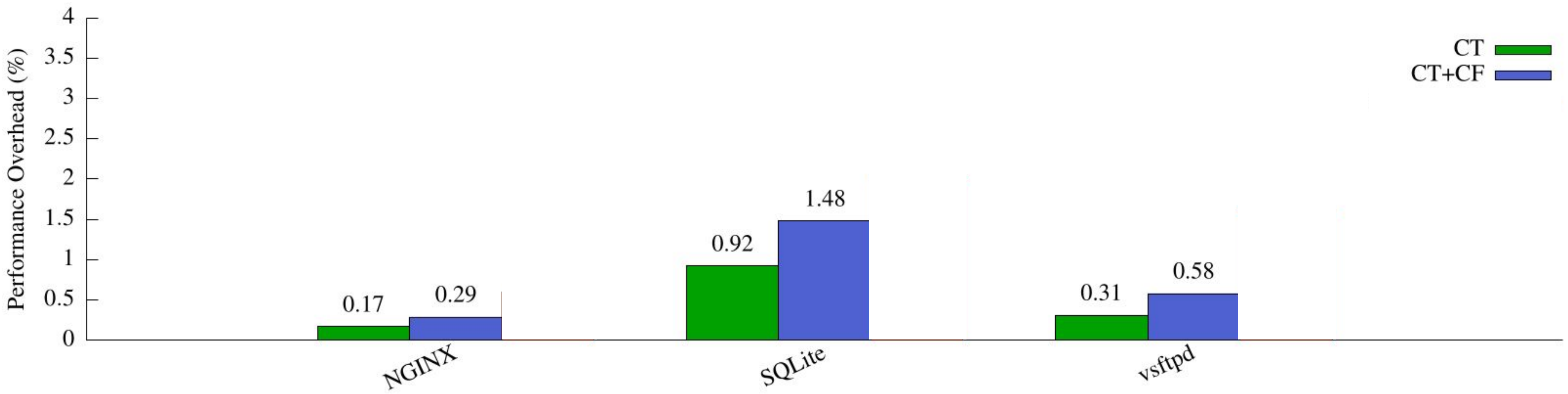
- **Argument Integrity** Context is BASTION's **most expensive** context to deploy
- BASTION **overall performance overhead** is **low** (<2.01%)

BASTION Performance



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

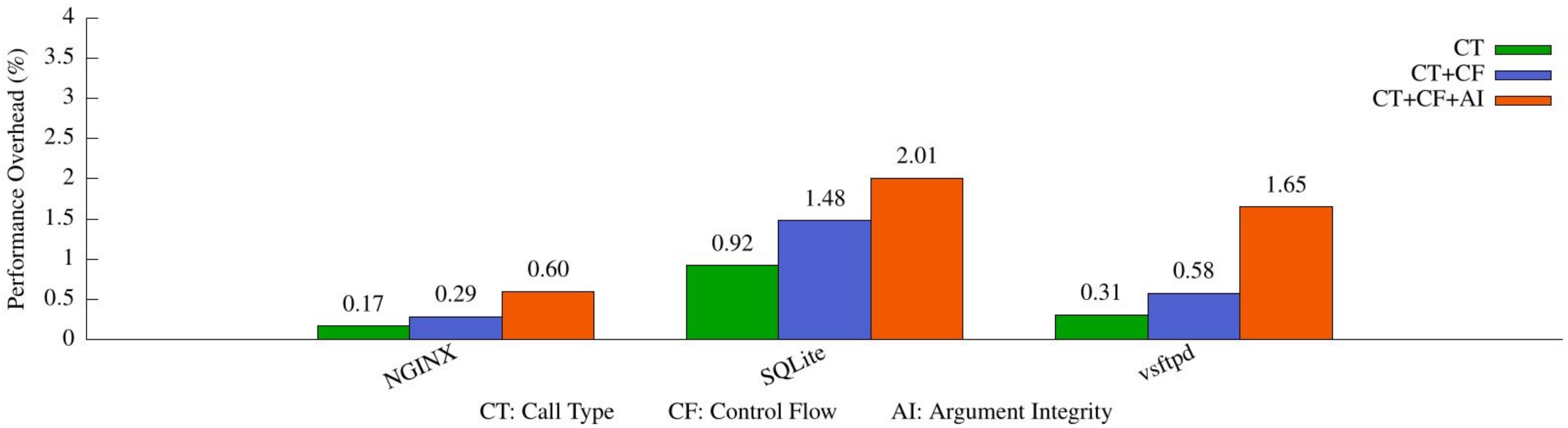
BASTION Performance



CT: Call Type CF: Control Flow AI: Argument Integrity

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

BASTION Performance



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

BASTION Security Analysis

	Violated System Call Integrity Context		
Attack Category	**Call Type**	**Control Flow**	**Argument Integrity**
Return-Oriented Programming (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 • All attacks attempt to corrupt arguments			
NEWTON CPI Attack *[SIGSAC'17]*	✗	✓	✓
AOCR Apache Attack *[NDSS'17]*	✗	✓	✓
AOCR NGINX Attack 2 *[NDSS'17]*	✗	✗	✓
COOP *[S&P'15]*	✗	✗	✓
Control Jujutsu *[CCS'15]*	✗	✗	✓

Conclusion

System Calls are an attacker gateway

- Coarse-grained filtering is not enough
- System call protection needs to be fine grained to be effective

System Call Integrity

- System Call Integrity hardens system calls by applying three specialized contexts
- Specialized coverage minimizes CPU interference while maximizing security around system calls

Looking Towards the Future

- BASTION can be a stepping stone to enable configurable system call protection
- BASTION can be expanded to add future contexts to protect against yet unknown system call threats
- BASTION can be used as starting framework to protect against other system call threats

EXTRA SLIDES

BASTION System Call Statistics

- Some system calls are called more than others (e.g., `accept4` vs `connect`)
- System calls have **sparse** callsites
- System calls **very rarely invoked indirectly**
- **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

Other Considerations

Attacks able to bypass BASTION?

- (subset of) **Data-only attacks**
- In practice, will be difficult to overcome BASTION constraints
  - most information can be deduced from static analysis

Deploying BASTION to real-world (2 main challenges)

- performance overhead - fine-grained defenses do constant checks to minimize deviation from correct control flow

Comparison to CFI

- Call Type + Control Flow Context are **NOT equivalent to CFI**
- Call Type is **NOT per callsite**
- Control Flow is not application wide (only covers paths that eventually lead to system calls)

Effectiveness of BASTION under arbitrary memory corruption

- info gained from static analysis significantly raises security
- attacker would need to accurately recreate a fake version of all 3 contexts
- In practice this would require MANY read/write operations to match constraints all the while STILL obeying all static constraints deduced from BASTION analysis

Other Considerations 2

Selection of "Sensitive System Calls"

- Targets system calls enabling common attacker strategies *aimed at escaping the scope of the victim application and reaching the underlying system*
  - arbitrary code execution
  - memory permission changes
  - privilege escalation
  - network reconfiguration
- We investigated open/write system call - this imposed significant performance overhead
  - We confirmed that overhead comes from fetching process state

Other competitors - Saffire (EuroS&P'20)

- Explore fine-grained syscall filtering (of arguments)
- BASTION is more secure as Saffire is a userspace solution (**works inside scope of vulnerable application**) and **relies on fine-grained CFI** to be in place to ensure their defense is not skipped
- BASTION is faster than Saffire since the **true performance cost** for them is: **CFI checking + Saffire checking**

Selection of benchmarks

- Did not look at compute bound benchmarks because these **very rarely** used security-sensitive system calls
- Further, all compute benchmarks **only used syscalls for initialization** of datasets and importing libraries. very very rarely during computation phase

BASTION System Call Statistics 2

- Even in the case of File system system calls, there was great contrast of call count (e.g., `open (light use)` vs `write (heavy use)` use in webserver)
- Heavy system call invocation bottlenecked BASTION at context switching (userspace/kernelspace)
- Would be resolved if BASTION was implemented directly in kernel (module)

BASTION Configuration	Runtime & % Overhead Added Per Checkpoint		
	NGINX	SQLite	vsftpd
BASTION + `file system syscalls` (seccomp hook only)	110.41 (0.15%)	36,993.27 (0.29%)	10.76 (0.08%)
BASTION + `file system syscalls` (fetch process state)	4.56 (95.88%)	7,461.18 (79.89%)	10.95 (1.85%)
BASTION + `file system syscalls` (full context checking)	3.65 (96.70%)	7,419.50 (80.00%)	11.01 (2.41%)