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

Christopher Jelesnianski[*]
Virginia Tech
Blacksburg, Virginia, USA
kjski@vt.edu

Mohannad Ismail
Virginia Tech
Blacksburg, Virginia, USA
imohannad@vt.edu

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

Dan Williams
Virginia Tech
Blacksburg, Virginia, USA
djwillia@vt.edu

Changwoo Min
Virginia Tech
Blacksburg, Virginia, USA
changwoo@vt.edu

ABSTRACT

System calls are a critical building block in many serious security attacks, such as control-flow hijacking and privilege escalation attacks. Security-sensitive system calls (*e.g.*, `execve`, `mprotect`), especially play a major role in completing attacks. Yet, few defense efforts focus to ensure their legitimate usage, allowing attackers to maliciously leverage system calls in attacks.

In this paper, we propose a novel *System Call Integrity*, which enforces the correct use of system calls throughout runtime. We propose three new contexts enforcing (1) which system call is called and how it is invoked (Call Type), (2) how a system call is reached (Control Flow), and (3) that arguments are not corrupted (Argument Integrity). Our defense mechanism thwarts attacks by breaking the critical building block in their attack chains.

We implement Bastion, as a compiler and runtime monitor system, to demonstrate the efficacy of the three system call contexts. Our security case study shows that Bastion can effectively stop all the attacks including real-world exploits and recent advanced attack strategies. Deploying Bastion on three popular system call-intensive programs, NGINX, SQLite, and vsFTPd, we show Bastion is secure and practical, demonstrating overhead of 0.60%, 2.01%, and 1.65%, respectively.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

System Call Protection, System Call Specialization, System Calls, Code Re-use Attacks, Argument Integrity, Exploit Mitigation

ACM Reference Format:
Christopher Jelesnianski, Mohannad Ismail, Yeongjin Jang, Dan Williams, and Changwoo Min. 2023. Protect the System Call, Protect (Most of) the World with BASTION. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3582016.3582066

[*]Now at Apogee Research, LLC

1 INTRODUCTION

System calls are a vital part in any program, providing an interface to interact with the operating system (OS). At the same time, they are a core component of various attacks (*e.g.*, arbitrary code execution, privilege escalation) enabling attackers to complete their attack.

Many defense techniques, such as debloating [24, 79, 80], system call filtering [36, 41, 42], and system call sandboxing [63, 94] aim to minimize the available attack surface by disabling unused system calls (*i.e.*, denylist). However, these defenses still allow system calls to be invoked, even if they are used illegitimately, as they remain needed for legitimate use as well.

In this paper, we propose *System Call Integrity*, a novel security policy that enforces the correct use of system calls in a program. The key idea is that a correct use of system calls should follow two properties: *(1) the control flow to the system call when invoked should be legitimate* and *(2) the arguments of the system call should not be compromised.* In order to capture these two properties effectively, we propose three system call contexts, namely Call Type, Control Flow, and Argument Integrity Contexts, which collectively represent how system calls are used in a given program. We enforce the correct use of system calls in a program throughout its runtime in order to break a critical part in attack chains – *i.e.*, illegitimate use of a system call – and negate attacks leveraging system calls. *Call Type Context* allows system calls to be invoked with only the calling convention (*i.e.*, direct call vs. indirect call) they are used within the application. It provides finer-grained system call filtering. *Control Flow Context* only allows system calls to be invoked through a legitimate runtime control-flow path. It can be considered as a scope-reduced CFI, purposely narrow and concentrated to focus on system call related control-flow paths. *Argument Integrity Context* ensures that the arguments of a system call are not compromised. It is data integrity, but only for system call arguments.

In order to enforce our three system call contexts, we propose Bastion, a prototype defense system for system call integrity. To enforce the contexts with minimal runtime overhead, we designed Bastion as a two part system, consisting of a custom compiler pass and a runtime enforcement monitor. Our Bastion compiler pass performs static analysis of system call usages within a program and extracts context information – such as function call types, control-flow paths, and argument tracing – for each system call. Our Bastion runtime monitor uses the compiler-generated metadata and minimal instrumentation to enforce all three system call contexts. Since system calls are often the most critical point in completing an attack, Bastion intervenes only when a system call is being attempted to be invoked where Bastion confirms if any of the three contexts have been violated.

We evaluate the strength of Bastion's security against both real-world exploits and synthesized attacks of advanced attack strategies [34, 47, 81, 82, 93]. Also, we evaluate our Bastion prototype using varied real-world applications to evaluate its efficiency. Our evaluation confirms that using all three contexts effectively protects sensitive system calls from illegitimate use. Bastion requires minimal instrumentation and narrow runtime interference for their protection, so Bastion imposes negligible performance overhead (0.60%-2.01%) even for system call-intensive applications: NGINX web server, SQLite database engine, and vsFTPd FTP server. Therefore, our Bastion design is lightweight and it is a practical solution to provide comprehensive hardening of system calls.

We make the following contributions in this paper:

- **Novel system call contexts for system call integrity.** We propose three system call contexts, namely Call Type, Control Flow, and Argument Integrity contexts. They collectively represent the legitimate use of a system call in a specific program for system call integrity.
- **Bastion defense enforcing system call integrity.** We design and implement Bastion, which enforces our proposed contexts. The Bastion compiler pass analyzes all system call usage in a program, performs necessary instrumentation, and generates accompanying metadata. Bastion's runtime monitor enforces all static and dynamic aspects of each system call context.
- **Security & performance evaluation.** We conducted security case studies of how Bastion defends against real-world exploits and synthesized attacks with advanced attack strategies. We also performed a performance evaluation with system call intensive real-world applications. Our evaluation shows our contexts can block advanced attacks effectively from illegitimate use of system calls with minimal performance overhead.

2 BACKGROUND

In this section, we first discuss how attackers can weaponize system calls (§2.1). Then we introduce the current defense techniques and discuss their limitations in securing system call usage in applications (§2.2).

2.1 System Call Usage in Attacks

While an attacker may utilize gadgets or ROP chains within a vulnerable program, their real objective is almost always to reach and abuse critical system calls to achieve arbitrary execution [25, 81, 93]. While there exist over 400+ system calls in recent Linux kernel versions [90], only a certain subset of system calls are actually desired by attackers. These *sensitive system calls* are responsible for critical OS operations, including process control and memory management. Thus, *sensitive system calls* particularly play a key role in attack completion. While recent works [33, 37, 48, 78, 91] have some overlap in sensitive system calls selected, their exact coverage varies.

With insight that a majority of critical attacks need some form of sensitive system call invocation, it raises the question as to why more care has not been taken by recent defense techniques to strengthen defenses around system calls. Current known and future unknown attacks may have different levels of complexity, different approaches, and different goals. However note that almost all must use sensitive system calls to complete their attack. Therefore, it is worthwhile to explore whether strong constraint policies can be created around system calls.

2.2 Current System Call Protection Mechanisms

Attack surface reduction. Debloating techniques [24, 79, 80] reduce the attack surface by carving out unused code in a program binary. They leverage static program analysis or dynamic coverage analysis using test cases to discover what code is indeed used. Hence, they can eliminate unnecessary system calls not used in a program. However, many sensitive system calls (*e.g.*, `mmap`, `mprotect`) are used for program and library loading so such sensitive system calls remain even after debloating.

System call filtering. `Seccomp` [54] is a system call filtering framework. A system administrator can define an allowlist/denylist of system calls for an application (*i.e.*, process) and, if necessary, restrict a system call argument to a constant value. However, `seccomp`'s allowlist/denylist approach cannot eliminate sensitive-but-necessary system calls (*e.g.*, `mmap`, `mprotect`). Moreover, constraining a system call argument to a constant value is applied across the entire application scope so this argument constraining policy could be more permissive than necessary (*e.g.*, an application uses a system call with two very different permissions flags like read-only vs. read-executable).

Control-flow integrity (CFI). Enforcing integrity of control flow can be one way to enforce legitimate use of system calls in a program. CFI defenses [22, 27, 40, 45, 46, 48, 57, 58, 66, 69, 89, 92, 95] aim to enforce the integrity of all forward and backward control flow transfers in a program. CFI-style defenses perform analysis to generate and define an allowed set of targets per-callsite, called an equivalence class (EC). The size and accuracy of ECs are dependent on the analysis technique used to derive legal targets for a given callsite. Imprecise (static) analysis lead to large ECs, allowing attackers to bypass CFI defenses [81, 93]. Enforcing an EC equal to one is ideal – *i.e.*, there is only one legitimate control transfer at a given moment in program execution. However, CFI techniques maintaining an EC equal to one [48, 58] incur high runtime overhead or consume a lot of system resources due to heavy program instrumentation and runtime monitoring (*e.g.*, Intel PT). Even with a perfect CFI technique (*i.e.*, EC=1, low overhead), an attacker can still divert the intended use of a system call by corrupting its arguments (*e.g.*, `pathname` in `execve`, `prot` flag in `mmap`).

Data-flow integrity (DFI). To enforce the integrity of data (*e.g.*, function pointers, system call arguments), DFI [31] tracks data-flow of each and every memory access, instrumenting every `load` and `store` instruction, thus incuring significant performance overhead [39]. Also, the effectiveness of DFI depends on the accuracy of the points-to analysis used to generate the data-flow graph.

3 CONTEXTS FOR SYSTEM CALL INTEGRITY

As discussed previously, debloating and system call filtering make a binary decision whether a system call is allowed to be used. Defining an allowlist or argument values for a program is too coarse-grained and ineffective to protect commonly used sensitive system calls because the context of system call usage is not considered at all. Meanwhile, CFI and DFI are application-wide defense mechanisms, but impose high runtime overhead and require significant system resources. Instead of carrying out fine-grained integrity enforcement, Bastion focuses on protecting an essential component – system call usage – in an attack chain, to thwart attacks.

A legitimate use of a system call should follow two invariants: (1) the control-flow integrity to a system call and (2) the data integrity of system call arguments. In order to enforce these two invariants effectively, we propose three contexts that are established from the system calls themselves. These contexts encompass system calls by incorporating (1) which system call is called and how it is invoked (§3.1), (2) how a system call is reached within a program (§3.2), and (3) if its arguments are sound (§3.3). Collectively, these system call contexts prevent system calls from being weaponized. In stark contrast to prior defense techniques, such as control-flow and data integrity, we selectively protect only program elements relevant to system calls, greatly minimizing invasive flow tracking and runtime overhead. We now describe each context in detail with two real-world code examples (§3.4).

3.1 Call-Type Context

We first propose *call-type context*, which is a per-system call context in a program. With this context, only permitted system calls are able to be called in their allowed manner, either through a direct or indirect call. By applying the call-type context, we are able to provide more fine-grained system call constraints by separating system calls into several sub-categories – (1) *not-callable*, (2) *directly-callable*, and (3) *indirectly-callable* – compared to the binary-decision debloating and system call filtering approaches. The call-type context blocks all unused system calls (*i.e.*, not-callable type). Note, the not-callable type is applied to all system calls, security-critical or not, thus protecting all system calls in a coarse-grained capacity. For allowed system calls, it divides them into ones that can be called from a direct call site (*i.e.*, directly-callable type) and ones that can be called from an indirect call site via a pointer (*i.e.*, indirectly-callable type). In our observation, it is rare for (especially sensitive) system calls to be called from an indirect call site so we can constrain accessibility of indirectly-callable system calls.

3.2 Control-Flow Context

Our *control-flow context* enforces that a (sensitive) system call is reached and called only through legitimate control-flow paths during runtime. Hence, it ensures that a control-flow path to a system call cannot be hijacked. All sensitive system calls in a program have their respective valid control-flow paths associated with one another to enforce this context at runtime. This context is specifically narrow to only cover those portions of code that actually reach a sensitive system call; the remaining unrelated control-flow paths are not considered or covered by this context. Note that our call-type context compliments this context by verifying whether a specific sensitive system call is permitted to be the target of an indirect callsite.

3.3 Argument Integrity Context

Our *argument integrity context* provides data integrity for all variables passed as arguments to (sensitive) system call callsites. By leveraging this context, a system call can only use valid arguments when being invoked even if attackers have access to memory corruption vulnerabilities. For this context to be complete, we classify a system call argument type as either (1) a *direct argument* or (2) an *extended argument*. In the direct argument type, the passed argument value itself is the argument (*e.g.*, `prot` flag in `mmap`) so we check the passed argument value for argument integrity. Alternatively, an extended argument type uses one or more levels of indirection of the passed argument for argument integrity checking. For example, in the case of `pathname` in `execve`, we need to check not only the `pathname` pointer but also whether the memory pointed to by `pathname` is corrupted or not, while taking care to avoid time-of-check to time-of-use issues (§6.3.2).

This context takes the expected argument type and its field into consideration for each individual argument for a given system call. For example, in Listing 1, the `path` field of a `ngx_exec_ctx_t` structure (*i.e.*, `ctx->path`) is the one that is verified. In this way, our *argument integrity context* is both a *type-sensitive* and *field-sensitive* integrity mechanism. Moreover, this context includes coverage of all non-argument variables associated with each argument's use-def chain. Thereby, attacks which try to corrupt an argument indirectly are still detected.

Compared to traditional data integrity defenses like DFI [31], we limit the scope to system call arguments (and their data-dependent variables). Further, the argument integrity context checks the integrity of argument values [49] rather than enforcing the integrity of data-flow, thus reducing the runtime overhead.

3.4 Real-World Code Examples

We use two code snippets from the NGINX web server [14] as running examples to demonstrate diversity in possible attacks and how all three proposed contexts can negate these attacks collectively. Note that use of `execve` and `mprotect` in Listing 1 and Listing 2, respectively, is legitimate in NGINX, so debloating and system call filtering mechanisms cannot protect these system calls from being weaponized. These attacks only assume the existence of a memory corruption vulnerability such as CVE 2013-2028 (NGINX) or CVE-2014-0226 (Apache).

(1) `execve()`. Listing 1 shows NGINX function `ngx_execute_proc()`, which legitimately uses the `execve` system call. This NGINX function is intended to update and replace the webserver during runtime. In particular, `execve` is a highly sought after system call

```
 1 // nginx/src/os/unix/ngx_process.c
 2 static void ngx_execute_proc(ngx_cycle_t *cycle, void *data){
 3     ngx_exec_ctx_t  *ctx = data;
 4     // Legitimate NGINX usage of execve system call
 5     if (execve(ctx->path, ctx->argv, ctx->envp) == -1) {
 6       ...
 7     }
 8     exit(1);
 9 }
10 // nginx/src/core/ngx_output_chain.c
11 ngx_int_t ngx_output_chain(ngx_output_chain_ctx_t *ctx,
12                            ngx_chain_t *in){
13     ...
14     if (in->next == NULL &&
15         ngx_output_chain_as_is(ctx, in->buf) ) {
16         return ctx->output_filter(ctx->filter_ctx, in);
17     }
18     ...
19 }
```

Listing 1: Legitimate use of the execve system call in NGINX.

```
 1 // nginx/src/http/ngx_http_variables.c
 2 ngx_http_variable_value_t *ngx_http_get_indexed_variable(
 3                      ngx_http_request_t *r, ngx_uint_t index){
 4     ...
 5     if (v[index].get_handler(r, &r->variables[index],
 6             v[index].data) == NGX_OK) {
 7
 8         ngx_http_variable_depth++;
 9         if (v[index].flags & NGX_HTTP_VAR_NOCACHEABLE) {
10             r->variables[index].no_cacheable = 1;
11         }
12         return &r->variables[index];
13         }
14     ...
15     return NULL;
16 }
```

Listing 2: Snippet of NGINX code that can be compromised to reach and call the `mprotect` system call elsewhere by corrupting `index` in vulnerable code pointer `v[index].get_handler()`.

for attackers; if it can be reached, attackers may invoke arbitrary code (*e.g.*, shell) and assume control of the victim machine.

There are two attack vectors against the `execve` system call. An attacker can reach the `execve` system call by hijacking the intended control flow (*e.g.*, corrupting a function pointer or return address) if a CFI-style defense is not deployed. Or she can corrupt the system call arguments (*e.g.*, `ctx->path`, `ctx->argv`) to launch a program illegitimately if no data integrity mechanism is used.

This attack scenario leverages an argument-corruptible indirect call site in the victim program such as `ctx->output_filter` (Listing 1, Line 16) in the function `ngx_output_chain()`. The function pointer at this callsite is maliciously redirected to the function `ngx_execute_proc()`, which contains the desired `execve` call. This is followed by corruption of the global `ctx` object, where it is set to attacker controlled values to enable arbitrary code execution.

Our call-type context allows only a direct call of the `execve` system call in NGINX. Also, the control path to `ngx_execute_proc()` is rarely used, only when a runtime update is necessary, so there are limited ways to reach `ngx_execute_proc()`. Our control-flow context enforcement, in tandem with the call-type context, is sufficient to catch such control-flow hijacking of the `execve` system call. Additionally, our argument integrity context can also detect the corrupted system call arguments (*e.g.*, `ctx`) to block this attack proposed by Control Jujutsu [38].

(2) `mprotect()`. Listing 2 is another NGINX code snippet that can be used maliciously. Here, the statement `v[index].get_handler(...)` (Listing 2, Line 5) is found within the function `ngx_http_get_indexed_variable()`. This statement is intended to be used as a generic handler for NGINX indexed variables that selects the appropriate callee from an array of structures with function pointers. Compared to our first attack scenario, `mprotect` does not need to be present within the `v[]` array to be reached.

An attacker can illegitimately reach an illegal offset beyond the `v[]` array by corrupting `index` and thus call the `mprotect` system call without manipulating pointer values. `mprotect` can be weaponized to change the protections of an attacker controlled memory region. To this end, she can corrupt `r` (or `$rsi`) and `v[index].data` (the third argument defining the permission) to **`PROT_EXEC`**`|PROT_READ|PROT_WRITE`.

This attack scenario is able to bypass both code and data pointer integrity (*e.g.*, CPI [60]) as described in the NEWTON attack framework [93]. Instead of corrupting code and data pointers, the attack relies on finding a callsite that is capable of being manipulated by non-pointer values. Here, the non-pointer variable `index` is manipulated to make the target address of the array `v[index].get_handler` go beyond the array bounds and be redirected to `mprotect`. This callsite's three arguments `r`, `&r->variables[index]`, and `v[index].data` are also all controllable via non-pointer variables. This allows the callsite to be crafted to an attacker desired function with malicious arguments.

Bastion blocks this attack from the perspective of the system call attempting to be invoked. Our control-flow context in tandem with the call-type context easily detects this control-flow hijacking reaching `mprotect`, whereas CFI and CPI type defenses would not detect this attack. `mprotect` is never invoked indirectly in this application, and is never assigned to `get_handler()`, violating call-type and control-flow contexts. Moreover, our argument integrity context can detect that the leveraged variables `r`, `&r->variables[index]`, and `v[index].data` are illegitimate and never used by any legal system call invocation.

4 THREAT MODEL AND ASSUMPTIONS

We assume a powerful adversary with arbitrary memory read and write capability by exploiting one or more memory vulnerabilities (*e.g.*, heap or stack overflow) in a program. We assume that common security defenses – especially Data Execution Prevention (DEP) [53, 67] and (coarse-grained) Address Space Layout Randomization (ASLR) [88] – are deployed on the host system. Hence attackers cannot inject or modify code due to DEP. We also assume a shadow stack will be employed (*e.g.*, CET [83]) as this technology is mature [28, 35] and is now available in hardware [62]. We assume that the hardware and the OS kernel are trusted, especially `secomp-BPF` [56] and the OS's process isolation. Attacks targeting the OS kernel and hardware (*e.g.*, Spectre [59]) are out of scope.

We focus on thwarting a class of attacks exploiting one or more (sensitive) system calls in their attack chain. This is because core attacker goals are to issue one or more (sensitive) system calls. System calls are the primary means to interact with the host OS. Attacks are therefore ineffective if they do not have access to these system calls [29]. For example, Göktas et al. [44] need to change

Table 1: Classification of sensitive system calls commonly leveraged by attackers. We classify each *security-critical* system call with the attack vector that commonly abuses it.

System Call Classification	Applicable System Calls
Arbitrary Code Execution	`execve, execveat, fork, vfork, clone, ptrace`
Memory Permissions	`mprotect, mmap, mremap, remap_file_pages`
Privilege Escalation	`chmod, setuid, setgid, setreuid`
Networking	`socket, bind, connect, listen, accept, accept4`

the permissions of an existing memory area (*e.g.*, `mprotect`) to achieve their attack goals. Therefore Bastion protects a subset of available system calls (Table 1) that can allow an attacker escape beyond the scope of an application to obtain control over the host system. Bastion concentrates on a subset of system calls as not all system calls perform meaningful security-critical actions. Likewise, Bastion protects a subset of data connected to system calls, instead of all program data.

5 BASTION DESIGN OVERVIEW

Bastion aims to strengthen the integrity surrounding sensitive system calls by enforcing the three proposed contexts (*i.e.*, Call-Type, Control-Flow, and Argument Integrity), such that attacks leveraging system calls can be mitigated. Similar to prior defenses [33, 37, 40, 48, 78, 92], we choose 20 sensitive system calls (Table 1). We chose these system calls such that Bastion can successfully defend against attacks that achieve arbitrary code execution, memory permission changes, privilege escalation, or rogue network reconfiguration. Note, this list can be easily extended to include other system calls.

As illustrated in Figure 1, Bastion consists of two core components: (1) a Bastion-compiler pass, which performs context analysis, instrumentation, produces a Bastion-enabled binary, and generates context metadata, and (2) a Bastion monitor, which enforces system call contexts during runtime. We now explain Bastion's compiler (§6) and runtime monitor (§7) in detail.

6 BASTION COMPILER

Our Bastion compiler derives the necessary information for proper enforcement of our system call contexts. Bastion also leverages a light-weight library API for dynamic tracking of sensitive data related to the argument integrity context to properly track relevant system call arguments. We now describe our program analysis for each context and creation of a Bastion-enabled binary.

6.1 Analysis for Call-Type Context

To enforce the call-type context, the Bastion compiler analyzes a program and classifies system calls in the program into three categories: (1) not-callable, (2) directly-callable, and (3) indirectly-callable types, as discussed in §3.1. It analyzes the entire program's LLVM IR instructions and checks all call instructions. If a system call is a target of a direct function call, the Bastion compiler classifies it into a directly-callable type. If the address of a system call is taken and used in the left-hand-side of an assignment, that system call can be used as an indirect call target, and the Bastion compiler classifies it as an indirectly-callable type. Note that a system call can be both directly-callable and indirectly-callable. All other arbitrary system calls not belonging to either type are set as not-callable. Any attempt by an attacker to reach a not-callable system call, security-sensitive or not, is not permitted by Bastion.

Table 2: Bastion library API for argument integrity context. Bastion manages shadow copies of sensitive variables (`ctx_write_mem`) and binds memory-backed (direct or extended) arguments and constant arguments (`ctx_bind_mem_X`, `ctx_bind_const_X`) to a specific position (`X`-th argument) of a system call callsite.

API	Description
`ctx_write_mem(p,size)`	Update the shadow copy of `p` in `size`
`ctx_bind_mem_X(p)`	Bind a memory `p` to `X`-th argument
`ctx_bind_const_X(c)`	Bind a constant `c` to `X`-th argument

After this analysis is completed, the Bastion compiler generates metadata which contains, (1) pairs of system call numbers and their call type, and (2) a list of *legitimate* indirect callsites (*i.e.*, offset in a program binary). This metadata is used by the Bastion runtime monitor to enforce the call-type context.

6.2 Analysis for Control-Flow Context

Bastion uses the control-flow context to prevent control-flow hijacking attacks that illegitimately reach system calls. Enforcement of the control-flow context is performed *only when a system call is invoked.* Our approach is different from conventional CFI techniques, which enforce the integrity of *every indirect control-flow transfer*.

Bastion analyzes a control-flow graph (CFG) of the entire program to identify all function *callee*⟶*caller* relationships that reach system call callsites. For each system call callsite in the CFG, the Bastion compiler recursively records all callee⟶caller associations. Recursive analysis stops once reaching either `main()` or an indirect function call. The Bastion runtime monitor verifies callee⟶caller relations until the bottom of the stack (*i.e.*, `main`), or an indirect callsite, by unwinding stack frames.

With the call-type context and control-flow context in tandem, Bastion verifies that control-flow reaching a system call follows (1) legitimate direct callee-caller relations and (2) is a legitimate indirect call from a valid indirect callsite. Once analysis for control-flow context is done, Bastion generates metadata which consists of the pairs of callee and caller addresses in a program binary.

6.3 Analysis for Argument Integrity Context

Lastly, the Bastion compiler analyzes a program to enforce argument integrity for system calls. Note that this is the only context that requires instrumentation. We now discuss what variables should be protected (§6.3.1), how to check if an argument is compromised (§6.3.2), how our instrumentation achieves dynamic tracking of arguments (§6.3.3), and what metadata is generated (§6.3.4).

6.3.1 *Protection Scope.*

In order to properly enforce argument integrity, Bastion needs to check not only system call arguments but also an arguments' data-dependent variables. We collectively call these protected variables as *sensitive variables.* Figure 2 shows how Bastion enforces the argument integrity context for each of the

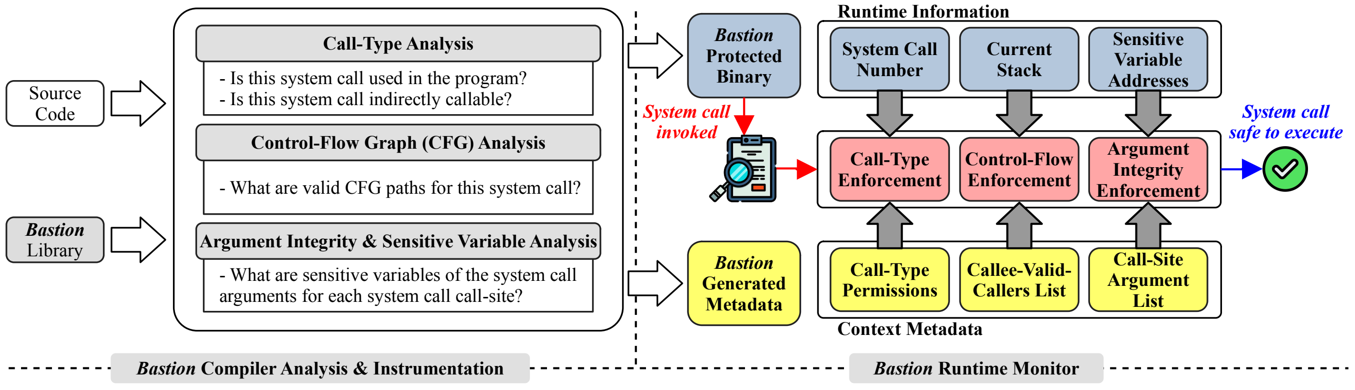
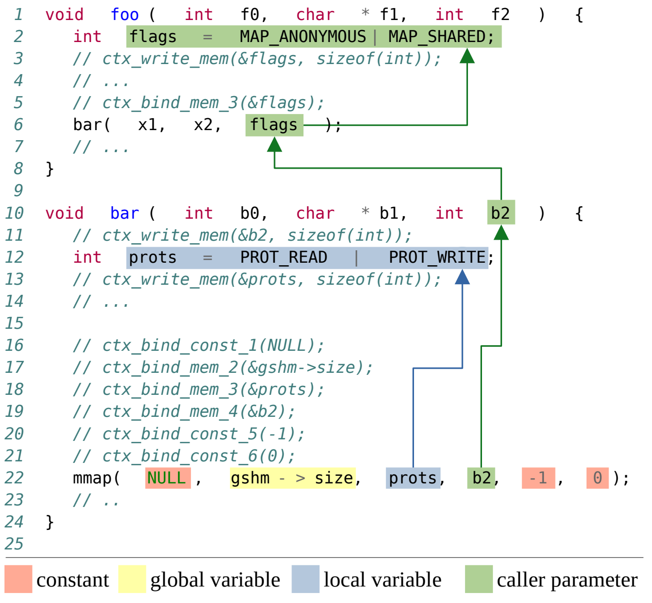

Source Code
Bastion Library
Call-Type Analysis
- Is this system call used in the program?
- Is this system call indirectly callable?
Control-Flow Graph (CFG) Analysis
- What are valid CFG paths for this system call?
Argument Integrity & Sensitive Variable Analysis
- What are sensitive variables of the system call arguments for each system call call-site?
Bastion Protected Binary
Bastion Generated Metadata
System call invoked
Runtime Information
System Call Number
Current Stack
Sensitive Variable Addresses
Call-Type Enforcement
Control-Flow Enforcement
Argument Integrity Enforcement
System call safe to execute
Call-Type Permissions
Callee-Valid-Callers List
Call-Site Argument List
Context Metadata
Bastion Compiler Analysis & Instrumentation
Bastion Runtime Monitor

Figure 1: Design overview of Bastion. At compile time, Bastion analyzes and generates a program's context metadata. For call-type and control-flow contexts, Bastion generates static metadata. For the argument integrity context, Bastion generates metadata for static argument values (*e.g.*, constants) and instruments the program to enable tracking of dynamic argument values. At runtime, the Bastion monitor hooks on all invocations of sensitive system calls and verifies all three contexts of the system call.



```
 1  void  foo (  int  f0,  char  * f1,  int  f2  )  {
 2     int  flags  =  MAP_ANONYMOUS | MAP_SHARED;
 3     // ctx_write_mem(&flags, sizeof(int));
 4     // ...
 5     // ctx_bind_mem_3(&flags);
 6     bar(  x1,  x2,  flags  );
 7     // ...
 8  }
 9
10  void  bar (  int  b0,  char  * b1,  int  b2  )  {
11     // ctx_write_mem(&b2, sizeof(int));
12     int  prots  =  PROT_READ  |  PROT_WRITE;
13     // ctx_write_mem(&prots, sizeof(int));
14     // ...
15
16     // ctx_bind_const_1(NULL);
17     // ctx_bind_mem_2(&gshm->size);
18     // ctx_bind_mem_3(&prots);
19     // ctx_bind_mem_4(&b2);
20     // ctx_bind_const_5(-1);
21     // ctx_bind_const_6(0);
22     mmap(  NULL ,  gshm - > size,  prots,  b2,  -1 ,  0 );
23     // ..
24  }
25
```

🟥 constant 🟨 global variable 🟦 local variable 🟩 caller parameter

Figure 2: Bastion instrumentation for argument integrity. Protection of memory-backed arguments is augmented using field-sensitive use-def analysis (size field of gshm, b2←flags) at an inter-procedural level.

arguments in the `mmap` system call. At Line 22, `NULL`, `-1`, and `0` are constant arguments; `gshm->size`, `prots`, and `b2` are memory-backed arguments; `flags` is a data-dependent sensitive variable, which is passed from the function `foo`.

6.3.2 Checking Argument Integrity. In order to verify the argument integrity for each system call, Bastion adopts data value integrity [50] for each sensitive variable. Bastion maintains a shadow copy of the sensitive variable's legitimate value in a shadow memory region and updates the shadow copy whenever the sensitive variable is updated legitimately. Before a system call is invoked, Bastion binds each argument to a certain position for the system call so the Bastion runtime monitor can check argument integrity using the Bastion-maintained shadow copy of the respective sensitive variable.

The Bastion runtime library provides the API shown in Table 2. `ctx_write_mem(p,size)` updates the shadow copy of a sensitive variable located at address `p` with `size`. Note that a constant argument (*e.g.*, `NULL`) does not need to have a shadow copy, because its legitimate value is determined statically at compile time. Hence `ctx_write_mem` is only used for memory-backed sensitive variables (*e.g.*, `gshm->size`, `prots`, `b2`, and `flags` in Figure 2). For argument binding, `ctx_bind_mem_X(p)` binds a memory-backed sensitive variable at `p` to the `X`-th argument of the associated system call callsite. Similarly, `ctx_bind_const_X(c)` binds the constant value `c` to the `X`-th argument of the callsite. Note that binding is applied to both system call callsites as well as callsites passing sensitive variables (*e.g.*, `bar()` callsite, Line 6 in Figure 2). We note that it is not necessary to instrument if an argument is a direct or extended argument as such a distinction is system call & position-specific. Instead, Bastion's monitor can recover the system call being verified so we design the monitor to handle this distinction, further minimizing instrumentation. For example, if Bastion's runtime monitor discerns `execve` is being verified, it knows the first argument of `execve` is an extended argument, and thereby will automatically verify not only the pointer value but also pointee memory contents for only the first argument, while the remaining arguments will be verified as direct arguments. To add, because the list of sensitive system calls is short, it is easy to specialize the rules for these arguments.

6.3.3 Sensitive Variable Instrumentation. At a high level, Bastion compiler performs field-sensitive, inter-procedural analysis to identify all sensitive variables, including *data-dependent variables*. To identify all sensitive variables, the Bastion compiler analysis performs three steps. First, it enumerates all variables used in system call arguments. These variables are the initial set of sensitive variables. Second, it performs a backward data-flow analysis, traversing the use-def chains to derive any other variables used to define sensitive variables. Such newly identified data-dependent variables are added to the set of sensitive variables. Third, if there is a write to a field of a struct (*e.g.*, `size` field of `gshm` in Figure 2), that write is added to the sensitive variables. Analysis repeats the second and third steps until no new sensitive variables are found. Note this field-sensitive analysis is also effective for tracking sensitive global and heap variables.

Bastion first follows and instruments the callsite's system call argument variables via intra-procedural analysis. If an argument variable is updated by a caller function's parameter (*e.g.*, `b2`$\longleftarrow$`flags` in Figure 2), Bastion adds the caller parameter to the sensitive variables and performs inter-procedural analysis between the caller and callee for the caller's parameter (*e.g.*, `flags` in Figure 2). This process is done recursively until an origin for the sensitive variable's use-def chain is found.

Once all sensitive variables are identified, Bastion instruments `ctx_write_mem` after any memory-backed sensitive variable `store` to keep its shadow copy up-to-date. Before each sensitive system call callsite, Bastion instruments `ctx_bind_mem_X` or `ctx_bind_const_X` to bind an arguments to their respective argument position `X`. Regarding the analysis of extended arguments, Bastion instruments all possible use-def chains. While this may incur more instrumentation, it does not lower security guarantees and we did not observe any additional performance overhead. In practice, the call depth to set system call arguments is fairly shallow – within the same function or only a few functions away.

Even if an attacker happens to skip our instrumentation, since non-system call relevant control-flow is not monitored or enforced by Bastion, malicious intent can still be detected by Bastion. In this case, legitimate system call argument values will not have been properly updated to their expected values.

6.3.4 Bastion-compiler Generated Metadata. In the case of argument integrity context metadata, the compiler specifies an entry for each sensitive system call callsite. Each callsite entry includes the callsite file offset and the argument types (*i.e.*, constant vs. memory-backed arguments, direct vs. extended arguments). For a constant argument, the expected value is recorded as well. For a memory-backed argument, the argument index denotes that a bound value should be retrieved from the Bastion shadow memory region and compared with the value in an argument register (*e.g.*, `$rdi`, `$rsi`, `$rdx`).

7 BASTION RUNTIME MONITOR

The Bastion monitor enforces all three of our proposed system call contexts at runtime. We specifically design the Bastion monitor as a separate process from the application being protected. In doing so, attackers are unable to avoid Bastion's runtime hooks that occur at *sensitive* system call invocations. We now describe how the monitor is initialized and how each context is enforced.

7.1 Initializing the Bastion Monitor

Loading metadata. The Bastion monitor is invoked with a target application binary path. The monitor retrieves `ELF`, `DWARF`, and linked library file information to recover symbol addresses. It then loads Bastion context metadata into the monitor's memory.

Launching a Bastion-protected application. After loading metadata, Bastion performs `fork` to spawn a child process where the child runs the Bastion-enabled application after synchronization is setup by the monitor. Specifically, Bastion initializes a shadow memory region under a segmentation register (`$gs` in Linux) for shared use between the application process and the Bastion monitor process. Bastion also initializes `seccomp` [54, 56] to trap on sensitive system calls in the child process and `ptrace` [64] to access the application's state (including the shared shadow region attached to the application's virtual address space). Once both the target application and Bastion are synchronized, the Bastion monitor allows the application to proceed and begins monitoring. Then, any sensitive system call invocation attempt will trap the Bastion monitor to perform context integrity verification by poking the application's execution state, before allowing the system call to be executed. Otherwise, Bastion monitor rests in an idle state until the next sensitive system call is invoked.

Trapping a system call invocation. The Bastion monitor initializes a custom `seccomp-BPF` filter to trap on the application's sensitive system call invocations using call-type metadata. `SECCOMP_RET_ALLOW` is specified for all non-sensitive system calls to ignore them when invoked, while `SECCOMP_RET_KILL` disables any not-callable system calls. `SECCOMP_RET_TRACE` is specified for directly- and indirectly-callable system calls so these system calls can be verified by the Bastion monitor. Note that a child process or thread spawned by a Bastion-protected application has the same `seccomp` policy as its parent process and is protected by the same Bastion monitor process.

Accessing application state & shadow memory. The Bastion monitor needs to retrieve an application's runtime information and its shadow memory to verify integrity of the system call contexts before allowing the execution of a sensitive system call. The Bastion monitor retrieves current register values and system call number via `ptrace`. It uses `process_vm_readv` [55] to access arbitrary memory in the application's address space, including stack, heap, and the shadow memory region.

As discussed, shadow memory is initialized when a Bastion-protected application is launched and belongs to the application's address space. It is an open-addressing hash table maintaining a shadow copy (*i.e.*, legitimate value) of a sensitive variable and argument binding information for the argument integrity context. The key to access this hash table data is an address; the sensitive variable address and callsite address are used to access its shadow copy and argument binding information, respectively. Note Bastion's shadow memory region relies on sparse address space support of the underlying OS like metadata store designs in prior studies [50, 60].

7.2 Enforcing Call-Type Context

For call-type context, the Bastion monitor retrieves the system call number and the program counter (`rip`) where the system call is invoked using `ptrace`. The Bastion monitor uses the `rip` to retrieve and check the call type of the callsite from its metadata. For example, the Bastion monitor decodes the `call` instruction at Line 22 in Figure 2 using `rip` to determine if the `mmap` call was made direct or indirect. If the call type is allowed, Bastion continues to the next context check. Otherwise, the Bastion monitor assumes this is an attack attempt and immediately kills the protected application and gracefully exits.

7.3 Enforcing Control-Flow Context

For control-flow context, the Bastion monitor retrieves a copy of the current stack trace when a system call attempt is made and verifies that the current call stack adheres to the CFG metadata, represented as a list of callees and their respective valid callers. The Bastion monitor unwinds the retrieved stack frame to get each function callsite offset. It then checks if the unwound caller is in the valid caller list of the callee in the CFG metadata. For example, in Figure 2, function `foo()` is a valid caller of function `bar()`. This is iteratively performed until the entire stack has been vetted or an indirect call is encountered. When handling an indirect call, Bastion ends verification at this point and verifies the partial stack trace encountered matches the expected one derived at compile time. Thus, there are no false-positives or false-negatives. If a mismatch in a control-flow transition is found, the Bastion monitor assumes this is a control-flow hijacking attempt to illegitimately reach this system call, and kills the application, as done for the call-type context.

7.4 Enforcing Argument Integrity Context

For argument integrity context, the Bastion monitor verifies integrity of all sensitive variables in the current call stack. Take Figure 2 as an example. At function `bar()`'s stack frame, the Bastion monitor verifies all bound constant variables (*i.e.*, `NULL`, `-1`, `0`) and memory-backed variables (*i.e.*, `gshm->size`, `prots`, `b2`). Once checking the current stack frame is done, it unwinds the stack and verifies function `foo()`'s sensitive variable (*i.e.*, `flags`). For each callsite, the Bastion monitor uses the current `rip` value to retrieve the associated argument integrity context metadata reporting types of each argument (constant vs. memory-backed arguments) to know how to verify it. Additionally, recall from §6.3.2, the monitor automatically processes each argument as direct or extended as needed. Particularly for non-system call callsites, Bastion also uses argument integrity context metadata to determine which arguments need to be verified. For each callsite's sensitive argument, the Bastion monitor compares the register (actual) argument value to the Bastion-traced value retrieved from shadow memory. Bastion iteratively traverses all function frames currently on the stack. If values match, the arguments are legitimate and the system call can proceed. Otherwise, the Bastion monitor kills the application and concludes runtime monitoring.

8 IMPLEMENTATION

We implemented Bastion on Linux x86-64 v5.19.14. Bastion analysis and instrumentation is implemented as an LLVM Module pass in 3,939 lines of code (LoC). Bastion's C runtime library (659 LoC) implements memory management functions to enable the monitor to read and bind arguments throughout the entire stack frame. All library functions are inlined to maximize performance. The Bastion runtime monitor is a C-program (7,313 LoC). It leverages `seccomp-BPF` and `ptrace` for triggering system call enforcement. Bastion also employs CET [83] a hardware-based shadow stack by specifying compiler flag `-fcf-protection=full`. Intel Tiger Lake and AMD Ryzen 7 processors onwards [62] with `Glibc` v2.28+ [61, 65], `Binutils` v2.29+ [84], and Linux kernel v5.18+ fully support CET.

9 EVALUATION

In this section, we evaluate how effective Bastion's system call contexts are. We measure the performance overhead of Bastion with three real-world applications: the NGINX web server [14], the SQLite database engine [85], and vsftpd [21], an FTP server. Our evaluation includes partitioning the individual costs of each Bastion system call context, examining the cost-benefit analysis for each as well as reporting static and runtime statistics.

9.1 Evaluation Methodology

Evaluation setup. We ran all experiments on an 8-core (16-hardware thread) machine featuring an AMD Ryzen 7 PRO 5850U processor and 16 GB DDR4 memory. All benchmarks were compiled with the Bastion LLVM compiler. Results are reported average over five runs.

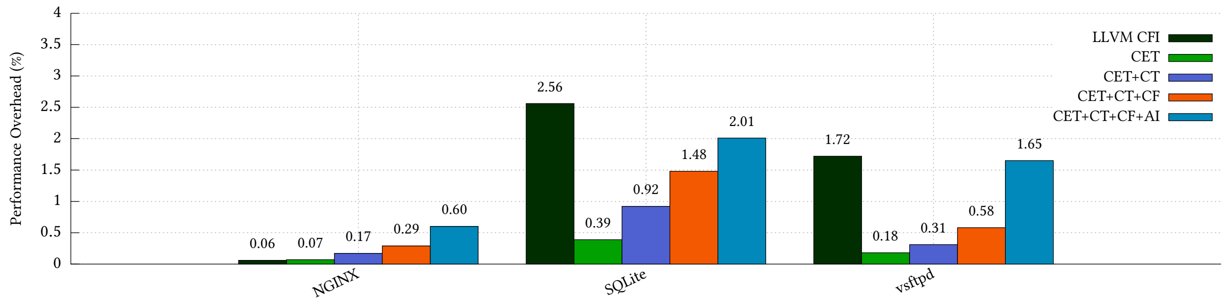
Applications. We ran NGINX, SQLite, and vsftpd as these real-world applications are widely deployed and as such, they are often victim to attack. Also, these are I/O-intensive and system call-heavy applications.

9.2 Performance Evaluation

For the performance evaluation of Bastion, we report both the relative performance overhead compared to the unprotected baseline version (Figure 3) as well as the raw performance numbers of all three applications (Table 3). We ran all evaluations with Address Space Layout Randomization (ASLR) [88] enabled as Bastion is based around relative-addressing and has no incompatibility issues with ASLR-based defenses. Before applying Bastion, we enabled CET and compared the runtime overhead compared to the baseline version. For all three applications, we observe CET incurs negligible overhead. Finally, we observed that the initialization cost of Bastion monitor is always negligible (on the order of ten to twenty milliseconds, *e.g.*, ≈21 ms for NGINX).

NGINX. To test NGINX, we use `wrk` [43], a HTTP benchmarking tool. `Wrk` sends concurrent HTTP requests to a web server and measures throughput. We place the `wrk` client on a different machine, but on the same local network as the NGINX webserver. NGINX is configured to handle a maximum of 1,024 connections per processor and have 32 worker threads. We measure throughput for a 20-second run. `wrk` spawns the same number of threads as NGINX's configured worker count where each `wrk` thread generates HTTP requests for a 6,745-byte static webpage.

The runtime overhead for NGINX minimally increased as each Bastion context was enabled. The overhead when applying full Bastion protection (all three contexts enabled, Call-Type, Control-Flow, & Argument Integrity) never incurred more than 0.60% degradation compared to the unprotected NGINX baseline. Figure 3 shows this breakdown in more detail. In NGINX, protecting the subset of 20 sensitive system calls with just the monitor adds barely



CFI: Control-Flow Integrity CET: Control-Flow Enforcement Technology CT: Call Type CF: Control Flow AI: Argument Integrity

Figure 3: Performance overhead of BASTION for NGINX, SQLite, and vsftpd. All values are compared to an unprotected baseline vanilla version. For reference to state-of-the-art, we also include the individual overhead for (coarse-grained) LLVM Control-Flow Integrity (LLVM CFI) [87] and Control-Flow Enforcement Technology (CET) [83].

Table 3: Benchmark numbers for NGINX, SQLite, and vsftpd on which Figure 3 is based on. We measure NGINX's throughput of data (MB/s). SQLite is evaluated using DBT2 [9] which records New Order Transactions Per Minute (NOTPM). Lastly, for vsftpd, we measure seconds elapsed to download a 100 MB file. For NGINX and SQLite, higher is better, whereas lower is better for vsftpd.

Application	Unprotected Vanilla	Runtimes with Hardware/Software Mitigations				
		LLVM Control-flow Integrity (LLVM CFI)	Control-flow Enforcement Technology (CET)	CET+DI	CET+DI+CF	CET+DI+CF+AI
NGINX (MB/sec)	110.61	110.54	110.52	110.42	110.28	109.94
SQLite (NOTPM)	37,107.41	36,156.15	36,961.91	36,764.50	36,560.02	36,360.85
vsftpd (sec)	10.75	10.93	10.77	10.79	10.81	10.93

≈1% overhead. As expected, the Argument Integrity context adds the most overhead following the monitor itself since this context is the most complex.

Analysis of NGINX source code reveals that NGINX utilizes a vast majority of sensitive system calls (*e.g.*, `mprotect`, `mmap`) during its initialization phase while seldom using any sensitive system calls when idle or processing requests. This results in BASTION rarely being triggered during runtime. Table 4 shows combined initialization and runtime statistics of all (sensitive) system calls invoked during benchmarking; it reveals that only a call to `accept4` is made per-request and dominates the system call invocation count. Additionally, evaluating NGINX showed that for all system call invocations (not including those originating from a library), the average call-depth is only 5.2 frames, with 4 and 9 being minimum and maximum stack call-depths encountered, respectively.

SQLite. SQLite [85] is a widely deployed, transactional SQL database engine. To test the performance throughput of SQLite, we use the DBT2 [9] database transaction processing benchmark. DBT2 mimics a mix of `read` and `write` SQL operations for large data warehouse transactions. We ran DBT2 with its default configuration with a 10 second new thread delay and a 10 minute workload duration. We measure DBT2's number of new-order transactions per-minute (NOTPM) for performance.

Figure 3 shows the BASTION performance breakdown for each context. Adding Call-Type and Control-Flow context checking increases BASTION's overhead to 0.92% and 1.48%, respectively. Subsequently, adding Argument Integrity context checking for SQLite's system calls brings BASTION's overhead to 2.01%. As in NGINX, Argument Integrity context enforcement contributes the most overhead, relative to Call-Type and Control-Flow contexts. However, note that since our Argument Integrity context strategically only requires tracing and enforcing value integrity for select sensitive variables used as system call arguments, this contexts performance overhead is magnitudes smaller than overhead imposed by conventional application-wide DFI-style defenses.

SQLite largely uses system calls to initialize and setup its worker threads based off the DBT2 configuration. We expect overhead is further amortized for SQLite in a long standing real-world application setting. Table 4 shows system calls invoked during SQLites runtime. The contrast in Argument Integrity costs can be attributed to differences in system call usage and argument patterns. SQLite relies more on `mprotect` compared to NGINX or vsftpd.

VSFTPD. vsftpd is evaluated using `dkftpbench` [10], an FTP benchmark program. `dkftpbench` simulates users downloading files from a FTP server. We place the `dkftpbench` client on the same machine as BASTION protected vsftpd. `dkftpbench` is configured to fetch a `100 MB` file from vsftpd launching clients one after another for a 120 second duration. All other options are left as default.

`Dkftpbench` showed negligible runtime overhead for all BASTION contexts compared to the unprotected baseline. In the worst case, BASTION incured 1.65% overhead.

Table 4 substantiates this performance overhead, showing that vsftpd invokes system calls far fewer times than either NGINX or SQLite. Similar to NGINX, `accept` accounts for most of BASTION monitor checks. When designing BASTION to support `accept` and `accept4`, we noted that only these two system calls had a more

Table 4: Sensitive system call usage from benchmarking.

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

Table 5: Instrumentation statistics for Bastion.

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

complex argument (*e.g.*, `struct sockaddr`) compared to other system call arguments. We therefore optimized the Bastion monitor to support verifying this structure in a specific way to improve argument integrity runtime.

System call statistics. Table 5 shows instrumentation statistics regarding Bastion including the total number of sensitive system call callsites and each Bastion API instrumentation count. This table shows that sensitive system calls have a very small footprint even in large production applications. Specifically, note the drastic difference between the number of application callsites (Table 5, Row 1) compared to the number of sensitive system call callsites (Table 5, Row 4). Consequently, Bastion's instrumentation footprint is also relatively small, with a maximum of 5,287 instrumentation points in NGINX. A key finding is that in all three real-world applications, sensitive system calls are *never* legitimately called indirectly via a function pointer (Table 5, Row 5). This facet allows Bastion to entirely invalidate attack schemes that rely on reaching system calls from a corrupted pointer.

Comparison against CET and LLVM CFI. We also compared Bastion with other state-of-the-art defenses, specifically LLVM CFI (forward edge protection) and CET (backward edge protection). Results are depicted in Figure 3 and Table 3. Note that we were not able to simultaneously enable CET with LLVM CFI as LLVM CFI does not function properly when paired with CET.

CET works by maintaining a secondary (shadow) stack that cannot be directly modified by applications. Upon returning from a function, the CPU compares return addresses in the shadow stack and the normal stack. If these two addresses differ, a protection fault is raised. Our results show that CET incurs negligible overhead (< 0.5%).

LLVM CFI [87] is a coarse-grained CFI technique. Our results show that LLVM CFI incurs low performance overhead (< 3%). LLVM CFI performs verification at every indirect callsite while Bastion is designed to only perform verification at sensitive system call invocations. LLVM CFI checks indirect and virtual function calls only using function type information. Since it is not fine-grained, it does not guarantee a unique target as in recent CFI works [48, 58]. Moreover, CFI does not have such a concept to check whether a function is allowed to be called directly or indirectly as done for our Call-Type context, nor does CFI verify argument integrity compared to Bastion. Therefore Call-Type and Control-Flow contexts cannot be directly replaced with CFI.

Summary. Our evaluation confirms our insights of sensitive system call usage patterns being used sparsely. For all real-world applications tested, we see Bastion instrumentation and monitoring for all three contexts incurs low (<3%) performance degradation. Compared to related defense strategies, we are on par or better (*e.g.*, 7.88% - uCFI [48], 7.6% - OS-CFI [58], 2.7% - OAT [86]).

10 SECURITY EVALUATION

We conducted case studies with 32 attacks in Table 6, which include ROP payloads, real-world CVE exploits, and synthesized attacks recent attack strategies [34, 47, 81, 82, 93]. The results confirm that even if one context is bypassed, another context in Bastion can compensate and still prevent the attack vector from being viable.

10.1 ROP Attacks

While CET is now available on the newest processors from Intel and AMD, older processors do not have a built-in shadow stack defense mechanism. We now explain how Bastion can defend ROP in the absence of CET.

ROP payloads work by stitching together various ROP-gadgets present in a victim application to create their attack. Generally, they will leverage one or more system calls (*e.g.*, `execve`, `mprotect`) to complete the attack.

ROP payloads can execute user (non-`root`) commands by leveraging the `libc` library call `system` (which internally calls `fork` and `execl` system calls), and passing the arguments `"/bin/sh"`. Or they can directly leverage an `exec`-type system call, to create access to a `root` shell. ROP payloads can also manipulate memory permissions and abuse a memory corruption vulnerability by using `mprotect` or `chmod` system calls to change memory or file permissions to be executable (*e.g.*, `RWX`) in order to setup their attacker directed script, before pivoting the stack or code pointer to the start location.

If the victim application does not use these system calls, Bastion blocks these invocations with the Call Type context. Or if these ROP payloads advance by directly manipulating control-flow and variables to reach and prepare these system calls, including setting flags to uncommonly used natively by applications (*e.g.*, making a memory region executable), Control Flow or Argument Integrity contexts will detect these attacks.

10.2 Direct Attacker Manipulation of System Calls

In this attack strategy, attacks go after system calls directly, crafting attacks that setup callsites and arguments to desired values via code pointer and variable corruption.

The CsCFI attack leverages `mprotect` to make the entire `libc` readable, writable, and executable, revealing the code layout to perform arbitrary code execution. AOCR's Attack 1 instead leverages `open` and `write` to reveal the code layout of NGINX to execute arbitrary code. In both cases, Bastion's Call-Type context blocks these attacks. In the CsCFI attack, `mprotect` is never used by the application. In Attack 1, `open` is legitimately used in NGINX, but only ever as a direct call. Moreover, Bastion records all valid call-traces for sensitive system calls as well as instrumenting all arguments for sensitive system calls, allowing Bastion's Control-Flow and Argument integrity contexts to also detect the anomaly of being called from an unexpected callsite with untraced arguments.

Several real-world CVE's also fall under this category. While they each individually affect different applications, they all rely on inherent memory corruption vulnerabilities and abuse them to maliciously manipulate program data including code pointers and argument variables. Therefore, Bastion can address all these CVE's because they aim to achieve arbitrary code execution by corrupting code pointers and arguments to point to system calls that are either never used or only used via direct callsites.

In comparison to Bastion, LLVM CFI cannot defend against either attack. Since LLVM CFI employs a coarse-grained defense scheme, it only enforces that function calls match the static type used at the callsite [87]. In the CsCFI attack, even though `mprotect` is never used by the application, its address is still taken as this system call is necessary to support dynamic loading of shared libraries. Related, AOCR Attack 1 leverages code pointers whose type matched to system calls `open` and `write`, allowing these control-flow violations to bypass LLVM CFI.

10.3 Indirect Attack Manipulation of System Calls

In this attack strategy, attacks evade popularly deployed defense strategies such as CFI, CPI, and system call filtering. Such strategies include attacks like full-function code re-use [38], data-oriented attacks [32?], and COOP [82].

The NEWTON CPI attack avoids corrupting any code or data pointers. It corrupts the index variable of an array of function pointers to make the array index point to a system call location. Bastion is able to defend against this attack with all three contexts. The Call-Type context blocks the invocation of a system call never used in the program code base. Similarly, Control-Flow and Argument Integrity context will not have legitimate system call information

Table 6: Real-world and synthesized exploits blocked by Bastion. ✓ denotes that a context can block an exploit, and × indicates that an exploit can bypass the context.

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

Attack Category & Type	Violated Context CT	CF	AI
Return-oriented programming (ROP)			
Execute user command [1, 3, 5, 7, 8, 11, 13, 15–20]	×	✓	✓
Execute root command [11]	×	✓	✓
Alter memory permission [2, 4, 6, 12]	×	✓	✓
Direct system call manipulation			
NEWTON CsCFI Attack [93]	✓	✓	✓
AOCR NGINX Attack 1 [81]	✓	✓	✓
CVE-2016-10190 ffmpeg [75] CVE-2016-10191 ffmpeg [76] CVE-2015-8617 php [74] CVE-2012-0809 sudo [70] CVE-2013-2028 nginx [71] CVE-2014-8668 libtiff v4.0.6 [73] CVE-2014-1912 python-2.7.6 [72]	✓	✓	✓
Indirect system call manipulation			
NEWTON CPI Attack [93]	✓	✓	✓
AOCR Apache Attack [93]	×	✓	✓
AOCR NGINX Attack 2 [81]	×	×	✓
COOP Against Google Chrome [34]	×	×	✓
Control Jujutsu against NGINX [38]	×	×	✓

for the corrupted callsite or arguments used thereby thwarting the attack as well.

The AOCR Apache attack finds an indirect code pointer of an `exec` function in `ap_get_exec_line()`. Because there is a legitimate indirect call to `exec`, Bastion's Call-Type context will be bypassed. Instead, the hijacking of `ap_get_exec_line()` onto a corruptable function pointer is what gives away the attack, and allows Bastion to block it with the Control-Flow context.

Lastly, AOCR NGINX Attack 2 [81], COOP [82], and Control Jujutsu [38] leverage inherent memory vulnerabilities as well as the inherent program control-flow against itself to gain hijack control.

Compared to Bastion, LLVM CFI cannot defend against this attack strategy – AOCR NGINX Attack 2, COOP, or Control Jujutsu – as these attacks specifically leverage legitimate control-flow transfers to reach security sensitive system calls. For instance, COOP starts with a buffer over-flow filled with fake counterfeit objects by the attacker and leverages virtual C++ functions that would appear as benign execution to LLVM CFI by not violating type. In AOCR NGINX Attack 2, with the help of the attacker-controlled worker, the attacker only needs to corrupt global variables to cause NGINX's master process loop to call `exec` under attacker chosen parameters.

In this case, Call-Type and Control-Flow contexts will seem legitimate for Bastion. However, these attacks still need to corrupt system call arguments to attacker directed values, thereby allowing Bastion to detect and block these attack vectors. Compared to LLVM CFI, Bastion employs the Argument Integrity context and is therefore tighter in its constraints compared to CFI. For this reason, Bastion is able to defend against indirect attack manipulation strategies.

11 DISCUSSION AND LIMITATIONS

11.1 Bastion under Arbitrary Memory Corruption

In theory, a powerful adversary with arbitrary memory `read`/`write` capability can circumvent all three of Bastion's contexts. However, in practice it will be very challenging particularly because our analysis pass is able to *statically* determine most constraints – direct vs. indirect call, legitimate stack traces, and constant arguments – to successfully enforce correct system call invocation in a program. For example, if `mprotect()` is used only with a constant value, `PROT_READ`, in a program, then it is impossible to call `mprotect()` with `PROT_EXEC` because such static constraints are maintained by the monitor processor running in a separate address space; this data is never available to the protected application. To bypass all three of Bastion's contexts, the attacker realistically would need to perform arbitrary `read`/`write` many times to match the expected context values without violating static constraints (*e.g.*, constant arguments). This is very challenging to carry out in real attack scenarios, thus Bastion raises the difficulty to complete such an attack.

11.2 Protecting Filesystem Related System Calls

One promising extension of Bastion is protecting file system related system calls to prevent information disclosure attacks. The main challenge is that this type of system call is called much more frequently than Bastion's sensitive system calls. To see the feasibility of such an extension, we extended Bastion's coverage to include all file system related system calls and their variants.

We present the performance overhead of this extension compared against the unprotected baseline in Table 7. Full Bastion context checking (Row 3) incurs high overhead – *e.g.*, 96.7% for NGINX. To understand where this overhead comes from, we break down Bastion into three sub-steps: 1) just hooking system calls (Row 1) to measure seccomp overhead, 2) fetching the protected program's information using `ptrace` (Row 2), and 3) verification of Bastion's three contexts (Row 3). Our results show overhead of hooking system calls (< 0.29%, Row 1) and full context checking (< 0.82%, delta between Rows 2 and 3) is negligible. A majority of overhead results from fetching protected process state using `ptrace` (< 95.7%, delta between Rows 1 and 2) due to the additional context switching overhead to access the protected program.

One approach to (almost) completely eliminate `ptrace` overhead would be to run the Bastion monitor inside the kernel as either a kernel module or via eBPF code. This solution would completely resolve overhead incurred from context switching; by being within the kernel, the Bastion monitor would now have transparent access to the protected application's process state to retrieve register values, etc. With the optimization of replacing `ptrace` with in-kernel execution, we can extend Bastion's system call protection scope with low performance overhead. Ultimately, Bastion can used as a foundational platform and be extended to implement defense for a range of threat models in the future, such as information disclosure.

Table 7: Bastion Performance overhead when file system-related system calls (*e.g.*, `open`, `read`, `write`, `send`, `recv`) and variants (*e.g.*, `openat`, `sendfile`) are protected. The baseline is the unprotected version. In this scenario, fetching process state using `ptrace` contributes the most performance overhead.

	Runtime & % Overhead Added Per Checkpoint		
Bastion Configuration	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%)

11.3 Impact of Not Protecting of All System Calls

We designed Bastion to deeply protect a subset of system calls that have security implications (*i.e.*, sensitive system calls). That being said, Bastion's Call-Type context can still sort out all not-callable system calls in a program (§3.1) so they cannot ever be weaponized. If an arbitrary (sensitive or not) system call is never used by an application, the Bastion monitor can still enforce the Call-Type context and disallow any use.

Not all system calls are security-critical (*e.g.*, `getpid`) as they do not perform any operation that can adversely affect the host system for attacker gains. Even if uncovered, non-sensitive system calls are used in an attack chain, an attacker should still exploit at least one sensitive system call. In this way, Bastion is able to block the attack by detecting unintended usage of a sensitive system call.

However, we do not claim that our sensitive system call selection is "perfect". Presently, there is no consensus of how to quantitatively assign a system call's "danger level". Thus far, this ongoing effort has been empirically deduced from performing case studies and examining real-world CVEs. Explicit system call classification is sparse in literature; Bernaschi et al. [26] provides some guidance, however their analysis is limited to buffer overflow-based attacks. More recent work ranks system call risk based on using information retrieval techniques in an exploit code analysis [52]. More comprehensive analysis remains an interesting future direction.

12 RELATED WORK

Since we already discussed the most closely related work in §2, we discuss other related studies in this section.

Advanced system call filtering. Static filtering makes filter creation and usage more accessible via libraries (`libseccomp` [51]), automation (`sysfilter` [36]), or architecture portability (ABHAYA [77]). Dynamic filtering improves soundness of system call filters using automated testing [63, 94] or dynamic profiling (Confine [41]). Some approaches leverage a temporal context [23, 42, 63] enabling specific system calls during distinct execution phases. These approaches are all coarse-grained and rely on whitelisting as a primary means of defense. Bastion goes beyond specifying a whitelist of allowed system calls, even if that whitelisting is made dynamic as in Temporal Filtering [42]. Notably, there exist several CVE's (e.g., Control Jujutsu [38], AOCR [81]) capable of bypassing Temporal Filtering as these attacks leverage system calls still permitted in the application's serving phase. For this reason, BASTION is significantly more secure than any coarse-grained system call filtering approach.

Data integrity. Data integrity seeks to protect data such that no data in a program is corruptible either by tracing or employing a secure data copy. Specialized data integrity narrows coverage to protect control-dependent data (OAT [86]), developer annotated data (Datashield [30]), or data passed as arguments in callsites (Saffire [68]). Compared to these data integrity mechanisms, BASTION only enforces argument integrity for *sensitive variables*.

13 CONCLUSION

This work is built on the insight that regardless of attack complexity or end goal, a majority of attacks *must* leverage system calls to complete their attack. Thus, we presented three specialized system call contexts (Call-Type, Control-Flow, & Argument Integrity) for securing their legitimate usage and implement them with our prototype, BASTION. Evaluating the performance impact of BASTION using system call-intensive real-world applications, we demonstrate a low runtime overhead. This shows BASTION is a practical defense on its own to block an entire attack class that leverages system calls.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and input to improve the paper. This work was partly supported by the National Science Foundation under the grants CCF-2153748, FAA ASSURE A38, A51, and A58 Grant.

REFERENCES

[1] 2022. 64-bit Linux Return Oriented Programming. https://crypto.stanford.edu/~blynn/rop/.
[2] 2022. 64-bit ROP | You rule 'em all! https://0x00sec.org/t/64-bit-rop-you-rule-em-all/1937.
[3] 2022. Analysis of Defenses against Return Oriented Programming. https://www.eit.lth.se/sprapport.php?uid=829/.
[4] 2022. ARM exploitation - Defeating DEP - executing mprotect. https://blog.3or.de/arm-exploitation-defeating-dep-executing-mprotect.html.
[5] 2022. Bypass DEP/NX and ASLR with Return Oriented Programming Technique. https://medium.com/4ndr3w/linux-x86-bypass-dep-nx-and-aslr-with-return-oriented-programming-ef4768363c9a/.
[6] 2022. Bypassing non-executable memory, ASLR and stack canaries on x86-64 Linux. https://www.antoniobarresi.com/security/exploitdev/2014/05/03/64bitexploitation/.
[7] 2022. Bypassing non-executable-stack during Exploitation (return-to-libc). https://www.exploit-db.com/papers/13204/.
[8] 2022. Crashmail 1.6 - Stack-Based Buffer Overflow (ROP). https://www.exploit-db.com/exploits/44331/.
[9] 2022. DBT-2. https://github.com/nuodb/dbt2.
[10] 2022. dkftpbench v0.45. http://www.kegel.com/dkftpbench/.
[11] 2022. Exploitation - Returning to libc. https://www.exploit-db.com/papers/13197/.
[12] 2022. HT Editor 2.0.20 - Local Buffer Overflow (ROP). https://www.exploit-db.com/exploits/22683/.
[13] 2022. Introduction to Return Oriented Programming (ROP). https://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html/.
[14] 2022. NGINX Web Server. nginx.org/.
[15] 2022. PHP 5.3.6 - Local Buffer Overflow (ROP). https://www.exploit-db.com/exploits/17486/.
[16] 2022. PMS 0.42 - Local Stack-Based Overflow (ROP). https://www.exploit-db.com/exploits/44426/.
[17] 2022. Return Oriented Programming and ROPgadget tool. http://shell-storm.org/blog/Return-Oriented-Programming-and-ROPgadget-tool/.
[18] 2022. Return-Oriented-Programming (ROP FTW). http://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf.
[19] 2022. ROP-CTF101. https://ctf101.org/binary-exploitation/return-oriented-programming/.
[20] 2022. Simple ROP Exploit Example. https://gist.github.com/mayanez/c6bb9f2a26fa75261a9a26a0a637531b/.
[21] 2022. vsftpd. http://www.kegel.com/dkftpbench/.
[22] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*. Alexandria, VA.
[23] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. 2021. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *30th USENIX Security Symposium (USENIX Security 21)*.
[24] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. 2019. Nibbler: Debloating Binary Shared Libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 70–83.
[25] Salman Ahmed, Ya Xiao, Kevin Z Snow, Gang Tan, Fabian Monrose, and Danfeng Yao. 2020. Methodologies for quantifying (Re-) randomization security and timing under JIT-ROP. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1803–1820.
[26] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V Mancini. 2000. Operating system enhancements to prevent the misuse of system calls. In *Proceedings of the 7th ACM conference on Computer and communications security*. 174–183.
[27] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 16.
[28] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.
[29] Nicholas Carlini and David Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses.. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. San Diego, CA.
[30] Scott A Carr and Mathias Payer. 2017. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 193–204.
[31] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 147–160.
[32] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats.. In *USENIX Security Symposium*, Vol. 5.
[33] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. 2014. ROPecker: A generic and practical approach for defending against ROP attack. (Feb. 2014).
[34] Stephen J Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's a TRaP: Table Randomization and Protection Against Function-reuse Attacks. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.
[35] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. Singapore, Republic of Singapore.
[36] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. 2020. `sysfilter`: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 459–474.
[37] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient protection of path-sensitive control security. In *Proceedings of the 26th USENIX Security Symposium (Security)*. Vancouver, BC, Canada.
[38] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. Denver, Colorado, 901–913.
[39] Lang Feng, Jiayi Huang, Jeff Huang, and Jiang Hu. 2021. Toward Taming the Overhead Monster for Data-flow Integrity. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 27, 3 (2021), 1–24.
[40] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Xi'an, China.
[41] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated system call policy generation for container attack surface reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 443–458.
[42] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal system call specialization for attack surface reduction. In *29th USENIX Security Symposium (USENIX Security 20)*. 1749–1766.
[43] Will Glozer. 2019. a HTTP benchmarking tool. https://github.com/wg/wrk.
[44] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE*

Symposium on Security and Privacy (Oakland). San Jose, CA.
[45] Jens Grossklags and Claudia Eckert. 2018. τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *Proceedings of the 21th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Heraklion, Crete, Greece.
[46] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent backward-edge control flow violation detection using intel processor trace. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*. Scottsdale, AZ.
[47] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of {Data-Oriented} Exploits. In *24th USENIX Security Symposium (USENIX Security 15)*. 177–192.
[48] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, ON, Canada.
[49] Mohannad Ismail, Jinwoo Yom, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. 2021. VIP: Safeguard Value Invariant Property for Thwarting Critical Memory Corruption Attacks. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. ACM, 1612–1626. https://doi.org/10.1145/3460120.3485376
[50] Ismail, Mohannad and Yom, Jinwoo and Jelesnianski, Christopher and Jang, Yeongjin and Min, Changwoo. 2021. VIP: Safeguard Value Invariant Property for Thwarting Critical Memory Corruption Attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1612–1626.
[51] Jake Edge. 2012. A library for seccomp filters. https://lwn.net/Articles/494252/.
[52] Sunwoo Jang, Somin Song, Byungchul Tak, Sahil Suneja, Michael V. Le, Chuan Yue, and Dan Williams. 2022. SecQuant: Quantifying Container System Call Exposure. In *Proceedings of the 27th European Symposium on Research in Computer Security (ESORICS)* (Copenhagen, Denmark). 145–166.
[53] Jonathan Corbet. 2004. x86 NX support. https://lwn.net/Articles/87814/.
[54] Jonathan Corbet. 2005. Securely renting out your CPU with Linux. (January 2005). https://lwn.net/Articles/120647/.
[55] Jonathan Corbet. 2019. New system calls for memory management. https://lwn.net/Articles/789153/.
[56] The kernel development community. [n. d.]. Seccomp BPF (SECure COMPuting with filters). https://lwn.net/Articles/656307/.
[57] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. 2019. Adaptive Call-site Sensitive Control Flow Integrity. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 95–110.
[58] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. 2019. Origin-sensitive control flow integrity. In *28th USENIX Security Symposium (USENIX Security 19)*. 195–211.
[59] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.
[60] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2014. Code-pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, 147–163.
[61] Larabel, Michael. 2018. Glibc 2.28 Released With Unicode 11.0 Support, Statx & Intel Improvements. https://www.phoronix.com/news/Glibc-2.28-Released.
[62] Larabel, Michael. 2020. Intel Confirms CET Security Support For Tiger Lake. https://www.phoronix.com/news/Intel-CET-Tiger-Lake.
[63] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. 2017. SPEAKER: Split-phase execution of application containers. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 230–251.
[64] Linux Programmer's Manual. 2023. PTRACE(2) – Linux manual page. https://man7.org/linux/man-pages/man2/ptrace.2.html.
[65] lwn.net. 2018. GNU C Library 2.28 released. https://lwn.net/Articles/761462/.
[66] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. Denver, Colorado.
[67] Microsoft Support. 2017. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in.
[68] Shachee Mishra and Michalis Polychronakis. 2020. Saffire: Context-sensitive function specialization against code reuse attacks. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 17–33.
[69] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, UK.
[70] National Institute of Standards and Technology. National Vulnerability Database. 2015. CVE-2012-0809. https://nvd.nist.gov/vuln/detail/CVE-2012-0809.
[71] National Institute of Standards and Technology. National Vulnerability Database. 2015. CVE-2013-2028. https://nvd.nist.gov/vuln/detail/CVE-2013-2028.
[72] National Institute of Standards and Technology. National Vulnerability Database. 2015. CVE-2014-1912. https://nvd.nist.gov/vuln/detail/CVE-2014-1912.
[73] National Institute of Standards and Technology. National Vulnerability Database. 2015. CVE-2014-8668. https://nvd.nist.gov/vuln/detail/CVE-2014-8668.
[74] National Institute of Standards and Technology. National Vulnerability Database. 2015. CVE-2015-8617. https://nvd.nist.gov/vuln/detail/CVE-2015-8617.
[75] National Institute of Standards and Technology. National Vulnerability Database. 2016. CVE-2016-10190. https://nvd.nist.gov/vuln/detail/CVE-2016-10190.
[76] National Institute of Standards and Technology. National Vulnerability Database. 2016. CVE-2016-10191. https://nvd.nist.gov/vuln/detail/CVE-2016-10191.
[77] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. 2020. Automated policy synthesis for system call sandboxing. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–26.
[78] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. 2013. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22th USENIX Security Symposium (Security)*. Washington, DC.
[79] Chenxiong Qian, Hong Hu, Mansour A Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA.
[80] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD, 869–886.
[81] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, et al. 2017. Address-Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.
[82] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.
[83] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. 1–11.
[84] sourceware.org. 2018. V2 [PATCH 24/24] Intel CET: Document –enable-cet. https://sourceware.org/legacy-ml/libc-alpha/2018-07/msg00550.html.
[85] SQLite. [n. d.]. SQLite. https://www.sqlite.org/index.html.
[86] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. 2020. OAT: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1433–1449.
[87] The Clang Team. 2022. Clang 16 documentation: CONTROL FLOW INTEGRITY. https://clang.llvm.org/docs/ControlFlowIntegrity.html.
[88] The PAX Team. 2003. Address Space Layout Randomization. https://pax.grsecurity.net/docs/aslr.txt.
[89] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. San Diego, CA.
[90] Torvalds, Linus. 2022. syscall_64.tbl. https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall$_6$4.tbl.
[91] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. Denver, Colorado.
[92] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 927–940.
[93] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. 2017. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.
[94] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. 2017. Mining sandboxes for linux containers. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 92–102.
[95] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22th USENIX Security Symposium (Security)*. Washington, DC.

Received 2022-10-20; accepted 2023-01-19