CASE: Minimizing Attack Surfaces based on Context-Aware System Call Enforcement

Man-Ni Hsu, Tsung-Han Liu, Hsuan-Ying Lee, Chun-Ying Huang *Senior Member, IEEE*
Department of Computer Science, College of Computer Science, National Yang Ming Chiao Tung University

***Abstract*—Invoking system calls in exploit implementation is a typical approach to compromising a system. A key objective of these attacks is to manipulate program execution paths, with a specific focus on invoking targeted system calls. Our study introduces Context-Aware System Call Enforcement (CASE), a software-based approach meticulously crafted to shrink the attack surface associated with system call-based exploits. CASE achieves this by rigorously validating the context, mainly backward function call paths and runtime stack states, to ensure the legitimacy of system call invocations. Our strategy incorporates innovative elements, including anchored entry points, return address-based validation, and frame size checks. We formalize our approach by creating NP-hard challenges for potential attackers and complete with a proof-of-concept (PoC) implementation that shields against attacks. Our PoC implementation introduces minimal overhead, less than 2%, for context validation. Simultaneously, it adeptly identifies and halts attacks of varying complexities, ranging from simple examples to real-world servers.**

***Index Terms*—Application Security, Backward Path Validation, Minimizing Attack Surfaces, System Call**

I. Introduction

SYSTEM calls serve as a pivotal linchpin in the intricate landscape of exploit development, acting as gateways for attackers to manipulate the underlying functionalities of an operating system. Based on the knowledgebase from MITRE defense [1], half of the attack tactics involve the use of system calls and can be defended by performing system call analysis. Analyzing shell codes retrieved from publicly available repositories [2] shows that all binary-based exploits use at least one system call. These observations show that the attack surfaces could be effectively minimized by managing the use of system calls. Although system calls are essential interfaces for attackers to carry out most attacks, most defense approaches concentrate on minimizing the system call set rather than ensuring their legitimate usage. Researchers have proposed solutions such as debloating [3–6], system call filtering [7–11], and system call sandboxing [12–15] to reduce the number of available system calls (e.g., creating a denylist), thus minimizing the attack surface. However, these defenses still leave a significant attack surface because those allowed system calls may still be used illegitimately. For example, suppose a web server executes an external Common Gateway Interface (CGI) program via the `fork` and `exec` system calls. A defense approach would allow the use of the two system calls in the web server process. Nevertheless, it is straightforward that invoking the system calls from the web server itself or a malicious shellcode injected by attackers must be handled independently. To this end, a more fine-grained control on enforcing system call usage is required to ensure a system can be well-protected.

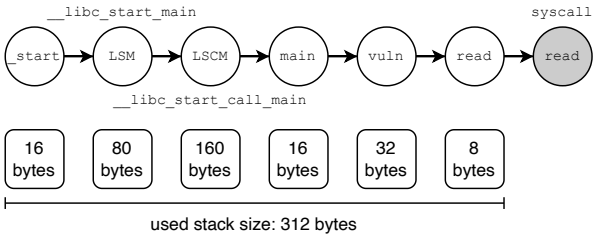
The works presented in this paper are supported, in part, by the National Science and Technology Council under the grants NSTC-113-2221-E-A49-186-MY3, 113-2634-F-A49-001-MBK, and 114-2218-E-A49-017. We thank the anonymous reviewers for their valuable and insightful comments for improving this paper. Corresponding Author: Chun-Ying Huang

```
1  /* header files omitted */
2  void shell() {
3    _exit(execlp("/bin/sh", "/bin/sh", NULL));
4  }
5
6  void vuln() {
7    char buf[16];
8    read(0, buf, 0x20);
9  }
10
11 int main() {
12   vuln();
13   return 0;
14 }
```

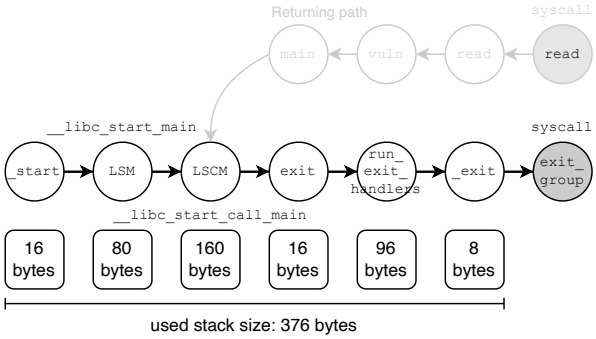
Fig. 1: A simple motivating example.

In this study, we propose a novel approach to enforce the usage of system calls by validating the calling context of targeted system calls, achieving a more fine-grained control of the system call usage. We use a motivating example to illustrate how the calling context of a system call can be retrieved from a running process and demonstrate several fundamental observations relevant to our defense approach. Figure 1 shows a simple program that contains a buffer overflow vulnerability. The program contains three functions, `main()`, `vuln()`, and `shell()`, where the `main()` function calls the `vuln()` function, and the `shell()` function is unreachable in this program. The program is compiled with *-no-pie* and *-fno-stack-protector* options so that an attacker can overwrite the return address of function *vuln()* to change the program flow.

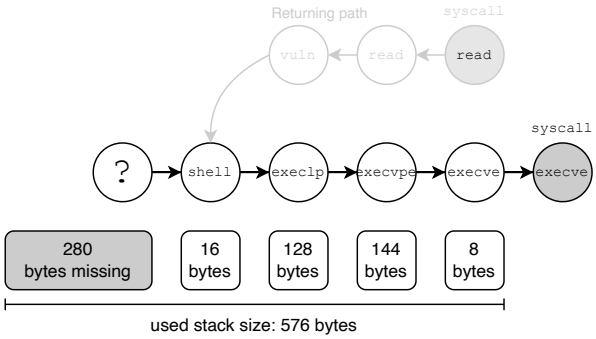
We look into the program state before the stack is overflowed by the `read()` function call. The `read()` function calls the `read` system call to receive input from the user. Figure 2a shows the call path from the program entry point to the invocation of the `read` system call. If a user does not overflow the stack, the program then terminates by returning from the `vuln()` function followed by returning from the `main()` function. The caller of the `main()` function, which is the `__libc_start_call_main()` function, then calls



(a) The call path on the invocation of system call 'read'



(b) The call path on the invocation of system call 'exit_group'



(c) The call path on the invocation of system call 'execve'

Fig. 2: Call paths for invoked system calls.

`exit()` function and finally reaches the `exit_group` system call, as shown in Figure 2b. However, suppose an attacker overflows the buffer by filling the return address of the `vuln()` function with the function address of the `shell()` function. In that case, the program flow is transferred to the `shell()` function and finally invokes the `execve` system call, as shown in Figure 2c. Note that we cannot determine the caller of the `shell()` function because the attacker has corrupted the stack.

We can have four fundamental observations from the program states shown in Figure 2. First, we can perform the *stack unwinding* [16] operation to retrieve the call path of the user-space program and the corresponding consumed stack sizes of each function on the path on the invocation of an arbitrary system call. Second, a *healthy* function call path should be backtraced to a *valid* source. In the example mentioned above, the valid source is the entry point, the `_start` function of the program. Third, even if we can backtrace a function call path to a valid source, the invocation of the `shell()` function is weird because it is originally unreachable in the program. Furthermore, if we sum up the used stack size of each recognized function call, the result does not match the expected stack size. It is worth noting that typical system call-based approaches cannot defend against the motivating example. Readers may refer to Section IV-G for the detailed comparison.

We design and implement our defense approach based on the observations. The general objective of our approach is to validate the program runtime state, i.e., context, to ensure that a program is on the right track. Our contribution is four-fold. First, we present CASE, a nimble solution designed to scrutinize the legitimacy of system call context meticulously. Second, through extensive observations, we demonstrate the compatibility of our approach with modern Linux distributions. Third, we formalize our approach, creating NP-hard challenges for potential attackers, and complete with a proof-of-concept (PoC) implementation that shields against attacks. Lastly, we subject our implementation to a comprehensive evaluation, showcasing its efficiency and effectiveness. Our PoC implementation introduces minimal overhead, less than 2%, for context validation. Simultaneously, it adeptly identifies and halts attacks of varying complexities, ranging from simple examples to real-world GUI applications and servers. The rest of this paper is organized as follows. We review several relevant research works in Section II. We introduce our approach in Section III. The evaluation of our proposed approach is performed in Section IV. A concluding remark is given in Section V.

II. Related Work

This section reviews relevant research works and approaches that can be used to determine the validity of a program's runtime state. Our approach is entirely software-based, and we omitted discussions with hardware-based approaches such as [17] due to space constraints.

A. Control Flow Integrity

Control flow integrity (CFI) [18] is a fine-grained approach to enforce the execution flow of a program. While CFI approaches must monitor all *call* and *return* instructions to ensure forward and backward integrities, they usually bring additional runtime costs and may require specific hardware assistance. Most CFI solution enforces their policy depending on a Control Flow Graph (CFG), and it is challenging to construct a precise CFG without source code. Furthermore, most existing binary-level CFI mechanisms use dynamic instrumentation [19–23] or static rewriting [24–27] to validate the current target of the call site instead of the whole path. To our knowledge, control flow enforcement approaches based on the full call path often depend on hardware tracing features to obtain the required information [28–31]. While many solutions enforce backward CFI policies [18, 20, 27, 32], they incur significant overhead. Although researchers have proposed solutions such as the parallel shadow stack [33] and the dual stack [34, 35] to reduce performance overheads, they could be bypassed by exploiting the information in the stack. Another type of CFI enforcement approach uses a coarse-grained policy to restrict a return address to only those adjacent to the call

site [21, 24, 28], which still leaves many attack surfaces. To this end, we propose a practical approach for CFI full-path enforcement without hardware tracing dependencies at the binary level. Our proposed approach differs from CFI approaches as we focus on the enforcement of system calls, the most critical interfaces for executing attacks.

B. System Call Enforcement

Several research works have been proposed to enforce system calls. Forrest et al. [36] first proposed a practical intrusion detection approach through the sliding window to analyze the sequence of system calls in 1996. They build up a database to define normal behavior and detect anomalies with the percentage of abnormal sequences. New model architectures for the system calls are then proposed in subsequent research works. Linn et al. [37] proposed a binary rewriting method to mitigate remote code injection attacks. Typically, they add the address of each system call instruction to the ELF executable as a new section called IAT, allowing it to detect injected system calls. Sensitive system calls play a vital role in various attacks. Therefore, system call filtering is one of the defense techniques. It aims to minimize the attack surface by disabling unallowed system calls. DeMarinis et al. [7] proposed a binary analysis-based framework named sysfilter. They restricted the system call set through static program analyses based on the function-call graph. Jelesnianski et al. [38] proposed a novel system call integrity called BASTION. They implement a compiler and a runtime monitor to demonstrate the three introduced context integrity. However, compared to CASE, BASTION needs source code. Ghavamnia et al. [9] present the limiting system call set for each initialization and serving phase. Nevertheless, these defense mechanisms still permit system calls to be invoked during legitimate execution phases. A more fine-grained control on enforcing system call usage is required to ensure a system can be well-protected.

III. Approach

In this section, we explained the design and challenges of our proposed approach. Our approach comprises two parts: static analysis (Sections III-B and III-C) and runtime enforcement (Sections III-D, III-E, and III-F). The static analysis is performed once per binary, while the runtime system loads the analyzed results and enforces protection with minimal overhead.

A. Architecture Overview and Threat Model

Figure 3 shows the general architecture of our proposed approach. The involved modules can be classified into two categories: the modules used in the *preprocessing phase* and the modules used in the *runtime phase*. Modules used in the preprocessing phase collect required information for runtime validation. The information includes the entry points of the executable, the required stack size information, and functions and the code segments that can form function call chains in the program. The preprocessing phase can be done offline.

In contrast, modules used in the runtime phase are used to monitor the program execution state and determine if the

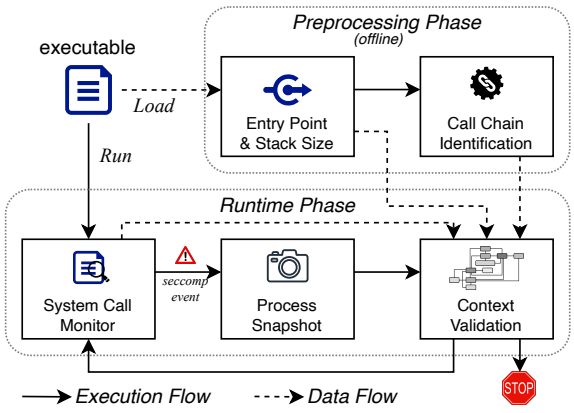


Fig. 3: Architecture overview of our proposed approach.

monitored state is valid or invalid. The program is terminated (or continued with alerts) when an invalid state is detected to prevent it from compromising the system. Before introducing our proposed approach, we list the assumptions and the threat model used in this study.

- Given that not all the targets have source codes accessible, we assume that both the proposed approach and the attackers can access only binary files without source codes.
- We assume the runtime phase is deployed by the system administrator. It runs within a privileged process different from the monitored executable in the user space and cannot be tampered with by attackers.
- We assume that the targets to be protected are not obfuscated, as our approach is designed to secure benign applications. Most legitimate applications avoid obfuscation to prevent being mistakenly classified as malicious.
- An attacker can access and analyze the target service binary files offline. It is a typical process in which attackers conduct reconnaissance before launching attacks against a target.
- An attacker invokes at least one system call in the attack process. System calls can be injected by an attacker or executed by leveraging gadgets in the target.
- The protected targets of the proposed approach are limited only to user space applications.

The details of the involved modules are further introduced in the rest of this section.

B. Entry Point and Stack Size Recognition

One essential module used in our context validation approach is recognizing a program's possible entry points and stack sizes. Our approach uses the recognized entry points and stack sizes to determine whether an identified call path is valid. The entry point of a program can be directly retrieved from the header of its executable file. For Executable and Linkable Format (ELF) files, the `e_entry` field in the ELF header records the entry point of the whole program. However, from the perspective of the program runtime state, a multi-threaded program would have multiple valid entry points for a call path in addition to the program entry point. Therefore, identifying thread entry points is also critical to entry point-based context validation.

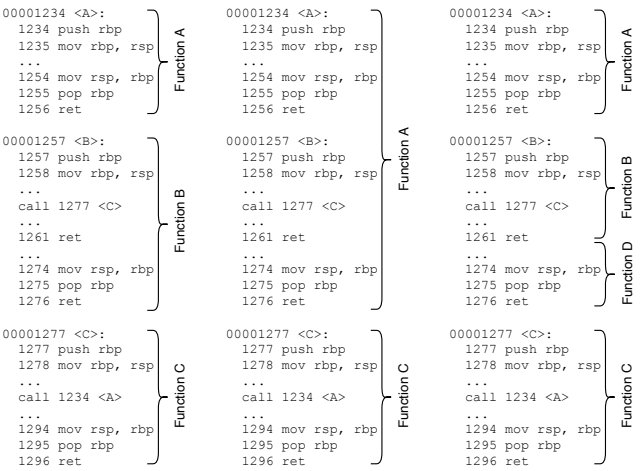
Suppose that only binary executables are available in the preprocessing phase. Special entry points, such as a thread's entry point (thread start routine) or a signal handler, can be identified through static taint analysis of pointers passed to relevant sink points, such as the standard `pthread_create` function call or the `clone` system call. Note that running taint analyses could be time-consuming or inaccurate. Therefore, our static analysis against thread entry points is limited only to pointers passed within the same function. For the rest of the cases, we use information collected in runtime by hooking the `pthread_create` function and the `clone` system call. The runtime parts are integrated with the system call monitor module introduced in Section III-D. Note that although thread creation can be done via different interfaces, e.g., `std::thread` in C++ or other variants, most of them still call the underlying `pthread_create` function or the `clone` system call. Therefore, hooking the aforementioned two types of application programming interfaces (APIs) is sufficient for most cases.

Note that an attacker may overflow the stack and then craft a valid call path. Therefore, we propose the *anchored entry points* approach to minimize possible attack surfaces by crafting call path attacks at runtime. The details of the approach are further introduced in Section III-D.

One essential piece of information required by our proposed approach is stack size information. Given the binary codes of a function, our approach has to determine the stack size consumed by the function when performing function calls and system calls. With the control flow graph of a function and the identified stack-relevant instructions, the size of the stack consumed by the function can be statically analyzed by performing taint analysis against instructions that manipulate stacks. The stack-relevant instructions include `push`, `pop`, and arithmetic operations (usually `add` and `sub`) that operate on the stack register, e.g., `rsp` on x86_64 architecture. Figure 4 shows a simple example of the `__libc_start_call_main` function. The first few instructions at the beginning of the function consume an additional 152 bytes of stack, which leads to 160 bytes of stack consumption when calling the `main` function (return address inclusive). The stack size consumed at any arbitrary address is obtained by analyzing assembly instructions for each recognized control flow. In our implementation, we leverage the open-sourced radare2 [39] tool to perform the static analysis. A simple script that works with the `aaaa` and `aflj` radare2 commands can be used to obtain functions and their corresponding basic blocks. Note that our static taint analysis marks a function as having an indeterministic stack size if it employs dynamic stack memory allocation, such as when using `alloca()` or variable-length arrays (VLA) with a value provided at runtime. However, dynamic stack memory allocation is generally discouraged and considered a bad practice from a secure programming perspective. For instance, the Linux kernel has been VLA-free since version 4.20 [40]. Industry standards also discourage dynamic stack allocation, such as C11 [41] (which makes VLAs optional) and MISRA C [42] (where rule 18.8 prohibits the use of VLAs). For real-world applications, only 52 out of 2558 functions (2%) in GNU libc and 1 out of 910 functions

```
1  0000000000029d10 <__libc_start_call_main>:
2  +8   29d10:  push   rax
3  -8   29d11:  pop    rax
4  +152 29d12:  sub    rsp,0x98
5       29d19:  mov    QWORD PTR [rsp+0x8],rdi
6       29d1e:  lea    rdi,[rsp+0x20]
7       29d23:  mov    DWORD PTR [rsp+0x14],esi
8       29d27:  mov    QWORD PTR [rsp+0x18],rdx
9       29d2c:  mov    rax,QWORD PTR fs:0x28
10      29d35:  mov    QWORD PTR [rsp+0x88],rax
11      29d3d:  xor    eax,eax
12      29d3f:  call   421e0 <_setjmp>
13      ...
```

Fig. 4: Disassembled `__libc_start_call_main` assembly codes from `glibc`.



(a) Correctly merged. (b) Incorrectly merged. (c) Incorrectly split.

Fig. 5: Correct and incorrect identification of function boundaries.

(0.1%) in the Firefox browser distributed with Debian Linux 11.6 employ dynamic stack memory allocation, indicating that our proposed approach can effectively secure the target in most cases.

C. Call Chain Identification

The call chain identification module is another essential module used in the preprocessing phase. This module aims to identify the required information for validating whether a call path in the current context is valid. The preprocessing phase statically analyzes both the ELF header and assembly codes of an ELF file. We decompose the details of this module into three parts, i.e., the identification of function boundaries, the identification of direct calls, and the identification of indirect calls.

1) Identification of Function Boundary: A call path is composed of multiple caller-callee relationships. By definition, a caller is a function that calls another function, and a callee is a function being called. To validate a call path, we have to know what caller-callee relationships are valid and then determine the validity of the entire call path. The first step to identifying correct caller-callee relationships is to recognize

function boundaries and identify all valid callees called by a caller. Identifying function boundaries is to retrieve the *precise* starting address and the ending address of a function. However, it is a non-trivial task to identify function boundaries. Figure 5a shows one piece of a program with three functions: A, B, and C. If we use the `call` instruction to identify the beginning of a function, we can get functions A and C but would miss function B, as shown in Figure 5b. Similarly, if we use the `ret` instruction to identify the end of a function, we may accidentally split one into two functions, as shown in Figure 5c. The most critical part here is that the function boundaries recognized here must be the same as those functions recognized when performing stack unwinding operations. Otherwise, it could lead to incorrect classification of valid or invalid program runtime states.

In addition to identifying function starting addresses using the `call` instruction, we use two existing pieces of information embedded in binary files to identify function boundaries precisely. The first one is the symbol table. We read the symbol table in binary executables when available. Symbol tables may not be available in stripped binaries. However, all library files, e.g., `.so` files, must have a dynamic symbol table. Although the dynamic symbol table may not cover all functions in a library file, it still provides accurate information for identifying exported functions.

The other reliable source for identifying functions is the Exception Handling (EH) frame. The EH frame information records *how* to find the starting address of a call stack frame for a function call. The starting address is usually relative to the content of a register such as `RBP`. With the information, a debugger of an exception handler can quickly identify the stack frame and perform required operations. Each frame description entry (FDE) stored in the ELF file's `.eh_frame` section usually records a function's starting address and size. Since the EH frame information is often used for exception handling, it is unlikely to be removed. Bastian et al. [16] proposed to rebuild missing EH frame information based on binary analysis tools [43]. We also show that the EH frame has been widely deployed in modern UNIX-like operating systems. More details can be found in Section IV-A.

2) Identification of Direct Caller-Callee Edge: Once function boundaries are recognized, the next step is to obtain valid caller-callee relationships of a calling function. Suppose a function call from caller A to callee B forms an edge. We must identify the caller's valid calling edges for each recognized function. We tackle this problem by inspecting all `call` and `jump`-like instructions within a caller. We discuss direct calls (jumps) in this sub-subsection and indirect cases in the following sub-subsection.

We use the address followed by the `call` instructions to identify a direct call edge. Suppose a target being called is an address within the executable. In that case, a calling edge is recorded as $f_A \Rightarrow f_B$, where f_A and f_B are the caller's and the callee's addresses, respectively. Similar to the `call` instruction, we use the same approach to recognize the targets of all `jump`-like instructions. However, readers must notice that *jump records are not visible on the stack*. Therefore, to obtain a caller-callee edge visible on the stack, we have to *follow* jumps to identify the real callee of a caller. Suppose we use the annotation $f_B \rightarrow f_C$ to represent a *jumping edge* from function B to function C. If there is an identified relationship $f_A \Rightarrow f_B \rightarrow f_C$, it implies a caller-callee edge $f_A \Rightarrow f_C$. To handle the exceptional cases caused by jumping edges inside an executable, which we call *internal jumps*, we add all the jump targets of a function to the callee set of the function. Internal jumps starting from a caller may happen multiple times before reaching a real callee, which forms an *internal jump chain*. Therefore, updating the callee set based on jumping targets must be recursively done.

Note that a `call` (`jump`-like) instruction may call (jump) to a function stored at an unpredictable address that requires runtime resolution. It usually happens for functions in a dynamic library. We call these cases *gadget call (jump) cases*. Our observations show that gadget cases are often implemented by calling (jumping to) gadgets generated in the `.plt` section. Although gadget call (jump) cases sound like indirect calls (jumps), we still classify them as direct calls (jumps) because the relative addresses of `.plt` symbols can be resolved offline, and their real addresses can be known once the base address is known on loading the library. We use two strategies to handle the gadget cases. In the first strategy, we preserve only the symbol name of the calling (jumping) gadgets. When performing stack unwinding operations, we resolve the actual address of the calling (jumping) targets in runtime. In the second strategy, we statically perform symbol name resolution based on the dynamic linker library dependencies and use the resolved target addresses as the calling (jumping) targets.

3) Handle Indirect Caller-Callee Edge: Another typical implementation of a function call is named an *indirect call (jump)*. Instead of calling a known address or symbol, an indirect call target is usually a register or a memory address that stores a callee's address. To validate a call path, we also have to handle indirect call cases and identify possible targets of an indirect caller. The significant challenge of handling indirect calls is that the actual addresses to be called are often known in runtime compared to direct calls. Therefore, this sub-subsection summarizes the statically analyzed heuristics we used to validate indirect call paths and narrow down possible callees of an indirect caller based on our observed real-world cases.

- **Return address-based validation.** It is straightforward that if we do not have caller-callee information for indirect calls, the validity of a call path cannot be validated. It implies that even if we can backtrack a call path to the program or the thread entry point, there is (at least) one *broken link* between a caller and a callee. A broken link between a caller and a callee means no valid caller-callee edge can be found. Assuming that indirect calls cause broken links, one simple heuristic to validate the broken link is to verify if a callee's return address is right after an indirect call from the caller or a direct call that calls the function with an indirect jump. The implementation of this heuristic is simple. Our call chain identification module only has to statically identify the calls used in each function, including indirect calls and direct calls whose targets contain indirect jumps, and then record their corresponding return addresses to support this

heuristic.

- **Argument-based validation.** Van der Veen et al. [22] also validate the number of arguments passed between a caller and its callee to determine the validity of a function call and mitigate the attacks leveraging indirect calls. We employ a similar check but enforce the policy when performing backtracking. Specifically, on backtracking (returning) from function b to function a, we compare the number of arguments handled by b against those passed from a.
- **Calling global variables.** One possible approach to narrow down the possible callees for indirect calls is to perform taint analysis against function pointers stored as global variables. Given that global variables often reside in the `.bss` (uninitialized data) or `.data` (initialized data) section. Suppose a calling (jumping) target is loaded from the `.bss` or the `.data` section. One can retrieve the actual callee address by analyzing the assignment operations to the global variable memory address based on the assigned values. The caller-callee relationship can then be built based on the retrieved addresses.
- **Calling dynamically loaded symbols.** One implementation to make an indirect call is to load symbols dynamically. We can recognize callers of dynamic symbol loading functions and retrieve the parameters passed to these functions to get the corresponding information, including library and symbol names. The analysis above can be done statically. With the obtained information, we can then use the symbol names to look up possible target addresses of this type of indirect call and add the target address to the callee set of the caller.

We perform the aforementioned static analysis for binary files and retrieve all the caller-callee relationships. If a broken link is recognized in the runtime, we use the *return address-based* and *argument-based validation* heuristics to perform further checks. Note that the heuristics return address-based validation, calling global variables, and calling dynamically loaded symbols do not produce false positives. A violation of the three heuristics can be rejected directly. However, the argument-based validation heuristic using only binary codes could be imprecise compared to source code-based analysis. Our *radare2* and *Ghidra*-based [44] implementation shows that the current implementation achieves 89.64% precision in recognizing the number of arguments. Once the calling (jumping) addresses and caller-callee relationships are recognized, we store all the information on a per-file basis for each statically analyzed executable. These files are loaded when an executable is launched by our program launcher, i.e., the system call monitor introduced later in Section III-D.

D. System Call Monitor

Due to the incurred overhead, it is infeasible to monitor and perform program state snapshots continuously. Therefore, our approach only monitors and takes program state snapshots when a system call is to be invoked. Since the sink of a typical attack would involve a system call to interact with the operating system, e.g., invoking a shell, we believe that performing system call filtering based on context validation is sufficient for defending against most attacks.

Many existing works [45–48] can be used for monitoring and hijacking system calls. To focus more on the design of the approach, we use the built-in mechanism in the modern Linux kernel. To intercept system calls, we first attempt to work with the `ptrace(PTRACE_SYSCALL)` feature. In the usage scenario of `ptrace`, there is a tracer and a tracee, where the tracer is the process to perform process state monitoring by invoking `ptrace`-relevant system calls, and the tracee is the process to be monitored. When `ptrace(PTRACE_SYSCALL)` is invoked, a tracee process is stopped whenever it is going to enter or leave a system call. Once a tracee is stopped, the tracer can access the tracee's registers and memory, which is suitable for retrieving the data needed to check the state. However, working with `ptrace(PTRACE_SYSCALL)` could stop a tracee even when it uses some harmless but frequently used system calls such as `futex`, `poll`, and `read`. System call interceptions incur context switches, which could degrade the performance and the user experiences of a monitored process. Therefore, the number of interceptions should be minimized.

To minimize the number of system call interceptions, we use `ptrace` with `SECCOMP_RET_TRACE` rules instead of using `ptrace(PTRACE_SYSCALL)`. `SECCOMP_RET_TRACE` is a feature provided by *seccomp* [49], a computer security feature offered by the Linux kernel to operate on the *secure computing* state of a process. With `SECCOMP_RET_TRACE`, a `ptrace`-based tracer can be notified before an execution of a system call. Working with `SECCOMP_RET_TRACE` has two significant benefits. First, a tracer can be notified only when entering a system call. Therefore, the number of interruptions is cut to half. Second, `SECCOMP_RET_TRACE` rules can be customized using `bpf` programs [50], which are built-in filtering mechanisms implemented in the OS kernel. With `bpf`, our approach can focus only on suspicious system calls instead of handling noises triggered by harmless system calls. Based on the aforementioned mechanisms, a monitored process only stops when a registered seccomp `SECCOMP_RET_TRACE` is triggered. When a `SECCOMP_RET_TRACE` rule is triggered, the tracer is notified with a `SIGSTOP` signal having `PTRACE_EVENT_SECCOMP` flag set. A notified tracer can then read registers and memories from a tracee to determine the validity of the program state.

We mentioned that seccomp allows a tracer to register `bpf`-based rules for system call filtering so that our approach can focus on suspicious system calls. To build the filtering list for suspicious system calls (or we called it a watch list for short), we collect 144 and 37 shellcodes running on x86 and x86_64 systems, respectively, from the shell-storm database [2] and emulate the shellcodes with the unicorn engine [51]. We then observe the system call usage from the emulated shellcodes and select suspicious system calls from the observations. The identified systems for constructing the watch list are summarized in Table I. Please note that the users can configure the selected system calls. Due to dataset availability and popularity, the system calls summarized in Table I are for demonstrating one possible set that works effectively for shellcode injection attacks.

As mentioned in Section III-B, we propose the *anchored entry points* approach to minimize possible attack surfaces

TABLE I: List of suspicious system calls from shellcodes.

socket	ioctl	open	vfork	fork	access
pause	chmod	connect	kill	mmap	creat
reboot	bind	setsid	listen	munmap	exit
pwrite64	clone	setreuid	chdir	mprotect	setuid
clone3	setregid	execve	lseek	execveat	setgid
rt_sigprocmask		sethostname			

by crafting call path attacks at runtime. The approach is implemented in runtime as one part of the monitor. The anchored entry point approach records the stack address that stores the return address to the entry point when loading a program and creating a new thread. Since the stack frame for the entry point does not change until the program (or a thread) terminates, validating the stack address that stores the return address to the entry point can guarantee a valid (or crafted) call path must be returned to the expected entry point. It increases the difficulties in crafting valid call paths.

E. Process Snapshot

On interruption of a program due to registered system calls, our approach takes a program runtime state snapshot and performs *stack unwinding* based on the snapshot. Stack unwinding is an operation that recognizes the stack frames of function calls back toward the previous caller. Stack unwinding is a critical operation in exception handling. When an exception is thrown, the program flow must be transferred to an appropriate exception handler, which could reside in a parent or grandparent function. Therefore, the exception-handling mechanism recovers stack states by performing stack unwinding operations. It ensures the stack state can be correctly recovered when running the exception handler.

In many processor architectures, including x86, x86_64, and ARM, executing a `call` instruction pushes the address of the subsequent instruction onto the stack as the return address. It then changes the program counter to the called function. Therefore, iteratively performing stack unwinding lets us know the past function calls at the current state, and this information can be used to recognize the call chain. Note that our approach performs stack unwinding just for building call paths. Unlike exception handlers, we do not modify or change the state of the stack. Once a call chain is identified, we pass it to the next stage, the context validation module. The call chain passed to the context validation module is based on the results of stack unwinding. For each identified function call on the stack, it has a four-tuple record in the following form:

$$\{base, function\text{-}offset, next\text{-}addr, stack\text{-}frame\text{-}size\}.$$

The *base* is the loaded memory base address of the executable (or library) containing the function. The *function-offset* is the function address relative to its *base*. The *next-addr* is the next program counter address to be invoked relative to the function address. Since we are performing stack unwinding, the *next-addr* is equivalent to the current program counter for the top stack frame. For each non-top stack frame, the *next-addr* is the return address of its succeeding frame. The *stack-frame-size* indicates the size of the current stack frame. Note that the size varies in different locations of the same function. This is because a function may allocate spaces on the stack for diverse purposes, and the stack-frame-size changes depending on how the stack is manipulated in the function.

Algorithm 1 The path validation algorithm

Input: *frames*: Stack unwinding results and the corresponding function information.
Input: *funclist*: List of all recognized functions.
Input: *indRetAddr*: Return addresses for all indirect calls.
Input: *anchors*: The anchors of the entry points.
Output: Validation result: *Valid* (True) or *Invalid* (False)

```
top = frames[0]     // frames[0] is the top stack frame
Refresh funclist & indRetAddr if top.base is unknown
if (top.base, top.offset) is not in funclist then
    return False
end if

caller = None; callee = top
for each caller in frames[1:] do
    Refresh funclist & indRetAddr if caller.base is unknown
    if (caller.base, caller.offset) is not in funclist then
        return False
    end if
    if callee in callee_set_of(caller) then
        if callee.return-addr ≠ caller.next-addr then
            return False
        end if
        callee = caller; continue
    end if
    if caller.next-addr in indRetAddr then
        if caller.argument < callee.argument then
            return False
        end if
        callee = caller; continue
    end if
    return False
end for
return True if the backtrace reaches a valid anchor and
    passes frame size check else False
```

F. Context Validation

The last module in our approach is the context validation module, which aims to check the validity of an obtained call path from the program runtime state snapshot. The call chain retrieved from the process snapshot module, along with the information received from the entry point identification module, call chain identification module, and system call monitor module, is used to perform the validation. The context is validated based on the call path and the stack size on the invocation of a system call. It is comprised of a path validation algorithm and stack size check mechanism. Note that if a call chain contains a function with an indeterministic stack size, it only needs to pass path validation. In general, the context validation is performed on a per-edge basis. Given the caller and the call of an identified edge in the call chain, the edge is

valid if we can find the callee from the callee set recognized by the call chain identification module. The entire call path is valid if every edge in the call path is valid and the sum of stack frame sizes matches the expected size.

The path validation algorithm is summarized in Algorithm 1. The algorithm starts from the top of the stack frame, the function that invokes the system call. We then check whether all the required information has been loaded in memory, including the base address of each involved executable and library, the required function list, and the return addresses for indirect calls. We must ensure that the last function being called resides in a valid segment in the process memory. When everything is ready, we start to backtrace all stack frame records one by one iteratively until all the frames have been validated. In each iteration, we have to ensure that all the functions on the stack frame are valid (lines 10–12), the callee is a valid one for the caller, and the return address of each callee is as expected (lines 13–18). If a broken link is recognized, we further verify whether the caller is invoking an indirect call and check the relation of the argument count of the caller and callee (lines 19–24). Once everything is done, we finally check if the backtraced path reaches the correct entry point and passes the *frame size check*, which recognizes the functions on the current call path and determines whether the total consumed stack size matches the expected size of the call chain.

We formalize the frame size check mechanism as a *subset-sum problem with digraph constraints* [52]. In the subset-sum problem, given a finite set $S \subset N$ and a target $t \in N$, we have to determine whether there is a subset $S' \subseteq S$ whose elements sum to t.

$$\text{subset-sum} = \{\langle S, t\rangle : \text{there exists a subset } S' \subseteq S \text{ st. } t = \textstyle\sum_{s \in S'} s\}. \quad (1)$$

The subset-sum with digraph constraints further extends the problem by adding more constraints. Given a directed graph $G = (S, A)$ and a non-negative weight $w(s)$ for every node $s \in S$. The weight of $S' \subseteq S$ must satisfy the following constraints.

$$\forall s \in S', (s, t) \in A \Rightarrow t \in S' \quad (2)$$

$$\sum_{s \in S'} w(s) = t \quad (3)$$

While the subset-sum problem is NP-complete [53], researchers have proved that the subset-sum problem with digraph constraints is an NP-hard problem. It means that given the function caller-callee relationships and the corresponding stack-frame-sizes, it is computationally difficult to find a valid set of functions to build a feasible attack path. If the caller-callee relationships are unavailable, the difficulty of valid path reconstruction falls back to the typical subset-sum problem, which is still an NP-complete problem. An attacker may analyze the target application offline based on our threat model. However, constructing a valid path also requires the four-tuple records of library functions deployed in the system, which could be unavailable to attackers. It could be even more challenging for attackers to bypass the context validation mechanism when the precise information of the libraries installed on the system is unknown.

TABLE II: Programs used for evaluations in this study.

Binary	Package	Version
`x264`	x264	2:0.163.3060+git5db6aa6
`tar`	tar	1.34+dfsg
`ls`	coreutils	8.32
`find`	findutils	4.8.0
`bzip2`	bzip2	1.0.8
`rg`	ripgrep	13.0.0
`openssl`	openssl	3.0.2
`diff`	diffutils	1:3.8
`ps`	procps	2:3.3.17
`gcc`	gcc	4:11.2.0
⟨Multiple⟩	SPEC CPU	2006
`nginx`	nginx	1.18.0
`sftp-server`	openssh-sftp-server	1:6.6p1
Live DVD	debian	live-11.6.0-amd64-gnome.iso
Live DVD	debian	live-11.6.0-amd64-kde.iso
Live DVD	manjaro	gnome-22.0.1-230124-linux61.iso
Live DVD	manjaro	kde-22.0.1-230124-linux61.iso
Live DVD	ubuntu	22.04.1-desktop-amd64.iso

It is worth noting that although the proposed approach reconstructs the context information by performing a return-address-based backtrace, it is not designed to tackle only the return-address-based control flow hijacking attacks. The context can be used to perform runtime state detection (based on caller-callee relationships) to determine whether a system call is valid. Therefore, the approach works even if an attacker manipulates a forward function call chain to launch attacks, e.g., by leveraging user-after-free, if any system call in the forward calling chain is invoked. Readers may refer to Section IV-F for more details.

IV. Evaluation

We evaluate our proposed approach in this section. We implement a proof-of-concept (PoC) implementation using standard features available in most Linux distributions. The system call monitor is implemented based on `ptrace` and seccomp `RET_TRACE` features. The stack unwinding feature is implemented by leveraging `libunwind`. We implemented the static analysis tool to build caller-callee relationships and the context validation part. Our implementation uses a single tracer to follow all `fork()` and thread-creating routines to ensure a target service can be fully protected. The entire code base contains about 3000+ lines of C++ and Python codes. Table II summarizes the targets we used for evaluation. Note that we only require SPEC CPU test cases to be compiled by ourselves. The rest of the binary files are downloaded from the package repositories of standard Linux distributions. No source codes are involved in the evaluations. We believe using binary files from publicly available repositories would be better for open research. All the experiments are performed on a server with an Intel i9-13900K CPU and 16GB RAM.

A. EH Frame and Backward Traceability

This section attempts to measure whether 1) EH frames have been widely deployed and cover the text segments for function boundary, and 2) The backtrace mechanism can always backtrace a program runtime state to its entry point. Therefore,

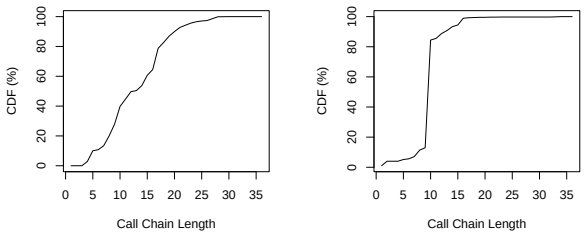
we perform a large-scale measurement against binary files available in Linux Live Desktop DVD ISO images listed in Table II.

To observe the deployment of EH Frame, we implement a scanner to scan ELF files stored in the `/usr/bin` directory and the directory stores `libc.so.6`. The scanner recognizes function symbols from the files and checks whether EH frame records cover the recognized functions. We boot each live DVD and run our scanner inside the live DVD runtime environment. We scanned about 16,353 executable and library files from the five DVDs and observed that about 2.5% (407) of the files had functions not covered by EH frames. It shows that EH frame information has been widely embedded into different flavors of Linux distributions. We further look into those functions not covered by EH frames. Many functions not covered by EH frames are constructor or deconstructor functions such as `_init`, `_fini`, `register_tm_clones`, `deregister_tm_clones`, `__do_global_dtors_aux`, and `frame_dummy`. One particular case is the OpenGL library, e.g., `libGL.so` and `libGLESv2.so`. If we look into the library files, we find that many of these functions not covered by EH frames are gadget-like implementations and irrelevant to stack operations, such as the `glTexCoord4s` function is `libGL.so`. In short, EH frames have been widely deployed in modern Linux distributions and can be effectively used for backtrace.

For backward traceability, we boot each live DVD and invoke all applications available on the desktop shortcut. Specifically, the evaluated desktop apps include browsers (Firefox and Konqueror), email apps (Evolution), file managers (Gnome- and KDE-builtin), music players (Rhythmbox), and office tools (LibreOffice). We implant our process snapshot module as a standalone executable to perform snapshots for all running processes in the live DVD. Finally, we validate if the backtrace can reach a valid entry point. All five live DVD environments contain a total of 543 running processes, and most of them can be correctly backtraced to the expected entry point. We observed that a few processes that cannot be correctly backtraced are implemented in the Go programming language, including `snap`, `snapd`, and `zsysd`, which is not currently supported by our entry point analysis module. Some non-standard entry points were also observed in our experiments. For example, the Konqueror (the Qt-based browser) uses `QtWebEngineCore::processMain` as the entry point for their child processes. These limitations can be handled by implementing additional plug-ins in our approach to recognize non-standard entry points.

B. Call Chain Path Length Statistics

Our context validation iterates through all functions in the call chain when a program invokes the system call to ensure path validity. To estimate the possible context validation costs, we measure the length of the call chain in this sub-subsection. We use the top ten selected programs listed in Table II for the evaluation. To prevent the selected program from terminating directly without doing anything, we use the corresponding commands listed in Table III to invoke the programs. The measured call chain length is presented in Figure 6. Figure 6a shows the cumulative distribution function (CDF) plot of call chain lengths for all system calls. The smoothly increasing curve in the plot indicates that the call chain lengths are evenly distributed. The longest call chain length is 36, which is observed from the `tar` binary. However, when a system call watch list is employed, it shows that more than 85% of the call chain lengths are less than or equal to 10, and more than 98% of the lengths are less than or equal to 16, as shown in Figure 6b. Our proposed context validation approach can be handled efficiently based on the observations. We also measure the performance of our context validation implementation. The processing time for validating a function call path requires about 1300ns for invoking the validation API, with an additional 26ns for validating a function call edge on average. Our proposed solution is performant, given that over 99% of the call chain paths are less than 30 in all cases.



(a) For all system calls (b) For selected system calls

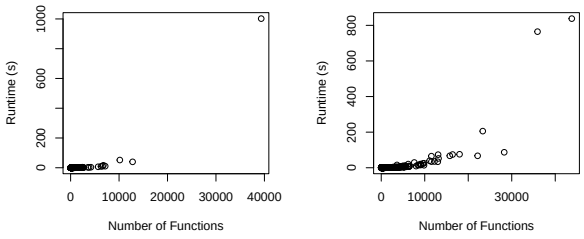
Fig. 6: Observations of the call chain length CDF.

TABLE III: List of commands to run the selected programs.

Binary	Command
x264	`x264 -o bbb.mp4 big_buck_bunny_720p_stereo.avi`
tar	`tar cf linux-5.15.90.tar linux-5.15.90/`
ls	`ls -l linux-5.15.90/`
find	`find linux-5.15.90/ -iname *.S`
bzip2	`bzip2 linux-5.15.90.tar`
rg	`rg -j1 ptrace linux-5.15.90/`
openssl	`openssl genrsa -out rsa.private 2048`
diff	`diff linux-5.15.90/arch/x86/entry/entry_64.S \ linux-5.15.90/arch/x86/entry/entry_32.S`
ps	`ps aux`
gcc	`gcc -S hello.c`

C. Performance Overhead

1) Static Analysis Performance: We also measure the overhead of static analysis using the Linux live DVD runtime environment. We run our call chain identification module against the binary files stored in the path `/usr/bin` and the shared library files stored in the same path as `libc.so.6`. Because the measurement results for different Live DVDs are similar, we only present the results obtained from Debian Live Gnome in Figure 7. The results show that the static analysis time depends mainly on the number of functions in an executable or library file. For executable files stored in `/usr/bin`, almost all of the analysis can be completed in 10 seconds, except for three outliers. These three outliers found in Figure 7a are `shotwell` (40s), `python3.9` (52s), and `lto-dump-10` (∼1000s). For the library files, 97.4% of



(a) `/usr/bin/*` (b) `/lib/x86-64-linux_gnu/*`

Fig. 7: Static analysis performance (Debian Live Gnome).

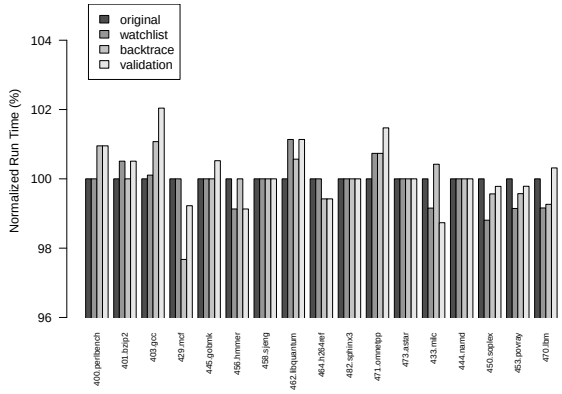


Fig. 8: Measured runtime performance.

the analysis can be finished in 10s, and 99.7% of the analysis can be finished within 90s. The three library files that take longer than 90s found in Figure 7b are `libmozjs-78.so` (205.79s), `libjavascriptcoregtk-4.0.so` (764.62s), and `libz3.so.4` (837.78s). Although statically analyzing the caller-callee relationships could be time-consuming, it can be done only once unless the analyzed binary files have been replaced or updated.

2) Runtime Performance and False Positive: In this subsection, we measure the runtime performance of our proposed approach and false positives. To break down each component's performance, we incrementally activate the modules and measure the execution time using the frequently used tools selected from the SPEC CPU 2006 test suite. The settings we used to measure the performance are summarized as follows.

1) **original (orig)**: Run the command directly.
2) **watchlist (nobt)**: Intercept system calls listed in the watch list but do not perform stack unwinding.
3) **backtrace (bt)**: (2), and performing stack unwinding.
4) **validation (va)**: Activate all the features.

Figure 8 shows the measured performance of tools selected from the SPEC CPU 2006 test suite. Note that our PoC implementation leverages standard features available in typical Linux distributions. Therefore, there would be many spaces for improving its performance. For example, `ptrace` may be replaced with other dynamic instrumentation tools. The `libunwind` implementation may be replaced with the one

TABLE IV: Performance measurement results for `nginx`.

Page size = 10 bytes				
	orig	nobt	bt	va
Requests/s (RPS)	24942.98	-2.02%	-0.36%	-2.94%
Transfer Rate (TR) (KB/s)	6065.24	-2.02%	-0.36%	-2.94%
RPS w/ tc	49.43	-0.04%	+0.04%	-0.22%
TR w/ tc	12.02	0%	0%	-0.25%
Page size = 1 kilobytes				
	orig	nobt	bt	va
Requests/s (RPS)	24193.96	-0.87%	-0.97%	-1.79%
Transfer Rate (TR) (KB/s)	29368.26	-0.87%	-0.97%	-1.79%
RPS w/ tc	49.41	+0.08%	-0.02%	+0.12%
TR w/ tc	59.98	+0.08%	-0.03%	+0.12%
Page size = 1 megabytes				
	orig	nobt	bt	va
Requests/s (RPS)	4410.85	-1.34%	-0.71%	+0.36%
Transfer Rate (TR) (KB/s)	4.31e6	-1.34%	-0.71%	+0.36%
RPS w/ tc	16.46	-0.36%	-0.06%	-0.61%
TR w/ tc	16080.12	-0.35%	-0.06%	-0.64%
Page size = 100 megabytes				
	orig	nobt	bt	va
Requests/s (RPS)	42.22	-0.07%	-1.02%	-0.40%
Transfer Rate (TR) (KB/s)	4.12e6	-0.08%	-1.01%	-0.41%
RPS w/ tc	1.78	0%	0%	0%
TR w/ tc	1.74e5	-0.02%	-0.05%	+0.04%

implemented by Théophile et al. [16]. The more features are activated, the more overhead can be observed. To focus on the performance of our proposed approach, we discuss more the cost incurred by context validation. The optimization of the involved standard components is out of the scope of this paper.

We selected 17 SPEC CPU tasks typically used in the other CFI works. SPEC CPU 2006 executes the selected tasks with predefined inputs sequentially. It runs for three rounds by default and reports the median to mitigate variances caused by different machine statuses. Figure 8 shows the overhead of the worst task 403.gcc is only 2%. The average overhead is less than 1%, which is much less than other works, including Bin CFI [21] (4.29%), ROPecker [30] (2.6%), PathArmor [31] (3%), BinCC [25] (22%), TypeArmor [22] (2.5%), PT-CFI [32] (21%), and τCFI [23] (2.89%). Our in-depth analysis shows that more than 90% of the overhead mainly comes from stack unwinding, indicating that the context validation implementation is quite efficient. As long as we can improve the efficiency of stack unwinding, the overhead can be further decreased.

We collect the false positives of our proposed approach when running the runtime performance evaluation. A *false positive* is defined as incorrectly determining a valid call path as invalid. We find no false positives from the test cases summarized in Table III and all the involved test programs from SPEC CPU 2006 test suite. Note that our implementation relies solely on CPU computational power, and we benchmarked its performance by executing 10,000 iterations of a 25-level recursive function call chain, correlating the measured execution time (*rt*) with the single-core event count (*evts*) from sysbench [54]. Testing across five hardware configurations, including four x86-64 machines and one ARM64 machine, showed a strong negative correlation (-0.97) between *rt* and *evts*, indicating consistent performance across architectures.

D. Performance on Real Server

To understand the performance impacts of our proposed approach, we benchmark the performance of a popular web server `nginx` secured by our approach. We use the Apache

TABLE V: Summary of ROP gadgets close to indirect call return addresses.

Distance (in Insn.)	Number of Gadgets	List of Controllable Registers
1	10 (4)	rax (3), rbx (4), rdx (3)
2	10 (5)	rax (2), rbx (4), rdx (2), rbp (2), r12 (1), r13 (1)
3	20 (7)	rax (1), rbx (5), rdx (2), rbp (5), r12 (1), r13 (1)
4	16 (6)	rbx (2), rdx (1), rbp (5), r12 (4), r13 (1)
5	15 (4)	rdx (1), rbp (2), r12 (2), r13 (1)
6	7 (3)	rbx (1), rbp (1), r12 (1), r13 (1)
7	13 (7)	rbx (5), rbp (6), r12 (3), r13 (2)
8	12 (3)	rbx (3), rbp (2)
9	10 (2)	rbx (2), rbp (1), r12 (1), r13 (1)
10	10 (0)	None

HyperText Transfer Protocol (HTTP) server benchmarking tool (`ab`) to perform the evaluation. The `nginx` server is configured to be a static web page server without enabling secured HTTP (HTTPS). We serve webpages of different sizes on the server and measure two performance metrics, the request per second (RPS) and the transfer rate (TR, in KBps), for different configurations. The benchmark is performed on the same machine via the loopback interface. To better emulate real-world network conditions, we consider adding a small value of latencies to the loopback interface using the `tc` tool. Durairajan et al. [55] measured the communication latencies between Network Time Protocol (NTP) clients and servers in the United States and reported that the latencies mostly fall in the range between 10ms and 40ms. Based on the measurement results, we only add 5ms latency to our loopback interface, which is much lower than the observed cases.

Table IV shows the measured results for the `nginx` server. The results show that the performance impact on the server is negligible. For both the RPS and the TR, the worst case is degradation by 2.94%. If network latency is configured on the loopback interface, the performance degrades at most 0.6%, which is almost equivalent to running the server without protections.

E. Minimizing Surfaces for Return Address-based Attacks

One strategy we used to determine a valid broken backtrace link is to validate whether the return address of a broken link returns to the address *right after* an indirect call. To validate how many attack surfaces can be eliminated based on the approach, we perform a measurement to understand the relationships between Return-Oriented Programming (ROP) gadgets and indirect calls.

We first collect ROP gadgets from `libc.so.6` from `glibc-2.35` using the default settings to run the command:

```
ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6.
```

There are 111,032 gadgets reports from the binary. Among all these gadgets, only the addresses of 38,822 (35%) gadgets are aligned to valid instruction address disassembly from `libc.so.6`. It means that the rest 65% of the ROP gadgets return to an unexpected address in a function, e.g., in the middle of a valid instruction, which could never be the address right after an expected indirect call. Of the 38,822 gadgets, only 231 are recognized from functions having indirect calls. We further measure the distance (in terms of the number of instructions) between a gadget and the closest indirect call. Only ten are right after an indirect jump, effectively minimizing the possible surfaces for attackers. The detailed statistics for the gadgets closest to indirect calls are summarized in Table V. For the case of the distance of one, i.e., the gadget right after an indirect call, there are ten gadgets, and four of them are capable of controlling some registers listed in the table. The number enclosed right after a register is how many gadgets can control the value of the register. The table shows that only a few limited registers can be controlled. Even if we look at all the 213 gadgets that might be reachable from an indirect call, only 68 of them are capable of controlling register values. The controllable registers are still limited only to `rax`, `rbx`, `rdx`, `r12`, and `r13`.

```
1  /* header files omitted */
2  typedef void (*fptr_t)(int s);
3  static fptr_t *ptr = NULL;
4
5  void shell() {
6    _exit(execlp("/bin/sh", "/bin/sh", NULL)); }
7  void handler(int s) { if(ptr!=NULL) (*ptr)(s); }
8  void good_bye(int s) {
9    fprintf(stderr, "Good Bye (%d)!\n", s); }
10
11 int main() {
12   char *msg = NULL;
13   if(ptr = (fptr_t *) malloc(sizeof(fptr_t))) {
14     *ptr = good_bye;
15     free(ptr);  /*** should not be freed ***/
16     signal(SIGINT, handler);
17   }
18   if(!(msg = (char *) malloc(17))) return -1;
19   if(read(0, msg, 16) < 0) return -1;
20   fprintf(stderr, "Press Ctrl-C to quit ...\n");
21   pause();
22   return 0;
23 }
```

Fig. 9: A case study with the use-after-free vulnerability.

F. Case Study: Use-After-Free Vulnerability

We use a case study involving a use-after-free (UAF) vulnerability to demonstrate that CASE can detect different flavors of attacks. Figure 9 presents source code containing a use-after-free vulnerability. Specifically, at line 15, the `free` function is improperly called, leading to the subsequent memory allocation function call at line 18 returning the same address pointed to by the function pointer stored in the global variable `ptr`. An attacker can exploit this behavior by supplying malicious input to manipulate the function call flow and execute the `shell`. Our proposed approach successfully mitigates this attack by defending against the invocation of the `execlp` call. This is achieved because the backtrace originating from within the `shell` function does not match any valid context created for the binary. Note that CASE successfully defends against the UAF attack because the prototype of the `shell` function differs from the type of the function pointer `fptr_t`. If the two prototypes were identical, the static analysis would consider `shell` a valid target, resulting in a possible evasion of the context validation. However, because the success of evasion depends on whether an attacker can find a useful gadget that has exactly the same number of arguments as a

writable function pointer, our proposed approach effectively reduces the attack surface by imposing stricter constraints on potential exploit paths for attackers.

G. Security Capability

In this section, we compare the capability of CASE against similar system call works and summarize the results in Table VI. We list the approaches mentioned in Section I and Section II. These approaches can primarily be categorized into debloating, system call filtering, and system call sandboxing. We further discuss the limitations of existing solutions, including whether they require access to the source code, the principle used to collect required information (static or dynamic analysis), runtime overhead, and protection granularity.

In the context of debloating research, exemplified by Nibbler [3], RAZOR [4], and μTrimmer [6], the identification of unused functions and instructions is performed at the binary level. These tools reduce the number of reachable syscalls at a coarser granularity, focusing on minimizing the attack surface rather than thoroughly validating their legitimate use. Although debloating can help reduce resource overhead by removing unnecessary system calls, they often focus on static code elimination. Thus, these methods do not incur additional runtime overhead. Also, even combined with CFI, debloating is incapable of preventing all code-reuse attacks.

System call filtering tools, such as sysfilter [7], Confine [8], sysverify [11], BASTION [38], and C2C [10], are used to deny specific system calls. These techniques generate the system call set via static analysis and then restrict the availability according to the list. Sysfilter resolves dependencies to dynamic shared libraries, constructs function-call graphs, and filters out unused syscalls. Confine identifies all the library functions that the program uses and the system calls that these library functions rely on. Both of them generate seccomp configuration solely based on whether the program will use these system calls. Therefore, we consider their granularity to be coarse. Sysverify checks if the indirect-call-related syscalls are really triggered by the secure path at runtime. BASTION not only blocks the unused system calls but also verifies the legitimate path to a system call and the arguments passed to the system call call-sites. C2C prunes configuration-dependent conditional branches in the control flow graph and allows precise basic block-level identification. These methods take into account more than just the usage of system calls, and as a result, we consider them as fine granularity approaches. Except for BASTION, permitted system calls can still be exploited illegitimately, resulting in a significant attack surface. While BASTION claims that it can defend against ROP and Counterfeit Object-Oriented Programming (COOP) attacks, it requires access to the source code, which can be challenging because commercial software is typically distributed without the source code. Compared to BASTION, CASE expands the protection to invisible library codes compared to source code-based solutions, minimizing possible attack surfaces led by control flow violations occurring in library calls. Furthermore, by incorporating both caller-callee relationships and stack size protection, CASE makes path reconstruction attacks computationally difficult for attackers.

Sandboxing methods such as SPEAKER [12] and Mining Sandbox [13] are designed to enhance container security by restricting access to OS resources. Mining Sandbox generates the set of system calls by automatic testing. SPEAKER removes unnecessary system calls from the running phase. These tools analyze docker images and block more system calls than the default sandbox. Nevertheless, the use of dynamic analysis may not be comprehensive and could lead to the omission of specific system calls during the preprocessing phase.

It is worth noting that system call-based approaches employ process-based enforcement for system calls, which is not sufficiently granular to secure scenarios such as the motivating example shown in Figure 1. While ROP is one of the major approaches for implementing exploitation and triggering system calls nowadays [56–58] due to the employment of emerging memory access protections, defending against ROP attacks would further improve overall security.

CASE performs stack unwinds to obtain the call path and validates both the sequence of the context and the sizes of the stack frames. Therefore, an attacker needs to fulfill the two conditions. First, they need to forge a stack with the same size as the original stack. Second, the function in the stack needs to pass the context validation. That is, the sequence of the function calls must be a valid record recognized in the preprocessing phase. We have shown that the problem could be more computationally difficult in Section III-F. Thus, CASE effectively enforces the system call usage at a finer granularity.

H. Real-World Vulnerable Server

To demonstrate that our approach can effectively protect a real-world server from a remote code injection attack, we evaluate it with the `sftp-server` available in the `openssh-6.6` package. Horn [59] disclosed the vulnerability that can be exploited if an `sftp` server is configured *without* enabling the `ChrootDirectory` option. Therefore, an `sftp` user can access `/proc/self/mem` and modify the stack content of the `sftp` process to execute any arbitrary instructions. For example, one return address on the stack can be set to the address of the `system()` function in the standard C library to execute arbitrary shell commands for attackers.

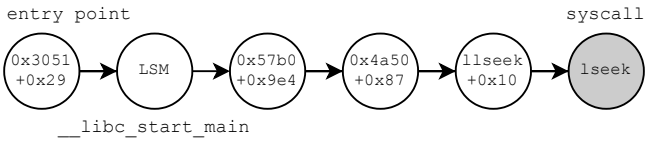
We reproduce the vulnerable environment by leveraging the root filesystem from the ubuntu-14.04-server cloud image, which has a preinstalled `openssh-6.6` package. We mount the root filesystem and then compile and install our PoC implementation to the root filesystem. We then 'chroot' into the root filesystem and launch the service. The exploit used to perform the attack is the PoC written by Simuntis and Slusnys and is available on ExploitDB [60]. The exploit first reads memory mappings to figure out the base address of `glibc` and the stack. It then edits `/proc/self/mem` to fill in the ROP payload on the stack, which redirects the control flow to execute the `system()` function and creates two files in `/tmp`.

The stack traces before and after the attack are depicted in Figure 10. Before the attack invokes a command, the last valid path we observed from the `lseek` system call is shown in Figure 10a, which attempts to move the file access position to a target address. In contrast, when the control flow

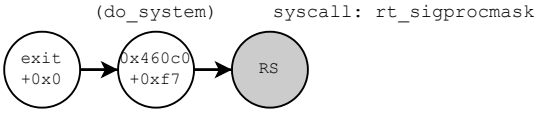
TABLE VI: Comparison of relevant system call protection approaches.

Type	Approach	Require Source	Information Gathering	Runtime Protection	Overhead	Granularity	ROP Protection
Filtering	CASE (our approach)	no	static analysis	ptrace+seccomp	$< 1\%$	fine-grained	yes
	sysfilter [7]	no	static analysis	static patch+seccomp	$< 1\%$	coarse-grained	no
	Confine [8]	no	static analysis	seccomp	–	coarse-grained	no
	sysverify [11]	yes	static analysis	syscall hook+seccomp	1%	fine-grained	no
	BASTION [38]	yes	static analysis	static instrumentation+seccomp	0.60%-2.01%	fine-grained	yes
	C2C [10]	yes	static analysis	static instrumentation+seccomp	1s-100s	fine-grained	no
Debloating	Nibbler [3]	no	static analysis[1]	–	–	coarse-grained	no
	RAZOR [4]	no	dynamic analysis[1]	–	1.7%[2]	coarse-grained	no
	μTrimmer [6]	no	static analysis[1]	–	–	coarse-grained	no
Sandboxing	Speaker [12]	no	dynamic analysis	seccomp	–	coarse-grained	no
	Mining Sandbox [13]	no	dynamic analysis	seccomp	0.6% - 2.14%	coarse-grained	no

[1]: Debloating determines the set of used functions rather than the set of system calls. Here, we discuss the methods they used to obtain the list of legitimate functions and instructions.
[2]:It is the average overhead to debloat programs, not the runtime overhead for RAZOR.



(a) The valid call path of the `lseek` system call used to change the file access offset of `/proc/self/mem`.



(b) The call path on the invocation of `rt_sigprocmask` system call from the `system()` function in the ROP payload.

Fig. 10: Call paths for benign and attacked system calls.

has been transferred to invoking the `system()` function, the `rt_sigprocmask` system call used in the `system` function triggers our context validation mechanism. The obtained call path is shown in Figure 10b, which cannot pass the context validation check, and the exploit is stopped from performing unauthorized operations.

V. Conclusion

We propose a practical approach that can prevent attackers from invoking inappropriate system calls to compromise a program. Our proposed approach is a software-only solution and prevents attacks by validating call paths snapshotted at our (or user) selected checkpoints. Several novel designs, including anchored entry points, return address-based validation, and frame size check, are employed in our design to harden the execution path of a protected program. We show that compromising our proposed approach is computationally difficult by formalizing CASE as creating NP-hard challenges for attackers. It can be used to defend against real-world code reuse attacks while the unwind and context validation operations incur less than 2% overhead in most cases. The performance evaluations show that our proposed approach is effective and efficient.

References

[1] MITRE D3FEND, "System call analysis - technique D3-SCA," Last accessed: 2023.03.01. [Online]. Available: https://d3fend.mitre.org/technique/d3f:SystemCallAnalysis/

[2] J. Salwan, "Shellcodes database for study cases," Last accessed: 2023.06.11. [Online]. Available: https://shell-storm.org/shellcode/index.html

[3] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating binary shared libraries," in *the 35th Annual Computer Security Applications Conference*, 2019, pp. 70—83.

[4] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "RAZOR: A framework for post-deployment software debloating," in *the 28th USENIX Security Symposium*, 2019.

[5] A. Quach, A. Prakash, and L. Yan, "Debloating software through Piece-Wise compilation and loading," in *the 27th USENIX Security Symposium*, 2018, pp. 869–886.

[6] H. Zhang, M. Ren, Y. Lei, and J. Ming, "One size does not fit all: Security hardening of MIPS embedded systems via static binary debloating for shared libraries," in *the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, p. 255–270.

[7] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "sysfilter: Automated system call filtering for commodity software," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses*, 2020, pp. 459–474.

[8] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses*, 2020, pp. 443–458.

[9] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *the 29th USENIX Security Symposium*, 2020, pp. 1749–1766.

[10] S. Ghavamnia, T. Palit, and M. Polychronakis, "C2c: Fine-grained configuration-driven system call filtering," in *ACM Conference on Computer and Communications Security*, 2022, pp. 1243—-1257.

[11] D. Zhan, Z. Yu, X. Yu, H. Zhang, and L. Ye, "Shrinking the kernel attack surface through static and dynamic syscall limitation," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1431–1443, 2023.

[12] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "Speaker: Split-phase execution of application containers," pp. 230–251, 2017.

[13] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining sandboxes for Linux containers," in *IEEE International Conference on Software Testing, Verification and Validation*, 2017, pp. 92–102.

[14] A. Bulekov, R. Jahanshahi, and M. Egele, "Saphire: Sandboxing PHP applications with tailored system call allowlists," in *the 30th USENIX Security Symposium*, 2021, pp. 2881–2898.

[15] N. Provos, "Improving host security with system call policies," in *the 12th USENIX Security Symposium*, 2003.

[16] T. Bastian, S. Kell, and F. Zappa Nardelli, "Reliable and fast dwarf-based stack unwinding," *the ACM on Programming Languages*, vol. 3, no. OOPSLA, 2019.

[17] H. Niu, Y. Xiao, X. Lei, L. Dan, W. Xiang, and C. Yuen, "Reconfigurable intelligent surface-assisted passive beamforming attack," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 8236–8247, 2024.

[18] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *the 12th ACM Conference on Computer and Communications Security*, 2005, pp. 340–353.

[19] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "Drop: Detecting return-oriented programming malicious code," in *Information Systems Security*, 2009, pp. 163–177.

[20] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: A detection tool to defend against return-oriented programming attacks," in *the 6th ACM Symposium on Information, Computer and Communications Security*, 2011, p. 40—51.

[21] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *the 22nd USENIX Security Symposium*, 2013, pp. 337–352.

[22] V. van der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 934–953.

[23] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert, "τ cfi: Type-assisted control flow integrity for x86-64 binaries," in *Research in Attacks, Intrusions, and Defenses*, 2018, pp. 423–444.

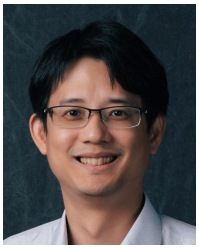
[24] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and

W. Zou, "Practical control flow integrity and randomization for binary executables," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 559–573.
[25] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng, "Binary code continent: Finer-grained control flow integrity for stripped binaries," in *the 31st Annual Computer Security Applications Conference*, 2015, p. 331—340.
[26] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity," in *Network and Distributed System Security Symposium*, 2015.
[27] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "Mocfi: A framework to mitigate control-flow attacks on smartphones," in *Network and Distributed System Security Symposium*, 2012.
[28] Y. Xia, Y. Liu, H. Chen, and B. Zang, "Cfimon: Detecting violation of control flow integrity using performance counters," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 2012, pp. 1–12.
[29] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing," in *the 22nd USENIX Conference on Security*, 2013, p. 447—462.
[30] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in *Network and Distributed System Security Symposium*, 2014.
[31] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive CFI," in *the 22nd ACM Conference on Computer and Communications Security*, 2015, p. 927—940.
[32] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace," in *the 7th ACM on Conference on Data and Application Security and Privacy*, 2017, p. 173—184.
[33] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *the 10th ACM Symposium on Information, Computer and Communications Security*, 2015, p. 555—566.
[34] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer integrity," in *the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 147–163.
[35] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "Aslr-guard: Stopping address space leakage for code reuse attacks," in *the 22nd ACM Conference on Computer and Communications Security*, 2015, pp. 280—-291.
[36] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff, "A sense of self for Unix processes," 1996, pp. 120–128.
[37] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman, "Protecting against unexpected system calls," in *the 14th USENIX Security Symposium*. USENIX Association, 2005.
[38] C. Jelesnianski, M. Ismail, Y. Jang, D. Williams, and C. Min, "Protect the system call, protect (most of) the world with bastion," 2023, pp. 528—-541.
[39] S. Alvarez, "Libre and portable reverse engineering framework," Last accessed: 2023.06.11. [Online]. Available: https://rada.re/n/
[40] M. Larabel, "The Linux kernel is now VLA-free: A win for security, less overhead & better for clang," October 2018. [Online]. Available: https://www.phoronix.com/news/Linux-Kills-The-VLA
[41] "Information technology – Programming languages – C," International Organization for Standardization, Standard, Dec. 2011.
[42] "MISRA C:2023," Motor Industry Software Reliability Association, Standard, Apr. 2023.
[43] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Computer Aided Verification*, 2011, pp. 463–469.
[44] National Security Agency, "GHIDRA: A software reverse engineering (sre) suite of tools developed by nsa's research directorate in support of the cybersecurity mission," September 2024. [Online]. Available: https://github.com/NationalSecurityAgency/ghidra/
[45] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, 2011, p. 9—16.
[46] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, p. 190—200, 2005.
[47] O. A. V. Ravnås, "Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers," Last accessed: 2023.06.11, https://frida.re/. [Online]. Available: https://frida.re/
[48] D. Bruening, V. Kiriansky, and T. Garnett, "DynamoRIO: Dynamic instrumentation tool platform," Last accessed: 2023.06.11, https://dynamorio.org/. [Online]. Available: https://dynamorio.org/
[49] K. Cook, W. Drewry, M. Kerrisk, T. Hicks, and T. Andersen, "seccomp(2) — Linux manual page," Last accessed: 2023.06.11. [Online]. Available: https://man7.org/linux/man-pages/man2/seccomp.2.html
[50] ——, "bpf(2) — Linux manual page," Last accessed: 2023.06.11. [Online]. Available: https://man7.org/linux/man-pages/man2/bpf.2.html
[51] N. A. Quynh and V. D. Hoang, "Unicorn engine," Last accessed: 2023.06.11. [Online]. Available: https://www.unicorn-engine.org/
[52] L. Gourvès, J. Monnot, and L. Tlilane, "Subset sum problems with digraph constraints," *Journal of Combinatorial Optimization*, vol. 36, pp. 937–964, 2018.
[53] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed., 2009.
[54] sysbench, "Scriptable database and system performance benchmark," April 2020. [Online]. Available: https://github.com/akopytov/sysbench
[55] R. Durairajan, S. K. Mani, J. Sommers, and P. Barford, "Time's forgotten: Using NTP to understand internet latency," in *the 14th ACM Workshop on Hot Topics in Networks*, 2015.
[56] A. Zaviyalov, "Defeating Windows DEP with a custom ROP chain," June 2023. [Online]. Available: https://www.nccgroup.com/us/research-blog/defeating-windows-dep-with-a-custom-rop-chain/
[57] S. Brizinov, N. Moshe, and T. Goldschmidt, "Pwn2Own: WAN-to-LAN exploit showcase, part 1," July 2024. [Online]. Available: https://claroty.com/team82/research/pwn2own-wan-to-lan-exploit-showcase
[58] B. Monie, "Exploiting a blind format string vulnerability in modern binaries: A case study from Pwn2Own Ireland 2024," October 2024. [Online]. Available: https://www.synacktiv.com/publications/exploiting-a-blind-format-string-vulnerability-in-modern-binaries-a-case-study-from
[59] J. Horn, "OpenSSH $\leq$ 6.6 SFTP misconfiguration exploit for 64bit Linux," 2014, Last accessed: 2023.06.11. [Online]. Available: https://seclists.org/fulldisclosure/2014/Oct/35
[60] A. Simuntis and M. Slusnys, "Openssh $\leq$ 6.6 sftp - command execution," Last accessed: 2023.06.11. [Online]. Available: https://www.exploit-db.com/exploits/45001

Man-Ni Hsu is a research assistant in the Department of Computer Science at National Yang Ming Chiao Tung University. She earned her B.S. degree in Computer Science and Information Engineering at National Central University in 2021 and her M.S. degree in Computer Science at National Yang Ming Chiao Tung University in 2023. Her research interests include control flow integrity, system security, and code reuse attacks.

Tsung-Han Liu is a research assistant in the Department of Computer Science at National Yang Ming Chiao Tung University. He earned his B.S. degree in Computer Science at National Yang Ming Chiao Tung University in 2023. His research interests include system programming, operating systems, and software engineering. He currently focuses on specification languages and verification for operating system development. Also, he was a Linux and FreeBSD system administrator at the IT center of the Computer Science Department in the same department, working on maintaining workstations and continuous integration for cross-platform projects.

Hsuan-Ying Lee is currently a graduate student at the Institute of Artificial Intelligence Innovation at National Yang Ming Chiao Tung University. She earned a B.S. degree at National Tsing Hua University. Her research interests include control flow integrity, system security and code reuse attack.



Chun-Ying Huang is a Professor at the Department of Computer Science, National Yang Ming Chiao Tung University. Dr. Huang leads the security and systems (SENSE) laboratory at National Yang Ming Chiao Tung University. His research interests include system security, multimedia networking, and mobile computing. Dr. Huang currently serves as the Institute of Network Engineering Director at National Yang Ming Chiao Tung University. He is a senior member of the Institute of Electrical and Electronics Engineers.