VIPER: Spotting Syscall-Guard Variables for Data-Only Attacks

Hengkai Ye, Song Liu, Zhechang Zhang, and
Hong Hu, *The Pennsylvania State University*

https://www.usenix.org/conference/usenixsecurity23/presentation/ye

This paper is included in the Proceedings of the
32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the
32nd USENIX Security Symposium
is sponsored by USENIX.

VIPER: Spotting Syscall-Guard Variables for Data-Only Attacks

Hengkai Ye Song Liu Zhechang Zhang Hong Hu

The Pennsylvania State University

Abstract

As control-flow protection techniques are widely deployed, it is difficult for attackers to modify control data, like function pointers, to hijack program control flow. Instead, data-only attacks corrupt security-critical non-control data (critical data), and can bypass all control-flow protections to revive severe attacks. Previous works have explored various methods to help construct or prevent data-only attacks. However, no solution can automatically detect program-specific critical data.

In this paper, we identify an important category of critical data, *syscall-guard variables*, and propose a set of solutions to automatically detect such variables in a scalable manner. Syscall-guard variables determine to invoke security-related system calls (syscalls), and altering them will allow attackers to request extra privileges from the operating system. We propose *branch force*, which intentionally flips every conditional branch during the execution and checks whether new security-related syscalls are invoked. If so, we conduct data-flow analysis to estimate the feasibility to flip such branches through common memory errors. We build a tool, VIPER, to implement our ideas. VIPER successfully detects 34 previously unknown syscall-guard variables from 13 programs. We build four new data-only attacks on sqlite and v8, which execute arbitrary command or delete arbitrary file. VIPER completes its analysis within five minutes for most programs, showing its practicality for spotting syscall-guard variables.

1 Introduction

Control-flow hijacking is the predominant approach to exploit memory errors in past decades [5, 61, 78]. Attackers deliberately modify control data, like function pointers, to divert program executions to malicious payloads. To prevent such attacks, researchers have developed and deployed many solutions [18, 30, 31, 53], like code pointer integrity (CPI) [48, 51] and control-flow integrity (CFI) [1, 8], to protect or validate control data. It is challenging to hijack control flow nowadays.

Among remaining hacking vectors, data-only attack shows great potential to be the next-generation exploit method [13]. Attackers just modify security-critical non-control data (*critical* data for short) to bypass control-flow protections and reach malicious goals. For example, modifying one variable `safemode` in Internet Explorer (IE) [87] enables attackers to execute arbitrary code remotely; stealing sensitive information such as password is also feasible by corrupting proper data [13, 39]. For attackers, such data are easy to modify using mature corruption techniques [23, 81] and countless memory errors [54, 83]; For defenders, protecting these bytes-long data can significantly reduce system attack surfaces. Therefore, critical data are becoming the main targets of memory attacks and defenses in the CFI/CPI era [14, 21, 57, 66, 67, 73, 75, 82].

However, no approach can automatically identify program-specific critical data, which makes large-scale attacks and comprehensive defenses debatable. Existing works usually spend tedious efforts to manually locate critical data, which can hardly support tremendous applications in a scalable way. For example, Chen *et al*. manually inspect the source code of vulnerable programs to label critical data [13]; To attack closed-source IE [87], security analyst has to investigate the code decompiled from binaries to identify the critical data `safemode`. Since attackers are eager to build data-only attacks to bypass control-flow defenses, it is urgent to develop automatic approaches to spot critical data for protection purposes.

Critical data are challenging to identify since they do not have common low-level properties, but are defined based on program-specific high-level semantics. Without concrete low-level properties, existing program analysis techniques can hardly identify them. For example, code pointers have distinct types (*i.e*., `PointerType` that holds `FunctionType` elements in LLVM IR), and we can detect them through type analysis [48]. However, critical data may have any type (*e.g*., `IntegerType` and `PointerType`) and could be stored in any memory location (*e.g*., stack or heap) [13, 39, 42]. This makes general analysis fail. Further, it is difficult and impractical yet to automatically infer high-level semantics for all program data, especially for large applications that contain thousands of variables [12, 35]. For example, given a four-byte variable in IE, we can hardly tell whether it stores critical security configurations,

represents merely an array index, or is used for other purposes. Most previous works take ad hoc methods to identify a few critical data only for demonstration purposes. It is unclear how many critical data are left unrevealed and unprotected.

In this paper, we aim to address the aforementioned challenges and achieve automatic scalable critical-data identification. We observe two common phenomena from previous data-only attacks [13, 39, 40, 42]. First, regardless of the program, most data-only attacks rely on security-related system calls (*syscall* for short) to achieve ultimate goals. Since data-only attacks have limited capability of manipulating control flow [42], they must utilize syscalls to request high-privileged operations or resources from the underlying operating system. For example, attackers must invoke syscall `execve` on Linux to create new processes. Second, security-related syscalls are commonly guarded by security checks, in the form of conditional branches. For example, IE checks several bits of `safemode` before silently executing untrusted code. We call such conditional branches *syscall-guard branches* and define checked variables as *syscall-guard variables*. Since syscall-guard variables determine to invoke security-related syscalls or not, their values have direct impact on the program security. This means that they should be considered as critical data.

Based on our observation, we present VIPER, a framework that automatically spots security-critical syscall-guard variables for data-only attacks and defenses. Our idea, *branch force*, flips every conditional branch triggered during normal executions, and monitors the execution to detect *interesting* syscalls. We define a syscall as interesting if it requests high-privileged resources or operations (*e.g.*, creating new processes) but is not invoked in the original execution. In this case, the high-privileged resources are obtained merely due to the flipped conditional branch. We treat such branches as candidates of syscall-guard branches. However, not all branches can be flipped equally. To identify corruptible syscall-guard branches, we perform dynamic data-flow analysis to identify syscall-guard variables, and estimate the feasibility of corrupting them through traditional memory errors. VIPER reports a set of syscall-guard variables and auxiliary information for each variable, including syscall-guard branches, triggered syscalls, triggering inputs and corruptibility assessment.

We design and implement VIPER as two components: `BranchForcer` and `VariableRator`. For each given input, `BranchForcer` runs the target program and records all triggered conditional branches and security-related syscalls. Then, `BranchForcer` runs the program again with the same input for multiple times. During each re-execution, it flips one triggered branch and monitors syscalls in the following execution. If the execution after the branch-flip invokes interesting syscalls, we treat this branch as a candidate of syscall-guard branches. To improve the test efficiency, we adopt the forkserver mechanism [90] from fuzzing techniques to reduce the overhead. In our experiments, we search online to find high-quality test cases, and meanwhile, utilize popular fuzzers [33, 50, 89] to cover more branches. However, generating high-quality inputs is out of the scope of this paper.

For each candidate, `VariableRator` constructs the data-flow graph for the branch condition. Based on the branch history, we reversely traverse the program, *i.e.*, starting from the branch instruction toward the program entry point. Along the traversal, we collect two attributes of each memory node for assessing the corruptibility, specifically, the memory location of the node (*e.g.*, stack or heap) and the number of memory-write instructions between a store and the following load. We use the former to estimate how easily the node can be corrupted, and take the latter to understand how likely the program contains a memory error in-between. We rate each branch based on all memory nodes along the data flow.

We apply VIPER on 20 real-world programs to evaluate its ability of detecting syscall-guard variables. These programs are common attacking targets or are well-tested in fuzzing works. VIPER successfully identifies 36 highly corruptible syscall-guard variables from 14 tested programs. By flipping one of these variables, the tested program will invoke new security-related syscalls, like `execve` that runs new code, `unlink` that deletes existing files and `chmod` that changes file permissions. `VariableRator` helps identify candidate branches that cannot be easily flipped, like constant branches and short-term variables. We check high-rank syscall-guard variables and confirm that common memory errors can corrupt them and trigger critical syscalls. We build four new data-only attacks by corrupting newly identified variables to achieve arbitrary command execution or arbitrary file deletion. The result shows that VIPER is practical to automatically identify syscall-guard variables for data-only attacks and defenses.

In summary, we make the following contributions.

- We identify an important category of program-specific security-critical non-control data, called syscall-guard variables. These variables are common targets of data-only attacks and should be protected properly.
- We design and implement, VIPER, a framework that can test a large number of programs in efficient ways to automatically pinpoint triggered syscall-guard variables.
- We apply VIPER on 20 real-world programs and detect 34 previously unknown syscall-guard variables. Altering these variables have significant security consequences, like arbitrary command execution and arbitrary file deletion. We build four end-to-end attacks on sqlite and v8.

Open Source. We will release VIPER source code and the details of detected critical data and constructed exploits at https://github.com/psu-security-universe/viper.

2 Background and Problem

```
1  void do_authentication(char *user, ...) {
2    int authenticated = 0;                          // non-control data
3    void (*authlog) (char *,...) = verbose;          // control data
4    ...
5    while (!authenticated) {
6     /* Get a packet from the client */
7     type = packet_read();                  // bug -> write primitive
8     ...
9     if (auth_password(user, password))
10      authenticated = 1;
11    ...
12    authlog(...);                                  // indirect call
13    if (authenticated) break;                       // check result
14   }
15   /* Perform session preparation. */
16   do_authenticated(pw);                            // open access
17 } +-> do_authenticated
18       +-> do_exec_pty
19           +-> do_child
20               +-> execve(shell, argv, env);          // syscall
```

Figure 1: Critical `authenticated` flag in openssh. Given wrong user name and password, `authenticated` remains 0 and the execution stays inside the loop. However, if attackers modify this variable to 1, they can login any account without correct password.

2.1 Data-Only Attack and Critical Data

Data-only attack is a general method to exploit memory errors. Given a memory error, like buffer overflow, attackers first manipulate program memory layouts [23,81] to construct memory-access primitives, like writing an arbitrary value to an arbitrary location [56]. The previously dominant control-flow hijacking attack will use the primitive to modify control data, like function pointers, and divert the execution to malicious payloads. As researchers develop solutions to protect control data [48,51] and verify their integrity [1,8], now it is challenging to hijack control flow. In contrast, a data-only attack will use the same primitive to modify *security-critical non-control* data for similar goals, like remote code execution. These attacks do not alter any control data, and thus can bypass current control-flow protections [1,48,51]. Data-only attack was first demonstrated feasible by Chen *et al*. in 2005 [13]. In recent years, it is receiving more attention from the security community [14,21,57,66,67,73,75,82,87] due to the wide adoption of control-flow protections [18,30,31,53].

Figure 1 shows our motivating example. The code is for openssh server to authenticate users. A loop at lines 5-14 retrieves user names and passwords from network and verifies their correctness (line 9). If the authentication succeeds, openssh saves the result in variable `authenticated`. It also records every login attempt through an indirect function call (line 12). Once the loop breaks, `do_authenticated` assumes the current user is authenticated and will allow her to access the server. However, old versions of openssh have an integer-overflow bug [55] that enables attackers to modify any memory location to any value in function `packet_read` (line 7). A control-flow hijacking attack can use the memory-write primitive to modify the control data `authlog` for ret2libc [61] or ROP attacks [78]. However, such attacks will be prevented by CFI that checks the validity of `authlog` or CPI that protects `authlog` from corruption. Instead, a data-only attack utilizes the same primitive to modify the non-control data `authenticated` to 1. This will force the loop to break (line 13), and attackers will obtain access to all user resources.

Security-critical non-control data, or *critical* data, are the empowering elements of data-only attacks. They are bound with particular security features of the program. Therefore, altering them will undermine the system security. These data usually have similar sizes as control data, like four or eight bytes. Therefore, attackers can reuse previous mature techniques [23,81] to corrupt them to revive old exploitations.

Several recent works demonstrate the feasibility of building data-only attacks without changing any critical data [10,40,42]. However, they must repeatedly trigger memory errors to modify many data, which is more challenging than altering critical data. For example, to chain enough basic gadgets for meaningful attacks [40], a data-oriented programming (DOP) attack sends over 700 network packets to `ProFTPD`. The huge number of modifications increase the risk of being detected [14,75], and exclude memory errors that can only be triggered once. Therefore, corrupting critical data is still the most convenient and powerful way to build data-only attacks.

2.2 Critical Data Identification

Critical data are prerequisites for data-only attacks, but it is nontrivial to identify them. First, critical data do not have common low-level attributes. For control data, code pointers have special types (*i.e.*, `PointerType` pointing to `FunctionType` in LLVM IR); return addresses are stored at particular locations (*i.e.*, bottom of stack frames). In contrast, critical data may have any type and could exist in any memory location. Second, data criticalness stems from their high-level semantics and security impacts, which are difficult to infer through program analysis [12,35]. For example, function `do_authentication` in Figure 1 has more than 10 local integer variables. It is hard to tell which one is more critical to openssh's security. The whole program may have hundreds to thousands of variables, which makes manual analysis impractical.

Due to these challenges, most previous works take ad hoc methods to identify critical data, but no one can automatically detect the `authenticated` variable in Figure 1. Chen *et al*. [13] manually identify critical data to demonstrate the feasibility of data-only attacks. They believe that "identifying security-critical non-control data and constructing corresponding attacks require sophisticated knowledge about program semantics". FLOWSTITCH [39] uses data-flow analysis to identify variables from configuration files and syscall arguments, where critical data come from explicit sources or are consumed in well-defined sinks. However, `authenticated` is not associated with any known source or sink, and attackers have to manually identify this critical data. Recent defense mechanisms [66,67,73] require users to annotate critical data. KENALI [79] utilizes error codes to help annotate critical data within Linux kernel. However, this method only works

Program	Critical Data	Ref.	Type	Semantics	Security Impact	Related Syscall
nginx	`clcf->root.data`	[39,57]	`u_char *`	root directory of web server	access any server file	A `send`
	`ctx`	[42]	`ngx_exec_ctx_t *`	execution context, like `argv`	execute arbitrary program	A `execve`
openssh	`authenticated`	[13,39]	`int`	authentication succeeds or not	login w/ wrong password	C `execve`
	`original_uid`	[39]	`uid_t`	numerical user ID	obtain root-user privilege	A `setuid`
sudo	`user_details.uid`	[39]	`uid_t`	numerical user ID	obtain root-user privilege	A `setuid`
null httpd	`config.server_cgi_dir`	[13,39]	`char [255]`	root directory of CGI binaries	execute arbitrary program	A `execve`
	`config.server_htdocs_dir`	[39]	`char [255]`	root directory of web server	access any server file	A `send`
ghttpd	`ptr`	[13,39]	`char *`	URL request from the client	execute arbitrary program	C `execve`
orzhttpd	`conn->basedir.path`	[39]	`char *`	root directory of web server	access any server file	A `sendfile`
wu-ftpd	`pw->pw_uid`	[13,39]	`uid_t`	numerical user ID	obtain root-user privilege	A `seteuid`
telnet	`loginprg`	[13]	`char *`	executable for authentication	execute arbitrary program	A `execve`
chromium	`m_universalAccess`	[44,75]	`bool`	whether to check SOP	disable same-origin check	C `open`
httpdx	`ftps.i["admin"].pass`	[39]	`char [255]`	password of administrator	admin login w/o password	C -
	`ftps.i["anon."].flags`	[39]	`int`	permission of anonymous	can detele file or directory	C `remove`,`rmdir`
	`ftps.i["anon."].root`	[39]	`char [255]`	root directory of anonymous	access any file on the server	A `open`, `send`
	`handlers[cgi].cmd`	[39]	`char [256]`	root directory of CGI binaries	execute arbitrary program	A `CreateProcess`
IE Browser	`safemode`	[87]	DWORD	security config of JS engine	execute arbitrary code	C `CreateProcess`

Table 1: Previous data-only attacks of privilege escalation. We study end-to-end attacks in previous works and list corrupted critical data based on our understanding. Same attacks are reported once. httpdx and IE run on Windows and others run on Linux. IE is closed-source.

for systems with well-defined error codes, and cannot handle diverse user-space applications. Further, not all error codes are directly related to the system security, and this method may introduce many false positives.

2.3 Our Focus: Syscall-Guard Variable

We investigate previous data-only attacks for privilege escalation [13,39,42,57,75,87] to understand how critical data affect the program security. We leave information leakage attacks to future work. Table 1 summarizes our findings. When details are unavailable, we try to find matched critical data based on our understanding of previous works. For each attack, we list the buggy program, the critical data, data type, data semantics, and the security impact. We only consider end-to-end attacks and exclude cases that merely build memory-access primitives (*e.g.*, several attacks in BOP [42]). We merge the same attacks used in multiple works and list them only once.

We observe that most attacks utilize security-related syscalls to achieve privilege escalation, as shown in Table 1. This is reasonable since data-only attacks must strictly conform to the legal control-flow graph and attackers can only reuse existing mechanisms for privilege escalation. For user-space programs, syscall is the most convenient method to prompt the privilege, like `execve` for creating new processes. Therefore, syscalls are commonly used in data-only attacks. The last column of Table 1 lists two common ways for critical data to affect syscalls. First, the data is used as one syscall argument ("A" in the table). For example, modifying the argument of `execve` enables attackers to run arbitrary program. Second, the data determines to invoke the syscall or not ("C" in the table). As an example, the bottom of Figure 1 shows the execution path from the corruption location (within `packet_read`) to the syscall `execve`. Otherwise, openssh will run inside the `while` loop and can never reach `execve`.

Since it is straightforward to identify syscall arguments through data-flow analysis [39], next we will focus on understanding and detecting critical data in the second category.

Definition 2.1 (Syscall-Guard Branch and Variable)**.** Since security-related syscalls significantly affect the program security, developers usually insert context checks before syscalls to avoid misuse. These checks are implemented as conditional branches, like `if` statements. We define branches that guard security-related syscalls as *syscall-guard branches*. Variables used in these branches determine to invoke security-related syscalls or not, and hence, are critical to the program security. We call them *syscall-guard variables*. Corrupting syscall-guard variables will change the target of syscall-guard branches, and finally affect the system security.

Syscall-guard variables are widely used in known data-only attacks. As shown in the last column of Table 1, except for syscall arguments (labeled as "A"), all six other critical data are syscall-guard variables (labeled as "C"). They guard invocations of critical syscalls such as `execve` on Linux and `CreateProcess` on Windows. Since syscall-guard variables are more challenging to identify than syscall arguments, it is not surprising that they are less frequently used than the later. However, these variables are critical to the program security. As researchers develop dedicated defenses to protect syscall arguments [43], the stealthy syscall-guard variables will likely receive more attention from attackers. We should spend more effort on identifying and protecting these critical data.

2.4 Challenges

We need to address two challenges to achieve automatic and scalable detection of syscall-guard branches and variables. First, to evaluate the criticalness of one variable, we need to figure out its *sole* contribution to triggering security-related

syscalls. Along the execution path, there are a large number of variables and branches. It is challenging to eliminate the impact of others and measure the sole contribution of a particular variable. For example, symbolic execution can identify a complete path from the program entry to a security-related syscall [39, 42]. However, it cannot tell which branch in the path is more critical than others. Attackers must treat all branches equally and find an input to satisfy all of them, which is more difficult than corrupting a syscall-guard variable.

Second, we need an efficient and scalable approach to handle real-world complicated applications. Manual analysis that requires huge human efforts is impractical, although most known critical data are identified manually. We also cannot afford program analyses that require heavy computing resources, like taint analysis [46, 62, 76] and symbolic execution [9, 16, 71, 74]. Static analysis can hardly satisfy our requirement due to its well-known limitations, like predicting indirect call targets and conducting inter-procedural analysis. For the example in Figure 1, the latest version of openssh extensively uses indirect function calls to dispatch various tasks; it also creates multiple processes to handle different functionalities, and relies on inter-process communication to achieve user authentication. In this case, we can hardly use static analysis to identify the syscall-guard variable `authenticated`.

3 Approach Overview

We design two steps to identify syscall-guard branches and variables. First, we propose the idea of *branch force* to collect candidates of syscall-guard branches (§3.1). Second, for each candidate, we use dynamic backward data-flow analysis to identify syscall-guard variables and measure the feasibility of corrupting them through common memory errors (§3.2).

3.1 Branch Force

Our insight stems from the definition of syscall-guard branches (Definition 2.1). Every syscall-guard branch may jump to two targets, the `true` target and the `false` target, but only one target will invoke the guarded syscall. If we change the branch condition from `true` to `false` or from `false` to `true`, we should observe that the syscall is invoked in one execution, not both. If both targets trigger the syscall, the branch is not qualified as a syscall guard; if both targets miss the syscall, then branch is not critical to the program security. For example, in Figure 1, with incorrect user name and password, variable `authenticated` remains 0. The branch at line 13 will jump to the `false` target, which goes back to the beginning of `while` and proceeds to another authentication attempt. If we change the condition to `true`, like through memory errors, the execution will break the loop and call function `do_authenticated`, which finally invokes syscall `execve`. Therefore, the branch at line 13 is a syscall guard, and `authenticated` is a syscall-guard variable.

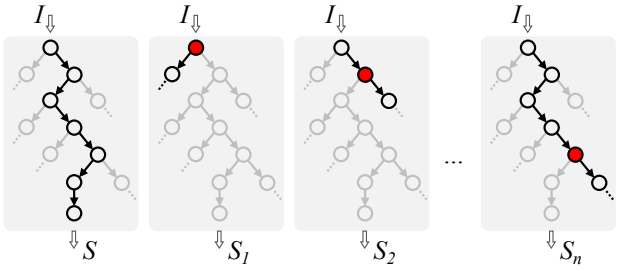

Figure 2: Demonstrate of branch force. We flip every executed branch in a brute-force manner. Hollow nodes and edges get executed, and gray ones are not. Red nodes indicate flipped branches. All executions share the same input, but may trigger different syscalls.

Our idea for identifying syscall-guard branches is to flip one and only one triggered conditional branch during the execution, and check whether the modified execution triggers new security-related syscalls. If so, we believe the flipped branch guards the new syscall, and is a candidate syscall guard; otherwise, we will check the next branch. Since we flip every branch in a brute-force manner, we call our method *branch force*. Figure 2 demonstrates how branch force works. Each node in the control-flow graph is a conditional branch, while each edge shows one possible branch target. Only hollow nodes and black edges get executed given the input I, and the gray ones are not executed. The red node in each graph is the only branch we flip during the execution with the same input. We collect the triggered syscalls for comparison purposes.

Branch force can address the two challenges of identifying syscall-guard branches discussed in §2.4. First, since we use the same input for two executions, the only difference is the target of the selected branch. Even if many branches are affected, all differences are rooted in our initial flip. Therefore, any newly invoked syscall must be caused by the branch flip. In this way, we eliminate the impact of other branches and obtain the sole contribution of the selected one on triggering new syscalls. Second, branch force takes lightweight instrumentation to dynamically alter branch targets and record execution contexts. We will discuss the implementation details in §4. Similar methods have been widely used in popular fuzzers to test diverse programs, like operating systems [17, 47, 65], program compilers [34, 69] and web browsers [24, 29, 77], and successfully detected a large number of vulnerabilities. We believe branch force can also handle tremendous programs.

3.2 Corruptibility Assessment

For each candidate of syscall-guard branches, we measure the feasibility of corrupting it through common memory errors for building data-only attacks. First, we need to find syscall-guard variables associated with candidate branches. Since attackers cannot directly flip a branch, they have to corrupt related condition variables to affect the branch. Second, not all branches are equally corruptible to attackers. Some could be easier, but others may require powerful memory-access

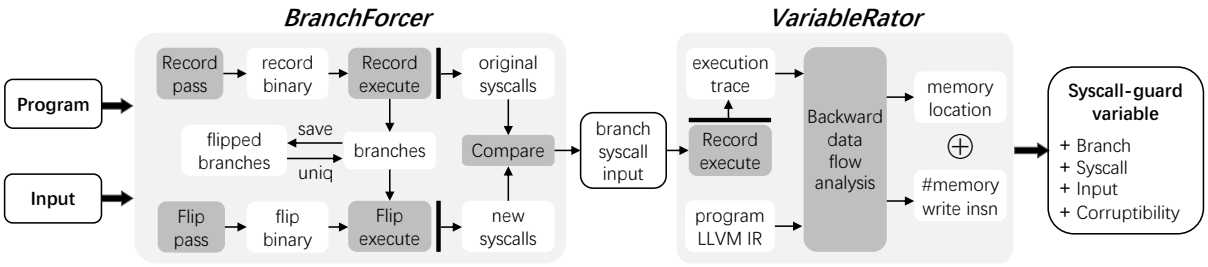

Figure 3: VIPER workflow. Given a program and its sample inputs, VIPER will report syscall-guard variables, together with related branches, syscalls, inputs and corruptibility assessment. VIPER has two components: `BranchForcer` uses branch force to identify candidates of syscall-guard branches; `VariableRator` locates syscall-guard variables and measures the feasibility of corruption through common memory errors.

primitives. In the worst cases, branches cannot be flipped at all if their conditions are completely out of attackers' control.

To address these problems, we first take dynamic data-flow analysis to identify all memory locations that contribute to the branch condition. Previous work [39] demonstrated that all memory nodes in the data flow could be corrupted by attackers for exploitation, so we should take all nodes into consideration. We adopt dynamic data-flow analysis since it provides concrete execution contexts, and is free from common challenges of static analysis, like pointer alias and indirect function calls.

Second, we collect two attributes of each memory node to assess the corruptibility, specifically, the memory location and the node lifespan. Memory locations have substantial impacts on data corruptibility since different locations may have unique defenses deployed [41, 48, 49]. For example, in non-PIE programs, heap and stack are randomized but global data are not [39]; stack canary is placed after stack buffers and cannot protect other regions [20]. We define three locations, stack, heap, and global, ranked from unlikely corruptible to highly corruptible. Further, variables with longer lifespan are more vulnerable to malicious modification [13, 39]. We take the number of memory-write instructions between variable defines and uses to calculate its lifespan. We observe that attackers must use memory-write primitives to corrupt variables. Without considering specific bugs, we assume every memory-write instruction could be abused by attackers for corruption with the same probability. More writes within the lifespan indicate higher likelihood of being corrupted [19, 28, 85].

Both data-only attackers and defenders can benefit from our corruptibility assessment. First, we can filter out incorruptible candidates. For example, if a branch compares `getuid()` with a constant, attackers cannot flip this branch since both operands are out of attackers' control. Second, we highlight candidates with high corruptibility. Attackers may corrupt these variables to quickly build exploits. Defenders can adopt the most promising (but could be heavy) protection on these variables. However, we should mention that our measurement is based on the common features of memory errors. Due to the diversity of vulnerabilities, the result could be inconsistent with some specific real-world exploitation scenarios. To exploit a concrete bug, attackers should consider more factors to find the optimal variable, like when the bug is triggered.

4 Design and Implementation

Figure 3 shows the workflow of VIPER, our framework for automatically detecting syscall-guard variables. Given the target program and a set of sample inputs, VIPER reports highly corruptible syscall-guard variables, with information such as flipped branches, triggered syscalls, triggering inputs and corruptibility assessment. VIPER has two main components: `BranchForcer` shortlists candidates of syscall-guard branches, and `VariableRator` measures the feasibility of corrupting syscall-guard variables. `BranchForcer` runs the program with each input, and records executed branches and syscalls. Then, it runs the program multiple times with the same input, and flips the target of one distinct branch for each re-execution. If the re-execution triggers new security-related syscalls, we add the current branch to our shortlist. For each candidate branch, `VariableRator` runs the program again, to record the complete trace of the execution, including the branch sequences, all memory-access addresses and the process memory layout. It conducts backward data-flow analysis on program IR with the help of the recorded trace. `VariableRator` checks the memory layout to identify the location of each memory node, and counts the number of memory-write instructions to estimate the node lifespan. We use the memory location and the node lifespan to represent the branch corruptibility.

4.1 Syscall-Guard Branch Identification

`BranchForcer` flips every conditional branch and checks whether the execution triggers new security-related syscalls.

4.1.1 Branch Recording

`BranchForcer` first instruments the program during compilation to general a binary for recording executed conditional branches, called record binary. There are multiple choices to record the branches. One way is to record all conditional branches in the order of execution. However, the method could record the same branch multiple times, which brings in two challenges. First, due to the huge number of branches, the execution will be slow and the trace files will be very large. Second, flipping a branch in this case emulates an attack that

```
1     ;----- [0] original conditional branch  ---------------
2     ; if (authenticated) break;
3
4     ;----- [1] LLVM IR of the original branch -------------
5     %11 = icmp ne i32 %10, 0, !dbg !50
6     br i1 %11, label %12, label %2, !llvm.loop !52
7
8     ;----- [2] recording unique branch  -------------------
9     %11 = icmp ne i32 %10, 0, !dbg !50
10  *  call void @record_br_uniq(i16 1122, i1 %11)
11    br i1 %11, label %12, label %2, !llvm.loop !52
12    ;
13    ; char uniq_branch_log[MAX] = {0};
14    ; void record_br_uniq(int branchID, bool value) {
15    ;   uniq_branch_log[branchID] |= ((value) ? 2 : 1);
16    ; }
17
18    ;----- [3] flipping one branch  -----------------------
19    %11 = icmp ne i32 %10, 0, !dbg !50
20  *  %new11 = call i1 @flip_br(i16 1122, i1 %11)
21  *  br i1 %new11, label %12, label %2, !llvm.loop !52
22    ;
23    ; int ID_to_flip = atoi(getenv("ID_TO_FLIP"));
24    ; bool flip_br(int branchID, bool value) {
25    ;   return (branchID == ID_to_flip) ? (!value) : value;
26    ; }
27
28    ;----- [4] recording all executed branches ------------
29    %11 = icmp ne i32 %10, 0, !dbg !50
30  *  call void @record_br_all(i16 1122, i1 %11)
31    br i1 %11, label %12, label %2, !llvm.loop !52
32    ;
33    ; FILE * branch_log_fd = fopen("...", "w");
34    ; void record_br_all(int branchID, bool value) {
35    ;   fputc((char) value, branch_log_fd);
36    ; }
```

Figure 4: Instrumenting target program for various purposes. `BranchForcer` performs all instrumentation during compilation.

only changes a branch one time among multiple dynamic uses. This assumes a very strong threat model where attackers have adequate ability to accurately control the flip. To avoid the slow execution, large trace and impractical threat model, we decide to record only unique executed branches.

In specific, we statically assign each conditional branch a unique identifier, called branch ID. Before each conditional branch, we insert an instruction to call a recording function. The function takes two arguments, a branch ID and a condition value. It creates a global array to record the branch, with the branch ID as the array index. We use `1` to represent a `false` condition and `2` to represent a `true` condition, and save the value to the proper element through bitwise-OR. In this way, we can tell whether the branch ever goes to the `true` target, the `false` target, or both along the execution. Our method is similar to block coverage collection in popular fuzzers [33, 50, 89], except that we also record condition values.

Figure 4 shows various instrumentation to the original code for different purposes. Line 2 shows the original conditional branch (line 13 in Figure 1). Line 5 and line 6 are the corresponding intermediate representation (IR) in LLVM, where `%10` represents `authenticated` and `%11` holds the result. Lines 9-11 demonstrate the instrumentation for unique branch recording, where we insert an instruction to call function `record_br_uniq` with the constant branch ID 1122 and the condition value as arguments. In lines 13-17, the function simply merges the condition value into the global array.

4.1.2 Branch Flipping

`BranchForcer` generates flip binary for flipping the selected branch. Similar to the instrumentation for recording, `BranchForcer` inserts an instruction to call another function, which takes a branch ID and a condition value as arguments. However, this function will check whether the branch ID is equivalent to the ID of the selected branch (retrieved from an environment variable). If so, it will return the negation of the condition value; otherwise, it will return the original value. We modify the branch instruction to use the return value as the new condition. Our current design only selects one branch, and flips all executions of that branch. We discuss multi-branch flip in §6. Lines 19-21 show the instrumented IR for branch flipping. Function `flip_br` is defined in lines 23-26. Return value `%new11` is used as the new condition, which is flipped if branch 1122 is the currently selected branch.

4.1.3 Designs for Efficiency

`BranchForcer` re-executes the program once for each executed branch, which may lead to a large number of re-executions. We adopt two mechanisms from fuzzing, unique branch flip and forkserver, to improve the system efficiency.

Unique Branch Flip. We identify branches that have never been flipped from previous experiments, and flip them to understand their criticalness. In other words, one branch will be flipped only once. However, one branch has two targets. As we do not know which target leads to security-related syscalls, we re-execute the program twice for each branch, one switching from `true` to `false` (if `true` is triggered), and another switching from `false` to `true` (if `false` is triggered). Since we log condition values, we can distinguish two flips.

Forkserver. In every re-execution, the program will process the same input using the same environment. To reduce the overhead, we will create a forkserver before the `main` function of the program. Every time when we need to re-execute the program, we just simply fork a new process. Based on previous fuzzing works, this method can help improve the throughput "by a factor of two or more" [90].

4.2 Syscall-Guard Variable Assessment

Given a candidate of syscall-guard branches, `VariableRator` locates syscall-guard variables through data-flow analysis, and evaluates their corruptibility. One way for data-flow analysis would focus on program binary [7, 80], as the runtime addresses match binary instructions. However, binary drops plenty of information, like variable types, making it hard to map variables to source lines. Another approach is to analyze source code or LLVM IR. However, it is nontrivial to connect runtime addresses to IR. Instead, we instrument the program so that the execution records IR-level branches and memory accesses, and we can conduct IR analysis with IR-level traces.

Algorithm 1: Path Construction

```
   Input  : module: program LLVM IR
            trace: dynamic execution trace
   Output : path: execution path
1  struct { insn; addr; } path[MAX];
2  pIdx ← 0
3  WalkFunc(module.getMainFunc())
4  Func WalkFunc(func)
5      block ← func.getEntryBlock ()
6      while block ≠ NULL do
7          block ← WalkBlock(block);
8  Func WalkBlock(block)
9      for instruction insn ∈ block do
10         idx ← pIdx;  pIdx ← pIdx + 1
11         path[idx].insn ← insn
12         switch insn.op do
13             case LOAD || STORE do
14                 path[idx].addr ← read8B(trace)
15             case direct CALL do
16                 WalkFunc(insn.getFunc())
17             case indirect CALL do
18                 target ← read8B(trace)
19                 WalkFunc(mapToFunc(target))
20             case unconditional BRANCH do
21                 ret insn.getBlock(0)
22             case conditional BRANCH do
23                 ret insn.getBlock(read1B(trace))
24             case SWITCH do
25                 ret insn.getBlock(read8B(trace))
26             case RETURN do ret NULL;
27             ...
```

Algorithm 2: Data-flow Analysis

```
   Input  : path: execution path (Algorithm 1)
            insn0: a candiate syscall-guard branch
   Output : dataflow: dataf flow of variable in insn
1  struct { defIdx; useIdx; } dataflow[MAX];
2  idx ← 0;   fIdx ← 0
3  while path[idx].insn ≠ insn0 do  idx ← idx+1;
4  S ← {tuple<insn0, NULL, idx>}
5  while idx ≥ 0 do
6      (i, a) ← path[idx];  ctuple ← tuple<i, a, idx>
7      for ptuple ∈ S do
8          (isDef, toDel) ← isDef(ctuple, ptuple)
9          if isDef = True
10             dataflow[fIdx] ← { ctuple.idx, ptuple.idx }
11             S ← S ∪ {ctuple};   fIdx ← fIdx + 1
12         if toDel = True  S ← S – {ptuple};
13     idx ← idx - 1;
14 Func isDef(c, p)
15     switch p.insn.op do
16         case LOAD do
17             if c.insn.op = STORE && c.addr = p.addr
18                 ret (True, True)
19         case STORE do
20             if c.insn.dstOprnd = p.insn.valueOprnd
21                 ret (True, True);
22         otherwise do
23             if c.insn.dstOprnd ∈ p.insn.srcOprnds
24                 if all p.insn.srcOprnds have defines
25                     ret (True, True)
26                 else ret (True, False)
27     ret (False, False)
```

Algorithm 3: Assessment

```
   Input  : path: execution path (Algorithm 1)
            dataflow: data flow (Algorithm 2)
   Output : assess: variable corruptibility
1  struct { glob; heap; stack; } assess;
2  for (defIdx, useIdx) ∈ dataflow do
3      (defInsn, defAddr) ← path[defIdx]
4      if defInsn.op = STORE
5          count ← 0
6          for idx ∈ range(defIdx, useIdx) do
7              if path[idx].insn.op = STORE
8                  count ← count + 1
9          if defAddr ∈ globals
10             assess.glob ← assess.glob + count
11         else if defAddr ∈ heap
12             assess.heap ← assess.heap + count
13         else if defAddr ∈ stack
14             assess.stack ← assess.stack + count
```

```
1  void output_reset(ShellState *p) {
2    if (p->doXdgOpen) {
3      char *zCmd = mprintf("xdg-open %s",
4                           p->zTempFile);
5      system(zCmd);      // invoke execve
6  }...}
7  void clearTempFile(ShellState *p){
8    if (p->zTempFile == 0) return;
9    // shellDeleteFile invokes unlink
10   if (shellDeleteFile(p->zTempFile))
11     return;
12 }
```

Figure 5: New syscall-guard variables in sqlite, guarding `execve` and `unlink`

4.2.1 Execution Trace Recording

We instrument the program IR to record execution contexts required for trace-based data-flow analysis, mainly including control flow and memory access. The control-flow information will help build the complete path on IR, including branch conditions, switch conditions and indirect call targets.

Branch Condition. In LLVM IR, a conditional branch is defined like `br i1 %1, label %2, label %3`. When %1 is `true`, the program jumps to basic block %2; otherwise, it jumps to basic block %3. When the execution reaches a conditional branch, we record the condition value. Different from the unique-branch recording in §4.1.1, now we record every condition in the execution order. Although this is slower than tracing unique branches, we only conduct this analysis on identified candidates of syscall-guard branches. Lines 29-31 in Figure 4 demonstrate the instrumentation. They are similar to unique-branch recording (lines 9-11), except that the function records all condition values sequentially.

Switch Condition. We record condition values of `switch` instructions. During the data-flow analysis, we use the condition value to determine the correct `case` or `default` to proceed.

Indirect Call Target. For a direct call, we get the called function from the instruction. However, for an indirect call, the target address is stored in a function pointer. We instrument the program IR to record all values of function pointers.

Memory Access. During backward data-flow analysis, when we encounter a load instruction, we need to find the related store instruction. Therefore, we record addresses and lengths of all memory access operations to connect load to store instructions. We also record addresses and lengths used in common memory access functions (*e.g.*, `memset` and `memcpy`).

4.2.2 Execution Path Construction

We restore the execution path on IR level, with the program IR and the execution trace. Algorithm 1 shows our algorithm. We use a flat array `path` to record all executed IR instructions and associated memory read/write addresses. The algorithm starts from the `main` function (line 3), and walks through all executed basic blocks. When reaching a new instruction, we add it into `path` (line 11). For load and store instructions, we retrieve memory addresses from the trace and save them into `path` (lines 14). For direct calls, we follow the call graph to step into the called function (line 16). For indirect calls, we utilize the recorded address to figure out the runtime target and step into it (lines 18-19). For unconditional branches, we get the successor from the instruction and proceed to process the successor (line 21). For conditional branches, we get the condition value from the trace, and use it to find the proper successor (line 23). We handle switch instructions in a similar way, except the condition has 8 bytes (line 25). For a return instruction, our algorithm will return to the callsite in the caller and continue the traversal (line 26). The algorithm will stop once it returns from `main` or we reach the end of the trace.

4.2.3 Backward Data-Flow Analysis

Based on the execution path, we conduct backward data-flow analysis to identify instructions that determine the condition of a syscall-guard branch. Algorithm 2 shows the algorithm. Given the branch instruction `insn0`, we scan array `path` to locate this instruction (line 3). `insn0`, together with its array index, forms the first element of set *S* (line 4), which contains all instructions queued for backward data-flow analysis. Next, we examine instructions in `path` in reverse order (lines 5-13). If one instruction *defines* one element in set *S*, we add one edge into the data flow `dataflow`, and add this instruction into *S*. The edge is directional, from the current instruction and to the element it defines (line 10). `isDef` checks whether instruction `c` defines `p`. When `p` is a load instruction, `c` defines `p` if `c` stores to the same address as that `p` loads from (lines 16-18). When `p` is a store instruction, `c` defines `p` if `c`'s output operand is `p`'s value operand (lines 19-21). Otherwise, if the output operand of `c` is the same as one source operand of `p`, `c` defines `p`. If we have found definitions for all source operands of `p` (lines 18, 21, 25), we should remove `p` from *S* (line 12).

4.2.4 Assessment Value Collection

With the variable data flow, we will collect metrics to represent the feasibility of flipping the guard branch. Algorithm 3 shows the algorithm. From the data flow we search for edges that start with a store instruction writing to address `defAddr` (line 4). Based on Algorithm 2, the other node of this edge will be a load instruction that retrieves value from the same memory address. Between these two instructions, if attackers can corrupt the content at address `defAddr`, the load instruction will retrieve the corrupted value for the program execution. To measure the likelihood of the corruption, we count the number of store instructions with this range (lines 6-8), assuming that every store instruction could be abused by attacker for memory corruption. Then we add this counter to different assessment aspects, based on whether `defAddr` points to globals, heap or stack (lines 9-14). Finally, the algorithm returns the counters as the assessment of the corruptibility.

4.3 Security-Related Syscalls & Test Cases

We need a list of security-related syscalls for identifying syscall-guard variables. After checking 313 syscalls on Linux and investigating related works [15, 22, 68, 84], we identify security-related ones shown in Table 4 (in Appendix). Attackers can modify the arguments of these syscalls to promote their privileges. For example, by manipulating the arguments of `execve`, they can run any program with any options. Based on the bug nature, attackers may customize the list to prioritize particular syscalls. For defense purposes, we consider all of them to seek maximum syscall-guard variables possible.

VIPER requires a set of test cases to trigger diverse program branches. Although obtaining high-quality test cases is out of the scope, we find multiple practical ways during our experiments. One method is to collect test cases from online resources, like project official test sets and third-party benchmarks. For example, sqlite [37], a popular DBMS, provides comprehensive test cases that cover almost all branches [36]. Another way is to generate new test cases through fuzzing. Fuzzing is a popular program-testing technique for detecting memory bugs [6, 89]. It uses code coverage to guide the generation of test cases, aiming to trigger more bugs. Researchers have made significant progress in fuzzing to achieve high coverage, which exactly matches our expectation. Moreover, many security research groups release high-coverage test cases obtained from long-time fuzzing [32, 64].

5 Evaluation

We apply VIPER on real-world programs to evaluate its effectiveness and efficiency in identifying syscall-guard variables.

Q1. Can VIPER identify new syscall-guard variables? (§5.1)

Q2. How does VIPER sieve branches and variables? (§5.2)

Q3. How effective is the corruptibility assessment? (§5.3)

Q4. Can VIPER handle diverse programs efficiently? (§5.4)

Target Programs. We select 20 programs for evaluation from three benchmarks. (1) FuzzBench [52], a widely adopted benchmark for evaluating fuzzing techniques. All programs in FuzzBench are popular targets of attacks. (2) Programs with known data-only attacks, as listed in Table 1. Although most known attacks corrupt syscall arguments, our goal is to detect syscall-guard variables. (3) To include more programs, we randomly select several programs from a large set [63]. For each program, we inspect the PLT section of the binary to understand whether it invokes any security-related syscalls or library calls. If so, we include it for evaluation. We leave Windows applications to future work. For Chromium, we test one of its core components, v8, the JavaScript engine. More details of tested programs are given in Table 5 (in Appendix).

Test Case Collection. Generating high-quality inputs is out of the scope of this paper. We try to use off-the-shelf test cases for detecting syscall-guard variables, as discussed in §4.3. For example, we adopt fuzzing results disclosed by FuzzBench team to test FuzzBench programs [32]. For programs that lack high-quality test cases, we use AFL++ [27] to fuzz each program for six hours and collect test cases that can trigger new branches. Table 5 (in Appendix) provides more details about how we collect test cases to evaluate programs.

Evaluation Environment. We conduct all of our experiments on a Ubuntu 20.04 system with two 28-core Intel(R) Xeon(R) Gold 6258R CPUs and 756GB memory.

5.1 New Syscall-Guard Variables

We identify 36 syscall-guard variables from 14 out of 20 tested programs, where 34 variables are previously unknown.

Program	Guard Variable	Branch Location	Syscall	Malicious Goal	Rate (S, H, G)	CK	CVE	Type	Cap
sqlite	mode	shell.c:5002	symlink	create symlinks to any file	(55, 0, 0)	◑			
		shell.c:5038	chmod	change any file to any mode	(75, 0, 0)	◑			
	p->doXdgOpen	shell.c:20270	execve	execute arbitrary program	(181770, 0, 0)	●	2017-6983	TC	AW
	p->zTempFile	shell.c:20560	unlink	delete any file	(86907, 0, 0)	●	2017-6983	TC	AW
	isDelete	sqlite3.c:42939	unlink	delete any file	(8353, 29276, 0)	●	2017-6983	TC	AW
	zPath	sqlite3.c:43094	unlink	delete any file	(57, 15036, 0)	◑			
	exists	sqlite3.c:60294	unlink	delete any file	(58, 15036, 0)	◑			
	isWal	sqlite3.c:58492	unlink	delete any file	(61, 15046, 0)	◑			
curl	tempstore	cookie.c:1732	rename	overwrite any file	(15, 0, 0)	◒	2019-3822	H/SBoF	AW
	tempstore	hsts.c:386	rename	overwrite any file	(15, 0, 0)	◒	2019-3822	H/SBoF	AW
	tempstore	altsvc.c:359	rename	overwrite any file	(15, 0, 0)	◒	2019-3822	H/SBoF	AW
harfbuzz	blob->mode	hb-blob.cc:453	mprotect	make RO memory writable	(31, 352, 0)	◒	2015-8947	HBoF	AW
nginx	sa_family	$_connection.c:631	chmod	change file mode	(0, 84831, 0)	◑			
	ngx_terminate	$_process_cycle.c:305	unlink	delete any file	(0, 0, 208640)	◒	2013-2028	SBoF	AW
	ngx_quit	$_process_cycle.c:305	unlink	delete any file	(0, 0, 208640)	◒	2013-2028	SBoF	AW
	ft.st_uid	($: ngx) $_file.c:631	chown	change owner of any file	(350832, 0, 0)	◑			
	ft.st_mode	$_file.c:640	chmod	change file mode	(175218, 0, 0)	◑			
openssh	result*	auth-passwd.c:128	execve	login without password	(5, 48153980, 0)	◑			
	received_sigterm	sshd.c:1163	unlink	delete any file	(0, 0, 1463147)	◑			
	received_sighup	sshd.c:1177	execve	execute arbitrary program	(0, 0, 1470603)	◑			
sudo	details->chroot	exec.c:173	chroot	change root path	(0, 0, 2039)	◒	2012-0809	FS	AW
	info	sudo.c:697	chdir	change directory path	(1702, 253382, 1982)	◒	2012-0809	FS	AW
null httpd	in_RequestURI	main.c:39	execve	enable CGI to run programs	(0, 525, 0)	◒	2002-1496	HBoF	AW
ghttpd	filename*	protocol.c:127	execve	enable CGI to run programs	(9, 0, 5912)	◒	2002-1904	SBoF	AW
wu-ftpd	RootDirectory	ftpd.c:1029	chroot	change root path of current user	(0, 0, 7322)	◑			
	anonymous	ftpd.c:2527	setgroups	obtain root privilege	(0, 0, 7432)	◑			
		ftpd.c:2893	chroot	change root path of anonymous	(0, 0, 8341)	◑			
	guest	ftpd.c:2893	chroot	change root path of guest	(0, 0, 37715)	◑			
	rval	ftpd.c:2708	setresuid	login without password	(8, 0, 0)	◑			
jhead	RegenThumbnail	jhead.c:978	execve	execute arbitrary program	(0, 0, 2856)	◒	2016-3822	IO	AW
	EditComment	jhead.c:1003	execve	edit any file using vi	(0, 0, 2856)	◒	2016-3822	IO	AW
	CommentInsertfileName	jhead.c:1003	execve	edit any file using vi	(0, 0, 2856)	◒	2016-3822	IO	AW
	CommentInsertLiteral	jhead.c:1003	execve	edit any file using vi	(0, 0, 2856)	◒	2016-3822	IO	AW
jasper	fileobj->flags	jas_stream.c:1392	unlink	delete any file	(0, 219062, 0)	◒	2020-27828	HBoF	AW
pdfalto	first	XRef.cc:240	unlink	delete files in specific folders	(1952, 214, 0)	◑			
	offsets[0]	XRef.cc:240	unlink	delete files in specific folders	(92, 117, 0)	◑			
gzip	fd	gzip.c:2111	unlink	delete any file	(0, 0, 11886)	◒	2010-0001	IO	AW
v8	enable_os_system	d8-posix.cc:762	execve	execute any program	(0, 0, 93512607)	●	2021-30632	TC	AW

●: exploits constructed with concrete bugs; ◒: exploitability checked with concrete bugs; ◑: primitives emulated with GDB. ● implies ◒, and ◒ implies ◑. Bug types: type confusion (TC); heap/stack buffer overflow (H/SBoF); format string bug (FS); integer overflow (IO). Capability: write anywhere any value (AW).

Table 2: Syscall-guard variables discovered by VIPER and corruptibility assessment. * means previously known variables.

Table 2 provides program names, variable names, source files, line numbers, guarded syscalls and high-level security impacts. 10 variables protect `execve` where corrupting them allows attackers to run arbitrary program. 15 variables guard `unlink` and `rename` and attackers can alter them to delete any accessible file. Four variables shield with file permission or owner syscalls `chmod` and `chown`, which can help attackers obtain extra file accesses. Five variables cover `chroot` or `chdir` that changes the root directory of current users, where attackers can abuse it to access more files. One variable secures `symlink`, and attackers can create any symlink to any file. This may allow attackers to replace executables or obtain accesses to extra files. Two variables are linked to `setgroups` or `setresuid`, where attackers may retain root privileges if they corrupt proper arguments. One variable takes care of `mprotect`, which attackers commonly abuse to create writable and executable pages. The sum of these numbers is over 36 as some variables (*i.e.*, `mode` in sqlite and `anonymous` in wu-ftpd) guard two security-related syscalls. Attackers can alter these variables in proper ways to invoke one or two syscalls. Three variables of jhead are checked in one line before invoking `execve`. Since different checks are combined through OR operation, attackers can modify any one to launch attacks.

Our evaluation covers eight out of 11 programs in Table 1. These eight programs have two known syscall-guard variables. VIPER successfully identifies both cases, specifically, `result` in openssh and `filename` in ghttpd, although variables have names different from previously attacked versions. We discuss the details of the difference of openssh in Appendix A.

Next, we provide case studies of two newly identified syscall-guard variables in sqlite. These variables guard different syscalls and we successfully build end-to-end exploits.

Case Study 1: Arbitrary Command Execution. VIPER reports a syscall-guard variable `p->doXdgOpen` that allows attackers to run arbitrary command in sqlite. Figure 5 shows the related function `output_reset`, where sqlite opens a tem-

porary file with an editor for users to modify the query result. In normal executions, `p->doXdgOpen` is `false` and sqlite will not invoke function `system`. If attackers flip the branch at line 2, sqlite will call `system` which internally invokes `execve`. Attackers can corrupt `p->zTempFile` to execute arbitrary program, like changing `zCmd` to `a.txt; cat ~/.ssh/id_rsa` to retrieve the private key. Pointer `p` points to a stack `ShellState` object that maintains database connection states and contains two members `doXdgOpen` and `zTempFile`. sqlite initializes this object at the early stage of the execution, and only invokes `output_reset` before exit. Given an empty query ".exit", `VariableRator` reports 181,770 memory-write instructions in the lifespan of `p->doXdgOpen` and 86,907 memory-write instructions in the lifespan of `p->zTempFile`.

We build an end-to-end exploit that allows attackers to run arbitrary command, based on the public tutorial [26]. CVE-2017-6983 is a type-confusion bug in sqlite, where attackers can fully control a pointer of an `Fts3Cursor` object. This bug has been fixed in version 3.40.1. We modify three lines in function `fts3FunctionArg` to reactivate it. Following the tutorial, we turn the bug into an arbitrary-memory-write primitive. We use the primitive to modify `doXdgOpen` and `zTempFile` for arbitrary command execution. We disable ASLR during exploit construction, as we can bypass ASLR easily through other bugs, like CVE-2017-6991 used in the tutorial.

Case Study 2: Arbitrary File Deletion. VIPER reports that altering `p->zTempFile` can trigger arbitrary file deletion. Specifically, sqlite uses temporary files in many ways. As shown in function `clearTempFile` of Figure 5, sqlite deletes any existing temporary files before creating a new one. If `p->zTempFile` is `NULL` (line 8), it will return immediately since no temporary file exists yet. Otherwise, it will call `shellDeleteFile` which internally calls `unlink` to delete existing files. An attacker can corrupt `p->zTempFile` to delete any sqlite-accessible file. We exploit the same bug as the previous one to achieve arbitrary file deletion.

5.2 Sieving Branches and Variables

To understand how VIPER identifies syscall-guard variables, we count the number of branches evaluated in different stages. Table 3 shows our statistics, including numbers of branches in each program (T), flipped by `BranchForcer` (F), whose flips invoke new syscalls (C), highly corruptible (A) and that confirmed by manual analysis (M). Due to different methods of test-case collection (shown in Table 5), each program is tested with one to 35,757 inputs, triggering different branch coverage from 4.0% to 71.4%. VIPER may detect more syscall-guard variables if we can improve the branch coverage.

`BranchForcer`. Results in columns F and C show that `BranchForcer` can filter out a large number of branches as they do not invoke new syscalls. After this step, most programs only have less than 20 branches left, except sqlite (21 branches) and sudo (26 branches). For five programs, `BranchForcer` does not find any branch that triggers new syscalls, caused by two reasons. First, security-related syscalls are not triggered by test cases even with flipping. For example, with 11,978 test cases, freetype2 does not invoke any syscalls listed in Table 4 (in Appendix). Adding diverse test cases [72, 88] could help detect more syscall-guard variables. Second, security-related syscalls are invoked in given test cases, and flipping branches does not trigger new ones. For example, telnet invokes `execve` with one test case, and flipping one branch will stop invoking this syscall. We can cover this example by considering flips that disable previously invoked syscalls. However, this could introduce false positives as any crash or timeout will stop invoking previously executed syscalls, even if the condition is not related to the syscall.

`VariableRator`. Results in column C and column A show that `VariableRator` removes incorruptible branches and reduces the number of candidates lower than 10 for all except two programs. `VariableRator` detects incorruptible branches in two cases. First, all memory nodes in the data-flow have very short lifespan, and it is hard for attackers to corrupt such variables. For example, for variable `m` at line 37364 of `sqlite3.c` in sqlite, there are only five memory-write instructions during its lifespan. Second, the new syscall is invoked through library functions, but attackers can hardly affect syscall arguments. For example, library function `getpwnam` internally invokes syscall `mprotect` with hard-coded constant arguments, where attackers have no way to modify these arguments. `VariableRator` currently does not support multi-threaded programs and fails to analyze systemd.

5.3 Exploitability Investigation

The goal of corruptibility assessment in VIPER is to estimate the likelihood of corrupting identified syscall-guard variables by common memory errors. To understand the usefulness of our estimation, we manually analyze the results through GDB emulation, CVE investigation and exploit construction.

GDB Emulation. Following the previous practice [42], we use the debugger GDB to emulate arbitrary memory-write primitives and check whether each variable guards security-related exploitable syscalls. Specifically, We launch the program with GDB, modify the syscall-guard variable within its lifespan, alter the syscall arguments and hook security-related syscalls. Column M of Table 3 shows the emulation results. Most branches and variables reported by VIPER (column A) are confirmed to guard likely exploitable security-related syscalls. We check excluded cases and identify three common reasons. ① The syscall has limited security impact, like `unlink(strcat(tmpFile, ".tmp"))` in sqlite only deletes files ended with ".tmp". ② The program enforces strong constraints on variables. For example, sqlite checks file size and only changes the mode of empty files. ③ Corrupting variables has negative side effects. For example, `jrnlOpen` in sqlite affects two branches, while triggering syscall `unlink`

Program	Version	kLoC	#Test	Branches/Variables						Time Cost					Stitch	
				T	F	F/T	C	A	M	Record	Flip	Rate	Total	Total/A		
sqlite	3.40.1	273	35,757	20,653	14,756	71.4%	21	9	7	288"	112"	378"	778"	87"		
curl	97f7f66	160	4,286	37,251	2,645	7.1%	13	3	3	23"	32"	689"	744"	248"		
harfbuzz	1.3.2	41	10,816	3,878	2,656	68.5%	1	1	1	17"	8"	8"	33"	33"		
systemd	v252	543	2,851	28,739	2,093	7.3%	5	✗	✗	69"	40"	-	>109"	>109"		
mbedtls	10ada35	128	1,078	8,754	1,528	17.5%	0	0	0	2"	6"	-	>8"	>8"		
openssl	3.0.7	483	2,515	32,063	5,448	17.0%	0	0	0	13"	61"	-	>74"	>74"		
freetype2	cd02d35	119	11,978	10,224	4,159	40.7%	0	0	0	18"	26"	-	>44"	>44"		
nginx	1.20.2	141	1,219	12,483	2,408	19.3%	7	5	5	238"	22"	329"	589"	118"	121"	
openssh	36b00d3	119	1	11,060	2,316	20.9%	4	3	3	1"	4722"	10624"	15347"	5116"	1110"	
sudo	1.9.9	110	329	6,894	2,411	35.0%	26	17	2	16"	16"	260"	292"	18"	393"	
null httpd	0.5.1	2	1	224	106	47.3%	1	1	1	1"	10"	31"	42"	42"	358"	
ghttpd	1.4.4	1	1	111	43	38.7%	2	2	1	1"	36"	72"	109"	55"	48"	
orzhttpd	0.0.6	3	1	191	100	52.4%	0	0	0	1"	32"	-	>33"	>33"	93"	
wu-ftpd	2.6.2	18	2	2,918	399	13.7%	9	8	4	1"	533"	189"	723"	91"	200"	
telnet	3f35287	11	1	420	147	35.0%	0	0	0	1"	144"	-	>145"	>145"		
jhead	3.04	4	72	616	287	46.6%	12	12	4	1"	2"	288"	291"	25"		
jasper	4.0.0	34	3,237	4,949	1,777	35.9%	4	1	1	37"	16"	84"	137"	137"		
pdfalto	0.4	76	16,954	14,069	4,581	32.6%	5	2	2	342"	116"	107"	565"	282"		
gzip	1.12	6	1,558	1,133	374	33.0%	1	1	1	6"	1"	19"	26"	26"		
v8	8.5.188	3,586	1	170,232	6,819	4.0%	2	1	1	1"	5833"	874"	6708"	6708"		

Table 3: Branches and time costs. We count the numbers of branches in the program (T), flipped by `BranchForcer` (F), triggering new syscalls (C), highly corruptible (A) and confirmed by us (M). We measure the time required for recording all test cases (Record), flipping all triggered branches (Flip) and assessing all candidates (Rate). ✗: pthread not supported. The last column lists the numbers reported in FLOWSTITCH.

requires changing one and only one of them. It is almost infeasible to exploit this variable since attackers must corrupt the same variable twice within a short time window.

CVE Investigation. To measure the feasibility of corrupting identified variables through *concrete* bugs, we conduct investigation using historical vulnerabilities [40,42]. For each program, we select one representative CVE and manually study the buggy function and attacker's capability. Then, we execute the program to check whether the buggy function can be triggered during the lifespan of each variable and whether the attacker's capability may cover the variable location. These two criteria are used by FLOWSTITCH [39] to automatically build data-only attacks. If we get positive answers to both, attackers possess a significant opportunity to corrupt the variable for attacks. Column CK of Table 2 shows our investigation results (◒ or ●). In total, we investigate 14 historical CVEs (all fixed in the latest version), one for each program. These CVEs have different types, like type confusion and buffer overflow, and most have been used to build control-flow or data-only attacks. We find that 20 identified variables from 11 programs are highly corruptible through these CVEs. For the other 16 variables, we either cannot find a path to trigger the selected bug within the variable lifespan, or the buggy code has been moved to a separate process. However, this does not mean these variables are completely safe since we only spend limited time to check one CVE for each program. Attackers may use other bugs or abuse different execution paths to corrupt these variables for attacks.

Exploit Construction. We build four new end-to-end data-only attacks, using two historical CVEs to corrupt four newly identified syscall-guard variables from two programs, indicated as ● in Table 2. We choose sqlite and v8 for exploit generation, where sqlite "is the most used database engine in the world" [37], while v8 is the JavaScript engine used in Chromium-based web browsers, like Chrome, Microsoft Edge and Opera. Three attacks against sqlite enable attackers to run arbitrary command (see Cast Study 1) or delete any files (see Case Study 2). The exploit on v8 allows malicious JavaScript code to run arbitrary command (Appendix B). We will release the details of these exploits on GitHub, together with the VIPER source code and identified syscall-guard variables.

5.4 Time Cost of VIPER

We measure the time cost of VIPER of finding syscall-guard variables. Table 3 shows times for testing all inputs and recording branches, flipping all triggered branches, and assessing the corruptibility of all candidates. We calculate the total time and the average time for each highly corruptible variable.

In general, VIPER analyzes programs with diverse costs. For all programs but openssh and v8, VIPER completes analyzing all test cases within 13 minutes, and on average, identifies one highly corruptible variable in five minutes. Testing openssh requires the most cost, which takes about 4.3 hours to complete. We identify two reasons of the slow performance. First, openssh server takes a long time to initialize. We have to set a large timeout threshold, two seconds, such that openssh can reach core functionalities. Second, flipping branches in openssh, introduces a lot of timeouts. One way to reduce the cost is to move our forkserver after the initialization process.

To understand the performance of VIPER during exploit generation, we check the cost of FLOWSTITCH, a dedicated

tool for building data-only attacks [39]. FLOWSTITCH was evaluated on eight programs, including seven on Linux and one on Windows. We apply VIPER on all Linux programs. The gray region of Table 3 compares the results of VIPER and the numbers reported in [39]. Since VIPER and FLOWSTITCH work on different tasks, it does not make much sense to compare absolute numbers. Instead, the time required for VIPER to identify a syscall-guard variable is within the same or smaller order of magnitude as that for FLOWSTITCH to construct an attack. Therefore, it is practical to combine VIPER and FLOWSTITCH to generate data-only attacks.

6 Discussion

In this section, we review several design issues of VIPER and discuss future directions for exploration and improvement.

Multi-Branch Flipping. Our current design flips a single conditional branch, and therefore, VIPER will miss syscalls guarded by multiple checks. We can extend VIPER to flip multiple branches to cover sophisticated checks. However, we find nontrivial challenges in multi-branch flipping. First, flipping one branch introduces negative inconsistency to the program state, which may lead to crash or hang. Flipping multiple branches will bring more inconsistency, and may cause more crashes and timeouts before reaching new syscalls. Second, for exploit generation, attackers must corrupt more variables to flip multiple branches, which is more difficult than single-branch flipping. Therefore, a simple extension will not work well. We may need to reduce results of multi-branch flip to single-branch flip to identify useful variables.

Multi-Threaded Application. Current version of VIPER does not support multi-threaded applications. In our evaluation, we limit the thread number to one so that VIPER can record traces and conduct data-flow analysis. To support multi-threaded programs, we need to extend `VariableRator` to (1) save traces of different threads into separate files; (2) hook and record synchronization primitives (*e.g.*, locks) to restore execution orders; (3) conduct data-flow analysis across different threads. We leave this extension to the future work.

Comparison with Data-Oriented Programming. Data-oriented programming (DOP) builds data-only attacks without relying on any critical data [40], while VIPER aims to automatically identify critical data for data-only attacks and defenses. These two works target different types of data-only attacks. VIPER can identify available critical data to simplify the exploitation, but the program does not always contain security-critical data; DOP can build exploits without critical data, but the process of exploit construction could be tedious and challenging. An optimal strategy for exploit generation could use VIPER to identify corruptible critical data as much as possible, and use DOP to connect remaining gaps.

7 Related work

Data-Only Attacks. Chen *et al*. propose non-control data attacks (*i.e*., data-only attacks) and demonstrate the feasibility of this new threat [13]. Follow-up works port this attack to diverse programs and systems [14], like kernels [4, 21, 86], browsers [44, 75, 87] and PDF readers [25]. Data-only attacks are also used to disable defenses. For example, manipulating kernel data can disable Linux auditing, AppArmor, ASLR and NULL-deference defense [86]. Corrupting browser variables can turn off the same-origin policy (SOP) [44, 75]. Altering metadata of shared libraries allows attackers to modify arbitrary code [40]. FLOWSTITCH [39] automatically connects disjoint data-flows for building data-only attacks. Recent works propose to build data-only attacks without corrupting critical data. Data-oriented programming (DOP) chains basic operations, like memory read and addition, to orchestrate expressive (even Turning-complete) attacks [40]. Block-oriented programming (BOP) automatically finds feasible paths from program entry to useful sinks such as memory accesses and syscalls [42]. Printf-oriented programming (POP) enables data-only Turing-complete attack via self-modified format strings [10]. However, these attacks require a lot of memory corruption, like over 700 packets to attack ProFTPD via DOP [40]. Such heavy memory corruption may attract attention from intrusion detection systems and anti-virus tools.

Critical-Data Identification. Most previous works take manual analysis to temporarily identify a few critical data for quick demonstration. Researchers manually check the program source [13] or even binary [87] to find useful variables. Jia *et al*. adopt debuggers, like GDB, to help identify differences between privileged and non-privileged executions [44]. However, manual analysis cannot afford large programs that contain thousands of variables [86]. For critical data coming from known sources (*e.g*., network socket) or used by well-defined sinks such as syscalls, previous works perform trace-based data-flow analysis to identify them [10, 39]. But they cannot identify critical data without well-recognized sources or sinks. Several projects build models to describe how to find particular critical data in memory [57, 75]. These solutions require predefined targets in order to train the model offline, and cannot report critical data with unknown patterns.

Defenses against Data-Only Attacks. The most comprehensive protection is to enforce data-flow integrity [11, 79] or full memory safety [2, 45, 58–60] to prevent memory errors in the first place. However, these solutions usually have unacceptable overhead and can hardly be adopted in production environment. A lightweight solution can identify data invariants through manual specification [38, 70], offline training [3] or program analysis [86], and enforce their integrity at runtime. However, they cannot protect data that are frequently updated. Recent works propose to adopt encryption, virtual machine and combine dynamic and static analysis to protect high-value critical data, and leave other less-critical

data unprotected [66, 67, 73]. Our work complements these selective protections by automatically identifying a list of program-specific security-critical non-control data.

8 Conclusion

In this paper, we propose branch force to automatically detect program-specific syscall-guard variables. Such variables can be modified by attackers to build data-only attacks. We design and implement VIPER, the first tool that can automatically and efficiently detect syscall-guard variables. With VIPER, we successfully detect 34 new syscall-guard variables from 13 programs. We build four new end-to-end attacks that enable arbitrary command execution or arbitrary file deletion.

Acknowledgment

We thank the anonymous reviewers, especially our shepherd, for their insightful comments and helpful feedback. This research was supported, in part, by an ICDS seed grant and an IST seed grant from the Pennsylvania State University.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, pages 340–353, Alexandria, VA, November 2005.

[2] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994.

[3] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Automatic Inference and Enforcement of Kernel Data Structure Invariants. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC)*, pages 77–86, Anaheim, CA, 2008.

[4] Arati Baliga, Pandurang Kamat, and Liviu Iftode. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data (Short Paper). In *Proceedings of the 28th IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 246–251, Oakland, CA, May 2007.

[5] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-reuse Attack. In *Proceedings of the 6th ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*, pages 30–40, Hong Kong, China, March 2011.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, pages 1032–1043, Vienna, Austria, October 2016.

[7] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, pages 463–469, Snowbird, UT, 2011. Springer.

[8] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)*, 50(1):16, 2017.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, December 2008.

[10] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, Washington, DC, August 2015.

[11] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-Flow Integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, November 2006.

[12] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting Decompiler Output with Learned Variable Names and Types. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*, pages 4327–4343, Boston, MA, USA, August 2022.

[13] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium (USENIX Security)*, pages 177–191, Baltimore, MD, August 2005.

[14] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N. Asokan, and Danfeng (Daphne) Yao. Exploitation Techniques for Data-Oriented Attacks with Existing and Potential Defense Approaches. *ACM Trans. Priv. Secur.*, 24(4), sep 2021.

[15] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. ROPecker: A Generic and Practical Approach for Defending against ROP Attack. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2014.

[16] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–278, Newport Beach, CA, March 2011.

[17] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. NTFUZZ: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 677–693, Virtually, May 2021.

[18] Clang Project. Clang Control Flow Integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html, 2022.

[19] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. Memfuzz: Using Memory Accesses to Guide Fuzzing. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 48–58, Xi'an, China, 2019. IEEE.

[20] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium (USENIX Security)*, pages 63–78, San Antonio, TX, January 1998.

[21] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2017.

[22] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient Protection of Path-Sensitive Control Security. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, pages 131–148, Vancouver, BC, Canada, August 2017.

[23] Yu Ding, Tao Wei, TieLei Wang, Zhenkai Liang, and Wei Zou. Heap Taichi: Exploiting Memory Allocation Granularity in Heap-spraying Attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, pages 327–336, Austin, TX, December 2010.

[24] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS)*, Virtually, February 2021.

[25] Francisco Falcon. Exploiting Adobe Flash Player in the Era of Control Flow Guard. https://www.blackhat.com/docs/eu-15/materials/eu-15-Falcon-Exploiting-Adobe-Flash-Player-In-The-Era-Of-Control-Flow-Guard.pdf, November 2015. Black Hat Europe.

[26] Siji Feng, Zhi Zhou, and Kun Yang. Many Birds, One Stone: Exploiting a Single SQLite Vulnerability Across Multiple Software. https://www.blackhat.com/docs/us-17/wednesday/us-17-Feng-Many-Birds-One-Stone-Exploiting-A-Single-SQLite-Vulnerability-Across-Multiple-Software.pdf, July 2017. Black Hat USA Briefings (Black Hat USA).

[27] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ Combining Incremental Steps of Fuzzing Research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*, pages 10–10, Virtual, August 2020.

[28] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path Sensitive Fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 679–696, San Francisco, CA, May 2018.

[29] Google. ClusterFuzz. https://google.github.io/clusterfuzz, 2012.

[30] Google. Android Control Flow Integrity. https://source.android.com/devices/tech/debug/cfi, 2018.

[31] Google. Chromium Control Flow Integrity. https://www.chromium.org/developers/testing/control-flow-integrity, 2022.

[32] Google. FuzzBench Data. https://commondatastorage.googleapis.com/fuzzbench-data/index.html, January 2023.

[33] Google. Honggfuzz. https://google.github.io/honggfuzz/, 2023.

[34] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.

[35] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting Debug Information in Stripped Binaries. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, pages 1667–1680, Toronto, Canada, October 2018.

[36] Richard Hipp. How SQLite Is Tested. https://www.sqlite.org/testing.html, May 2022.

[37] Richard Hipp. What is SQLite? https://www.sqlite.org/index.html, December 2022.

[38] Owen S Hofmann, Alan M Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring Operating System Kernel Integrity with OSck. *ACM SIGARCH Computer Architecture News*, 39(1):279–290, 2011.

[39] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic Generation of Data-Oriented Exploits. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, pages 177–192, Washington, DC, August 2015.

[40] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 969–986, San Jose, CA, May 2016.

[41] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. The Taming of the Stack: Isolating Stack Data from Memory Errors. In *Proceedings of the 29th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, April 2022.

[42] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, pages 1868–1882, Toronto, Canada, October 2018.

[43] Christopher Jelesnianski, Mohannad Ismail, Yeongjin Jang, Dan Williams, and Changwoo Min. 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 (ASPLOS)*, pages 528–541, Vancouver, Canada, March 2023.

[44] Yaoqi Jia, Zheng Leong Chua, Hong Hu, Shuo Chen, Prateek Saxena, and Zhenkai Liang. The "Web/Local" Boundary Is Fuzzy – A Security Study of Chrome's Process-based Sandboxing. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, pages 791–804, Vienna, Austria, October 2016.

[45] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, 2002.

[46] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2011.

[47] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2020.

[48] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–163, Broomfield, Colorado, October 2014.

[49] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[50] LLVM. LibFuzzer - A Library for Coverage-guided Fuzz Testing. http://llvm.org/docs/LibFuzzer.html, 2023.

[51] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 941–951, Denver, Colorado, October 2015.

[52] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, page 1393–1403, New York, NY, USA, August 2021. ACM.

[53] Microsoft Corporation. Control Flow Guard. https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx, 2016.

[54] Matt Miller. Trends, Challenges, And Strategic Shifts In The Software Vulnerability Mitigation Landscape. https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf, 2019. BlueHat IL.

[55] MITRE. Integer Overflow Vulnerability in SSH CRC-32 Compensation Attack Detector. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0144, 2001. CVE.

[56] MITRE. CWE-123: Write-what-where Condition. https://cwe.mitre.org/data/definitions/123.html, July 2006. CWE: Common Weakness Enumeration.

[57] Micah Morton, Jan Werner, Panagiotis Kintis, Kevin Snow, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Security Risks in Asynchronous Web Servers: When Performance Optimizations Amplify the Impact of Data-Oriented Attacks. In *Proceedings of the 3rd IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 167–182, London, United Kingdom, April 2018.

[58] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[59] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, pages 31–40, 2010.

[60] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[61] Nergal. The Advanced Return-into-lib(c) Exploits (PaX case study). http://phrack.org/issues/58/4.html, December 2001. Phrack.

[62] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2005.

[63] Manh-Dung Nguyen. A Collection of Widely-Fuzzed Targets. https://github.com/strongcourage/fuzzing-targets, August 2016.

[64] Manh-Dung Nguyen. My Fuzzing Corpus. https://github.com/strongcourage/fuzzing-corpus, April 2020.

[65] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pages 729–743, Baltimore, MD, August 2018.

[66] Tapti Palit, Fabian Monrose, and Michalis Polychronakis. Mitigating Data Leakage by Protecting Memory-Resident Sensitive Data. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, pages 598–611, San Juan, Puerto Rico, December 2019.

[67] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 1919–1937, Virtually, May 2021.

[68] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22th USENIX Security Symposium (USENIX Security)*, pages 447–462, Washington, DC, August 2013.

[69] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing JavaScript Engines with Aspect-Preserving Mutation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 1629–1642, Virtually, May 2020.

[70] Nick L Petroni Jr, Timothy Fraser, AAron Walters, and William A Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *Proceedings of the 15th USENIX Security Symposium (USENIX Security)*, Vancouver, Canada, July 2006.

[71] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't Interpret, Compile! In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, pages 181–198, Virtually, August 2020.

[72] Sebastian Poeplau and Aurélien Francillon. Symbolic Execution with SymCC: Don't Interpret, Compile! In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, Virtually, August 2020.

[73] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 563–577, Virtually, May 2020.

[74] David A. Ramos and Dawson Engler. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, pages 49–64, Washington, DC, August 2015.

[75] Roman Rogowski, Micah Morton, Forrest Li, Fabian Monrose, Kevin Z. Snow, and Michalis Polychronakis. Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 366–381, Paris, France, April 2017.

[76] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (But Might Have Been Afraid to Ask). In *Proceedings of the 31th IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 317–331, Oakland, CA, May 2010.

[77] Kostya Serebryany. Sanitize, Fuzz, and Harden Your C++ Code. In *ENIGMA*, San Francisco, CA, 2016. USENIX Association.

[78] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 552–561, Alexandria, VA, October–November 2007.

[79] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[80] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS)*, pages 1–25, Hyderabad, India, December 2008. Springer.

[81] Alexander Sotirov. Heap Feng Shui in JavaScript. *Black Hat Europe*, 2007:11–20, 2007.

[82] Bing Sun, Chong Xu, and Stanley Zhu. The Power of Data-Oriented Attacks: Bypassing Memory Mitigation Using Data-Only Exploitation Technique. https://www.blackhat.com/docs/asia-17/materials/asia-17-Sun-The-Power-Of-Data-Oriented-Attacks-Bypassing-Memory-Mitigation-Using-Data-Only-Exploitation-Technique.pdf, March 2017. Black Hat Asia Briefings (Black Hat Asia).

[83] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 48–62, San Francisco, CA, May 2013.

```
void Shell::AddOSMethods(Isolate* isolate,
                         Local<ObjectTemplate> os_templ) {
  if (options.enable_os_system) {
    os_templ->Set(isolate, "system",
      FunctionTemplate::New(isolate, System));
  } ...
}
```

Figure 6: Altering `options->enable_os_system` will activate `system` function in v8

```
// Use CVE-2021-30632 to corrupt options->enable_os_system.

var workerScript = `os.system("bash")`;
var worker = new Worker(workerScript, {type: "string"});
```

Figure 7: The exploit code to execute arbitrary program in v8

[84] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 927–940, Denver, Colorado, October 2015.

[85] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory Usage Guided Fuzzing. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, pages 765–777, Seoul, South Korea, July 2020.

[86] Jidong Xiao, Hai Huang, and Haining Wang. Kernel Data Attack is a Realistic Security Threat. In *Proceedings of the 11th EAI International Conference on Security and Privacy in Communication Networks (SecureComm)*, pages 135–154, Dallas, TX, October 2015. Springer.

[87] Yang Yu. Write Once, Pwn Anywhere. https://www.blackhat.com/docs/us-14/materials/us-14-Yu-Write-Once-Pwn-Anywhere.pdf, August 2014. Black Hat USA Briefings (Black Hat USA).

[88] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, Baltimore, MD, August 2018.

[89] Michal Zalewski. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl, November 2017.

[90] Michal Zalewski. Fuzzing Random Programs Without Execve(). https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html, 2019.

A Case Study of openssh

We test openssh, the origin of our motivating example, and expect it to report some syscall-guard variables named `authenticated`. openssh has evolved from the vulnerable version 2.2.0 (released in 2000) to version 9.1 (released in 2022). From the latest version VIPER does not identify any variable named `authenticated`, but reports a variable `result` triggering syscall `execve`. We alter this variable using GDB and successfully login with incorrect password. This confirms that VIPER identifies a new syscall-guard variable. However, openssh source code contains multiple variables named `authenticated`, which makes us wonder whether VIPER fails to identify these variables.

We manually inspect the source code of the latest openssh, and confirm that VIPER is correct. Among all branches using `authenticated` variables, no one can decide to invoke syscall `execve`. Especially, the latest openssh adopts mandatory user-privilege separation, and creates two processes for authentication. The child process runs in a loop to read network packets and send them to the parent, while the parent executes in another loop to check whether the child data passes the authentication. If the authentication succeeds, the parent will share the result with the child. After that, they will break the loops, synchronize the key state, and create the shell for the client. Both processes save the result into `authenticated` variables, but these variables are local and cannot affect each other. If VIPER only flips an `authenticated`-related branch in the parent, the child still has the `false` result, which fails the key-state synchronization; if VIPER only flips a such branch in the child, the parent will stay in the loop and never create the shell. Therefore, VIPER did not report any such checks. Instead, before sharing the result, the parent keeps the value in a `result` variable, and conducts a conditional check on it. When VIPER flips this check, the parent will update the value to `true`, and share it with the child. VIPER captures this check as a syscall-guard branch, and reports `result` as the syscall-guard variable.

This example demonstrates the effectiveness of VIPER. Although the latest design is more complicated and the previously known variable does not work, it can automatically explore other branches to identify new syscall-guard variables. Moreover, openssh contains many obstacles that hinder static analysis, like asynchronous event processing, indirect function call and inter-process communication. VIPER circumvents these challenges and successfully spots critical data.

B Case Study of v8 Exploit

VIPER reports a syscall-guard variable `options.enable_os_system` in v8 JavaScript engine that allows attackers to execute any program. Figure 6 shows the related function `Shell::AddOSMethods` where v8 prepares available methods for each isolate thread. When initializing a new worker thread, v8 checks the value of variable `options.enable_os_system` to decide whether to enable the `system` function, while this function invokes syscall `execve` internally. By default, users do not turn on this option. Therefore, `options.enable_os_system` is `false` and v8 will not activate `system`. However, if attackers flip the branch at line 3, all subsequently created threads can reactivate `system`. By corrupting the variable and subsequently creating new threads using crafted JavaScript code, attackers can obtain the ability to execute any program, *e.g.*, spawning an interactive shell. `VariableRator` reports 93,512,607 memory-write instructions in the lifespan of `options.enable_os_system`. It is highly corruptible after breaking ASLR.

We build an end-to-end attack, exploiting a historical type confusion vulnerability (CVE-2021-30632) caused by improper JIT optimization. v8 mistakenly interprets `SMI` array as `double`, enabling attackers to manipulate the memory layout and achieve arbitrary memory read and write. Figure 7 shows a simplified version of our exploit code. We customize the public proof-of-concept to modify the store address `rwxAddr` to the address of `options.enable_os_system` (retrieved from information leakage through the same CVE), and alter the value `shellcode` to `1`. Next, we create a new worker object, which leads v8 to create a new thread internally and specify the worker to execute `os.system("bash")`. As a result, the worker thread can run the command and get a shell eventually.

Syscall	Related Library Call	Description
mmap, mremap	mmap, mremap, realloc, getpwnam	map/remap files or devices into memory
mprotect	mprotect, getpwnam, getgrnam, pthread_create	change memory access protections
execve	execve, execlp, execv, system	execute a program
chdir, fchdir, chroot	chdir, fchdir, chroot	change the working/root directory of the current process
rename	rename	change the name or location of a file
unlink, unlinkat	unlink, remove, unlinkat	delete a name and possibly the file it refers to
symlink, symlinkat	symlink, symlinkat	create a symbolic link
chmod, fchmod	chmod, fchmod	change file mode bits
chown, fchown, lchown, fchownat	chown, fchown, lchown, fchownat	change ownership of a file
setuid, setgid, setpgid	setuid, setgid, setpgid	set the process user/group ID
setreuid, setregid, setresuid	setreuid, setregid, setresuid, seteuid	set real, effective, saved user/group ID
setgroups	setgroups	set list of supplementary group IDs
setfsuid, setfsgid	setfsuid, setfsgid	set user/group identity used for filesystem checks
uselib	-	load shared library
setrlimit	setrlimit	set resource limits
init_module, delete_module	init_module, delete_module	load/unload a kernel module

Table 4: Security-related syscalls and related library calls considered in our evaluation. Attackers can abuse these syscalls to compromise the vulnerable programs and systems. Red calls represent unexploitable wrappers since attackers have limited control over the syscall arguments.

Source	Program	Version	BranchForcer Evaluation		VariableRator Evaluation	
FuzzBench	sqlite	3.40.1	✔	(corpus from source code repository)	✔	
	curl	dd486c1e5	✔	(FuzzBench Dataset [32])	✔	
	harfbuzz	f73a87d9a	✔	(FuzzBench Dataset [32])	✔	
	systemd	v252	✔	(FuzzBench Dataset [32])	✗	[pthread to be supported]
	mbedtls	10ada3501	✔	(FuzzBench Dataset [32])	∅	[no syscall-guard branches]
	openssl	3.0.7	✔	(FuzzBench Dataset [32])	∅	[no syscall-guard branches]
	freetype2	cd02d35	✔	(FuzzBench Dataset [32])	∅	[no syscall-guard branches]
	php	8.1.0	✗	[failed to get PHP return value]		
	re2	954656f47	∅	[no critical syscall/libcall in PLT]		
	woff	4721483ad	∅	[no critical syscall/libcall in PLT]		
	bloaty	52948c107	∅	[no critical syscall/libcall in PLT]		
	zlib	02a6049eb	∅	[no critical syscall/libcall in PLT]		
	jsoncpp	8190e061b	∅	[no critical syscall/libcall in PLT]		
	littlecms	f7db22d3e	∅	[no critical syscall/libcall in PLT]		
	libjpeg-turbo	d859232da	∅	[no critical syscall/libcall in PLT]		
	libpng	9923515ff	∅	[no critical syscall/libcall in PLT]		
	vorbis	c8391c2b2	∅	[no critical syscall/libcall in PLT]		
	proj4	d00501750	∅	[no critical syscall/libcall in PLT]		
	libpcap	59aab18ee	∅	[no critical syscall/libcall in PLT]		
	openthread	5b0af03af	∅	[no critical syscall/libcall in PLT]		
	libxml2	59b336617	∅	[no critical syscall/libcall in PLT]		
Programs with known attacks	nginx	1.20.2	✔	(one simple file and fuzz for 6 hours)	✔	
	openssh	9.1	✔	(a correct username and a wrong password)	✔	
	sudo	1.9.9	✔	(two simple files and fuzz for 6 hours)	✔	
	null httpd	1.20.2	✔	(one command "wget index.html")	✔	
	ghttpd	1.4.4	✔	(one command "wget index.html")	✔	
	orzhttpd	0.0.6	✔	(one command "wget index.html")	✔	
	wu-ftpd	2.6.2	✔	(two packets with correct/incorrect password)	✔	
	telnet	3f352877e	✔	(one self-generated pcap file)	∅	(no syscall-guard branches)
	chromium		✗	[huge program, to be supported]		
	⊢ v8	8.5.188	✔	(one JS statement that run local command)	✔	
	httpdx		✗	[Windows program, to be supported]		
	IE browser		✗	[Windows binaries, to be supported]		
More [63]	jhead	3.04	✔	(corpus from source code repository)	✔	
	jasper	4.0.0	✔	(online corpus [64] and fuzz for 6 hours)	✔	
	pdfalto	0.4	✔	(online corpus [64] and fuzz for 6 hours)	✔	
	gzip	v1.12	✔	(online corpus [64] and fuzz for 6 hours)	✔	

Table 5: Programs for evaluation. We select 20 programs from three benchmarks, FuzzBench, programs with known attacks and common fuzzing targets. For BranchForcer evaluation, we list supported programs (✔) with adopted test cases, and unsupported programs (✗) and untested programs (∅) with reasons. For VariableRator evaluation, we list supported programs and unsupported programs with reasons.