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
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 increases 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.

### 2.1 Data-Only Attack and Critical Data

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” \[40\]. 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...
We investigate previous data-only attacks for privilege escalation. Therefore, syscalls are commonly used in data-only attacks. 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 form to the legal control-flow graph and attackers can only modify the data to affect syscalls. First, the data is used as one syscall argument. Whether it is a syscall-guard variable or not determines 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.

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

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.

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.

3.2 Controllability 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
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.

**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.

**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 generate 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...
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 condition, and save the 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 of the original branch. 

Figure 4 shows various instrumentation to the original code for different purposes. Lines 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.
We instrument the program IR to record execution contexts and values to determine the correct control flow. For a direct call, we get the called function information 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`).

### 4.2.1 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 the `if` statement. When `%i` 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 value. Different from §4.1.1, now we record every condition value. 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 the `path` (line 11). For load and store instructions, we retrieve memory addresses from the trace and save them into the `path` (lines 14). For direct calls, we follow the call graph to step into the called function (lines 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 insn, we scan array path to locate this instruction (line 3). insn, 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 sysems 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 sysems. 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.
Table 2: Syscall-guard variables discovered by VIPER and corruptibility assessment. * means previously known variables.

<table>
<thead>
<tr>
<th>Program</th>
<th>Guard Variable</th>
<th>Branch Location</th>
<th>Syscall</th>
<th>Malicious Goal</th>
<th>Rate (S, H, G)</th>
<th>CK</th>
<th>CVE</th>
<th>Type</th>
<th>Cap</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlite</td>
<td>mode</td>
<td>shell.c:5002</td>
<td>symlink</td>
<td>create symlink to any file</td>
<td>(55, 0, G)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>shell.c:5038</td>
<td>chmod</td>
<td>change any file to any mode</td>
<td>(75, 0, G)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p-&gt;doXdgOpen</td>
<td>shell.c:20270</td>
<td>execve</td>
<td>execute arbitrary program</td>
<td>(181770, 0, 0)</td>
<td>☀</td>
<td>2017-6983</td>
<td>TC</td>
<td>AW</td>
<td></td>
</tr>
<tr>
<td>p-&gt;zTempFile</td>
<td>shell.c:20560</td>
<td>unlink</td>
<td>delete any file</td>
<td>(86907, 0, 0)</td>
<td>☀</td>
<td>2017-6983</td>
<td>TC</td>
<td>AW</td>
<td></td>
</tr>
<tr>
<td>isDelete</td>
<td>symlink3.c:42399</td>
<td>unlink</td>
<td>delete any file</td>
<td>(8353, 29276, 0)</td>
<td>☀</td>
<td>2017-6983</td>
<td>TC</td>
<td>AW</td>
<td></td>
</tr>
<tr>
<td>zPath</td>
<td>symlink3.c:43094</td>
<td>unlink</td>
<td>delete any file</td>
<td>(57, 15356, 0)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exists</td>
<td>symlink3.c:60294</td>
<td>unlink</td>
<td>delete any file</td>
<td>(58, 15356, 0)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isW1</td>
<td>symlink3.c:58492</td>
<td>unlink</td>
<td>delete any file</td>
<td>(61, 15046, 0)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>curl</td>
<td>tempstore</td>
<td>cookie.c:1732</td>
<td>rename</td>
<td>overwrite any file</td>
<td>(15, 0, 0)</td>
<td>☀</td>
<td>2019-3822</td>
<td>H/SBoF</td>
<td>AW</td>
</tr>
<tr>
<td></td>
<td>tempstore</td>
<td>hsts.c:386</td>
<td>rename</td>
<td>overwrite any file</td>
<td>(15, 0, 0)</td>
<td>☀</td>
<td>2019-3822</td>
<td>H/SBoF</td>
<td>AW</td>
</tr>
<tr>
<td></td>
<td>tempstore</td>
<td>altsvc.c:359</td>
<td>rename</td>
<td>overwrite any file</td>
<td>(15, 0, 0)</td>
<td>☀</td>
<td>2019-3822</td>
<td>H/SBoF</td>
<td>AW</td>
</tr>
<tr>
<td>harfbuzz</td>
<td>blob-&gt;mode</td>
<td>hb-blob.c:453</td>
<td>mprotect</td>
<td>make RO memory writable</td>
<td>(31, 352, 0)</td>
<td>☀</td>
<td>2015-8947</td>
<td>HBoF</td>
<td>AW</td>
</tr>
<tr>
<td>nginx</td>
<td>sa_family</td>
<td>$connection.c:631</td>
<td>chmod</td>
<td>change file mode</td>
<td>(0, 84831, 0)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ngxTerminate</td>
<td>$process_cycle.c:305</td>
<td>unlink</td>
<td>delete any file</td>
<td>(0, 208640, 0)</td>
<td>☀</td>
<td>2013-2028</td>
<td>SBoF</td>
<td>AW</td>
</tr>
<tr>
<td></td>
<td>ngxQuit</td>
<td>$process_cycle.c:305</td>
<td>unlink</td>
<td>delete any file</td>
<td>(0, 208640, 0)</td>
<td>☀</td>
<td>2013-2028</td>
<td>SBoF</td>
<td>AW</td>
</tr>
<tr>
<td></td>
<td>ft.stat_uid</td>
<td>($nxg)$file.c:631</td>
<td>chown</td>
<td>change owner of any file</td>
<td>(350832, 0, 0)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ft.stat_mode</td>
<td>$file.c:640</td>
<td>chmod</td>
<td></td>
<td>(175218, 0, 0)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>openssl</td>
<td>result*</td>
<td>auth-passwd.c:128</td>
<td>execve</td>
<td>login without password</td>
<td>(5, 48153908, 0)</td>
<td>☀</td>
<td>2012-0809</td>
<td>FS</td>
<td>AW</td>
</tr>
<tr>
<td></td>
<td>received_SIGTERM</td>
<td>sshd.c:1163</td>
<td>unlink</td>
<td>delete any file</td>
<td>(0, 1463147, 0)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>received_SIGHUP</td>
<td>sshd.c:1177</td>
<td>execve</td>
<td>execute arbitrary program</td>
<td>(0, 1470063)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sudo</td>
<td>details-&gt;chroot</td>
<td>exec.c:173</td>
<td>chroot</td>
<td>change root path</td>
<td>(0, 2039)</td>
<td>☀</td>
<td>2012-0809</td>
<td>FS</td>
<td>AW</td>
</tr>
<tr>
<td></td>
<td>info</td>
<td>sudo.c:697</td>
<td>chdir</td>
<td>change directory path</td>
<td>(1702, 253382, 1982)</td>
<td>☀</td>
<td>2012-0809</td>
<td>FS</td>
<td>AW</td>
</tr>
<tr>
<td>null http</td>
<td>in_RequestURI</td>
<td>main.c:39</td>
<td>execve</td>
<td>enable CGI to run programs</td>
<td>(0, 525, 0)</td>
<td>☀</td>
<td>2002-1496</td>
<td>HBoF</td>
<td>AW</td>
</tr>
<tr>
<td>ghtpd</td>
<td>filename*</td>
<td>protocol.c:127</td>
<td>execve</td>
<td>enable CGI to run programs</td>
<td>(9, 0, 5912)</td>
<td>☀</td>
<td>2002-1904</td>
<td>SBoF</td>
<td>AW</td>
</tr>
<tr>
<td>wu-ftpd</td>
<td>RootDirectory</td>
<td>ftpd.c:1029</td>
<td>chroot</td>
<td>change root path of current user</td>
<td>(0, 7322)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>anonymous</td>
<td>ftpd.c:2527</td>
<td>setgroups</td>
<td>obtain root privilege</td>
<td>(0, 7432)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>guest</td>
<td>ftpd.c:2893</td>
<td>chroot</td>
<td>change root path of anonymous</td>
<td>(0, 8341)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>guest</td>
<td>ftpd.c:2893</td>
<td>chroot</td>
<td>change root path of guest</td>
<td>(0, 37715)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jhead</td>
<td>RegenThumbnail</td>
<td>jhead.c:978</td>
<td>execve</td>
<td>execute arbitrary program</td>
<td>(0, 2856)</td>
<td>☀</td>
<td>2016-3822</td>
<td>IO</td>
<td>AW</td>
</tr>
<tr>
<td></td>
<td>EditComment</td>
<td>jhead.c:1003</td>
<td>execve</td>
<td>edit any file using vi</td>
<td>(0, 2856)</td>
<td>☀</td>
<td>2016-3822</td>
<td>IO</td>
<td>AW</td>
</tr>
<tr>
<td></td>
<td>CommentInsertfileName</td>
<td>jhead.c:1003</td>
<td>execve</td>
<td>edit any file using vi</td>
<td>(0, 2856)</td>
<td>☀</td>
<td>2016-3822</td>
<td>IO</td>
<td>AW</td>
</tr>
<tr>
<td></td>
<td>CommentInsertLiteral</td>
<td>jhead.c:1003</td>
<td>execve</td>
<td>edit any file using vi</td>
<td>(0, 2856)</td>
<td>☀</td>
<td>2016-3822</td>
<td>IO</td>
<td>AW</td>
</tr>
<tr>
<td>jasper</td>
<td>fileobj-&gt;flags</td>
<td>jas_stream.c:1392</td>
<td>unlink</td>
<td>delete any file</td>
<td>(0, 219062, 0)</td>
<td>☀</td>
<td>2020-27828</td>
<td>HBoF</td>
<td>AW</td>
</tr>
<tr>
<td>pdfalto</td>
<td>first</td>
<td>XRef.c:240</td>
<td>unlink</td>
<td>delete files in specific folders</td>
<td>(1952, 214, 0)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>offsets[]</td>
<td>XRef.c:240</td>
<td>unlink</td>
<td>delete files in specific folders</td>
<td>(92, 117, 0)</td>
<td>☀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gzip</td>
<td>fd</td>
<td>gzip.c:2111</td>
<td>unlink</td>
<td>delete any file</td>
<td>(0, 11886, 0)</td>
<td>☀</td>
<td>2010-0001</td>
<td>IO</td>
<td>AW</td>
</tr>
<tr>
<td>v8</td>
<td>enable_os_system</td>
<td>d8-posix.c:762</td>
<td>execve</td>
<td>execute any program</td>
<td>(0, 93512607)</td>
<td>☀</td>
<td>2021-30632</td>
<td>TC</td>
<td>AW</td>
</tr>
</tbody>
</table>

Note: exploits constructed with concrete bugs; ☀: exploitability checked with concrete bugs; ☀: primitives emulated with GDB. Implicates ☀ and ☀ implicates ☀. Bug types: type confusion (TC); heap/stack buffer overflow (H/SBoF); format string bug (FS); integer overflow (IO). Capability: write anywhere any value (AW).
porary file with an editor for users to modify the query result. In normal executions, \texttt{p->doXdgOpen} is false and sqlite will not invoke function \texttt{system}. If attackers flip the branch at line 2, sqlite will call \texttt{system} which internally invokes \texttt{execve}. Attackers can corrupt \texttt{p->zTempFile} to execute arbitrary program, like changing \texttt{zCmd} to \texttt{a.txt; cat ~/.ssh/id_rsa} to retrieve the private key. Pointer \texttt{p} points to a stack \texttt{ShellState} object that maintains database connection states and contains two members \texttt{doXdgOpen} and \texttt{zTempFile}. sqlite initializes this object at the early stage of the execution, and only invokes \texttt{output_reset} before exit. Given an empty query “.exit”, \texttt{VariableRator} reports 181,770 memory-write instructions in the lifespan of \texttt{p->doXdgOpen} and 86,907 memory-write instructions in the lifespan of \texttt{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 \texttt{Fts3Cursor} object. This bug has been fixed in version 3.40.1. We modify three lines in function \texttt{fts3FunctionArg} to reactivate it. Following the tutorial, we turn the bug into an arbitrary-memory-write primitive. We use the primitive to modify \texttt{doXdgOpen} and \texttt{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 \texttt{p->zTempFile} can trigger arbitrary file deletion. Specifically, sqlite uses temporary files in many ways. As shown in function \texttt{clearTempFile} of Figure 5, sqlite deletes any existing temporary files before creating a new one. If \texttt{p->zTempFile} is NULL (line 8), it will return immediately since no temporary file exists yet. Otherwise, it will call \texttt{shellDeleteFile} which internally calls \texttt{unlink} to delete existing files. An attacker can corrupt \texttt{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 \texttt{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 \texttt{m} at line 37364 of \texttt{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 \texttt{getpwnam} internally invokes syscall \texttt{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 \texttt{unlink(strcat(tempfile, ".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, \texttt{jrn10pen} in sqlite affects two branches, while triggering syscall \texttt{unlink}
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). A ✓ pthread not supported.

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>kLoC</th>
<th>#Test</th>
<th>Branches/Variables</th>
<th>Time Cost</th>
<th>Stitch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Record</td>
<td>Flip</td>
</tr>
<tr>
<td>sqlite</td>
<td>3.40.1</td>
<td>273</td>
<td>35,757</td>
<td>20,653 14,756 71.4%</td>
<td>21 9 7</td>
<td>2880' 112' 378' 778' 87'</td>
</tr>
<tr>
<td>curl</td>
<td>97E7E66</td>
<td>160</td>
<td>4,286</td>
<td>37,251 2,645 7.1% 13 3 3</td>
<td>23' 32' 689' 744' 248'</td>
<td></td>
</tr>
<tr>
<td>haribuzz</td>
<td>1.3.2</td>
<td>41</td>
<td>10,816</td>
<td>3,878 2,656 68.5%</td>
<td>1 1 1</td>
<td>17' 8' 8' 33' 33'</td>
</tr>
<tr>
<td>systemd</td>
<td>v252</td>
<td>543</td>
<td>2,851</td>
<td>28,739 2,093 7.3%</td>
<td>5 ✗ ✗</td>
<td>69' 40' - &gt;109' &gt;109'</td>
</tr>
<tr>
<td>mbedtsl</td>
<td>1.8ada35</td>
<td>128</td>
<td>1,078</td>
<td>8,754 1,528 17.5%</td>
<td>0 0 0</td>
<td>2' 6' - &gt;8' &gt;8'</td>
</tr>
<tr>
<td>openssl</td>
<td>3.0.7</td>
<td>483</td>
<td>2,515</td>
<td>32,063 5,448 17.0%</td>
<td>0 0 0</td>
<td>13' 61' - &gt;74' &gt;74'</td>
</tr>
<tr>
<td>freetype</td>
<td>2db2d35</td>
<td>119</td>
<td>11,978</td>
<td>10,224 4,159 40.7%</td>
<td>0 0 0</td>
<td>18' 26' - &gt;44' &gt;44'</td>
</tr>
<tr>
<td>nginx</td>
<td>1.26.2</td>
<td>141</td>
<td>1,219</td>
<td>12,483 2,408 19.3%</td>
<td>7 5 5</td>
<td>238' 22' 39' 58'</td>
</tr>
<tr>
<td>opensslh</td>
<td>368d4d3</td>
<td>119</td>
<td>1 11</td>
<td>11,060 2,316 20.9%</td>
<td>4 3 3</td>
<td>1' 4722' 10624' 15347' 5116' 1110'</td>
</tr>
<tr>
<td>sudo</td>
<td>1.9.9</td>
<td>110</td>
<td>329 6,841</td>
<td>2,411 35.0%</td>
<td>26 17 2</td>
<td>16' 16' 260' 292' 18' 393'</td>
</tr>
<tr>
<td>null httpd</td>
<td>0.5.1</td>
<td>2 1 124</td>
<td>206 104 47.3%</td>
<td>1 1 1</td>
<td>1' 10'</td>
<td>31' 42' 42' 358'</td>
</tr>
<tr>
<td>ghtpd</td>
<td>1.4.4</td>
<td>1 1</td>
<td>111 43</td>
<td>38.7%</td>
<td>2 2 1</td>
<td>1' 36' 72' 109' 55' 48'</td>
</tr>
<tr>
<td>orzhttpd</td>
<td>0.6.6</td>
<td>1 1</td>
<td>191 100</td>
<td>52.4%</td>
<td>0 0 0</td>
<td>1' 32' - &gt;33' &gt;33' 93'</td>
</tr>
<tr>
<td>wu-ftpd</td>
<td>2.6.2</td>
<td>18 2</td>
<td>2,918 399</td>
<td>13.7%</td>
<td>9 8 4</td>
<td>1' 533' 189' 723' 91' 200'</td>
</tr>
<tr>
<td>telnet</td>
<td>3f35287</td>
<td>11 1 420</td>
<td>147 35.0%</td>
<td>0 0 0</td>
<td>1' 144' - &gt;145' &gt;145'</td>
<td></td>
</tr>
<tr>
<td>jhead</td>
<td>3.94</td>
<td>4 47</td>
<td>616 287</td>
<td>46.6%</td>
<td>12 12 4</td>
<td>1' 2' 2' 288' 291' 25'</td>
</tr>
<tr>
<td>jasper</td>
<td>4.0.0</td>
<td>34 72</td>
<td>3,237 4,949</td>
<td>1,777 35.9%</td>
<td>4 1 1</td>
<td>37' 16' 84' 137' 137'</td>
</tr>
<tr>
<td>pdfalto</td>
<td>0.4</td>
<td>76 16,954</td>
<td>14,069 4,581</td>
<td>32.6%</td>
<td>5 2 2</td>
<td>342' 116' 107' 565' 282'</td>
</tr>
<tr>
<td>gzip</td>
<td>1.12</td>
<td>6 1,558</td>
<td>1,133 374</td>
<td>33.0%</td>
<td>1 1 1</td>
<td>6' 1' 9' 19' 26' 26'</td>
</tr>
<tr>
<td>v8</td>
<td>8.5.188</td>
<td>3,586</td>
<td>1    170,232 6,819 4.0%</td>
<td>2 1 1</td>
<td>1' 5833' 874' 6708' 6708'</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Branches and time 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]. Print-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 privilege 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


```javascript
void Shell::AddOSSMethods(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

```javascript
// 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

---

[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::AddOSSMethods 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. VariableWorker 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 STM 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 radxaddr 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.

---

[A] References:


---

**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::AddOSSMethods 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. VariableWorker 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 STM 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 radxaddr 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.
We select 20 programs from three benchmarks, FuzzBench, programs with known attacks and common
untested programs (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.