SPINNER: Automated Dynamic Command Subsystem Perturbation

Meng Wang, Chijung Jung, Ali Ahad, and Yonghwi Kwon
University of Virginia
Charlottesville, Virginia 22904, USA
{mw6td,cj5kd,aa5rn,yongkwon}@virginia.edu

ABSTRACT

Injection attacks have been a major threat to web applications. Despite the significant effort in thwarting injection attacks, protection against injection attacks remains challenging due to the sophisticated attacks that exploit the existing protection techniques' design and implementation flaws. In this paper, we develop SPINNER, a system that provides general protection against input injection attacks, including OS/shell command, SQL, and XXE injection. Instead of focusing on detecting malicious inputs, SPINNER constantly randomizes underlying subsystems so that injected inputs (e.g., commands or SQL queries) that are not properly randomized will not be executed, hence prevented. We revisit the design and implementation choices of previous randomization-based techniques and develop a more robust and practical protection against various sophisticated input injection attacks. To handle complex real-world applications, we develop a bidirectional analysis that combines forward and backward static analysis techniques to identify intended commands or SQL queries to ensure the correct execution of the randomized target program. We implement SPINNER for the shell command processor and two different database engines (MySQL and SQLite) and in diverse programming languages including C/C++, PHP, JavaScript and Lua. Our evaluation results on 42 real-world applications including 27 vulnerable ones show that it effectively prevents a variety of input injection attacks with low runtime overhead (around 5%).

1 INTRODUCTION

Injection attacks have been a long-standing security problem, listed as the first security risk in the OWASP Top 10 security risks [116]. Among them, input injection (e.g., shell command/SQL injection) is one of the most prevalent injection attacks. It happens when malicious inputs (shell commands or SQL queries) are injected and executed on the victim system. Despite the effort in thwarting injection attacks [16, 21, 27, 30, 68–71, 75, 80, 84, 101, 106, 123, 137, 145, 158, 159, 164], injection vulnerabilities are still pervasive in practice because, in part, the ever-evolving attacks exploit the limitations of the prevention measures.

Existing Prevention Techniques. Input sanitization/validation is a recommended practice to prevent input injection attacks [10, 16, 75]. However, implementing a sanitizer that can filter out all malicious inputs is extremely challenging due to the large and complex input space (e.g., grammars for OS/shell commands and SQL are expressive, allowing various inputs). Another straightforward approach is first identifying all allowed inputs on each call-site of APIs and only allowing them. However, this cannot prevent attacks that inject the allowed inputs twice. For instance, attackers can inject new "`rm`" commands to a vulnerable code snippet "`system("rm logfile $opt")`" (Details can be found in Appendix 9.3.1). There are more advanced prevention techniques, such as those leveraging dynamic taint analysis [30, 68, 70, 106, 123, 145]. However, they suffer from over/under tainting issues and runtime overhead. Techniques that build models of benign commands/SQL queries to detect anomalies [21, 69, 137] require accurate modeling of attackers and target applications, which have been evolving over the years.

Randomization-based Prevention. There are techniques [27, 120] that randomize SQL keywords (in SQL engine and benign SQL queries) to prevent the execution of injected SQL queries that are not randomized. While the idea is effective, they have a critical limitation in their design choices. To deploy the techniques, they rely on a proxy to translate a randomized query to a standard query using a *parser*. If the proxy's translator fails because of sophisticated SQL queries and grammar differences between SQLs (e.g., SQL dialects [85] as discussed in Section 5.4.1), malicious queries can be injected or benign queries may not be properly executed. Diglossia [140] is an injection attack prevention technique that proposes the dual-parsing approach. Unfortunately, it also relies on the accuracy of the parser used in the dual-parser (Details on how Diglossia will fail are presented in Section 5.4.2). Other randomization techniques [27, 120] are susceptible to attacks that leak randomization key because their randomization scheme is not dynamically changing. Attackers can then prepare and inject a randomized command.

Our Approach. We propose a robust and practical randomization-based technique called SPINNER to prevent input injection attacks. The technique works by randomizing words in inputs (e.g., commands and SQL queries) and the subsystems (e.g., shell process and SQL engine) that parse and run the inputs. The randomized subsystems does not allow commands that are not properly randomized to be executed. For instance, if '`rm`' is randomized to '`xc`' (`rm` ↦ `xc`), the original command '`rm`' will result in an error (i.e., the command not found error) while '`xc`' command will work as same as the original '`rm`.' To ensure the *intended benign commands* from applications work correctly with the randomization, we analyze target programs to identify and instrument the intended commands to be randomized. To this end, legitimate commands are correctly randomized at runtime, while injected commands are not randomized and prevented from being executed.

1) Revisiting Design Choices: To mitigate sophisticated attacks evading existing randomization based preventions [27, 120], we revisit the design choices made by existing techniques. First, we eliminate the proxy and parser requirement for the shell process randomization by hooking APIs called before and after the shell process's original parser. Second, for SQL engines that are difficult

to blend our technique in, we develop a *bidirectional randomization* scheme based on a scanner (Details in Section 4.2.2) to prevent sophisticated attacks (e.g., those exploiting bugs/flaws of parsers). Third, SPINNER changes the randomization scheme at runtime so that even if an attacker learns a previously used randomization key and injects a randomized command, the attack will fail.

2) Program Analysis Approaches for Input Randomization: We propose practical program analysis techniques that can effectively analyze large and complex programs for input randomization. In particular, we show that our approach, bidirectional data flow analysis (Section 4.1.1), is scalable to real-world applications including WordPress [65]. Our contributions are summarized as follows:

- We propose an approach that can prevent various types of input injection attacks by randomizing the subsystems that run or process the inputs.
- We design and develop an effective static data flow analysis technique called bidirectional analysis that can identify intended commands in complex real-world applications.
- We implement a prototype of SPINNER in diverse programming languages, including C/C++, PHP, JavaScript and Lua.
- Our evaluation results show that it prevents 27 input injection attacks with low overhead (≈5%).
- We release our implementation and data-sets publicly [146].

2 DEFINITIONS AND BACKGROUNDS

Scope of Inputs for Randomization. We consider three types of inputs to randomize: OS/shell commands, SQL queries, and XML queries. This is because they are commonly exploited in web server applications that SPINNER aims to protect, according to the OWASP Top 10 document [116]. Those inputs are used by a program to leverage external programs' functionalities. For example, a program can compress files by executing a shell command that executes '`gzip`'. SQL queries are for SQL engines to store and retrieve values to/from the database. An XML query is an interface for interacting with XML entities (e.g., reading and writing values in the entities).

Choice of Term 'Command.' SPINNER focuses on preventing three different input injection attacks: shell injection, SQL injection, and XXE injection attacks. In this paper, we use the term *command* to include the three input types to facilitate the discussion. We consider SQL queries and XXE entities *commands* as they eventually make the subsystem run or execute particular code.

Command Execution APIs. We define a term *Command Execution API* to describe APIs that execute a command or a query. A list of command execution APIs is shown in Table 1.

Command Specification. A command passed to a command execution API should follow a certain specification. Specifically, shell commands should use correct command names or external executable binary file names. SQL queries should follow the predefined SQL keywords and grammar. If a command does not follow the specification (e.g., a wrong file name), its execution will fail.

Input Injection Vulnerability. Input injection happens when an attacker injects malicious inputs to the composed command string or SQL query string passed to a command execution API as an argument. In practice, programs may try to validate and sanitize suspicious inputs that might contain malicious inputs. Typically, when a program composes a command, the command name (e.g., '`gzip`') is defined as a constant string or loaded from configuration files that are not accessible to attackers (hence can be trusted). However, some programs allow users to define arguments of the command. As a result, attackers aim to inject malicious commands through the arguments. After a command is composed, the program calls command execution APIs (e.g., `exec()`, `system()`, or `mysql_query()`) to fulfill the command execution.

Limitations of Existing Randomization Techniques. There are existing techniques [26, 27, 120] that randomize the keywords and grammars. While we share the similar idea to them, our work differs from them as we aim to solve the following three limitations.

First, existing techniques leverage parsers to randomize/derandomize commands. Unfortunately, attackers often exploit bugs or design flaws in parsers to evade the prevention techniques that rely on them. In particular, the parsers may not handle complicated benign inputs (e.g., because of the use of SQL dialects [85]), breaking benign functionalities or allowing injection attacks. We elaborate details of such weaknesses of existing techniques in Section 5.4.1.

> SPINNER handles this by integrating our randomization scheme to the internal parser in the shell process and leveraging our bidirectional randomization scheme for SQL engines. The bidirectional randomization is grammar and keywords agnostic, meaning that attacks exploiting flaws of parsers will be prevented.

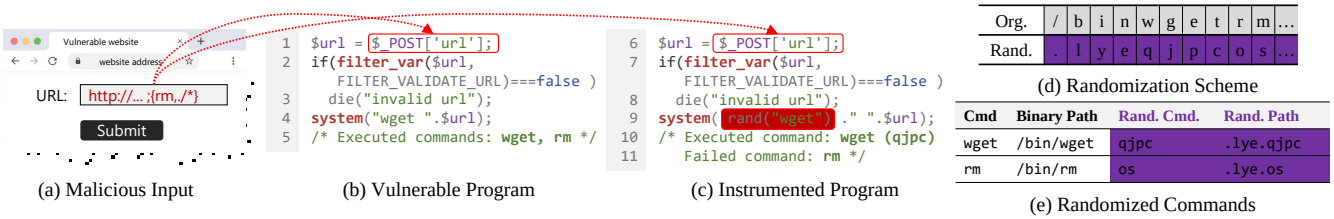
Second, existing automated approaches [120, 149] leverage static analysis techniques to identify intended (i.e., benign) commands in the source code. We find that their static analysis techniques are not scalable to complex real-world applications.

> We propose a practical and scalable bi-directional data flow analysis (Details in Section 4.1.1) that can effectively identify benign commands in complex real-world applications.

Third, existing techniques randomize the command specification statically, meaning that the commands are randomized only once. If an attacker learns the randomization scheme (e.g., via information leak vulnerabilities), the attacker can inject randomized commands which will not be prevented.

> SPINNER dynamically randomizes the command specification whenever a command execution API is called. To this end, even if an attacker learns a previously used randomization key, it will not help subsequent attacks.

Threat Model. SPINNER aims to prevent remote input injection attacks (including SQL/XXE injections) on server-side applications. We expect server-admins and web-developers as typical users of SPINNER. Client-side attacks such as XSS (Cross Site Scripting) and XSRF (Cross Site Request Forgery) are out of the scope. We assume the subject program and inputs from trusted sources (defined by the user) are benign, but inputs from untrusted sources can include malicious commands. Typical trusted sources are local configuration files. We trust local software stacks, including OS kernel, applications, and libraries. If they are compromised, attackers can disable SPINNER. SPINNER does not focus on preventing attacks that compromise non-command parts such as arguments of commands (e.g., a directory traversal attack). SPINNER does not aim to prevent binary code injection (e.g., shellcode injection).



(a) Malicious Input (b) Vulnerable Program (c) Instrumented Program (d) Randomization Scheme (e) Randomized Commands

Figure 1: Example of SPINNER Preventing a Command Injection Attack

3 MOTIVATING EXAMPLE

Figure 1 shows an example of a shell command injection attack to a vulnerable server-side program written in PHP. From a website shown in Figure 1-(a), an attacker sends a malicious input (with a malicious command) through the textbox on the webpage. The server-side program, shown in Figure 1-(b), is vulnerable because it directly passes the input to `system()`, executing the injected command (line 4). It tries to sanitize inputs via `filter_var()` at line 2 (commonly recommended [11, 22, 151]), but it fails.

Command Specification Randomization. In this example, SPINNER randomizes commands in the shell process. There are two types of shell commands [89]: (1) internal commands that are implemented inside of OS/shells such as '`cd`' and (2) external commands that are implemented by separate binaries such as '`grep`'.

For internal commands, we randomize the command names by hooking and overriding APIs in the shell process (Details in Section 4.2.1). For external commands, since the shell process will look up a binary file for the external command to execute (i.e., check whether a binary for the command exists), we randomize the binary file names and paths (e.g., '`rm`' ↦ '`os`' as shown in Figure 1-(e)) in the file I/O APIs. This will prevent injected malicious (and *not randomized*) commands from being executed. For the randomization, we use a one-time substitution cipher. Specifically, as shown in Figure 1-(d), we create a mapping between the original input and its randomized character. To execute a command "`wget`" under this randomization scheme, one should execute "`qjpc`" as shown in Figure 1-(e). To prevent brute-force attacks against the randomized commands, SPINNER provides two mitigations. First, SPINNER creates a new randomization scheme on *every new command* to mitigate attacks leveraging previously used randomized commands. Second, to further make the brute-force attacks difficult, SPINNER supports one to multiple bytes translation, enlarging the searching space. Details can be found in Appendix 9.3.3.

Instrumentation by SPINNER. Once the commands are randomized in the shell process, the system cannot understand commands that are not randomized. In other words, it affects every command in the program including intended and benign commands, breaking benign functionalities. To ensure the correct execution of intended commands, we statically analyze the program to identify intended (hence benign and trusted) commands that are originated from trusted sources (e.g., defined as a constant string or loaded from a trusted configuration file). We describe our bidirectional command composition analysis for identifying intended commands in Section 4.1.1. Then, we instrument the target program to randomize intended commands. Figure 1-(c) shows the instrumented program. At line 9, as "`wget`" is the intended command (because it is a constant string), it is instrumented with "`rand()`". Note that `$url` that includes an injected command "`{rm,./*}`" is not instrumented because it is originated from an untrusted source (`$_POST['url']`).

```
1 "Constant"      :"String",
2 "Configuration":["/var/www/html/wp/wp-config.php",
                   "/var/www/html/wp/wp-config-sample.php",
                   "/etc/php5/cli/php.ini"],
3 "API"           :["ini_get"]
```

(a) For WordPress

```
4 "Constant":"String",
5 "API"     :["getenv"]
```

(b) For Composer

```
6 "Constant" :"String",
7 "Directory":["/var/www/html/cgi-bin"]
```

(c) For Lighttpd

Figure 2: Trusted Command Specification Examples

4 DESIGN

Figure 3 shows the workflow of SPINNER with two phases.

4.1 Instrumentation Phase

SPINNER takes a target program to protect and specification of trusted commands as input. It analyzes the target program to identify intended commands to instrument randomization primitives.

Target Program. SPINNER analyzes and instruments target programs' source code. Hence, it requires the target program's source code. Note that we do not require source code of the subsystems (e.g., shell process and database engines).

Trusted Command Specification. Another input that SPINNER takes is the trusted command specification which is a list of trusted source *definitions* as shown in Figure 2. We provide a semi-automated tool to derive the trusted command specification as well (Details can be found in Appendix 9.1.7). There are four types of trusted source definitions: (1) a constant string containing known command names such as hard-coded command (Lines 1, 4, and 6), (2) a path of a configuration file that contains definitions of trusted commands (Line 2), (3) a path of a folder where all the files in that folder are trusted (Line 7), and (4) APIs that read and return values from trusted sources (Lines 3 and 5).

The first type represents a command in a constant string. We consider a hard-coded command is *an intended command by the developer*. The second type is to handle a command defined in the program's configuration file. For example, a PHP interpreter can execute other applications (e.g., `sendmail` for `mail()`) which is defined in `php.ini`. We include the `php.ini` file in our analysis as shown in Figure 2 at line 2. The third type is a folder. It is to define all files under the folder to be trusted. For instance, a web-server may trust all the CGI (Common Gateway Interface) programs found at

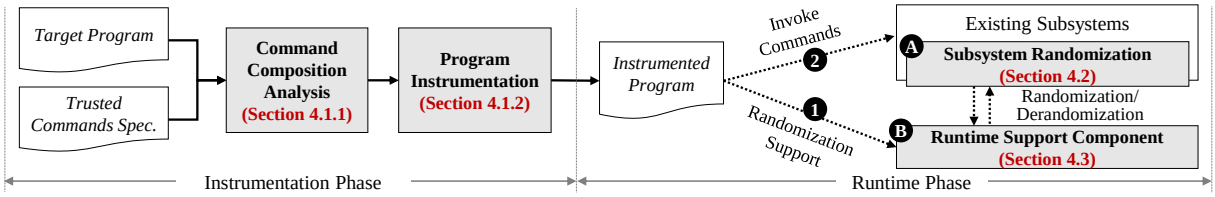


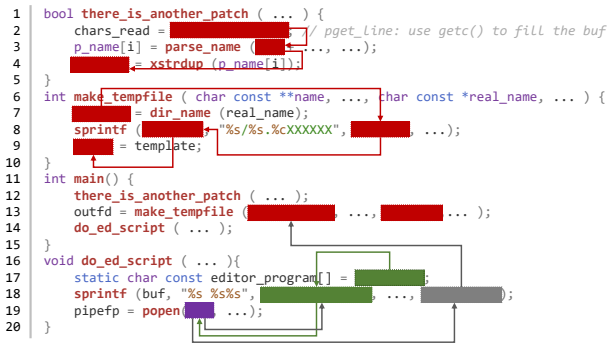
Figure 3: Overview and Workflow of SPINNER (Design details are presented in the annotated sections)

the time of offline analysis with a configuration shown in Figure 2 at line 7. The fourth type describes APIs that read trusted sources. For instance, `getenv()` returns values of local environment variables. If a user assumes that the local environment variables cannot be modified at runtime, it can add the API to the trusted sources.

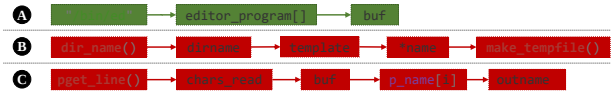
For most applications, specifying the first type (i.e., hard-coded commands) as a trusted source is sufficient. For some applications, configuration files may define trusted commands. In such a case, the command specification should include the trusted configuration files' file paths so that, at runtime, commands originated from the configuration file are trusted. Note that SPINNER checks whether a trusted source (e.g., configuration file) can be modified by remote users. If there is any data flow between untrusted sources and trusted sources, SPINNER notifies the users to redefine the trusted command specification. Similar to the values from configuration files, values from databases are trusted, only if there is no data-flow from untrusted sources to the database. We define *untrusted sources* as any sources *controlled by remote users (i.e., potential attackers)*. Further, to prevent modifications of trusted sources, we hook APIs that can change the trusted sources (e.g., `setenv()`) to make them read-only and detect attempts to modify during our evaluation.

4.1.1 Command Composition Analysis. Analyzing data-flows from the trusted command definitions (i.e., sources) to command execution APIs (i.e., sinks) is a challenging task, particularly for complex real-world applications. A naive approach that uses forward analysis (e.g., taint analysis) from trusted sources to sinks often leads to the over-approximation (i.e., over-tainting), resulting in instrumenting variables that are not relevant to the commands (i.e., false positives) hence breaking benign functionalities. On the other hand, backward data-flow analysis from the sinks (i.e., tracing back the origins of variables from arguments of command execution APIs) to trusted sources is also difficult due to the complicated data dependencies. After analyzing challenging cases from both analyses, we realize that combining two analyses can significantly reduce their limitations (i.e., over and under-approximations).

Bidirectional Command Composition Analysis. We propose a bidirectional analysis, which is the key enabling technique that makes SPINNER effective in analyzing complex data-flow in real-world applications. Specifically, we conduct forward data flow analyses (1) from trusted sources to identify variables holding trusted commands and (2) from untrusted sources to identify variables that are not relevant to commands. SPINNER *automatically* derives the definitions of untrusted sources: (a) return values of APIs that are not included in the trusted command specifications (e.g., `gets()`) and (b) constant strings that do not contain command names. From the forward data flow analysis, we obtain two sets of variables: a set of trusted variables and another set of untrusted variables.



(a) Source code of GNU-Patch executing a command with `popen()`



(b) Forward Data Flow Analysis Results

(c) Backward Data Flow Analysis and Bidirectional Analysis Results

Figure 4: Bidirectional Analysis on GNU-Patch [66]

Next, we conduct a backward data flow analysis from the arguments passed to command execution APIs (e.g., `system()`). While we analyze the program to trace back the origin of the arguments, if we encounter a node originated from a variable in the trusted variable set, we conclude the argument is an intended command hence instrumented. If it meets a node originated from a variable from the other set, untrusted variable set, we stop the backward analysis and conclude that the argument is not relevant to the command.

Running Example. Figure 4-(a) shows a real-world program GNU-Patch [66]'s code snippet consisting of 4 functions. At line 19, it calls `popen()` which executes a shell command composed at line 18. The `buf` variable contains the composed command from `editor_program` and `TMPOUTNAME` via `sprintf()`. First, `editor_program` is defined as a constant string containing a known binary program path in line 17, meaning that it is an intended command and hence instrumented. Second, `TMPOUTNAME` is defined through multiple functions. It is used as an argument of `make_tempfile()` function (line 13). Inside the function, it is defined by `sprintf()` where its value is originated from the `dir_name()` function. As described, there are multiple functions involved to define the value `TMPOUTNAME`, making it difficult to trace back the origin.

1) Forward Analysis: Figure 4-(b) shows the results of our forward data flow analysis from the trusted/untrusted sources. Ⓐ shows the data flow graph from the trusted source. It shows the `/bin/ed` command is propagated to `buf`. Ⓑ and Ⓒ show two graphs from

untrusted sources. Specifically, Ⓑ shows that the return value of `dir_name()` (line 7) is propagated to `*name` (line 9), affecting the first argument of the `make_tempfile()`. As a result, the graph include the `make_tempfile()` as a node, meaning that the function's return values are untrusted and not intended commands. Ⓒ also shows that the values from untrusted source `pget_line()` (reading inputs from the standard input) are propagated to `outname`.

2) Backward Analysis: Figure 4-(c) shows our backward analysis from the sink function: `popen()`. Table 1 shows sink functions (i.e., Command Execution APIs) for each command subsystem. From the argument of `popen()`, `buf`, we analyze how the argument is composed. First, it identifies `editor_program[]` is concatenated via `sprintf()` at line 18. Second, it finds out that `TMPOUTNAME` is a part of the command and it is defined by `make_tempfile()`, which can be found in the forward data flow analysis result Ⓑ.

3) Connecting Forward and Backward Analysis Results: The bidirectional analysis merges results from forward and backward analysis together as shown in Ⓓ and Ⓔ. Note that our backward analysis will terminate when it reaches any nodes in the forward data flow analysis results. This effectively reduces the complexity of the data flow analysis. Typically, the forward analysis is mostly localized and the backward analysis quickly reaches nodes in the forward analysis results. Note that SPINNER conducts inter-procedural analysis if function arguments (e.g., `name` at line 9) or global variables (e.g., `TMPOUTNAME` at line 18) are affected.

Table 1: Command Execution APIs (Sink Functions).

Type	Function	Lang.
Shell	`exec()`[1], `system()`, `popen()`	C/C++
	`passthru()`, `system()`, `popen()`, `shell_exec()`, `exec()`, `proc_open()`	PHP
	`os.execute()`, `io.popen()`	Lua
	`spawn()`[2], `exec()`[2], `execFile()`[2]	JS
XML	`xmlParseFile()`, `xmlParseChunk()`	C
	`simplexml_load_file()`, `simplexml_load_string()`, `xpath()`, `xml_parse()`	PHP
Database (MySQL)	`mysqli::multi_query()`, `mysqli::prepare()`	PHP
	`mysql_query()`, `mysql_real_query()`	C/C++
Database (SQLite)	`sqlite_query()`, `sqlite_exec()`	PHP
	`sqlite3_prepare()`, `sqlite3_exec()`	C/C++

[1]Including `exec()`, `execvpe()`, `execvp()`, `execv()`, `execlp()`, `execle()`, `execl()`, `execve()`.
[2]Including `spawnSync()`,`execSync()`, `execFileSync()`.

Algorithm. Alg. 1 shows our bidirectional data flow analysis algorithm for identifying variables used to create commands.

– *Step 1. Bidirectional Analysis:* BIDIRECTIONALANALYSIS takes a set of functions of a target program F_{set} as input. Then, it conducts the forward analysis (lines 3-8). Specifically, for each variable (lines 3-4), if a variable is from a trusted source (line 5), it creates a tree that describes dependencies between variables as shown in Figure 4-(b)-Ⓐ. The return value is the root node of the tree and it is passed to FORWARDANALYSIS (line 6). Similarly, we also build trees for untrusted sources (lines 7-8) as shown in Figure 4-(b)-Ⓑ and Ⓒ. Next, it starts the backward analysis (lines 9-14). In each function and each statement (lines 9-10), it searches for invocations of sink functions (line 11). For each identified sink function, we obtain variables used as arguments of the function (line 12). For each argument V_i, we call the BACKWARDANALYSIS (line 14) that identifies the commands that need to be instrumented.

Algorithm 1: Bidirectional Analysis for Instrumentation

```
Input: F_set: a set of functions in a target program.
Output: Ins_out: a set of variables to instrument.
1  function BidirectionalAnalysis(F_set)
2      Ins_out ← {}
3      for ∀F_i ∈ F_set do
4          for ∀v_i ∈ F_i do
5              if v_i is from a trusted source then
6                  ForwardAnalysis(CreateDepTree(v_i, F_i, TRUSTED), v_i)
7              else if v_i is from an untrusted source then
8                  ForwardAnalysis(CreateDepTree(v_i, F_i, UNTRUSTED), v_i)
9      for ∀F_i ∈ F_set do
10         for ∀S_i ∈ F_i do
11             if S_i is a sink function then
12                 V_args ← ARGS(S_i)
13                 for ∀V_i ∈ V_args do
14                     BackwardAnalysis(V_i, F_i)
15     return Ins_out
16 procedure ForwardAnalysis(T_cur, V)
17     for ∀V_use ∈ GetUses(V) do
18         if V is an argument x of a function F then
19             ForwardAnalysis(AppendNode(T_cur, F), x)
20         else if V is a variable in an assignment 'x = expression' then
21             ForwardAnalysis(AppendNode(T_cur, x), x)
22 procedure BackwardAnalysis(V, F)
23     if V is a command from a trusted source then
24         Ins_out ← Ins_out ∪ V
25     else if V is from an untrusted source then
26         return
27     else
28         V_defs ← GetDefVars(V)
29         for ∀V_i ∈ V_defs do
30             if V_i ∈ ARGS(F) then
31                 F_callers ← GetCallers(F)
32                 for ∀F_i ∈ V_callers do
33                     BackwardAnalysis(GetCallerArg(V_i, F_i), F_i)
34             else if V_i is a global variable then
35                 V_gdefs ← GetGlobalDefVars(V_i)
36                 for ∀V_j ∈ V_gdefs do
37                     BackwardAnalysis(V_j, GetContainingFunc(V_j))
38             else
39                 BackwardAnalysis(V_i, F)
```

– *Step 2. Forward Analysis:* Given a variable V, it enumerates all the statements that use V via the GETUSES function, which returns the results of the standard def-use analysis [72, 141]. For each statement that uses V, if it is used as an argument x of a function call F (line 18), we add the function as a node to the tree (T_{cur}) via APPENDNODE which returns a subtree where the added node is the root of the subtree. It continues the analysis by recursively calling FORWARDANALYSIS with the subtree and the variable x (line 19). If V is used in an assignment statement 'x = *expression*', where *expression* contains V, it adds the node x to the tree, and call FORWARDANALYSIS with the subtree and x (lines 20–21).

– *Step 3. Backward Analysis:* From a variable V and a function F containing V, BACKWARDANALYSIS identifies variables that are used to compute the value of V recursively (lines 23–39). For the identified variables, it checks whether the variable is a command and is from a trusted source (i.e., it is found during the trusted forward analysis results) (line 23). If so, the variable is added to Ins_{out}, which is a set

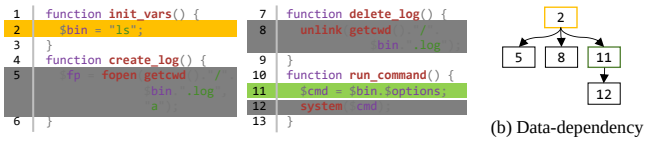
that contains variables to be instrumented. If the variable can be found in the untrusted results, it terminates (lines 25–26).

If V is also computed from other variables (e.g., V=V_x+V_y), we also find origins of the contribution variables (e.g., V_x and V_y) (line 28). Specifically, GetDefVars(V) returns such contributing variables at the last definition of the variable V (e.g., V_x and V_y). The contributing variables are stored in V_{defs}. We check the variable's type of each variable V_i in V_{defs}. If it is an argument of the current function, we extend our analysis into the caller function. GetCallers returns all of them. To find out the corresponding variable passed to the function in the caller, we use GetCallerArg(V_i). Then, we continue the analysis in the caller function F_i (lines 32–33). If V_i is a global variable, it searches all statements that define the variable, then get rvalues of the statements via GetGlobalDefVars (line 35). It recursively calls BackwardAnalysis to extend the analysis on the functions defining the global variable via GetContainingFunc (line 37). Lastly, if V_i is a local variable (line 38), it recursively conducts backward analysis (line 39).

Inter-procedural Analysis. We build a call graph [131] of a target program for inter-procedural analysis. In Alg. 1, GetCallers uses the call graph. After our intra-procedural analysis, we leverage the call graph to identify callers and obtain backward slices from them. We repeat the analysis until there are no more callers to analyze.

Indirect Calls. Call targets of indirect function calls are determined at runtime. As they are not included in the call graphs we generate, they may cause inaccurate results. To handle this problem, given an indirect call, we conservatively assume that the call target can be any functions in the program that have the same function signature (e.g., number of arguments and types). However, as this is a conservative approximation, we may include more callers. To mitigate this, we check the origins of the variables passed to callee functions. If the origins are not relevant to commands (i.e., they are not passed to command execution APIs), we prune out the caller.

4.1.2 Program Instrumentation. We instrument the variables identified in the previous section (Section 4.1.1).



(a) Source Code

(b) Data-dependency Graph for $bin

Figure 5: Command used in Multiple Places

Avoiding Instrumenting Non-command Strings. If an instrumented variable is used in other contexts that do not execute commands, it could break the benign execution. Figure 5 shows an example. The sink function, `system()`, executes `$cmdline`, which is composed by concatenating `$bin` and `$options` (line 11) where `$bin` at line 2. Our analysis described in Section 4.1.1 will attempt to instrument `"ls"` at line 2, adding a randomization primitive to the definition of the command: "`$bin = rand("ls")`". In an original execution, "`ls.log`" file is created at line 5 and unlinked at line 8. However, the instrumentation at line 2 will change the file name to a randomized name. For instance, if the randomized name is "`mt`" (e.g., `ls` ↦ `mt`), the instrumented program will create and unlink "`mt.log`", which is different from the original program.

To solve this problem, we leverage dependency analysis to find a place to instrument that does not affect the other non-command execution APIs (e.g., `fopen()` and `unlink()` at lines 5 and 8). Specifically, we obtain a data-dependency graph, as shown in Figure 5-(b). Nodes are statements in line numbers, and edges between the nodes represent the direction of data flow. From a target variable for instrumentation (`$bin`), we identify statements that use the target variable. If we instrument at the root node (`$bin`), it affects all the child nodes, including those with non-randomized functions (lines 5 and 8). Hence, among the nodes between the root node and the node including the `system()` function (line 12), we pick the node line 11 to instrument. This is because instrumenting at line 11 only affects the command execution API `system()`. Essentially, from the root node, we pick a child node along the path to the sink function. We move toward the sink function until the picked node's children do not include any non-randomized functions.

4.2 Runtime Phase

4.2.1 OS/Shell Command Processor Randomization. We randomize the OS/shell command processor by hooking two critical paths of the command execution: (1) the creation of the shell process and (2) file I/O and shell APIs that access external binary files in the shell process. Recall that there are two types of OS/shell commands: internal and external [89]. For all commands, a program spawns a shell process (e.g., '`/bin/sh`'). The shell process, which contains the implementation of internal commands, directly executes internal commands (e.g., `cd`). External commands are executed by further calling APIs (e.g., `execve`) that run an external program.

Figure 6 shows how Spinner randomizes internal and external commands, following the typical execution flows. To execute an OS/shell command, the program often composes a command via string operations. If a command is composed of trusted inputs, the command names are randomized via the instrumentation (❶). Commands originated from the untrusted inputs are not randomized (❷). The composed command is then passed to the command execution APIs such as `system()`. In the following paragraphs, we explain how Spinner works after the command execution APIs are called depending on whether the command is internal or external.

Internal Commands. To execute an internal command, an application calls a command execution API, which spawns a shell process (❸) and passes the command to the spawned process. As the internal commands are implemented within the shell process, it does not make further API calls to access external binary files.

External Commands. After the shell process is spawned (❸), if the command is an external command (❺), the shell process calls a few files I/O APIs such as `stat()` to check whether the binary file for the command exists or not (❻). If the binary exists, it will execute the binary (❼). We provide a randomized view of the underlying file system by hooking file I/O and shell APIs and only allowing access with properly randomized file paths. If the command is not randomized, API calls such as `stat()` will fail, preventing the execution of the command. A randomized command is derandomized and executed via APIs such as `execve()` (❼).

4.2.2 Database Engine Randomization. Database engines are complicated and some are proprietary (i.e., closed source), meaning

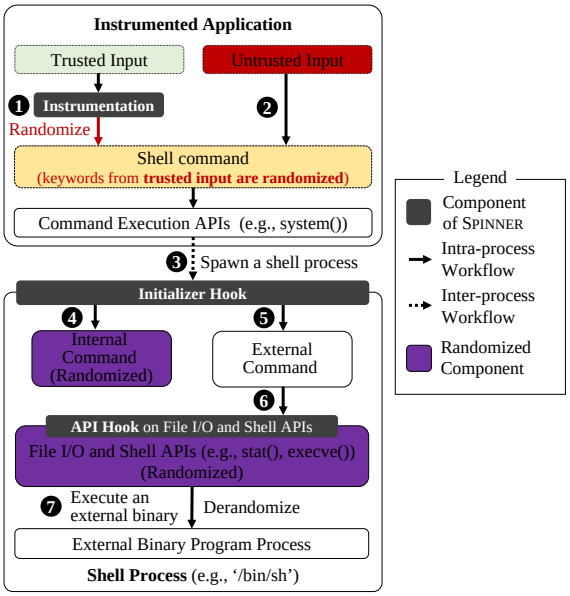


Figure 6: OS/Shell Command Processor Randomizer

that it is difficult to randomize them in practice. As a result, previous approaches (e.g., [27]) leverage a database proxy to parse a randomized query and rewrite it to a standard (i.e., derandomized) query. Implementing a robust parser for multiple database engines is challenging as shown in Section 5.4.1. Moreover, they rely on a list of known SQL keywords to randomize and derandomize, failing to prevent sophisticated attacks presented in Section 5.4.2.

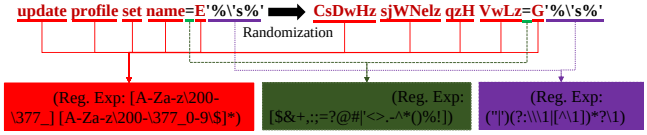


Figure 7: Scanner Recognizing Words for Randomization

Bidirectional Randomization with Scanner. We propose a *bidirectional randomization approach* that applies the randomization scheme twice, one for randomization and another for reverse-randomization. Unlike existing techniques requiring knowledge of known SQL keywords and grammar, SPINNER uses a scanner that works without such knowledge. As shown in Figure 7, SPINNER only needs patterns of words, special characters, and strings. For each identified word, it (1) derandomizes randomized intended queries and (2) randomizes (and breaks) injected malicious queries at the same time. Specifically, in a program that accesses a database, SPINNER instruments strings that are used to compose a SQL query as shown in Figure 8 (❶). Note that untrusted inputs are not randomized by this instrumentation (❷). Finally, randomized trusted inputs and untrusted inputs are combined to compose a query and then passed to a SQL API such as `mysql_query()`. We hook such SQL APIs to apply our reverse-randomization (or derandomization) scheme before the query is passed to the database engine (❸). We apply it for every recognized term (*not only for the SQL keywords* because our scanner does not have the notion of known SQL keywords), resulting in derandomizing all the randomized terms as well as randomizing (with the reverse-randomization scheme) SQL queries from untrusted inputs. To this end, if all terms in a SQL query are from the trusted sources, the resulting query can be successfully executed. However, if some terms are from the untrusted inputs, they are randomized (via the reverse-randomization) and cannot be executed, preventing injection attacks.

– *Handling Escaping String Constant:* Note that Figure 7 shows an example of PostgreSQL's unique feature of escape string constant, which is a special way of defining a string with a capital letter 'E' before a string. Since it is a unique grammar for PostgreSQL, many parsers [24, 81, 102, 148, 160] do not support it, resulting in a parsing error. SPINNER considers the 'E' as a word, and randomize/derandomize correctly, preserving content in the string.

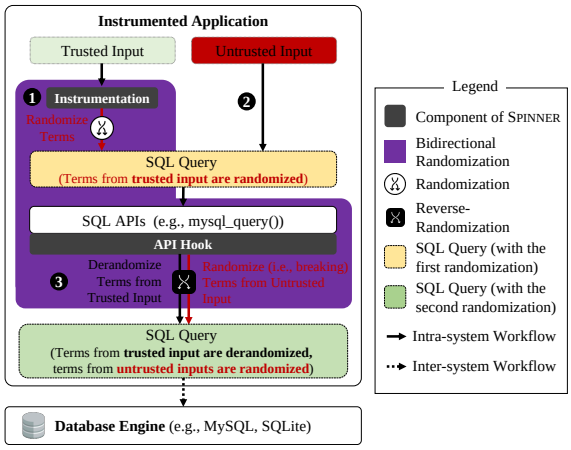


Figure 8: Randomization for Database Engines

Execution of Intended SQL Queries. Figure 9-(a) shows examples of how a benign SQL query is processed by SPINNER with a randomization scheme shown in Figure 9-(c), along the execution path of Figure 8. Specifically, when an intended SQL statement is executed, it goes through the instrumentation (❶) hence randomized and then is passed to a SQL API. The randomized query is shown in the second row of Figure 9-(a). All recognized terms, including the table name '`users`', are randomized. Then, in our hook function of the SQL API, we apply our reverse-randomization for all terms. Since there are no terms from untrusted input, every term is derandomized (❸), as shown in the third row. The last column of Figure 9-(a) shows whether the query can be executed without errors or not. The query after ❸ is executable.

Execution of Injected SQL Queries. An injected SQL query is not randomized because it is not instrumented (❶). Figure 9-(b) shows an example query with an injected query highlighted in red. Specifically, assume that `select * from users where id='$id'` is the vulnerable SQL query, and an attacker injects a query by providing a value highlighted in Figure 9-(b) to `$id`. When the query is passed to the SQL APIs, the beginning part of the query (from trusted inputs up until the single quote) is randomized, but the later part outside the quotes (i.e., the injected query) is not. On the hooked API, SPINNER applies the reverse-randomization in every term we recognize. As a result, it effectively *derandomizes* the (trusted) beginning part of the query while *randomizing* the later part of the query from untrusted inputs. Note that the reverse-randomization applies the substitution rule in the reverse order (i.e.,

	Query	Executable
Original	**select * from users where id**='123'	Yes
After ❶	**xoyozi * lvrj kxovx teovo na**='123'	No
After ❸	**select * from users where id**='123'	Yes

(a) Benign SQL Query Example

	Query	Executable
Original	**select * from users where id**='**1' and exec proc** ;'	Yes
After ❶	**xoyozi * lvrj kxovx teovo na**='**1' and exec proc** ;'	No
After ❸	**select * from users where id**='**1' drl hshu tooa** ;'	No

(b) Injected SQL Query Example (Injected Query is Highlighted)

Org.	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	…	r	s	t	u	…	w	…
Rand.	u	s	z	a	o	l	m	e	n	p	b	y	j	g	r	c	…	v	x	i	k	…	t	…

(c) Randomization Scheme
(Randomization: Org. → Rand., Reverse-Randomization: Rand. → Org.)

Figure 9: Example of Benign and Injected Query Execution

Randomized ↦ Original). For example, '`a` ↦ `f`' is a randomization rule of a reverse-randomization '`f` ↦ `a`'. After ❸, the injected query is prevented as it is reverse-randomized.

Randomized Table Name Translator. The bidirectional randomization scheme randomizes all terms that are not originated from trusted sources. As a result, we observe that if a table name in a SQL query is originated from untrusted sources without quotes (e.g., '`select * from $input`'), the table name can be randomized, resulting in a wrong query. While using input as a table name is a poor programming practice, there exist programs composing queries in that way. To this end, we additionally instrument `tbl_derand()` to the variables that are not quoted. At runtime, it will check whether the instrumented string contains a randomized table name. *If and only if it contains a single table name*, we derandomize it to the original table name. Note that it does not derandomize if the instrumented string contains multiple terms (i.e., words) to prevent injection attacks targeting the instrumented variables.

4.2.3 XML Processor Randomization. An XML processor is a program or module that parses an input XML file and executes the annotated actions in parsed XML elements described via tags.

XML External Entity (XXE) Attack. Among the entities, there is an XML External Entity (XXE) which refers to data from external sources (e.g., other files or networks). The entity can refer to a sensitive password file using the following entity: "`<!ENTITY xxe SYSTEM "file://FILE">`". An XML processor parses the entity, then it reads to include the content of '`FILE`' in the output.

XML Processor Randomization. We randomize *external resources' namespaces* such as file names and network addresses at APIs that access them (e.g., file I/O APIs and network APIs). With the randomization, only the API calls with randomized file names, paths, IPs, and URLs will succeed. To ensure benign requests are properly handled, we analyze a target program to identify all intended XML files. Specifically, we identify XML files and data passed to the XML sink functions shown in Table 1. If they are originated from trusted sources (e.g., constants or from configuration files), we mark them to be randomized at runtime. At runtime, when a trusted XML file is loaded, we randomize resource names/paths of XXEs in the file. As we randomized the namespaces in the application through APIs, intended accesses through the randomized XXEs will be successful. For untrusted XMLs, file names, paths, and URLs in XXEs are not randomized and passed to the file I/O and network APIs, resulting in errors and preventing injection attacks. Note that when Spinner analyzes the program to identify trustable XML files, we assume all the local XML files during the offline analysis are not compromised. Spinner's goal is to prevent future XXE injection after the analysis.

4.3 Spinner Runtime Support

4.3.1 Dynamic Randomization Support. Spinner randomizes commands in the subsystems at runtime dynamically. We change our randomization scheme (or table) on every command execution function invocation (or per input) so that knowing previously used randomization schemes will not help subsequent attacks.

4.3.2 Randomization Primitives. The runtime support provides two primitives: randomization and derandomization primitives.

– *Randomization Primitive* is a function that takes a string as input and returns a randomized string via a mapping between each byte in the input and randomized byte(s). To mitigate brute-force attacks against the randomized commands, the mapping is created per input. It also supports multiple randomization schemes that convert 1 byte to 2 bytes ('`x`' ↦ '`ab`'), 4 bytes ('`x`' ↦ '`cdef`'), and 8 bytes ('`x`' ↦ '`ghijklmn`'). Details can be found in Appendix 9.3.3.

At runtime, we maintain a pair of a randomized string and its randomization table, which we call *randomization record*. The record is later used in the derandomization function. Note that two different strings can be randomized into the same string with two different one-time pads. For example, a one-time pad '`a` ↦ `c`' and another one-time pad '`b` ↦ `c`' will randomize both '`a`' and '`b`' to '`c`', leading to the ambiguity in derandomization. To solve this problem, when it randomizes, it checks whether the randomized string exists in the existing randomization records. If it exists, it randomizes the input string again until there are no matching strings in the records.

– *Derandomization Primitive* takes a randomized string as input and returns the original value of the string. Given a list of randomization records, it finds a record that has a matching randomized string. Then, it leverages the record's randomization scheme to derandomize the input string.

5 EVALUATION

Objectives. We evaluate Spinner on four aspects. First, we present analysis results on the instrumented code and its impact to show the correctness of Spinner (Section 5.1). Second, we run PoC exploits against a set of vulnerable programs and their Spinner instrumented versions, to show the effectiveness of Spinner in preventing command injection attacks (Section 5.2). Third, we measure the performance overhead of Spinner (Section 5.3). Fourth, we present case studies to show the effectiveness of Spinner in advanced command injection attacks (Section 5.4).

Implementation. We implement our static analysis tool by leveraging LLVM [4] for C/C++, php-ast [108] and Taint'em All [125] for PHP, Acorn [92] for JavaScript and Lua SAST Tool [32] for Lua. Spinner uses `LD_PRELOAD` that requires access to the shell. Hence, it does not support a web hosting service such as cPanel [33].

Setup. All the experiments were done on a machine with Intel Core i7-9700k 3.6Ghz, 16GB RAM, and 64-bit Linux Ubuntu 18.04.

Program Selection. We search publicly known input injection vulnerabilities (including SQL and XXE injections) in recent five years. Among them, we reproduced 27 vulnerabilities and used the

Table 2: Selected Programs for Evaluation and Instrumented Results

ID	Name	Size	Vulnerability	Language(s)	# Instrumentations					# Instr. Affecting			# Affected Vars./Funcs.						Dep. Analysis	
					Const.	Dynamic			Sinks	BB[1]	Fn[2]	Fns[3]	Local[4] (Total)		Global[5] (Total)		Funcs[6] (Total)		Forward	Backward
						1-5	6-10	>11												
s1	WordPress [65]	42.60 MB	Cmd.[7] [35]	PHP	38	279	127	18	7	3	1	458	11.39	(178)	2.95	(15)	7.04	(90)	10.2$^{\alpha}$	6.9$^{\beta}$
s2	Activity Monitor [57]	0.99 MB	Cmd.[7] [38]	PHP	6	12	9	0	6	2	4	21	9.89	(27)	2.77	(2)	7.53	(34)	7.7	6.3$^{\gamma}$
s3	AVideo-Encode [163]	8.93 MB	Cmd.[7] [110]	PHP	2	48	8	3	27	3	37	21	0.98	(63)	0	(0)	1.36	(79)	1.7	6.7
s4	Pepperminty-Wiki [143]	23.00 MB	XXE[8] [36]	PHP†	0	2	0	0	2	0	2	0	0	(2)	0	(0)	1	(2)	1	1
s5	PHPSHE [2]	11.91 MB	XXE[8] [43]	PHP‡	54	183	26	7	5	54	0	236	3.82	(96)	0	(0)	3.76	(67)	5.7	3.6
s6	Pie Register [139]	5.51 MB	SQL[9] [37]	PHP*	0	68	5	0	2	0	0	73	3.06	(26)	3	(3)	4.28	(27)	6.3	7.2$^{\delta}$
s7	Lighttpd [88]	17.40 MB	SQL[9] [34]	C	5	4	0	1	10	5	5	0	0.5	(5)	0	(0)	0.5	(5)	1.6	7.3
s8	Leptonica [23]	24.10 MB	Cmd.[7] [41]	C	0	0	0	2	2	0	2	0	2	(2)	0	(0)	2	(2)	2.4	12.1
s9	GNU-Patch [66]	4.96 MB	Cmd.[7] [42]	C	0	0	0	7	2	0	1	6	1.00	(6)	0.86	(5)	3.57	(1)	4.9	10.2
s10	Goahead [60]	18.20 MB	Cmd.[7] [47]	C	0	0	0	1	1	0	1	0	1	(1)	0	(0)	1	(1)	3	9
s11	LuCI [112]	43.10 MB	Cmd.[7] [52]	C, Lua	19	102	17	2	52	19	37	84	2.24	(136)	0	(0)	1.96	(132)	2.4	6.4
s12	jison [166]	1.25 MB	Cmd.[7] [45]	JS§	0	2	0	0	2	2	0	0	0	(0)	0	(0)	0	(0)	1	3
s13	Kill-port [73]	34.80 KB	Cmd.[7] [129]	JS§	0	5	0	0	2	5	0	0	0	(0)	0	(0)	0	(0)	1	4.5
s14	egg-scripts [58]	58.40 KB	Cmd.[7] [49]	JS§	2	1	0	0	3	3	0	0	0	(0)	0	(0)	0	(0)	1	3.3
s15	node-df [7]	40.00 KB	Cmd.[7] [104]	JS§	0	1	0	0	1	0	1	0	1	(1)	0	(0)	1	(1)	1	2
s16	PM2 [144]	4.42 MB	Cmd.[7] [126]	JS§	7	25	2	1	34	21	23	2	0.77	(21)	0	(0)	1.95	(31)	1.3	5.3
s17	fs-git [95]	130.00 KB	Cmd.[7] [46]	JS§	0	1	0	0	1	0	0	1	1	(1)	0	(0)	2	(2)	2	4
s18	Meta-git [157]	262.00 KB	Cmd.[7] [98]	JS§	0	1	0	0	3	0	0	1	2	(2)	0	(0)	2	(2)	1	5
s19	Listening Process [96]	131.00 KB	Cmd.[7] [109]	JS§	0	3	0	0	3	3	0	0	0	(0)	0	(0)	0	(0)	1	3
s20	NPM lsof [54]	18.00 KB	Cmd.[7] [51]	JS§	0	3	0	0	3	3	0	0	0	(0)	0	(0)	0	(0)	1	2
s21	NPM opencv [121]	22.60 MB	Cmd.[7] [50]	JS§	1	2	0	0	3	3	0	0	0	(0)	0	(0)	0	(0)	1	2.3
s22	logkitty [118]	514.00 KB	Cmd.[7] [44]	JS§	0	2	0	0	2	0	0	2	2	(3)	0	(0)	2.5	(4)	3	3
s23	gitpublish [28]	32.00 KB	Cmd.[7] [99]	JS§	0	9	0	0	3	2	0	7	2	(8)	0	(0)	2	(8)	1	5.7
s24	codecov [19]	290.00 KB	Cmd.[7] [53]	JS§	4	0	2	0	6	4	2	0	0.5	(3)	0	(0)	0.5	(3)	1	7.4
s25	pdfinfojs [63]	77.00 KB	Cmd.[7] [48]	JS§	0	3	0	0	3	3	0	0	0	(0)	0	(0)	0	(0)	1	4.3
s26	libnmap [79]	157.00 KB	Cmd.[7] [39]	JS§	0	1	0	0	1	0	0	1	4	(4)	1	(1)	4	(4)	3	4
s27	pdf-image [94]	14.00 KB	Cmd.[7] [40]	JS§	0	1	1	0	2	0	0	2	0	(2)	0	(0)	2	(4)	2	7.5

1: Basic block. 2: Function. 3: Multiple Functions. 4: Local variable (Avg.). 5: Global/member variable (Avg.). 6: Functions (Avg.). 7: Shell Command Injection. 8: XXE Injection. 9: SQL Injection. †: PHP and XML. ‡: PHP, XML, and SQL. ∗: PHP and SQL. §: JavaScript. α: 4 FN (False negative) cases. β: 24 FN cases. γ: 3 FN cases. δ: 2 FN cases. (α, β, γ, δ): No FN cases when we apply the bidirectional analysis. FN cases are caused when only forward or backward analysis is applied alone.

vulnerable programs as shown in Table 2. Note that the versions of the evaluated programs can be found in Appendix 9.1.6 (Table 4). The selected programs are diverse, including popular programs such as WordPress [65] and OpenCV [121]. They are also written in diverse programming languages such as PHP, C/C++, Lua, and JavaScript. The programs and vulnerabilities are on [146].

Input Selection. To obtain realistic test input (or test data) that can cover diverse aspects of the program, we leverage publicly available input data sources. For instance, Leptonica [23] provides 278 test cases with 192 images. Other programs also have developer provided test cases: NPM-opencv [121], fs-git [95], PM2 [144] and codecov [19]. For the programs with less than 100 test cases, we extend them on different inputs with around 100 cases. For the programs that accept PDFs (e.g., pdfinfojs), videos (e.g., Avideo-Encoder), and patches (e.g., GNU-Patch), we crawl more than 100 samples for each type from public websites [3, 105]. To run the programs for the performance evaluation (Section 5.3), we leverage Apache Jmeter [64] and Selenium [138] for web applications. We also use Selenium scripts provided by [15] to simulate requests and interactions for web services such as WordPress. OLPTBench [55], which aims to conduct extensive tests on relational database systems, is used to test diverse SQL queries. In addition, a large publicly available XML data-set (1026 MB total) [100] is used. We also include popular web servers [13, 29, 88, 111], SQL engines [115, 142], and XML libraries [90, 122, 130, 153] in our evaluation.

5.1 Instrumentation Results and Correctness

Table 2 presents the results of our instrumentation in detail.

Statistics. The "Const." and "Dynamic" columns represent the number of completely constant commands and the number of dynamically composed commands with other values respectively. There are three groups based on the number of variables involved in creating a command or query dynamically. The first group includes cases where 1~5 variables are involved, that are trivial to verify that they do not break benign functionalities. Most cases belong to this group. The second and third groups indicate 6~10 and more than 11 variables are involved respectively. We checked them all that they do not break benign functionalities. Examples and details can be found in Appendix 9.2.6. The "Sinks" column represents the number of sink functions identified by SPINNER. Note that while there are applications that require many instrumentations (e.g., 462 for WordPress), most of them are constants or dynamic cases with only a few variables are involved. WordPress is a content management system stores/retrieves contents from databases, PHPSHE is a website builder, and Pie Register is a user registration form service. LuCI is a web interface for configuring OpenWrt [114] that runs various commands in nature. These programs include many SQL queries, leading to a large number of instrumentations. However, patterns of queries in those programs are simple and similar to each other. Further, we analyze the dynamically composed commands.

Most cases are appending file names to base folders to compose paths and adding table names in queries.

Correctness. We run test cases and analyze all the instrumented code to show the instrumented programs' correctness.

– *Testing Instrumented Programs:* To empirically show that our instrumentation does not break the original functionalities, we run test cases that can cover instrumented code and other parts of the program code affected by the instrumentation. We leverage test cases provided by developers of the target applications. If there are no provided test cases or test cases are not sufficient, we manually extend test cases to cover those. All the test cases are presented in Table 7. In total, we run 15,916 test cases for the 27 programs, achieving the average code coverage of 78.17%. For the code that is not covered by the test cases, we manually checked that they are not affected by our instrumentation.

– *Manual Analysis of Instrumentation:* We analyze the impact of our instrumentations and categorize them into three types: instrumentations that affect (1) a single basic block (the BB^1 column), (2) a single function (the Fn^2 column), and (3) multiple functions (the Fns^3 column). The first category only affects statements within its basic block. Mostly, they are the cases where a constant string is instrumented and directly passed to a sink function. For this case, it is trivial to prove that it does not impact the correctness of the program as the impact of the instrumentation is contained within the current basic block. For the second category (i.e., single function), the instrumented values are stored into local variables, but it does not affect other functions (i.e., they are not returned or passed to other functions). Hence, the impact of instrumentation is limited within the function. The last category (i.e., multiple functions) means that the instrumentation affects multiple functions because the instrumented value is stored to a variable shared between functions (e.g., global variable) or passed to other functions as arguments. We verify all the cases in the three categories that they do not break the original functionalities of the target programs by tracing dependencies caused by our instrumentations. Details with example code for the three categories are in Appendix 9.2.6.

We also inspect local and global variables and functions affected by the instrumentations. In the next three columns, the average number of variables/functions affected by each instrumentation is presented, followed by the total number of variables/functions affected in the entire program. The average number of variables and functions affected per instrumentation is not large: less than 12 local variables, 2 global variables, and 8 functions. We verify all of them and Spinner does not break the benign functionality.

5.2 Effectiveness

5.2.1 Against PoC (Proof of Concept) Exploits. We reproduce 27 PoC exploits on Spinner instrumented programs, as shown in Table 2. The "Vulnerability" column shows attack type (e.g., Command injection, XXE injection, and SQL injection) with citations. All the PoC (Proof of Concept) attacks are successful in the vanilla versions, while prevented in the Spinner protected programs.

5.2.2 Against Automated Vulnerability Discovery Tools. To see whether Spinner can prevent *diverse malicious commands*, in addition to the tested CVEs in Section 5.2.1, we leverage three automated vulnerability discovery tools, Commix [31], sqlmap [20], and xcat [150], to test a diverse set of known malicious commands. We launch 102 command injection attacks, 97 SQL injection attacks, and 24 XXE injection attacks, leveraging the three tools. They essentially brute-force the target programs' inputs using the known malicious commands. Then, they check whether it is vulnerable to command injection attacks. The result shows that Spinner successfully prevents all 223 tested attacks. Details are in Appendix 9.1.2.

5.2.3 Bidirectional Analysis Compared to Backward and Forward Analysis. We apply forward and backward data flow analysis alone to the programs and presented the average length of dependency chain obtained from each analysis in the last two columns of Table 2. We observe that data-flow analysis accuracy, including forward and backward analyses, decreases as the data dependency chain's length becomes larger than 10 in general, causing false negatives. We find such cases in WordPress [65], Activity Monitor [57], and Pie Register [139], marked with α, β, γ, and δ with the red cell background color. We manually verified that all the results from our analysis are true-positives. In particular, we run other static/dynamic taint-analysis techniques [5, 6, 119] and manually verify that the dependencies identified by the existing techniques but not by ours are false-positives. Appendix 9.2.4 and 9.2.5 provide more details including examples and accuracy of the bidirectional analysis.

5.3 Performance Evaluation

Runtime Overhead (Overhead: ≈5%). We measure the runtime overhead of Spinner on the 27 programs in Table 2 as shown in Figure 10. Note that each application has 4 measures as we use 4 different randomization schemes mapping 1 byte to 1, 2, 4, and 8 bytes. In each bar, the bottom black portion represents the overhead caused by creating randomization tables, including those for rerandomization, while the top gray portion is the overhead from the computations for randomization. For each program, we use 100 typical benign test inputs that cover instrumented statements. For each input, we run ten times and take the average. The average overhead is 3.64%, 3.91%, 4.28% and 5.01% for 1, 2, 4, and 8 bytes randomization schemes respectively.

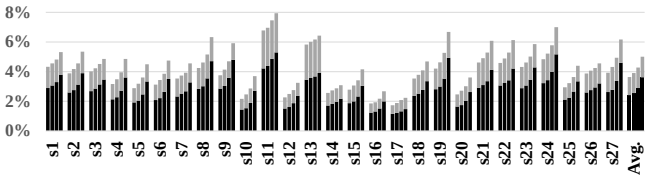


Figure 10: Runtime Overhead

Throughput of Full Stack Web Servers (WordPress). We measure the overhead on throughput of full-stack web services to understand the performance overhead of realistic deployment of Spinner. We applied Spinner to four different web servers: Apache 2.4.41, Lighttpd 1.4.55, Cherokee 1.2.102, and Openlightspeed 1.5.11. We also apply Spinner to SQLite 3.31.0, PHP 7.2, and WordPress 4.9.8, along with the listed vulnerable plugins. Apache Jmeter [64] is used to request 10,000 concurrent webpages, covering various functionalities of WordPress, including posting blogs, changing themes, and activating/deactivating/configuring plugins. The average overhead on throughput is 3.69% (4.33%, 3.76%, 3.18%, and 3.47% overhead on Apache, Lighttpd, OpenLightSpeed, and Cherokee respectively).

Overhead on Database Engines and XML Parsers. We apply SPINNER to SQLite and MySQL and run various SQL queries using data-sets from OLTP-Bench [55]. The result shows that the overhead with SQLite is 4.9% and MySQL is 5.3%. We also measure the overhead on four XML parsers [90, 122, 130, 153]. The average overhead is 1.4% (Details in Appendix 9.1.4).

Memory Overhead. SPINNER needs to maintain randomization tables on memory during execution. Memory overhead for one randomization table is 54 bytes, 106 bytes, 209 bytes and 417 bytes respectively when SPINNER is configured to randomize 1 to 1, 2, 4, and 8 bytes. At runtime, the memory overhead is ~1MB on large programs such as WordPress, with 8 bytes randomization scheme.

5.4 Case Study

5.4.1 Advanced SQL Injections Exploiting Parsers. We present a few sophisticated injection attacks exploiting flaws of 11 popular parsers [9, 24, 25, 76, 78, 81, 91, 102, 107, 148, 160], showing the weaknesses of the parser-based for randomization approaches [27, 120, 140]. All of the cases are successfully prevented by SPINNER, demonstrating the effectiveness of SPINNER.

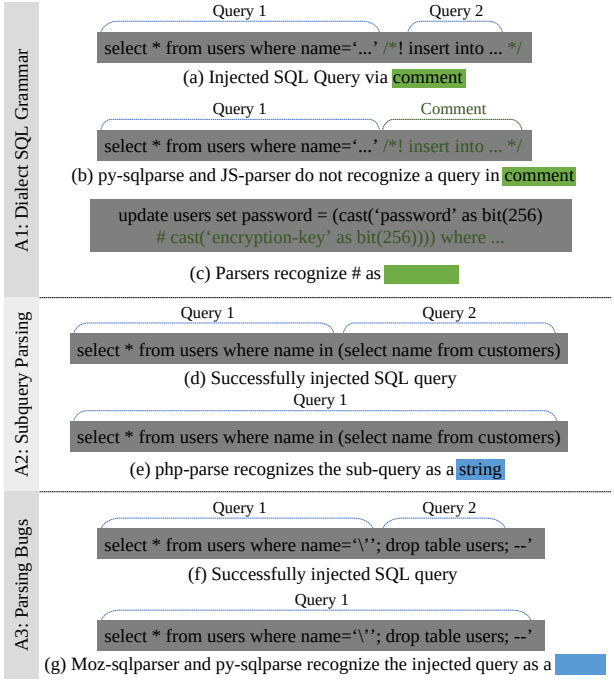


Figure 11: SQL Injections that Parsers Fail to Recognize (Yellow: keywords, Blue: strings)

A1: Dialect SQL Grammar ((a), (b), and (c)). The dialect SQL attack shows that using a parser is an insecure design choice of the existing techniques. For instance, MySQL supports a SQL dialect: if a query in a comment starts with "`/*!`", it can be executed, as shown in Figure 11-(a). However, many parsers do not support this dialect. An attacker can inject a malicious payload inside the comment, exploiting parsers that cannot recognize queries in a comment. We confirmed py-sqlparse [24] and JS-parser [148] fail to recognize injected queries in a comment as shown in Figure 11-(b). In addition, as shown in Figure 11-(c), PostgreSQL [67] considers the '#' symbol as an XOR operator, while others typically consider it as a single line comment operator. An attacker can also inject a malicious query with '#'. Note that some techniques may automatically remove queries after '#', removing injected queries. However, this will break benign queries using # is an XOR operator as shown in as shown in Figure 11-(c): doing a simple XOR encryption on a password.

A2: Sub-query Parsing Error ((d) and (e)). Attackers can inject malicious queries as a subquery to exploit approaches relying on parsers that cannot parse sub-queries correctly. For instance, Figure 11-(d) shows a SQL statement including two queries where the second query is a sub-query. As shown in Figure 11-(e), php-parse [78] parses the entire sub-query as a string. Note that we present a specific case study for this attack type in Section 5.4.2.

A3: String Parsing Error ((f) and (g)). Moz-sqlparser [102] and py-sqlparse [24] have a bug in parsing a string [12], allowing injected queries to be considered as a string that is not a randomization target. For example, Figure 11-(f) shows two SQL queries where the Query 2 is an injected malicious query. [24, 102] mistakenly consider the entire second query as a part of a string (blue marked).

5.4.2 Comparison with Existing Techniques. We compare SPINNER with two state-of-the-art techniques [140, 149].

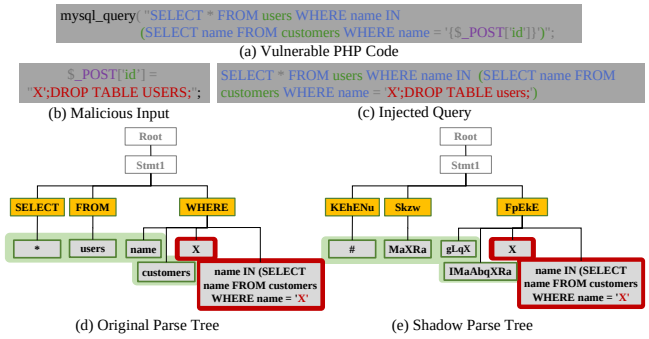


Figure 12: Diglossia with php-parse [78]

Diglossia [140] vs SPINNER. Diglossia runs two parsers, the original parser and the shadow parser, together. The shadow parser is created to use a different language than the original parser and its input is obtained by translating the original input into the other language. At runtime, it obtains two parse trees from the parsers. Different nodes between the trees indicate the parts originated from untrusted sources. If identical nodes are representing keywords (not strings/numbers), it detects an injection attack. In this experiment, we implement our own version of Diglossia using php-parse [78] and SPINNER's randomization scheme for the translation, since Diglossia does not provide its source code.

Figure 12-(a) shows a vulnerable PHP code. Given the malicious input shown in Figure 12-(b), the malicious query is injected as shown in Figure 12-(c). As explained in Section 5.4.1, the parser failed to parse the subquery after the `IN` keyword, resulting in an incorrect tree as shown in Figure 12-(d). The last two children nodes (with red borders) of `WHERE` are unknown type nodes. When the malicious input is injected, both parse trees have the injected query as unknown nodes, resulting in a broken trees. As a result, it failed to recognize injected query. Note that Figure 12-(d) and (e) show

that they have identical nodes, marked with red borders. However, they are considered as literal nodes, hence not considered as an injected code. Worse, while the parser fails to process the query, it does not show error messages but silently suppresses the errors, missing the opportunities to detect the attack. On the other hand, SPINNER successfully prevents the injected SQL query `DROP TABLE users` from being executed. Note that the performance of SPINNER (about 5%) is slightly better than and Diglossia (7.54%).

sqlrand-llvm [149] vs SPINNER. sqlrand-llvm [149] is an implementation of SQLRand using LLVM. It hooks `mysql_query()` to tokenize a randomized input query and compare each token with a list of randomized SQL keywords. It then derandomizes the matched tokens and then pass the deranomized query to the SQL engine.

	Query	Executable
Original	**select * from users where id='1' and exec proc ;'**	Yes
Randomized	**select123 * from123 users where123 id='1' and exec proc ;'**	No
Derandomized	**select * from users where id='1' and exec proc ;'**	Yes

Figure 13: SQL Injection Example with sqlrand-llvm [149]

1) SQL Injection with New Keywords: sqlrand-llvm maintains a list of known SQL keywords and another list of randomized keywords. If a query with a keyword that is not included in the list are injected, it cannot prevent. For instance, as shown in Figure 13, SQL keywords in the original query are randomized by appending `123` to the keywords. During the derandomization process, if it encounters a SQL keyword that is not randomized, it considers the keyword is injected. However, it does not support `exec` and `proc` keywords according to the sqlrand-llvm's source code [149]. As shown in Figure 13, `exec` and `proc` in the injected query (highlighted) are not detected. Note that `exec proc` can execute a stored procedure called `proc`. SPINNER prevents the attack with a similar performance: SPINNER (5%) and sqlrand-llvm (4.13%).

2) Breaking Benign Queries: sqlrand-llvm uses `strtok()` to randomize/derandomize keywords even if they are a part of a string. This results in an error if a string contains a SQL keyword. Consider a query "`select * from users where name='$name'`", where the value of `$name` is '`grant`'. Its randomized query is "`select123 * from123 users where123 name='grant'`". The value `grant` is from the user at runtime, hence not randomized. Unfortunately, `grant` is one of the known SQL keywords used in sqlrand-llvm, meaning that it will detect an injection attack (false positive) because the `grant` is not randomized. SPINNER does not have this issue as it does not randomize string type values.

6 DISCUSSION

Prepared Statements. Prepared statements [103] aim to prevent SQL injections by separating input data from a SQL query during the query construction. While effective, they have limitations. First, some SQL keywords are not supported in prepared statements such as `PASSWORD` and `DESC` (5 more in Appendix 9.3.2). Second, changing existing SQL queries to prepared statements requires manual effort. Note that we manually check 866 SQL queries from all our target programs, and none of them is a prepared statement, showing the needs of SPINNER in practice (Appendix 9.3.2).

Memory Disclosure on Randomization Records. SPINNER maintains randomization records that contain previously used randomization schemes. Attackers who can leak the memory pages containing the records may obtain SPINNER's previously used keys. However, SPINNER chooses a new randomization key on every new input. Hence, knowing previous randomization keys does not help in launching subsequent attacks. Also, existing memory protection techniques [77] can be used to protect the records.

Limitations. When a target application is updated, one needs to run SPINNER to analyze and instrument the updated target application. Typically, this simply requires re-running SPINNER on the updated application. However, if an update significantly changes program code relevant to the trusted commands, it requires manual efforts to redefine the trusted command specifications. Further, we analyze updates of 42 applications, including popular programs, to check whether updates in practice lead to changes in trusted command specifications. The results show that they do not change trusted command specifications. Details are in Appendix 9.3.4. If an application runs a completely dynamic command (e.g., `system("$_GET['cmd']")`, SPINNER blocks it and notifies users to fix the program.

7 RELATED WORK

Runtime Protection of Web Application. There have been many researchers that have proposed runtime protection systems against command and SQL injection attacks [21, 27, 30, 68–71, 106, 123, 137, 145]. Taint tracking techniques track untrusted user inputs in server-side applications at runtime [30, 68, 106, 123, 145]. [69, 137] leverage static analysis to infer possible benign commands and use them to detect injection attacks. CANDID [21] employs dynamic analysis to extract and model an accurate structure of SQL queries. [70] proposes positive tainting that dynamically tracks trusted inputs. Unlike them, SPINNER focuses on randomizing trusted commands, which is more lightweight than existing approaches (e.g., up to 19% overhead in [70]). Diglossia [140] proposes a dual parsing technique that uses different languages during the parsing to detect injected SQL queries. However, it relies on parsers which can be exploited as shown in Sections 5.4.1 and 5.4.2.

Among the existing approaches, SQLRand [27, 120] is the closest work to our approach. It randomizes SQL keywords and uses a proxy that can parse and derandomize the randomized SQL statements. Compared to SQLRand, SPINNER does not rely on parsers which can be attacked and exploited as presented in Section 5.4.1. There are also randomization based techniques such as Instruction Set Randomization [17, 26, 82]. While sharing the randomization idea, SPINNER's design provides solutions for preventing advanced attacks exploiting ambiguous grammars [85], as shown in Section 5.4.1. [26] randomizes a programming language, leveraging a similar method to SQLRand. However, it is vulnerable to attacks exploiting language specification changes across different versions.

Security Analysis of Web Applications/Randomization. Researchers have proposed various techniques to analyze vulnerabilities in web applications [16, 75, 80, 84, 101, 158, 159, 164]. [75] uses static analysis to identify vulnerabilities in PHP applications. Xie et al. [164] propose a symbolic execution based program analysis

technique to find SQL injection vulnerabilities. String-taint analysis [101, 158, 159] tracks untrusted substrings from user inputs to prevent information leak attacks. [16] combines dynamic and static analysis to find vulnerabilities in input sanitizers. SPINNER also uses static taint analysis and data flow analysis. [8] studies the impact of timing of rerandomization. SPINNER rerandomizes subsystems per input event, following the paper's recommendation.

Security Testing for Web Applications. Security testing aims to identify inputs that can expose input validation vulnerabilities in web applications [16, 18, 56, 74, 83, 93, 97, 132, 133]. [74] is a pioneer of web application testing by injecting XSS and SQL attacks. Mcallister et al. [97] propose a guided and stateful fuzzing technique to improve the performance. Doupé et al. [56] propose incrementally building a state machine during crawling to understand the internal structure of the web applications for better web application fuzzing. To enhance input generation efficiency, Martin et al. [93] leverage model checking and static analysis, ARDILLA [83] applies symbolic execution, and Saxena et al. [132, 133] use both dynamic taint analysis and symbolic execution for input mutation space pruning. [127] systematically measures security issues in the payment card industry's webservices. SPINNER aims to provide runtime protection. They are orthogonal to SPINNER and are complementary.

8 CONCLUSION

In this paper, we introduce SPINNER, a randomization based input injection prevention technique. SPINNER is more robust than state-of-the-art randomization techniques. Our extensive evaluation results show that SPINNER successfully prevents advanced attacks with low overhead (<4%). We release our tool's source code and result to public [146].

REFERENCES

[1] Dependency Manager for PHP. https://github.com/composer/composer.
[2] Online Shopping Website Framework. https://gitee.com/koyshe/phpshe.
[3] TED Ideas worth spreading. https://www.ted.com/talks.
[4] The LLVM Compiler Infrastructure Project. https://llvm.org/.
[5] 2020. GitHub - vimeo/psalm: A static analysis tool for finding errors in PHP applications. https://github.com/vimeo/psalm.
[6] abiusx. 2015. Taint Tracking and Inference analysis and breaking tool. https://github.com/abiusx/taintless/.
[7] Adriano D.Giovanni. 2020. A cross-platform Node.js wrapper around the standard Unix program df. https://github.com/adriano-di-giovanni/node-df.
[8] Salman Ahmed, Ya Xiao, Kevin Z Snow, Gang Tan, Fabian Monrose, and Danfeng Yao. 2020. Methodologies for quantifying (Re-) randomization security and timing under JIT-ROP. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1803–1820.
[9] Alibaba. 2020. Generic SQL engine for Web and Big-data. https://github.com/alibaba/nquery.
[10] Muath Alkhalaf. 2014. *Automatic Detection and Repair of Input Validation and Sanitization Bugs.* Ph.D. Dissertation. University of Californida, Santa Barbara.
[11] Anastasionico. 2019. Good Practices: how to sanitize, validate and escape in PHP. https://dev.to/anastasionico/good-practices-how-to-sanitize-validate-and-escape-in-php-3-methods-139b.
[12] Andi Albrecht. 2020. Multiple parsing failures identifying Comment Tokens . https://github.com/andialbrecht/sqlparse/issues/558.
[13] Apache. 2019. Apache Web Server. https://httpd.apache.org/.
[14] Automattic. 2020. Automatically checks all comments and filters out the ones that look like spam. https://wordpress.org/plugins/akismet/.
[15] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1697–1714. https://www.usenix.org/conference/usenixsecurity19/presentation/azad
[16] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. 2008. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *2008 IEEE Symposium on Security and Privacy (S&P 2008)*. 387–401. https://doi.org/10.1109/SP.2008.22
[17] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. 2003. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*. Association for Computing Machinery, New York, NY, USA, 281–289.
[18] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. 2010. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *2010 IEEE Symposium on Security and Privacy*. 332–345. https://doi.org/10.1109/SP.2010.27
[19] Joe Becher. 2019. Codecov NodeJS Uploader. https://www.npmjs.com/package/codecov.
[20] Bernardo Damele A. G. and Miroslav Stampar. 2020. sqlmap. https://github.com/sqlmapproject/sqlmap.
[21] Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. 2010. CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. *ACM Trans. Inf. Syst. Secur.* 13, 2, Article Article 14 (March 2010), 39 pages. https://doi.org/10.1145/1698750.1698754
[22] BitDegree. 2017. Learn PHP Sanitize Input: Example of Input Sanitization Included. https://www.bitdegree.org/learn/php-sanitize-input.
[23] Dan Bloomberg. 2020. Leptonica. http://www.leptonica.org/.
[24] John Bodley. 2020. A non-validating SQL parser module for Python. https://github.com/andialbrecht/sqlparse.
[25] BorseGo AG. 2019. Parse SQL (select) statements into abstract syntax tree (AST) and convert ASTs back to SQL. https://github.com/godmodelabs/flora-sql-parser/.
[26] Stephen W. Boyd, Gaurav S. Kc, Michael E. Locasto, Angelos D. Keromytis, and Vassilis Prevelakis. 2010. On the General Applicability of Instruction-Set Randomization. *IEEE Trans. Dependable Secur. Comput.* 7, 3 (July 2010), 255–270.
[27] Stephen W. Boyd and Angelos D. Keromytis. 2004. SQLrand: Preventing SQL Injection Attacks. In *Applied Cryptography and Network Security*, Markus Jakobsson, Moti Yung, and Jianying Zhou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 292–302.
[28] Frank Lyder Bredland. 2016. git-publish. https://www.npmjs.com/package/git-publish.
[29] Cherokee. 2019. Cherokee is an innovative, feature rich, lightning fast and easy to configure open source web server designed for the next generation of highly concurrent secured web applications. https://cherokee-project.com/.
[30] Erika Chin and David Wagner. 2009. Efficient Character-Level Taint Tracking for Java. In *Proceedings of the 2009 ACM Workshop on Secure Web Services (SWS '09)*. Association for Computing Machinery, New York, NY, USA, 3–12.
[31] Commix Project. 2020. Automated All-in-One OS command injection and exploitation tool. https://github.com/commixproject/commix.
[32] Andrei Costin. 2017. Lua Code: Security Overview and Practical Approaches to Static Analysis. In *38th IEEE Symposium on Security and Privacy Workshops (SPW)*. IEEE. https://doi.org/10.1109/spw.2017.38
[33] cPanel. 2021. Hosting Platform of Choice. https://cpanel.net/.
[34] CVE. CVE-2014-2323. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2323.
[35] CVE. CVE-2016-10033. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10033.
[36] CVE. CVE-2017-10004. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-10004.
[37] CVE. CVE-2018-10969. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10969.
[38] CVE. CVE-2018-15877. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15877.
[39] CVE. CVE-2018-16461. https://nvd.nist.gov/vuln/detail/CVE-2018-16461.
[40] CVE. CVE-2018-3757. https://www.cvedetails.com/cve/CVE-2018-3757/.
[41] CVE. CVE-2018-3836. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3836.
[42] CVE. CVE-2019-13638. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13638.
[43] CVE. CVE-2019-976. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-976.
[44] CVE. CVE-2020-8149. https://nvd.nist.gov/vuln/detail/CVE-2020-8149.
[45] CVE. CVE-2020-8178. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8178.
[46] CVE. 2017. CVE-2017-1000451. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000451.
[47] CVE. 2017. CVE-2017-17562. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17562.
[48] CVE. 2018. CVE-2018-3746. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3746.
[49] CVE. 2018. CVE-2018-3786. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3786.

[50] CVE. 2019. CVE-2019-10061. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-10061.
[51] CVE. 2019. CVE-2019-10783. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-10783.
[52] CVE. 2019. CVE-2019-12272. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-12272.
[53] CVE. 2020. CVE-2020-7597. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-7597.
[54] Dav Glass. 2015. lsof. https://www.npmjs.com/package/lsof.
[55] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288.
[56] Adam Doupé, Bryce Boe, Christopher Kruegel, and Giovanni Vigna. 2011. Fear the EAR: Discovering and Mitigating Execution after Redirect Vulnerabilities. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, New York, NY, USA, 251–262.
[57] Edward. 2018. Plain View Activity Monitor. https://wordpress.org/plugins/plainview-activity-monitor.
[58] Egg. 2019. eggscripts. https://www.npmjs.com/package/egg-scripts.
[59] Elementor. 2020. A website builder that delivers high-end page designs and advanced capabilities. https://wordpress.org/plugins/elementor/.
[60] Embedthis. 2019. GoAhead. https://www.embedthis.com/goahead/.
[61] Fabien Potencier. 2020. free feature-rich PHP mailer. https://packagist.org/packages/swiftmailer/swiftmailer.
[62] Fabien Potencier. 2020. Symfony Console Component. https://packagist.org/packages/symfony/console.
[63] Fagbokforlaget V&B AS. 2018. pdfinfojs. https://www.npmjs.com/package/pdfinfojs.
[64] Apache Software Foundation. 2019. Apache JMeter. https://jmeter.apache.org/.
[65] WordPress Foundation. 2019. WordPress. https://wordpress.com/.
[66] GNU. 2018. Patch. https://savannah.gnu.org/projects/patch/.
[67] PostgreSQL Global Development Group. 2020. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/docs/9.4/functions-bitstring.html.
[68] Vivek Haldar, Deepak Chandra, and Michael Franz. 2005. Dynamic Taint Propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC '05)*. IEEE Computer Society, USA, 303–311.
[69] William G.J. Halfond and Alessandro Orso. 2005. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the International Conference on Automated Software Engineering*. Long Beach, California, USA.
[70] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. 2006. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In *Proceedings of the Symposium on the Foundations of Software Engineering*.
[71] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. 2008. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *Transactions on Software Engineering* 34, 1 (2008), 65–81.
[72] Mary Jean Harrold and Mary Lou Soffa. 1994. Efficient Computation of Interprocedural Definition-Use Chains. *ACM Trans. Program. Lang. Syst.* 16, 2 (March 1994), 175–204.
[73] Daniel Hillmann. 2019. kill-port-processes. https://www.npmjs.com/package/kill-port-process.
[74] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. 2003. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *Proceedings of the 12th International Conference on World Wide Web (WWW '03)*. Association for Computing Machinery, New York, NY, USA, 148–159.
[75] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. 2004. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. ACM, New York, NY, USA, 40–52.
[76] HYRISE. 2020. SQL Parser for C++. Building C++ object structure from SQL statements. https://github.com/hyrise/sql-parser.
[77] Intel. 2019. Software Guard Extensions. https://software.intel.com/en-us/sgx.
[78] Isaac Bennetch. 2020. SQL Parser. https://github.com/phpmyadmin/sql-parser.
[79] Jason Gerfen. 2019. NPM API to access nmap from node.js. https://www.npmjs.com/package/libnmap.
[80] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P '06)*. IEEE Computer Society, USA, 258–263. https://doi.org/10.1109/S&P.2006.29
[81] Justin Swanhart. 2019. A pure PHP SQL (non validating) parser w/ focus on MySQL dialect of SQL. https://github.com/greenlion/PHP-SQL-Parser.
[82] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. 2003. Countering Code-Injection Attacks with Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*. Association for Computing Machinery, New York, NY, USA, 272–280.
[83] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. 2009. Automatic Creation of SQL Injection and Cross-Site Scripting Attacks. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 199–209. https://doi.org/10.1109/ICSE.2009.5070521
[84] Engin Kirda, Christopher Krugel, Giovanni Vigna, and Nenad Jovanovic. 2006. Noxes: A client-side solution for mitigating cross-site scripting attacks. In *SAC'06*.
[85] Kevin E. Kline and Daniel Kline. 2001. *SQL in a Nutshell*. O'Reilly.
[86] Lerna. 2020. A tool for managing JavaScript projects with multiple packages. https://github.com/lerna/lerna.
[87] Jinyuan Li, Maxwell N Krohn, David Mazieres, and Dennis E Shasha. 2004. Secure Untrusted Data Repository (SUNDR).. In *Osdi*, Vol. 4. 9–9.
[88] Lighttpd. 2019. Lighttpd Web Server. https://www.lighttpd.net/.
[89] LinuxConfig.org. 2015. Internal vs External Linux shell commands - LinuxConfig.org. https://linuxconfig.org/internal-vs-external-linux-shell-commands.
[90] LuaExpat. 2020. XML Expat parsing for the Lua programming language. https://matthewwild.co.uk/projects/luaexpat/.
[91] Margaret Brewster. 2019. Parses Sql to an AST and re-stringifies SQL ASTs. https://www.npmjs.com/package/druid-sql-parser.
[92] Marijn Haverbeke. 2020. A small, fast, JavaScript-based JavaScript parser. https://github.com/acornjs/acorn.
[93] Michael Martin and Monica S. Lam. 2008. Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking. In *Proceedings of the 17th Conference on Security Symposium (SS'08)*. USENIX Association, USA, 31–43.
[94] Masafumi Oyamada. 2018. NPM Provides an interface to convert PDF's pages to png files in Node.js. https://www.npmjs.com/package/pdf-image.
[95] Masahiro Wakame. 2017. fs-git. https://www.npmjs.com/package/fs-git.
[96] Matthew Gonzalez. 2017. listening-processes. https://www.npmjs.com/package/listening-processes.
[97] Sean McAllister, Engin Kirda, and Christopher Kruegel. 2008. Leveraging User Interactions for In-Depth Testing of Web Applications. In *Recent Advances in Intrusion Detection*, Richard Lippmann, Engin Kirda, and Ari Trachtenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 191–210.
[98] Michele Romano. 2019. Hackerone-728040. https://hackerone.com/reports/728040.
[99] Michele Romano. 2020. Hackerone-730121. https://hackerone.com/reports/730121.
[100] Gerome Miklau. 2019. xmldata. http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/.
[101] Yasuhiko Minamide. 2005. Static Approximation of Dynamically Generated Web Pages. In *Proceedings of the 14th International Conference on World Wide Web (WWW '05)*. ACM, New York, NY, USA, 432–441.
[102] Mozilla. 2020. Moz SQL Parser. https://github.com/mozilla/moz-sql-parser.
[103] MySQLTUTORIAL 2020. MySQL Prepared Statement. https://www.mysqltutorial.org/mysql-prepared-statement.aspx/.
[104] National Vulnerability Database. 2019. CVE-2019-15597. https://nvd.nist.gov/vuln/detail/CVE-2019-15597.
[105] Trent Nelson. 2020. Technically-oriented PDF Collection. https://github.com/tpn/pdfs.
[106] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. 2005. Automatically Hardening Web Applications Using Precise Tainting. In *Security and Privacy in the Age of Ubiquitous Computing*. Springer, 295–307.
[107] Nick Galbreath. 2018. SQL / SQLI tokenizer parser analyzer. https://github.com/client9/libinjection.
[108] Nikita Popov. 2020. Extension exposing PHP 7 abstract syntax tree. https://github.com/nikic/php-ast.
[109] notpwnguy. 2018. Hackerone-511459. https://hackerone.com/reports/511459.
[110] NVD. 2019. CVE Details: CVE-2019-5127. https://nvd.nist.gov/vuln/detail/CVE-2019-5127.
[111] OpenLiteSpeed. 2019. OpenLiteSpeed is the Open Source edition of LiteSpeed Web Server Enterprise. https://openlitespeed.org/.
[112] OpenWrt. 2019. LuCI. https://openwrt.org/docs/guide-user/luci/start.
[113] OpenWrt. 2019. uHTTPd. https://openwrt.org/docs/guide-user/services/webserver/uhttpd.
[114] OpenWrt 2020. OpenWrt Project. https://openwrt.org/.
[115] Oracle. 2019. Mysql. https://www.mysql.com/.
[116] OWASP. 2019. OWASP Top Ten. https://owasp.org/www-project-top-ten/.
[117] Packagist. 2020. The PHP Package Repository. https://packagist.org.
[118] Pawel Trysla. 2020. Display pretty Android and iOS logs without Android Studio or Console.app, with intuitive Command Line Interface. https://github.com/zamotany/logkitty.
[119] PECL. 2021. PECL :: Package :: taint. https://pecl.php.net/package/taint.
[120] Jeff Perkins, Jordan Eikenberry, Alessandro Coglio, Daniel Willenson, Stelios Sidiroglou-Douskos, and Martin Rinard. 2016. AutoRand: Automatic Keyword Randomization to Prevent Injection Attacks. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721 (DIMVA'16)*. Springer-Verlag, Berlin, Heidelberg, 37–57.
[121] Peter Braden. 2019. OpenCV. https://www.npmjs.com/package/opencv.

[122] PHP. 2019. SimpleXML Extension. https://www.php.net/manual/en/book.simplexml.php.
[123] Tadeusz Pietraszek and Chris Vanden Berghe. 2005. Defending against injection attacks through context-sensitive string evaluation. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 124–145.
[124] QEMU. 2019. Generic and open source machine emulator and virtualizer. https://www.qemu.org/.
[125] Quan Yang. 2019. Taint'em-All: a taint analysis tool for the PHP language. https://github.com/quanyang/Taint-em-All.
[126] Rafal Janicki. 2019. Hackerone-633364. https://hackerone.com/reports/633364.
[127] Sazzadur Rahaman, Gang Wang, and Danfeng Yao. 2019. Security Certification in Payment Card Industry: Testbeds, Measurements, and Recommendations. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. ACM, New York, NY, USA, 481–498.
[128] RaymondDesign. 2012. Advanced-XML-Reader. https://wordpress.org/plugins/Advanced-XML-Reader/.
[129] Renan Rocha. 2019. Hackerone-661959. https://hackerone.com/reports/661959.
[130] Robbie Chipka. 2020. GitHub - libxmljs:libxml bindings for v8 javascript engine. https://github.com/libxmljs/libxmljs.
[131] B. G. Ryder. 1979. Constructing the Call Graph of a Program. *IEEE Trans. Softw. Eng.* 5, 3 (May 1979), 216–226.
[132] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Xiaodong Song. 2010. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS*.
[133] Prateek Saxena, David Molnar, and Benjamin Livshits. 2011. SCRIPTGARD: Automatic Context-Sensitive Sanitization for Large-Scale Legacy Web Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, New York, NY, USA, 601–614.
[134] Sebastian Bergmann. 2020. Library that helps with managing the version number of Git-hosted PHP projects. https://packagist.org/packages/sebastian/version.
[135] Sebastian Bergmann. 2020. PHPUnit is a programmer-oriented testing framework for PHP. https://phpunit.de/.
[136] Sebastian Bergmann. 2020. Provides functionality to handle HHVM/PHP environments. https://packagist.org/packages/sebastian/environment.
[137] R. Sekar. 2009. An Efficient Black-box Technique for Defeating Web Application Attacks. In *Network and Distributed System Security Symposium (NDSS'09)*.
[138] Selenium. 2021. SeleniumHQ Browser Automation. https://www.selenium.dev/.
[139] Genetech Solutions. 2020. Pie Register - Custom Registration Form, Invitation based Registrations and User Login WordPress Plugin. https://wordpress.org/plugins/pie-register/.
[140] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. 2013. Diglossia: Detecting Code Injection Attacks with Precision and Efficiency. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. Association for Computing Machinery, New York, NY, USA, 1181–1192.
[141] Amie L. Souter and Lori L. Pollock. 2003. The Construction of Contextual Def-Use Associations for Object-Oriented Systems. *IEEE Trans. Softw. Eng.* 29, 11 (Nov. 2003), 1005–1018.
[142] SQLite. 2019. What Is SQLite. https://www.sqlite.org/index.html.
[143] Star Beam Rainbow Labs. 2020. Pepperminty-Wiki. https://github.com/sbrl/Pepperminty-Wiki.
[144] Alexandre Strzelewicz. 2019. PM2. https://www.npmjs.com/package/pm2.
[145] Zhendong Su and Gary Wassermann. 2006. The Essence of Command Injection Attacks in Web Applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. Association for Computing Machinery, New York, NY, USA, 372–382.
[146] Spinner. 2020. Spinner Project Website. https://github.com/cmd-spinner/commandrandom-spinner-php.
[147] Takayuki Miyoshi. 2020. Contact Form 7 can manage multiple contact forms. https://wordpress.org/plugins/contact-form-7/.
[148] Tao Zhi. 2020. Nodejs SQL Parser. https://www.npmjs.com/package/node-sql-parser.
[149] Theofilos Petsios. 2014. sqlrand-llvm. https://github.com/nettrino/SQLRand.
[150] Tom Forbes. 2020. Github-orf/xcat:Automate XPath injection attacks to retrieve documents. https://github.com/orf/xcat.
[151] Joe Topjian. 2009. Sanitize and Validate Data with PHP Filters. https://code.tutsplus.com/tutorials/sanitize-and-validate-data-with-php-filters--net-2595.
[152] TryGhost. 2020. The #1 headless Node.js CMS for professional publishing. https://github.com/TryGhost/Ghost.
[153] Daniel Veillard. 2019. libxml. http://xmlsoft.org/.
[154] Vercel. 2020. Generate changelogs. https://github.com/vercel/release.
[155] Veselin. 2020. Easy package.json exports. https://www.npmjs.com/package.
[156] Voidcosmos. 2020. KILLO: List any node_modules directories in your system. https://github.com/voidcosmos/npkill.
[157] Matt Walters. 2019. meta-git. https://www.npmjs.com/package/meta-git.
[158] Gary Wassermann and Zhendong Su. 2007. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 32–41.
[159] Gary Wassermann and Zhendong Su. 2008. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 171–180.
[160] Wenbin Xiao. 2018. SQL Parser implemented in Go. https://github.com/xwb1989/sqlparser.
[161] WordPress. 2020. The WordPress Importer will import the content from a WordPress export file. https://wordpress.org/plugins/wordpress-importer/.
[162] WordPress. 2020. WordPress Plugins. https://wordpress.org/plugins.
[163] World Wide Broadcast Network. 2020. AVideo-Encoder. https://github.com/WWBN/AVideo-Encoder.
[164] Yichen Xie and Alex Aiken. 2006. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th Conference on USENIX Security Symposium (Security'06)*. USENIX Association, USA, Article Article 13.
[165] Yoast BV. 2020. Yoast SEO. https://yoast.com/wordpress/plugins/seo/.
[166] Zach Carter. 2017. An API for creating parsers in JavaScript. https://www.npmjs.com/package/jison.

9 APPENDIX

9.1 Supplementary Text and Experiment

9.1.1 Sink Functions. In addition to Table 1, Table 3 provides additional sink functions for XML and database subsystems.

Table 3: Sink Functions

Sink Functions	Subsystem	Language
`mysqli::multi_query()`, `mysqli::prepare()`, `mysqli::real_query()`, `mysqli::select_db()`, `mysqli::send_query()`	MySQL	PHP
`mysql_create_db()`, `mysql_drop_db()`, `mysql_query()`, `mysql_real_query()`, `mysql_select_db()`	MySQL	C/C++
`sqlite_array_query()`, `sqlite_exec()`, `sqlite_open()`, `sqlite_query()`, `sqlite_popen()`, `sqlite_single_query()`, `sqlite_unbuffered_query()`	SQLite	PHP
`sqlite3_get_table()`, `sqlite3_exec()`, `sqlite3_prepare()`[1], `sqlite3_prepare16()`[2], `sqlite3_open()`[3]	SQLite	C/C++
`libxml.parseXmlString()`, `parser.parseString()`, `parser.push()`, `element.find()`, `element.get()`	XML	JavaScript
`callbacks.StartDoctypeDecl()`, `parser:parse()`	XML	Lua

1: inlcuding `sqlite3_prepare_v2()`, `sqlite3_prepare_v3()`.
2: inlcuding `sqlite3_prepare16_v2()`, `sqlite3_prepare16_v3()`.
3: inlcuding `sqlite3_open16()`, `sqlite3_open_v2()`.

9.1.2 Automated Vulnerability Discovery Tools. Commix [31] is an automated testing tool that aims to find command injection vulnerabilities on web server-side applications. We test all programs except for `s4` and `s5` which do not have functions executing OS/shell commands. Commix identified vulnerabilities shown in Table 2, and successfully executed 102 malicious commands, while it failed to do so for Spinner protected programs. sqlmap [20] is a penetration testing tool for SQL injection vulnerability testing. We apply sqlmap to the applications that use SQL database engines: `s1` (WordPress), `s5` (Pie Register), and `s6` (Lighttpd). We instruct sqlmap to inject the SQL statements through typical input channels (e.g., `GET` and `POST` requests). sqlmap supports various types of injection attack payloads, including boolean blind SQL injection, error-based SQL injection, stacked queries SQL injection, and time blind SQL injection, just to name a few. Spinner mitigates all the injected statements. xcat [150] is a command line tool to exploit and investigate XML injection vulnerabilities. We tested `s4` and `s5`,

which are vulnerable to the XXE injection. xcat successfully discovers XXE injection vulnerability in the original programs while it failed with the SPINNER protected application.

9.1.3 Overhead on Database Engines. We use OLTP-Bench [55], which is an extensible testbed for benchmarking relational databases. It provides 15 data-sets. However, when we test the data-sets on the vanilla MySQL and SQLite, *only three data-sets (TPC-C, Wikipedia, and Twitter) were successfully completed* while all others lead to crashes. Hence, we select the three working data-sets. The average overheads are 4.9% and 5.3% for SQLite and MySQL respectively.

9.1.4 Overhead on XML Library. We use Libxml [153], SimpleXML [122], libxmljs [130], and LuaExpat [90]. For XML test-data, we download a data-set (1GB in total) from the University of Washington [100]. The average overheads are 1.5%, 1.38%, 1.43%, and 1.29% for Libxml, SimpleXML, LuaExpat, and libxmljs respectively.

9.1.5 Overhead on OpenWrt. We applied SPINNER to the OpenWrt firmware's uHTTPd [113] web server and LuCI web configuration interface [112]. We use QEMU [124] to run OpenWrt ARM firmware with 256MB RAM, which represents the standard router hardware specification. We use Apache Jmeter to generate 1,000 concurrent requests to visit the LuCI interface to get system status information. Note that 1,000 parallel requests are sufficient to exhaust the test system's resources and the test workload is more intensive than the common usage. The average overhead is 5.83%.

9.1.6 Versions of the Evaluated Programs. Table 4 shows the versions of all the evaluated programs including those in Table 6.

Table 4: Versions of the Evaluated Programs

ID	Version	ID	Version	ID	Version	ID	Version
s1	5.3.2	s12	0.4.18	s23	0.2.4-beta	s34	5.1.3
s2	20161228[1]	s13	1.1.0	s24	3.6.1	s35	2.0.4
s3	2.3	s14	2.6.0	s25	0.3.6	s36	6.2.3
s4	0.15	s15	0.1.4	s26	0.4.13	s37	3.0.2
s5	1.7	s16	3.5.0	s27	1.0.2	s38	3.36.0
s6	3.0.9	s17	1.0.1	s28	5.2.2	s39	3.22.1
s7	1.4.35	s18	1.1.2	s29	15.2	s40	0.7.2
s8	1.74.4	s19	1.2.0	s30	4.1.7	s41	6.3.0
s9	2.7.6	s20	0.1.0	s31	3.0.12	s42	1.0.0-pre.45
s10	3.6.5	s21	6.0.0	s32	0.7		
s11	0.10	s22	0.7.0	s33	5.1.8		

1: This project does not have an explicit version number. This is the date of the last commit.

9.1.7 Trusted Command Specification (TCS) Generation Tool. We provide an automated trusted command specification generation tool [146] that takes a list of sink-functions (e.g., Table 1) and trusted-folders (e.g., `/var/www/`) as input. It derived all the TCSs used in the paper without significant domain-knowledge and completed the analysis in less than four minutes. It can also detect incomplete specifications (e.g., untrusted commands passed to sink-functions). Note that we did not observe incomplete specifications.

Performance of TCS Generator. Table 5 shows the time required to generate the TCS by our TCS generator. We generate the same TCS used in our evaluation. Note that generating TCS for Leptonica (s8) took the longest time: 217.75 seconds, which is 3 min 37.75 seconds.

Table 5: Time to generate TCS for each application

ID	Time (s)	ID	Time (s)	ID	Time (s)
s1	165.59	s15	1.25	s29	31.96
s2	7.82	s16	22.29	s30	4.13
s3	5.18	s17	1.18	s31	29.24
s4	7.99	s18	1.59	s32	2.88
s5	14.73	s19	1.14	s33	4.71
s6	6.23	s20	1.22	s34	7.31
s7	100.32	s21	0.83	s35	28.64
s8	217.75	s22	0.75	s36	18.38
s9	12.28	s23	0.74	s37	9.23
s10	54.02	s24	1.17	s38	15.19
s11	146.36	s25	1.35	s39	8.28
s12	9.76	s26	0.58	s40	0.98
s13	1.18	s27	1.28	s41	1.19
s14	1.61	s28	9.68	s42	0.87

9.2 Effectiveness of SPINNER

9.2.1 Applicability of SPINNER. To understand whether SPINNER can be a generic solution for various applications, we additionally collect the five most popular applications from three well-known open-source package managers (NPM [155], Packagist [117] and WordPress Plugin [162]) as shown in Table 6. We prune out programs that are not meant to be deployed such as a unit-test framework [135]. SPINNER successfully handled them without errors

9.2.2 Correctness of Instrumentation. Table 7 shows the number of test cases. Our additional test cases to cover all the instrumented code and increase code coverage are shown in the "Added" column.

9.2.3 Supporting Polymorphic Objects. SPINNER supports complex real-world applications including OOP programs such as WordPress in Table 2. In this example, we show that our analysis handles polymorphism and dynamic bindings. In particular, Figure 14 shows how SPINNER analyzes polymorphic objects in a WordPress plugin called Elementor [59]. From line 11, we identify an instantiation of an object with a string (`$class_name`). We backtrace the string variable (annotated via red arrows), identifying that the class name starts with "`Control_`". However, as the return value of `get_control_names()` can be updated at runtime, we conservatively assume that any class that has name starting with `Control_` (i.e., `Control_*`) can be created at line 11.

Then, we conduct a forward analysis to identify the object's usage (annotated through black arrows). Figure 14 shows only a few of the forward flows due to space. We check all the omitted flows and they are not relevant to command execution.

9.2.4 Effectiveness of Bidirectional Analysis. This section provides an example of the effectiveness of bidirectional analysis.

1) Backward Analysis: The backward flow analysis begins from the sink function `mysql_query()` at line 32. Following the backward data flow (depicted as purple arrows), it reaches to line 38, which is a SQL query. However, the backward analysis alone is not able to identify the original of `$wpdb->users` to determine

Table 6: Popular Applications From Pakcage Managers

ID	Name	Size	Source	# Instrumentation				
				Const.	Dynamic			Sinks
					1-5	6-10	>11	
s28	Contact-Form-7 [147]	744.00 KB	WordPress	1	4	0	0	5
s29	Yoast SEO [165]	13.70 MB	WordPress	6	12	9	0	6
s30	Akismet Spam Protection [14]	288.00 KB	WordPress	0	17	0	0	17
s31	Elementor Website Builder [59]	18.00 MB	WordPress	2	21	0	0	23
s32	WordPress Importer [161]	100.00 KB	WordPress	0	2	0	0	2
s33	Symfony Console [62]	584.00 KB	Packagist	8	10	0	0	15
s34	Environment [136]	49.00 KB	Packagist	3	0	0	0	3
s35	Composer [1]	120.00 KB	Packagist	4	4	0	0	8
s36	Swiftmailer [61]	2.08 MB	Packagist	0	1	0	0	1
s37	Version [134]	20.00 KB	Packagist	1	0	0	0	1
s38	Ghost [152]	58.80 MB	NPM	1	0	0	0	1
s39	Lerna [86]	12.10 MB	NPM	1	0	0	0	1
s40	Npkill [156]	7.69 MB	NPM	2	7	0	0	9
s41	Release [154]	900.00 KB	NPM	0	3	0	0	3
s42	Yalc [134]	704.00 KB	NPM	0	4	0	0	4

Table 7: Test Cases and Code Coverages

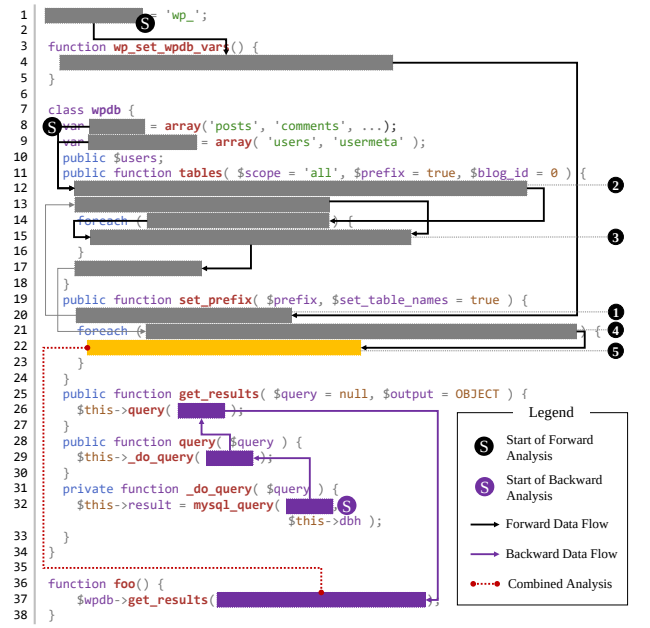
ID	Line of Code	# of Test Cases		Cov–erage	ID	Line of Code	# of Test Cases		Cov–erage
		Added	Total				Added	Total	
s1	116,356	200	11,677	58.54%	s15	146	39	63	76.14%
s2	9,881	100	142	75.76%	s16	12,281	50	332	72.78%
s3	67,560	404	404	70.49%	s17	402	50	60	88.33%
s4	6,124	161	161	82.30%	s18	69	49	53	86.21%
s5	12,872	219	219	71.56%	s19	78	51	54	89.74%
s6	9,944	188	188	73.92%	s20	76	50	58	83.85%
s7	42,840	100	459	64.00%	s21	1,912	49	141	86.00%
s8	86,668	97	443	65.38%	s22	999	40	53	79.17%
s9	30,011	100	107	71.00%	s23	301	60	62	79.46%
s10	58,994	233	289	68.96%	s24	792	38	122	83.22%
s11	34,237	396	396	72.42%	s25	145	38	48	91.19%
s12	1,431	10	122	78.23%	s26	284	60	64	74.11%
s13	174	50	53	83.73%	s27	143	55	57	87.50%
s14	281	55	89	96.65%					

whether this is from a trusted source (hence requires instrumentation) or not. Note that the value of `$wpdb->users` is assigned by dynamic construct, which is unreachable by the backward analysis.

2) Forward Analysis: Our forward analysis starts from trusted sources such as constant strings at lines 1, 8, and 9. `$table_prefix` (global variable) is assigned to `$this->base_prefix` at line 20 (❶), which will be used at line 13 in `tables()`. Figure 15-(b) shows values of variables at the lines marked by circled numbers. From lines 8 and 9, the two arrays are merged at line 12 (❷), resulting in an array shown in Figure 15-(b). At line 15, it constructs an array consisting of pairs of table names and table names with `wp_` prefix (❸). The composed new table (`$_table`) is returned by `tables()`, which is called at line 21 in `set_prefix()`. Hence, we further analyze `set_prefix()` which iterates arrays shown in Figure 15-(b)-❹. Note that PHP allows a string variable to be used to specify a member variable's name in an object (line 22). To this end, the line 22 essentially executes statements shown in Figure 15-(b)-❺. Note that the first statement define `$this->users`, where `$this` is essentially `$wpdb`.

3) Combining Forward and Backward Analysis: Recall that the backward analysis stops at line 37, as it was unable to resolve `$wpdb->users`, while the forward analysis can resolve the value.

Figure 14: Handling Polymorphic Objects (Red and black arrows represent backward and forward analysis respectively)



(a) Bidirectional Command Composition Analysis on WordPress

❶ `$this->base_prefix = 'wp_';`

❷ `$_tables=(0=>'users', 1=>'usermeta', 2=>'posts',...);`

❸ `$_tables=('users' =>'wp_users', 'usermeta'=>'wp_usermeta', 'posts' =>'wp_posts',...);`

❹ `$tablename=('users', 'usermeta', 'posts',...);`
`$prefixed_table=('wp_users', 'wp_usermeta', 'wp_posts',...);`

❺ `$this->users = 'wp_users'; // $this=$wpdb`
`$this->usermeta = 'wp_usermeta';`
`$this->posts = 'wp_posts'; ...`

(b) Example Concrete Values of Variables

Figure 15: Bidirectional Analysis on WordPress

Bidirectional analysis successfully finds out that the variable in the query (`$wpdb->users`) is a constant string from a trusted source.

9.2.5 Evaluation of Bidirectional Analysis's Accuracy. In this section, we explain the details of how we evaluate Spinner's bidirectional analysis's effectiveness and correctness. Note that since obtaining the ground-truth is challenging, we try to evaluate the bidirectional analysis's accuracy as follows. We manually verified that all the results from our analysis are true-positives. We also run other static/dynamic taint-analysis tools [5, 119, 125] and compare the results (i.e., dependencies) from them with the result from Spinner. As shown in Table 8, we observe that Spinner covers the majority of the dependencies chains that are generated by the other tools. For dependencies not covered by Spinner, we manually check them and find that they are false-positives (hence we are not missing anything covered by other tools).

Note that during this process, we have updated and implemented a few tools. First, we update the AST parser of taintless [6] to the new version and add extra rules to help it handle WordPress's callback function hook. Second, we add additional plugins to psalm [5] to enhance its ability on tracing data flow on object inheritance. Third, we add additional sinks to PECL taint [119] for tainting WordPress.

Procedure of the Evaluation. We do our evaluation as follows.

1. Run the bi-directional analysis and manually verify the dependency chains identified by the analysis. a) Manually check the propagation rules applied by the bi-directional analysis (both forward and backward analyses). b) Verify that all the dependencies identified by the bi-directional analysis are true-positives.
2. Run other static/dynamic analysis tools to get dependency chains (Note that static/dynamic analysis tools suffer from over and under-approximations). a) If other tools find more dependencies, then they might be potential false-negatives of the bi-directional analysis. We manually verify them all, and the result shows that they are all false-positives, meaning that we did not find false-negatives from the bi-directional analysis. b) If other tools find lesser dependencies, then they might be potential false-positives of the bi-directional analysis. We manually verify them all, and the result shows that they are all false-negatives, meaning that we did not find false-positives from the bi-directional analysis.

Procedure and Method for Manual Analysis. Our manual analysis leverages existing static/dynamic analysis techniques. While they are inaccurate, we only apply them for a single dependency chain and reason about the result. Since we only reason a single dependency at a time, the task was manageable even though it is a time-consuming task. We conduct inter-procedural manual analysis, meaning that we follow through the callee functions' arguments if values propagate through the functions. The analysis finishes when the data reaches a trusted/untrusted source. In addition to the static/dynamic taint analysis techniques, we manually run the programs and observe how the concrete values are propagated by changing inputs and checking output differences. Note that if an output value is changed from the above testing due to the input change, there is a dependency.

To make sure Spinner's bi-directional analysis does not miss anything, we compared the results with existing techniques (Taintless, Psalm, and PECL taint). We manually analyzed them and verified that all the results from bi-directional analysis are true-positives. Details on the notable cases are as follows.

Table 8: Effectiveness of Spinner's bidirectional analysis compared with existing techniques

Testbeds	Spinner	Taintless	Psalm	PECL taint
WordPress*	462	413	426	537
Activity Monitor*	27	16	17	27
Avideo Encoder*	61	66	61	61
PHPSHE*	270	301	266	223
Pie Register*	73	79	77	73
Pepperminty WiKi	2	2	2	2
Contact-Form-7	5	5	5	5
Yoast SEO	27	27	27	27
Akismet Spam Protection	17	17	17	17
Elementor Website Builder	23	23	23	23
WordPress Importer	2	2	2	2
Symfony Console	18	18	18	18
Environment	3	3	3	3
Composer	8	8	8	8
Swiftmailer	1	1	1	1
Version	1	1	1	1

*: Except for these 5 applications, there is no difference between the tools.

1. *WordPress:* Compared to Taintless, Taintless has 49 false negatives. Among them, 24 false negatives are caused as described in Figure 15. 5 false negatives are caused by handling PHP dynamic function call (e.g., `call_user_func_array()`). 20 false negatives are caused by handling WordPress `apply_filter` which invokes a function by the nickname registered by `add_filter`. Compared to Psalm, Psalm has 24 false negatives as described in Figure 15. Psalm is not accurate in handling object inheritance. It will miss the data dependencies from subclass methods to base class methods in 36 cases. Compared to PECL taint, PECL taint has 35 false positives caused by handling WordPress `do_action` dynamic function hook. PECL taint has 40 false positives caused by string array filtering operation.
2. *Activity Monitor:* Compared to Taintless, Taintless has 11 false negatives. Among them, 3 false negatives are caused as shown in Figure 15. 8 false negatives are caused by not supporting WordPress `apply_filter` which invoke a function registered by `add_filter` dynamically. The data flow will be broken when it goes into such APIs. Compared to Psalm, Psalm has 14 false negative and 4 false positive cases. Among them, 3 false negatives are caused as shown in Figure 15. 8 false negatives are caused by `add_filter` and `apply_filter`. 3 false negatives are caused by mishandling object inheritance. Variables defined in base class will not be recognized in subclass. 4 false positive cases are caused by mishandling regex matching API `preg_match`.
3. *Avideo-Encoder:* Compared to Taintless, Taintless has 2 false negatives and 7 false positives. Among them, 2 false negatives are caused by unsupported API `DateTime()` which should be considered as trusted. 7 false positives are caused by mishandling regex API `preg_match`.
4. *PHPSHE:* Compared to Taintless, Taintless has 16 false negatives and 47 false positives. Among them, 16 false negatives are caused by parsing error on one PHP file. Internal bug on an old version of PHP-Parser. 47 false positives are caused by history upgrading scripts. Compared to Psalm, Psalm has 15 false negatives and 11 false positives. Among them, 3 false negatives are caused by `time()` API. 12 false negatives are caused by class object

inheritance. 11 false positives are caused by SQL keywords in arguments used matching pattern of `preg_match` functions. Compared to PECL taint, PECL taint has 47 false negatives because of PHP fatal error in executing database update script

5. *Pie-register:* Compared to Taintless, Taintless has 2 false negatives and 8 false positives. Among them, 2 false negatives are caused by the case shown in Figure 15. 8 false positives are caused by SQL keywords in the embedded HTML while they are not SQL statements. Compared to Psalm, Psalm has 2 false negatives caused by the case shown in Figure 15.

```
1  function getProcesses (command) {
2    execSync(rand('lsof')+` -i TCP -P -n | ` +
3             rand('grep')+`'${command}\\s.*:[0-9]* (LISTEN)' |` +
4             rand('cat'), {encoding: 'utf-8'})
5             .toString().split('\n');
6    ...
```

(a) Instrumentation affecting a single basic block

```
7  l_int32 gplotMakeOutput(GPLOT *gplot)
8  {
9      char buf[L_BUF_SIZE];
10     snprintf(buf, L_BUF_SIZE, "%s %s", rand("gnuplot"), ...);
11     ... = system(buf);
12 }
```

(b) Instrumentation affecting a single function

```
13 class WP_Site_Health {
14   private function prepare_sql_data() {
15     ... = $wpdb->get_var( [redacted] );
16   }
17 }
18 class wpdb {
19   var $last_query;
20   public function get_var( [redacted] ) {
21     if ( ... ) $this->query( [redacted] );
22   }
23   public function query( [redacted] ) {
24     [redacted]
25     $this->_do_query( [redacted] );
26   }
27   private function _do_query( [redacted] ) {
28     ...
29     $this->result = [redacted]
30     ...
31   }
32   public function print_error( $str = '' ) {
33     ...
34     [redacted] = sprintf( 'WordPress database error ...',
35                           $str, [redacted] );
36     ...
37     [redacted]
38     ...
39   }
40 }
```

(c) Instrumentation affecting multiple functions (5 functions)

Figure 16: Examples of Impact of Instrumentation

9.2.6 Impact Analysis for Instrumentated Code. Figure 16 shows examples of instrumentations impacting a single basic block (a), a single function (b), and multiple functions (c).

Single Basic Block (the BB column in Table 2). This is the simplest type of instrumentation. As shown in Figure 16-(a), all the instrumented commands (i.e., `lsof`, `grep`, and `cat`) are directly fed into the sync function (`execSync` at line 2). The instrumented commands are not saved and transferred to other functions.

Single Function (the Fn column in Table 2). Instrumented commands can affect or stored in local variables. However, they only affect statements within the same function and do not propagate to other functions. In Figure 16-(b), the instrumentation (`rand()` at line 10) affects a local variable `buf`. However, the local variable does not affect any other statements nor passed/returned to other functions. Note that it is relatively easy to verify the impact of instrumentation since it only requires analysis within the function.

Multiple Functions (the Fns column in Table 2). In this type, an instrumentation affects multiple functions through function calls and global/member variables. Figure 16-(c) shows an example. The instrumented SQL query is shown at line 15. The randomized query is passed to `get_var()` (❶). The query is then used to call `query()` function (❷) and passed to the function again (❸). In the `query()` function, it is stored to the `$last_query` member variable (at line 24, ❹) and passed to the `_do_query()` function (❹). Finally, in the `_do_query()` function, the query is used to call a sink function which is `mysql_query()`. Note that the `$last_query` variable that stores the randomized query is used later in the `print_error()` function at lines 34 and 37 (❼ and ❽). In this example, the instrumentation at line 15 affects 5 functions (`prepare_sql_data()`, `get_var()`, `query()`, `_do_query()`, and `print_error()`).

9.3 Additional Discussions

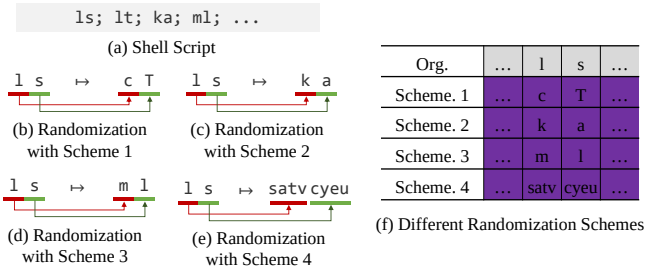
9.3.1 Alternative Approach: Screening Unintended Commands. One can develop an approach that only allows intended commands identified. For instance, given a function call "`system("rm file $opt")`", the approach will only allow the "`rm`" command. Such an approach (i.e., allowlist method) is fundamentally different from SPINNER since it cannot distinguish *different instances of commands* and enforce the same rule for every commands on an API. For example, it cannot prevent if an attacker injects the same command (e.g., "`rm`" in this case). SPINNER randomizes the first "`rm`" and leaves the second "`rm`" command, which is injected, preventing the attack. For SQL injections, approaches relying on known/allowed SQL keywords cannot prevent attacks leveraging keywords that are not considered (e.g., Section 5.4.2) while SPINNER can prevent them.

9.3.2 Prepared Statements in Practice. As mentioned in Section 6, prepared statements are not well adopted in practice. We analyze all the SQL queries in the applications used in our evaluation. We find that 866 SQL queries from WordPress [65] (459 queries), Pie Register [139] (70 queries), PHPSHE [2] (277 queries), AVideo-Encode [163] (39 queries), and Plainview Activity Monitor [57] (21 queries). None of them use the prepared statements.

Unsupported Keywords. The following SQL keywords are not supported: `DESCRIBE` (or `DESC`), `ALTER DATABASE`, `LOAD DATA`, `LOAD XML`, `RENAME USER`, and `SHOW TABLES LIKE`. In particular, WordPress (`s1`) is using the unsupported keywords, i.e., `DESC`, `SHOW TABLES LIKE`, in their queries, making it challenging to convert.

9.3.3 Brute-force Attack SPINNER. Attackers may inject multiple commands (or a shell script file containing multiple commands) to try out a number of guesses of randomization schemes. From the attacker's perspective, if any of the guesses lead to the successful execution of the command, the attack is successful. Figure 17-(a) shows such a shell script containing multiple commands. We find that the Linux shell process handles individual commands separately, causing multiple command execution API invocations for each command. Recall that SPINNER uses different randomizations on command execution API invocations. To this end, we randomize the subsystems differently, as shown in Figure 17-(b-e). The first command failed because we randomize '`ls`' ↦ '`cT`'. The second

attempt also failed as 'ka' is expected. Even if an attacker learned this previous randomized command and injects ka next time, as shown in this example, it still fails as SPINNER changes the randomization scheme to 'ls' ↦ 'ml'. Finally, one may try to inject a large number of the same command (e.g., millions of sl), waiting for our randomization scheme to become 'ls' ↦ 'sl'. Unfortunately, SPINNER allows can be configured to use multiple bytes translation rules. For example, the randomization scheme 4 translates a single byte to 4 bytes. With this, searching space is practically too large to brute-force. Specifically, assume our randomization schemes use all printable ASCII characters (94 of them) to substitute, two-byte commands such as 'ls', can be randomized to 8,741 ($=P(94,2)-1$) different two-byte characters. For 4 bytes commands, the space becomes extremely large: $P(94^4, 2) - 1$.



Org.	…	l	s	…
Scheme. 1	…	c	T	…
Scheme. 2	…	k	a	…
Scheme. 3	…	m	l	…
Scheme. 4	…	satv	cyeu	…

Figure 17: Randomization Schemes Used for Each Command

Effectiveness of Dynamic Randomization. SPINNER dynamically changes randomization scheme on every command, which we call dynamic randomization. To understand the effectiveness of dynamic randomization compared with the static randomization which uses a single randomization scheme during the entire execution, we tried brute-force attacks on both static and dynamic randomization approaches. In general, attackers need to try twice more attacks to break the dynamic approach than the static approach. For instance, using the dynamic approach (1-to-1 mapping) for three characters-long commands requires 70,191 more attempts to succeed the attack (which we believe quite effective) than the static approach.

Experiment Results. We conduct brute-force attack experiments. Specifically, we brute-force four different randomization schemes to show the effectiveness of the dynamic randomization scheme.

Table 9: Brute force attacks on static and dynamic randomization schemes

	1 to 1	1 to 2	1 to 3	1 to 4
Static randomization	71.1K	9.8M	1,389T*	195Q*
Dynamic randomization	141.3K	19.7M	2,779T*	391Q*

T: Trillion. Q: Quintillion. *: Estimated value.

Table 9 shows the number of failed attempts before the first correct guess, leading to a successful attack. For instance, using the 1 to 1 mapping scheme, the static method prevents 71.1K attempts successfully. With the dynamic randomization scheme, the attack has to run 141.3K commands until the first successful guess. Note that we decided to use the estimation for 1 to 3 and 1 to 4 randomization schemes because the experiment did not finish within 10 hours. We observe this result follows the distribution (i.e., static randomization approach follows the uniform distribution and dynamic randomization approach follows the geometric distribution). According to this observation, we put the expected value through the statistical method. For the case of 1 to 2 scheme, using dynamic approaches for this command requires 9,807,906,470 more attempts to succeed the attack than static randomization.

Table 10: Update History of All Evaluated Programs

ID	Trusted Src.[1]	Language	# S[2]	# V[3]	Timeline (dd/mm/yyyy)	Dur.[4]
s1	Const.[5], Conf.[6]	PHP	7	16	11/16/2017 ~ 10/29/2020	35
s2	Const.[5], Conf.[6]	PHP	6	17	05/11/2014 ~ 08/26/2018	51
s3	Const.[5]	PHP	27	3	08/12/2017 ~ 01/13/2020	29
s4	Const.[5]	PHP	2	20	11/25/2014 ~ 09/11/2020	69
s5	Const.[5], Conf.[6]	PHP	5	3	01/01/2017 ~ 09/05/2018	20
s6	Const.[5], Conf.[6]	PHP	2	19	10/04/2011 ~ 10/22/2020	108
s7	Const.[5], Path	C	10	25	01/02/2016 ~ 10/25/2020	57
s8	Const.[5]	C	2	20	01/14/2016 ~ 07/28/2020	54
s9	Const.[5]	C	2	7	09/12/2012 ~ 02/06/2018	64
s10	Const.[5], Path	C	1	3	12/22/2018 ~ 07/15/2020	18
s11	Const.[5]	Lua	52	13	10/09/2014 ~ 09/28/2020	71
s12	Const.[5]	JavaScript	2	54	12/28/2009 ~ 06/18/2012	29
s13	Const.[5]	JavaScript	2	9	01/05/2018 ~ 10/01/2019	20
s14	Const.[5]	JavaScript	3	23	08/02/2017 ~ 02/24/2020	30
s15	Const.[5]	JavaScript	1	4	06/03/2014 ~ 02/16/2018	44
s16	Const.[5]	JavaScript	34	20	09/15/2016 ~ 09/29/2020	48
s17	Const.[5]	JavaScript	1	14	09/20/2014 ~ 06/01/2017	32
s18	Const.[5]	JavaScript	3	6	03/03/2017 ~ 11/26/2019	32
s19	Const.[5]	JavaScript	3	5	08/12/2017 ~ 08/18/2017	<1[8]
s20	Const.[5]	JavaScript	3	2	05/23/2014 ~ 01/06/2020	67
s21	Const.[5]	JavaScript	3	4	12/23/2013 ~ 05/16/2020	76
s22	Const.[5]	JavaScript	2	15	01/25/2019 ~ 01/10/2020	11
s23	Const.[5]	JavaScript	3	6	04/07/2016 ~ 03/23/2017	11
s24	Const.[5]	JavaScript	6	23	10/16/2015 ~ 05/09/2017	18
s25	Const.[5]	JavaScript	3	11	02/22/2013 ~ 06/28/2018	64
s26	Const.[5]	JavaScript	1	25	11/23/2016 ~ 10/30/2019	35
s27	Const.[5]	JavaScript	2	4	06/30/2015 ~ 01/30/2016	7
s28	Const.[5], Conf.[6]	PHP	5	35	05/06/2013 ~ 10/21/2020	89
s29	Const.[5], Conf.[6]	PHP	6	28	09/03/2019 ~ 10/15/2020	13
s30	Const.[5], Conf.[6]	PHP	17	27	03/04/2016 ~ 10/15/2020	55
s31	Const.[5], Conf.[6]	PHP	2	23	05/30/2016 ~ 10/20/2020	52
s32	Const.[5], Conf.[6]	PHP	2	11	10/25/2010 ~ 04/04/2020	113
s33	Const.[5]	PHP	15	32	01/07/2015 ~ 10/04/2020	68
s34	Const.[5]	PHP	3	10	02/18/2014 ~ 09/28/2020	29
s35	Const.[5], Env.[7]	PHP	8	14	04/15/2016 ~ 10/24/2020	54
s36	Const.[5]	PHP	1	13	12/19/2016 ~ 11/12/2019	34
s37	Const.[5]	PHP	1	10	03/03/2017 ~ 09/28/2020	42
s38	Const.[5]	JavaScript	1	38	03/26/2014 ~ 10/20/2020	78
s39	Const.[5]	JavaScript	1	32	12/04/2015 ~ 06/08/2020	54
s40	Const.[5]	JavaScript	7	20	07/29/2019 ~ 01/20/2020	5
s41	Const.[5]	JavaScript	3	21	12/28/2016 ~ 07/28/2020	43
s42	Const.[5]	JavaScript	2	27	11/22/2017 ~ 10/22/2020	35

1: Trusted Sources. 2: Versions. 3: Sinks. 4: Duration in months. 5: Constant String. 6: Configuration File. 7: Environment Variable. 8: Less than 1 month.

9.3.4 Impact of Software Updates on SPINNER. As discussed in Section 6, if software updates of a target application cause changes in the trusted command specification, manual analysis of the target application is required. To understand how prevalent such cases are in practice, we study the update history of 42 applications (27 applications in Table 2 and 15 programs in Table 6). As shown in Table 10, we track major version updates from the first stable version to the most recent major update until November 2020. We analyze

each major update to understand whether the trusted command specification of an old version should be updated for a new version to use Spinner. The result shows that none of the trusted command specifications are changed between versions.

Table 10 shows the results. All 42 applications use constant strings as a trusted source. There are 9 programs that have both configuration files and constant strings as trusted sources. Their trusted command specification is similar to Figure 2-(a). `s7` and `s10` have folder paths and constant strings as trusted sources and can be defined as shown in Figure 2-(c). `s35` requires the environment variable as a trusted source. For this program, to prevent attacks that attempt to compromise environment variables, we hook `setenv` and `getenv`.

Table 11: Performance of Spinner

Program	Version used in Section 5	Latest version
WordPress	4.33% (released in 12/18/19)	4.41% (released in 2/22/21)
Leptonica	4.25% (released in 6/11/17)	4.21% (released in 7/28/20)

Spinner on Different Versions of Target Programs. To understand the impact of software updates on the performance of Spinner, we applied Spinner to WordPress (v5.6.2; released on Feb 22, 2021) and Leptonica (v1.8; released in July 28, 2020) in addition to the versions we have evaluated in Section 5, as shown in Table 11. The resulting protected programs are correct where we observe a similar average overhead of 4.31%.

9.3.5 Preventing Trusted Sources from Being Compromised. Spinner often trusts configuration files that cannot be modified by remote attackers. However, if our analysis is incomplete or the system has other vulnerabilities that allow attackers to compromise the trusted configuration files, Spinner's protection can be affected. As a mitigation, we implement a kernel module that denies any modifications to the configuration files. We also tried secure file systems [87] to prevent unauthorized modifications to the configuration files. We enabled them during our evaluation, and we do not observe any errors caused by them, meaning that users may also use such approaches to protect Spinner.

9.4 Diglossia [140] vs Spinner

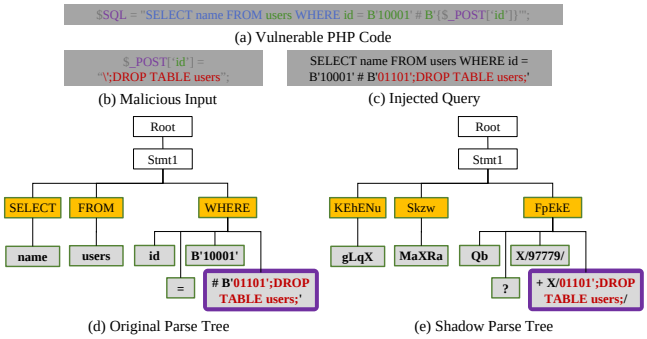
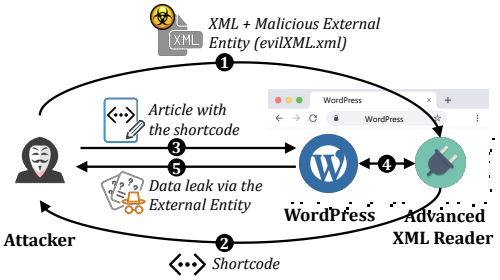


Figure 18: Failure Case of Diglossia with PHP-SQL-Parser

In addition to Section 5.4.2, we compare Spinner with another our own implementation of Diglossia [140] using PHP-SQL-Parser [81], which is the most popular SQL Parser for PHP in GitHub. Figure 18-(a) shows a vulnerable PHP program's code. Given the malicious input shown in Figure 18-(b), the malicious query is injected as shown in Figure 18-(c). Figure 18-(d) and (e) show parse trees from the original parser and the shadow parser. Nodes with yellow backgrounds represent keywords while nodes with gray backgrounds represent strings or numbers which are allowed to be injected. Nodes with green borders are correctly translated in the shadow parser, meaning that they are intended nodes. Nodes with the violet borders are those that are *not fully translated*, meaning that some values (i.e., the first 2 characters) are translated and some are not. Note that Diglossia detects an injected input by identifying nodes with the same values between the two parse trees. In this case, we do not have such nodes, meaning that Diglossia will miss the attack. The injected code is not properly parsed due to the bug of the parser. It fails to recognize SQL grammar after the # symbol, an XOR operator in PostgreSQL.

Spinner uses a scanner and applies reverse-randomization scheme to the injected query, preventing the attack.

9.4.1 Preventing XXE Injection. XML External Entity (XXE) injection allows attackers to inject an XML external entity in an XML file. XML external entity is a custom XML tag that allows an entity to be defined based on the content of a file path or URL. An attacker can abuse the external entity to leak the content of arbitrary files. In this case, we use a WordPress plugin, Advanced XML Reader [128], to demonstrate how Spinner prevents the XXE injection attack.



(a) Attack Scenario

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <!DOCTYPE foo [
3 <!ENTITY [redacted] SYSTEM [redacted]>]>
4 <doc> Contents of passwd: [redacted] </doc>
```

(b) XML + Malicious External Entity (evilXML.xml)

```
[advanced-xml tag=[redacted]]
```

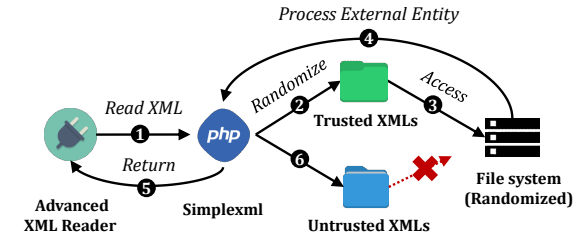
(c) Article with the Shortcode

Figure 19: XXE Injection Attack Scenario

Figure 19-(a) shows an attack scenario. The attacker first sends a malicious XML file with a malicious external entity (❶). The malicious XML file's content is shown in Figure 19-(b). The XML file defines an `<!ENTITY SYSTEM>` tag with a file path `/etc/passwd`. The tag is used in line 4, which will be the content of the XML file when it is requested. The vulnerable plugin uploads the XML file and returns a shortcode (❷), which is essentially the name of the uploaded file to refer to the XML content in the future. Now, the attacker posts an article with the shortcode (❸) as shown in Figure 19-(c). Note that the tag value indicates the uploaded file's name. Once the post is uploaded, WordPress sends it to the plugin (Advanced XML Reader), which will parse and resolve the XML file

referred to in the post (❹). During the processing, the plugin reads the password file and returns the content. When the posted article is requested, the password file's content will be delivered (❺).

Figure 20 shows how Spinner ensures benign operations while preventing the XXE injection attack described in Figure 19.



(a) Preventing XXE Injection

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <!DOCTYPE e [
3 <!ENTITY x SYSTEM "file:/[redacted]" >]>
4 <doc> &x; </doc>
```

(b) Trusted XML Files (Randomized)

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <!DOCTYPE e [
3 <!ENTITY x SYSTEM "file:/[redacted]" >]>
4 <doc> Contents of passwd: &x; </doc>
```

(c) Untrusted XML Files

Figure 20: Preventing an XXE Injection Attack

Benign Operation. When the plugin reads an XML file (❶), Spinner intercepts the file I/O and check whether it reads a trusted XML file or not. Note that Spinner maintains a list of trusted XML file paths. Typically, those are the XML files provided by an administrator, not the files that are uploaded by remote users. If the file path of the XML file is in the list, Spinner randomizes the external entities' file contents (❷). Then, it tries to access the file system with the randomized file name. As the file paths of the file system are randomized by Spinner, it successfully reads the file and returns (❸ and ❹). Finally, the content is returned (❺).

Preventing XXE Injection. When an attacker uploads a malicious XML file, it is not included in the trusted XML file list. When the plugin tries to read an XML file that is uploaded by a remote user (which is not trusted, ❻), the XML file's entity will not be randomized. As a result, it will not be able to access the file system correctly, leading to a file open error.