SPIDER: Enabling Fast Patch Propagation in Related Software Repositories

Aravind Machiry†, Nilo Redini†, Eric Camellini¶, Christopher Kruegel†, and Giovanni Vigna†
†University of California, Santa Barbara
{machiry, nredini, chris, vigna}@cs.ucsb.edu
¶Politecnico di Milano
eric.camellini@gmail.com

***Abstract*—Despite the effort of software maintainers, patches to open-source repositories are propagated from the main codebase to all the related projects (e.g., forks) with a significant delay. Previous work shows that this is true also for security patches, which represents a critical problem. Vulnerability databases, such as the CVE database, were born to speed-up the application of critical patches; however, patches associated with CVE entries (i.e., CVE patches) are still applied with a delay, and some security fixes lack the corresponding CVE entries. Because of this, project maintainers could miss security patches when upgrading software.**

In this paper, we are the first to define safe patches (*sp*s). An *sp* is a patch that does not disrupt the intended functionality of the program (on valid inputs), meaning that it can be applied with no testing; we argue that most security fixes fall into this category. Furthermore, we show a technique to identify *sp*s, and implement SPIDER [1], a tool based on such a technique that works by analyzing the source code of the original and patched versions of a file. We performed a large-scale evaluation on 341,767 patches from 32 large and popular source code repositories as well as on 809 CVE patches. Results show that SPIDER was able to identify 67,408 *sp*s and that most of the CVE patches are *sp*s. In addition, SPIDER identified 2,278 patches that fix vulnerabilities lacking a CVE; 229 of these are still unpatched in different vendor kernels, which can be considered as potential unfixed vulnerabilities.

I. INTRODUCTION

The open-source software model revolutionized the software industry, and prior research shows that it is more secure [70], [33] than its closed-source counterpart. However, propagating changes and patches from the main repository of an open-source project to all the *related projects* (forks) is a major problem [66]. While related projects share a common "ancestry," their code bases typically diverge over time, as different teams add different features and capabilities to each branch. As a result, when a problem is found and fixed in one branch, it is not always easy to directly apply this patch to another one [67]. For example, Android depends on a Linux kernel fork, and upgrading it with patches from the main kernel repository without thoroughly testing their effects could break Android.

To avoid this problem, and still be able to keep their software up-to-date, the maintainers of related projects need to carefully track other branches. When they see a fix that might be relevant for their codebase, they have to "cherry-pick" the patch: That is, they have to understand the patch and its behavior, adapt it to their own code base, and finally ensure that the whole system, after applying the patch, still works as expected. Not surprisingly, this is a manual and resource-intensive process [72], [17]. As a result, as shown in the Appendix D of our extended version [7], changes in the main code base of a project are usually applied to the code of dependent software with a significant delay [68]: Android 10, for example, is based on Linux kernel 4.19, while the latest release of the Linux kernel is version 5.3.8 [2].

[1] This is a short form for Safe Patch fInDER.

This problem becomes critical when we consider security patches: in these cases, the fixes should propagate to all the codebases as soon as possible. Vulnerability databases such as the Common Vulnerabilities and Exposures (CVE) database were born to facilitate this process: project maintainers can take them as a reference to know which security-related patches they need to apply, without having to find them manually. Despite the existence of these databases, security patches still take a substantial amount of time to propagate to all the project forks [57], [53], [22], [41]. In the year 2016, the Android maintainers patched 76 publicly known vulnerabilities (i.e., CVEs) from the year 2014, two from 2013, and two from 2012, which means that 80 disclosed vulnerabilities remained unpatched in the Android code base for more than one year [1]. This revelation attracted considerable public interest [5]. Recent work [48] shows that attackers who monitor source repositories often get a head start of weeks (and sometimes months) on targeting vulnerabilities prior to any public disclosure. Furthermore, as we will show in this study (Section VII-D), it is possible that the maintainers of a project underestimate the severity of a patched bug, and fail to request a corresponding entry in a vulnerability database (a CVE ID) [10]. When this happens, maintainers of related projects are not aware that a patch actually addresses a security problem. This is a growing problem, as exemplified by the recent VLC security issue [16], which is caused because developers of `libebml` failed to associate the corresponding security fix with a CVE ID [11], and the vulnerability existed for nearly two years after the fix was available. Unfortunately, hackers are known to scan source repository commits for fixes that might address vulnerabilities, and then check for the presence of these vulnerabilities in related repositories [74]. Therefore, the security fixes lacking a CVE ID provide a potential source of unfixed vulnerabilities as they are most likely not ported to related repositories.

Existing approaches that ease the process of cherry-picking relevant patches rely on commit-related information, such as code *diff* or commit messages [65], [76], [21], or they look for specific patterns [56]. These tools have the advantage of being fast, lightweight, scalable, and suitable to be used on large codebases. However, either they only match simple patches, or analyze commit messages, which are often not expressive enough to convey the scope and effect of a change [24], [69], [4]. Other techniques attempt to go a step further and analyze the semantic differences introduced by a patch using static analysis [44], [45], [25], [26], [64] and symbolic execution [52], [29], [62], [37]. Unfortunately, these techniques suffer from scalability issues. Moreover, some of these approaches also require the exact build environment [30] of the whole code base, restricting their practicality and applicability to complex software, such as the Linux kernel, the VLC player, the OpenBSD OS, etc., as these software have many possible configurations [12].

Intuitively, an ideal solution, which would help maintainers in selecting and applying important changes, would be a system that is capable of identifying those patches that do not affect the

intended functionality of the software. If the intended functionality of the software is not changed by a patch, this patch can be applied without the need for testing: we call these changes *safe patches*. In this paper, we argue that a significant portion of all security-related fixes falls under the category of safe patches [60]. Thus, a tool that can identify safe patches could be used to monitor the main repository and automatically alert or apply this kind of patches on a target forked repository. This observation is also confirmed by our anonymous survey (Appendix J) of maintainers and developers of various open-source software projects.

To be effective and usable on large codebases, a system to identify safe patches should at least satisfy the following requirements:

- **R1**: Only rely on the original and patched versions of the modified source code file, without any other additional information (e.g., commit message, build environment, etc.)
- **R2**: Be fast, lightweight and scalable.

In this work, we design and implement a static analysis approach that aims to identify safe patches and that satisfies both the requirements above. Our approach is designed specifically to target source code changes and to identify patches that could be applied with minimal testing, as they do not modify the program's functionality. Specifically, we make the following contributions:

- We provide the first formal definition of safe patches, and design a general technique to identify them.
- We implement SPIDER, a system based on this technique, that takes as input only the source code of the original and patched file.
- We evaluate SPIDER on 341,767 commits taken from 32 source code repositories (Linux kernel repositories, Android kernel repositories, interpreters, firmware, utilities and various other repositories), as well as on 809 CVE patches.
- We identify 67,408 safe patches and show that SPIDER could help developers in the process of selecting and testing changes, resulting in a speed-up in the propagation of security fixes.
- We also provide the Security Patch mode of SPIDER that can precisely identify security patches. It identified 2,278 patches that most likely fix security vulnerabilities, despite the fact that they were not associated with any CVE entry. 229 of these issues are still unpatched in several kernel forks. As such, they can be considered unfixed vulnerabilities.
- We are releasing SPIDER and the corresponding `git` server-side hook configuration at github.com/ucsb-seclab/spider.

Unlike previous work, our approach is the first that focuses on determining those patches that can be propagated to related projects with minimal effort, and without defining *a priori* specific types of changes or semantic characteristics that should be detected (i.e., we do not just target patches that fix a specific type of vulnerability). We envision our system to be part of the recently introduced Github security alerts [8], or it could be used to build a variant of the *git rebase* feature that suggests patches that are most likely safe and should be prioritized.

II. SAFE PATCHES

Our goal is to identify patches that can be applied without subsequent testing. We call such patches *safe patches* (*sp*s). Intuitively, for a patch to be considered an *sp*, it should satisfy the following two conditions:

- **Non-increasing input space (C1)**: The patch *should not* increase the valid input space of the program. That is, the patched version should be more restrictive in the inputs that it accepts. The assumption is that some of inputs that the original program accepted resulted in security violations, and the patched version "removes" these inputs as invalid.
- **Output equivalence (C2)**: For all the valid inputs that the patched program accepts, the output of the patched program must be the same as that of the original program.

The condition $C1$ ensures that there is *no need to add new test cases*, as there are no new inputs that are accepted by the patched program [2]. Furthermore, the condition $C2$ ensures that there is *no need to run the existing test cases* as the output will be the same as that of the original program (for all the valid inputs). Consequently, if a patch satisfies the above two conditions then it can be applied without any effect on the existing test cases. Of course, the purpose of testing is to ensure that the program behaves as expected, so it is always a recommended step after applying a patch. In Section II-B, we define more formally the two conditions above.

A. Running Example

Listing 1 shows our running example, a C language example of a safe patch in the unified *diff* format (i.e., where $+$ and $-$ indicate inserted and deleted lines, respectively). In this example, the programmer decided that it was necessary to add an extra length check (Lines 3-5), presumably to protect against a buffer overflow later in the program. In addition, the patch also includes the length of the header (HDR) as part of a size check in Line 10.

This patch is safe. The inserted modifications to the variables `len` and `tlen` do not change the output of the function. Moreover, the extra conditional statement in Line 3 adds a missing length check, thereby restricting the input space. That is, all inputs where `t->len` is larger than `MAX_LEN` now lead to the function returning an error, while those inputs were accepted by the original function.

Figure 1 shows the control flow graph (CFG) after the application of this example patch: underlined text indicates the pieces of code inserted, while the left (blue) and right (red) children of each basic block are the true and false branches, respectively.

B. Formal Definition

We first define terminology used throughout the paper:

- *Input* i to a program: The input data with which the program is executed; I indicates the set of all the possible inputs to the program.
- *Function of a program*: The symbol f denotes the original function, and any subscript to it identifies its patched version. For example: f_p indicates the function f after applying the patch p.
- *Error-handling basic blocks*: The symbol BB_{err} denotes the basic blocks of a function that are part of its error-handling functionality. In Figure 1, BB_2 is an

[2] However, for regression testing purposes, one may want to add a test case that checks that the inputs are indeed invalid and the corresponding security flaw is patched.

```
1  long get_read_size(struct dring *t) {
2      long len, tlen;
3  +   if(t->len > MAX_LEN) {
4  +       return -1;
5  +   }
6      ...
7  -   len = t->len;
8  +   len = t->len + 4;
9      ...
10     if(len % 2) {
11         len += DEF_SIZE;
12     }
13     ...
14 -   tlen = len;
15 +   tlen = len - 4;
16     ...
17     t->total = tlen;
18     ...
19     return tlen;
20 }
```

Listing 1: Running Example of a safe patch.

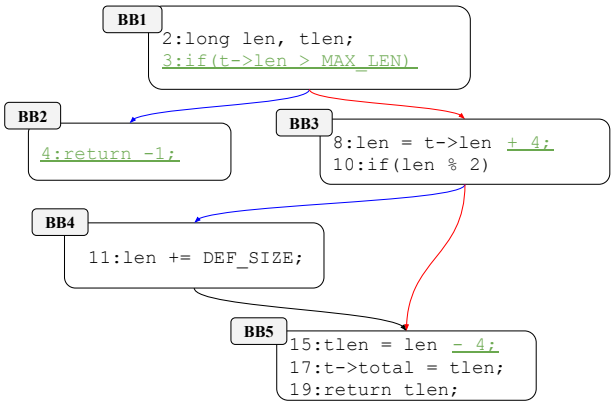


Fig. 1: Control flow graph of the patched program from Listing 1

error-handling basic block. We will explain later how error-handling basic blocks are identified.

- We use the notation $i \hookrightarrow f$ to indicate that input i *successfully executes* through function f. That is, starting from the entry basic block of f, and given input i, none of the error-handling basic blocks (BB_{err}s) of f will be reached. In other words, i represent a valid input to the function f.
- *Output of a function*: The *output* of a function f is its return value and all the externally visible changes to the program's data. Specifically, the output includes the return value, all writes to heap and global variables, and the arguments to all function calls. For instance, the output of the function `get_read_size` in Listing 1 is its return value (line 19), and the value written to the pointer variable `t->total` (line 17). Furthermore, $output(i,f)$ indicates the output of the function f when run with input i.

Now, we will use the definitions we introduced to formally define two conditions ($C1$ and $C2$) introduced at the beginning of Section II.

1) Non-increasing input space (C1): The non-increasing input space (C1) condition requires that the patched program does not accept any inputs as valid that are not also accepted as valid by the original program. This condition can be defined at the granularity of functions; that is, for C1 to hold, we require that all patched functions, individually, do not accept any additional valid inputs. In other words, any valid input to a patched function must also be a valid input to the corresponding original function. More formally:

$$\forall i \in I \,|\, (i \hookrightarrow f_p) \rightarrow (i \hookrightarrow f). \tag{1}$$

In the case of Listing 1, the patch restricts the original input space by adding an additional constraint (i.e., $t \rightarrow len > MAX\_LEN$ in Line 3). As a result, all valid inputs to the patched function are also valid inputs to the original function (but not vice versa). This satisfies Equation 1.

2) Output correspondence (C2): The output correspondence (C2) condition requires that, for all valid inputs, the output of the patched program must be the same as the output of the original program. This condition, again, can be defined at the function granularity: For each patched function, for all corresponding valid inputs, the patched function must produce the same outputs as the original function. More formally:

$$\forall i \in I \,|\, (i \hookrightarrow f_p) \rightarrow (output(i,f_p) = output(i,f)). \tag{2}$$

In the case of Listing 1, although the patch inserts changes that modify the values of some variables (for example, `len`), the changes do not affect the externally visible data of the program, and thus, they do not change the output of the function, thereby satisfying Equation 2.

If all the patched functions satisfy both Equation 1 and Equation 2, then we can say that the patch satisfies the conditions $C1$ and $C2$. As a result, the patch can be considered as a *safe patch (sp)*. Note that, as a trivial case, an empty patch ($f_p = f$) satisfies Equations 1 and 2, making it an *sp*. Furthermore, there exist patches that do not satisfy the above conditions but still could be applied without testing, making our conditions sufficient but not necessary. We refer all the interested readers to Appendix A, where we explain our formalism with more examples.

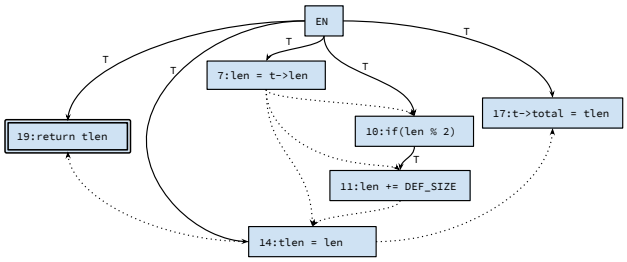
III. Identifying Safe Patches

In this section, we introduce a general technique to determine whether a given patch is an *sp*.

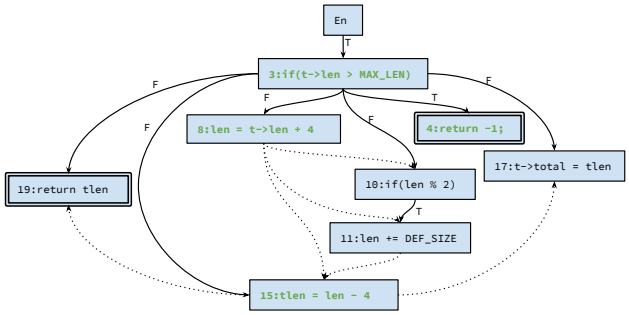
A. Program Dependency Graph (PDG)

Our technique leverages the concept of a program dependency graph (PDG). A program dependency graph [36] captures both data and control dependencies in a single graph. Formally, the *PDG* of a function f, denoted as $PDG(f) = (V,C,D)$, is a directed graph where

- $V = \{v_0, v_1, ..., v_n, E_n\}$ is a set of nodes, one for each *instruction* ($v_\bullet$) of the function. The additional node, E_n represents the function entry.
- C is a set of directed, labeled edges, where each edge $(v_i, v_j, T|F)$ represents the (direct) control dependency of v_j on v_i. An instruction v_j is control-dependent on v_i, and the edge is labeled as *true* (T) [or *false* (F)], when v_j is executed if and only if v_i evaluates to true [or *false*]. To complete the PDG, if an instruction v is not control-dependent on any other instruction in the function (in other words, it does not have any incoming control flow edges), we connect it to the function entry node (E_n). That is, we add the edge (E_n, v, T) to C. Note that all source nodes of control-flow edges are either conditional statements (`if`, `while`, etc.) or the function entry node



(a) *PDG* of the *original* function in Listing 1



(b) *PDG* of the *patched* function in Listing 1

Fig. 2: Program Dependency Graph (*PDG*) of the original and patched function in Listing 1, where dotted and solid edges represent data and control dependencies, respectively. The function exit points are in double-bordered boxes.

E_n. In addition, all conditional statements will have at least one outgoing control-dependency edge.

- D is a set of directed edges, where each edge (v_i, v_j) represents a data dependency. That is, instruction v_i defines a variable that can reach the corresponding use in instruction v_j.

For our running example in Listing 1, Figures 2a and 2b show the program dependency graphs for the original and the patched function, respectively. The labels on the control dependency edges [$true(T)$ or $false(F)$] indicate whether the destination node is reachable from the source node via the *true* or *false* branch.

Control dependency versus control flow: The concept of control dependency is different from the more commonly-used concept of control flow. Control flow captures *possible flows* of execution, while control dependency captures the *necessary conditions* that must hold for the execution to reach a particular statement. We refer all the interested readers to Appendix B, where we explain this in detail.

Control-Dependency Path: Given a $PDG(f) = (V, D, C)$ of a function f, we say that a control dependency path exists from instruction $x \in V$ to instruction $y \in V$, denoted as $x \mapsto_c y$, if there exists a path in the *PDG* from x to y that only follows control-dependency edges. Formally,

$$x \mapsto_c y = \{ < x, v_1, v_2, ..., v_n, y > | \, v_\bullet \in V \wedge (x, v_1, \bullet) \in C \wedge (v_n, y, \bullet) \in C \wedge \forall_{1 \leq i < n} (v_i, v_{i+1}, \bullet) \in C \}.$$

In the *PDG* shown in Figure 2b, there exists a control-dependency path (a path along solid edges) from the instruction at Line 3 to the instruction at Line 11: $\{3, 10, 11\}$.

Path Constraint (*PC*): For any instruction v, the condition derived from the sequence of nodes and edges (with their labels) along the control-dependency path from the function entry E_n to v is called its *path constraint*. For example, consider the instruction at Line 11 in the *PDG* in Figure 2b. The control-dependency path from E_n to Line 11 is $\{E_n, 3, 10, 11\}$. The corresponding path constraint is $PC(E_n \mapsto_c 11) = ((E_n == T) \wedge (\texttt{(t->len > MAX\_LEN)} == F) \wedge ((\texttt{(len \% 2)} \neq 0) == T))$. That is, Line 11 is only executed if `(t->len <= MAX_LEN)` and `(len % 2)` $\neq$ `0`.

Data-Dependency Path: Given a $PDG(f) = (V, D, C)$ of a function f, we say that a data-dependency path exists from instruction $x \in V$ to instruction $y \in V$, denoted as $x \mapsto_d y$, if there exists a path in the *PDG* from x to y that only follows data-dependency edges. More formally:

$$x \mapsto_d y = \{ < x, v_1, v_2, ..., v_n, y > | \, v_\bullet \in V \wedge (x, v_1) \in D \wedge (v_n, y) \in D \wedge \forall_{1 \leq i < n} (v_i, v_{i+1}) \in D \}.$$

In the *PDG* shown in Figure 2b, there exists a data-dependency path (a path along dotted edges) from instruction at Line 8 to the instruction at Line 19: $\{8, 15, 19\}$. We say that a given data-dependency path $x \mapsto_d y = < x, v_1, v_2, ..., v_n, y >$ is *complete* if there is no data dependency path to x. Formally, $(\bullet, x) \notin D$. The data dependency path example from Line 8 to 19 is complete as there is no data dependency path to Line 8.

Also, note that although a data dependency path exists from the instruction at Line 8 to instruction at Line 19, there is no control-dependency path between these instructions. This is because the execution of the instruction at Line 19 is not controlled by the instruction at Line 8.

B. *The* SPIDER *Approach*

Our system is given as input a patch p, with f and f_p being a function before and after applying the patch, respectively. The technique to detect whether p is a safe patch works in four steps, as outlined in the following four sections.

1) Checking modified instructions: We first need to identify what statements are affected by a patch, and determine whether these modifications can be soundly analyzed given our requirement $R1$. Recall that $R1$ requires that the analysis operates directly on the original and patched versions of the modified source code file, without any other additional information (e.g., commit message, build environment, etc.).

Affected Statements: A statement can be affected either *directly* or *indirectly* by the patch. We call a statement *directly affected* if it is modified, inserted, deleted, or moved by the patch. A statement is *indirectly affected* if it is either *control-* or *data-* dependent on any of the directly affected statements. Given the set of directly affected statements A_d and the *PDG* of the corresponding function, all the instructions *reachable* from the statements in A_d, either through control flow or data flow edges, are indirectly affected.

Consider the patch for our running example in Listing 1. Here, the directly affected statements are at Lines 3, 4, 8, and 15. However, looking at the corresponding *PDG* in Figure 2b, we can see that all instructions are reachable from the node that corresponds to the instruction at Line 3. Consequently, all statements are affected by the patch.

Locally analyzable statement: We call a directly affected statement *locally analyzable* if all the writes made by the statement can be captured *without* any interprocedural and pointer analysis. Specifically, the modifications made by the patch should not involve any new function calls or pointer manipulation. Consider the patch represented by Listing 2: The inserted statement at Line 4 is locally analyzable. However, the inserted statement at Line 5 is not locally analyzable, because it involves a new function call.

If a patch has any directly affected statements that are not locally analyzable, we *do not* consider it an *sp*. This is because we cannot soundly analyze the affected statements without analyzing the effects on the whole program. Moreover, performing whole-program analysis requires a static analysis tool (like LLVM), which in turn, requires access to the sources of the entire program, violating our requirement $R1$.

```
1  int kthread_init() {
2      ...
3  -   total_size = file->size;
4  +   total_size = header + file->size;
5  +   init_cleanup();
6      ...
7      if(total_size > MAX_SIZE) {
8          ...
9      }
10     ...
11 }
```

Listing 2: Patch illustrating locally analyzable statements.

2) Error-handling basic blocks: In the next step, we need to identify all the error-handling basic blocks (BB_{err}s) in f and f_p, so that all the changes to the statements within BB_{err}s are discarded and not considered in the next steps. This decision is based on the assumption that any changes to error basic blocks do not disrupt the original functionality (i.e., they just result in better or adjusted error-handling). The remaining statements affected by p are then analyzed to check if Equations 1 and 2 can be proved. We leverage previous work [42], [73] to identify error-handling basic blocks, as discussed in more detail in Section IV-D.

3) Non-increasing input space (C1): To verify the non-increasing input space condition (C1), we need to ensure that the patch does not accept more inputs than the original function. In other words, the patch must not increase the valid input space for the modified function.

Intuitively, if a patch does not affect any control-flow statements (such as `if`, `while`, `for`, etc.), then it cannot change the input space of the function. However, if a patch affects one or more control-flow statements, we must verify that no additional inputs can successfully execute through the function.

This can be done by first identifying the valid exit points (*VEP*) of a function. The valid exit points of a function are those instructions that, if reached during the execution of an input, imply that the input successfully executed through the function. For instance, in the case of our running example in Listing 1, the `return tlen` instruction at Line 19 is a valid exit point.

We consider all `return` statements as possible valid exit points. However, a function might exit because of an error (for instance, Line 4 in Listing 1), and the corresponding return statement does not represent a valid exit point. Hence, to identify the *VEP* set, we need to filter out all the return statements that are part of error basic blocks (BB_{err}).

In summary, to identify the *VEP* set of a function f with $PDG(f) = (V, D, C)$, we need to find all the exit points of f, i.e., $E_x(f)$, and filter out all the return instructions that belong to error basic blocks. More formally:

$$VEP(f) = \{r \mid ((r \in E_x(f)) \wedge (BB(r) \notin BB_{errs}(f)))\}.$$

where $BB(r)$ indicates the basic block of instruction r.

To ensure that a patch satisfies condition $C1$, we need to verify that all inputs that go through the valid exit points, i.e., *VEP* of the patched function f_p, also go through the valid exit points of the original function f.

We observe that, in order for an input i to be successfully executed by a function, the input must satisfy the path constraint (*PC*) of a valid exit point. Thus, all the inputs that are accepted as valid by a function, which we denote as $vinputs(f)$, are the union of all the inputs that satisfy the path constraints for *at least* one valid exit point. More formally, the constraints on the inputs that are successfully executed by the function f are captured by the following disjunction:

$$vinputs(f) = \bigvee_{i \in VEP(f)} (PC(i)). \tag{3}$$

If we have $vinputs(f_p) \rightarrow vinputs(f)$, which shows that all the inputs that can be successfully executed by the patched function f_p are also successfully executed by the original function f, we have succeeded in proving condition $C1$.

For our running example in Listing 1, with the *PDG* of the patched function in Figure 2b, the valid exit point is at Line 19 (`return tlen`). By following the solid edges backwards and computing the path constraints for the patched function, we obtain $vinputs(f_p) = ((E_n == T) \wedge (\texttt{t->len} > \texttt{MAX\_LEN} == F))$. For the original function, whose *PDG* is in Figure 2a, we obtain $vinputs(f) = (E_n == T)$. We can easily see that $vinputs(f_p) \rightarrow vinputs(f)$, thus satisfying $C1$.

To perform this step, we use symbolic interpretation to convert the C language statements into symbolic expressions (as discussed in more detail below). Then, we prove the implications between the two symbolic expressions using a SAT solver [34] (more details are provided in Section IV-E).

4) Output equivalence (C2): To verify the output equivalence condition (C2), we need to verify that all externally visible changes (as described in Section II-B) in the patched function are the same as that of the original one. Specifically, we want to ensure that for any input that successfully executes through the patched *and* original function, the output of the two functions will be identical.

We first look at all the affected (non-control-flow) statements. First, we discard all the statements that modify local variables. While local variables can have an indirect effect on a function's output (which we take into account, as explained below), the local variables themselves are not externally visible. Thus, we do not need to consider them in this step. In the next step, we need to verify that all the updates (writes) to non-local (global and pointer) variables, function call arguments, and return values in the patched function are the same as that of the original function. In other words, we aim to prove that all global and pointer variables have the same values after the patched function has executed (compared to the original function), the patched function returns the same value, and it calls the same functions with the same arguments (and in the same order). When we are able to prove this, we are sure that, for every valid input, the patch does not change the externally visible effect of executing this function.

Given a statement t, we need to show that for all (valid) inputs that reach t in the patched and the original function, their outputs will be the same. More formally:

$$\forall i \in I \,|\, (i \hookrightarrow t_p) \rightarrow (output(t_p) = output(t))$$

The output value of a statement depends on the values of the inputs (input variables). Consider, for example, the statement `c = a + b`. Here, the output is assigned to the variable `c`, and the value depends on the inputs `a` and `b`. We can determine where these inputs come from by looking at the data-dependency graph for the statement. Of course, the inputs for a statement could come from multiple data-dependency paths. Consider again the *PDG* in Figure 2b. For the statement at Line 15, there are two complete data dependency paths: $<8,15>$ and $<8,11,15>$. The execution can take two different paths to reach this line, based on whether the function input satisfies the constraint on Line 10 or not.

For a given statement t, and for each data-dependency path to this statement, we compute a symbolic expression for the possible output values (along these paths). The idea is that the union of the symbolic expressions (overall data-dependency paths) for t are the same for the patched function as for the original one. While this intuitively makes sense, there is one additional consideration. It is not enough to ensure that just the symbolic expressions are the same; they need to be the same under the same path constraints. Thus, we need to extend the symbolic expressions with their corresponding path constraints. We refer to these extended symbolic expressions as *symbolic output-constraint pairs*, which are computed as described hereinafter:

For a given statement t in a function f and the corresponding $PDG(f) = (V,D,C)$, we can compute the output-constraint pairs from *all* the complete data dependency paths to t. For each such path, we compute an output-constraint pair as:

$$\Psi_s = (interpret(<x_1,x_2,...,x_n,t>), \bigwedge_{1 \leq i \leq n} PC(x_i)).$$

where $interpret$ represents the symbolic expression that is computed by interpreting each of the instructions in sequence, and $PC(\bullet)$ is the path constraint of the corresponding instruction in the *PDG*.

Let $\Psi_p(t)$ and $\Psi(t)$ be the symbolic output-constraint pairs for the statement t in the patched and original function, respectively. We say that the output of statement t is equivalent in the original and the patched function, denoted as $\Psi_p(v) \equiv \Psi(v)$, if the following equation holds:

$$\forall (o_x,c_x) \in \Psi_p(v) \cdot \exists (o_y,c_y) \in \Psi(v) \vdash (o_x == o_y) \wedge (c_x \rightarrow c_y). \quad (4)$$

Note that $o_\bullet$ are not concrete values but rather symbolic values.

It is possible that there is an infinite number of data dependency paths that lead to a statement. This happens when there are loops or cyclic dependencies in the data dependency graph (for example, when a value is updated inside the body of a loop and later used by an affected statement). We will show in Section IV-E how we resolve cycles in the data dependency graph. We will further argue that our approach is safe for a subset of instances, and we only consider these cases as safe patches.

Consider how we verify that condition C2 holds for our running example in Listing 1: The affected statements are at Lines 3, 8, 10, 11, 15, 17, and 19. Recall that we only consider non-control-flow statements. Thus, we can remove Line 3 and 10 from further consideration. Next, we can discard all statements that write to local variables, which removes Lines 8, 11, and 15. We end up with the statements at Lines 17 and 19, which write to a non-local variable through a pointer and return a value, respectively.

Looking at the *PDG* for the patched function (in Figure 2b), we see that there exist two complete data dependency paths for Line 17: $< 8,15,17 >$ and $< 8,11,15,17 >$. The symbolic interpretation steps for both paths is shown in Table I. For every path, we first initialize each of the variables with a unique symbol, and then start interpreting each instruction according to its semantics. The symbolic output with corresponding path constraints along the path $< 8,15,17 >$ is $(o_p^1, c_p^1) = (\texttt{t->total} = sym2, ((En == T) \wedge ((sym2 > sym3) == F) \wedge ((((sym2+4)\%2) \neq 0) == F)))$. For the path $< 8, 11, 15, 17 >$, the result is $(o_p^2, c_p^2) = (\texttt{t->total} = sym2 + sym5, ((En == T) \wedge ((sym2 > sym3) == F) \wedge ((((sym2+4)\%2) \neq 0) == T)))$.

For interpreting the original function, we start with the same initial symbols for the same variables that were used in the patched function. From the original function's *PDG* in Figure 2a, for Line 17, there are also two data dependency paths: $<7,14,17>$ and $<7,11,14,17>$. The symbolic output along with the corresponding path constraints are $(o_c^1, c_c^1) = (\texttt{t->total} = sym2, ((En == T) \wedge (((sym2\%2) \neq 0) == F)))$ and $(o_c^2, c_c^2) = (\texttt{t->total} = sym2 + sym5, ((En == T) \wedge (((sym2\%2) \neq 0) == T)))$.

We can see that $o_p^1 == o_c^1 \wedge c_p^1 \rightarrow c_c^1$ and $o_p^2 == o_c^2 \wedge c_p^2 \rightarrow c_c^2$. Hence, Equation 4 holds.

Similarly, we can show that the output at Line 19 is equivalent in both the patched and the original function. As a result, our system has verified that the patch satisfies condition $C2$, and the patch is safe. For a patch that affects multiple functions, the steps described above are performed for each function.

IV. SPIDER: DESIGN AND IMPLEMENTATION

In this section we show the details of SPIDER, a tool, that satisfies our requirements $R1$ and $R2$, uses the approach described in Section III to analyze a given C source code patch and determine if it is an *sp*. The steps that SPIDER performs are detailed in the remaining part of this section.

A. Preprocessing

SPIDER starts by handling the C preprocessor directives. File inclusions (i.e., `#include`) are ignored, since as a requirement we do not want to collect information outside of the two input source code files. Macro definitions are ignored as well: macro calls will be treated as regular function calls, as explained later. The system then uses the *unifdef*[1] tool to handle conditional code inclusion directives (e.g., `#ifdef`, `#ifndef`, etc.): the output of *unifdef* is a valid C source file, without any of these constructs. Note that this step could exclude certain code segments. Section V explains this in detail. This first step outputs two C source files ready to be parsed.

B. Parsing

The preprocessed source files are parsed using the Joern [77] fuzzy parser, which provides an Abstract Syntax Tree (AST) for all the functions in the file. Although Joern also provides a Control Flow Graph (CFG), with nodes linked to the ones in the AST, we had to modify it to suite our needs. Specifically, we had to implement the reaching definitions analysis [58], simple type inference [63], control dependency analysis [18], and, finally, program dependency graph [36]. At the end of this phase, SPIDER has access to the AST, CFG, and PDG for each of the functions affected by the patch.

Current Statement	Symbolic State	
	Input	Output
For path: $<8,15,17>$ starting with initial state		
`8: len = t->len + 4`	`len = sym1, t->len = sym2, MAX_LEN = sym3 tlen = sym4, DEF_SIZE = sym5, t->total = sym6`	`len = sym2 + 4`
`15: tlen = len - 4`	`len = sym2 + 4, t->len = sym2, MAX_LEN = sym3 tlen = sym5, DEF_SIZE = sym5, t->total = sym6`	`tlen = sym2`
`17: t->total = tlen`	`len = sym2 + 4, t->len = sym2, MAX_LEN = sym3 tlen = sym2, DEF_SIZE = sym5, t->total = sym6`	`t->total = sym2`
For path: $<8,11,15,17>$ starting with initial state		
`8: len = t->len + 4`	`len = sym1, t->len = sym2, MAX_LEN = sym3 tlen = sym4, DEF_SIZE = sym5, t->total = sym6`	`len = sym2 + 4`
`11: len += DEF_SIZE`	`len = sym2 + 4, t->len = sym2, MAX_LEN = sym3 tlen = sym4, DEF_SIZE = sym5, t->total = sym6`	`len = sym2 + 4 + sym5`
`15: tlen = len - 4`	`len = sym2 + 4 + sym5, t->len = sym2, MAX_LEN = sym3 tlen = sym4, DEF_SIZE = sym5, t->total = sym6`	`tlen = sym2 + sym5`
`17: t->total = tlen`	`len = sym2 + 4 + sym5, t->len = sym2, MAX_LEN = sym3 tlen = sym2 + sym5, DEF_SIZE = sym5, t->total = sym6`	`t->total = sym2 + sym5`

TABLE I: Symbolic interpretation of the data-dependency path $<8,15,17>$ and $<8,11,15,17>$ of the *PDG* in Figure 2b.

C. Fine-grained diff

SPIDER uses function names to pair the functions in the original file with the corresponding ones in the new files, assuming patches that insert, delete, or rename one or more functions not to be *sp*s. SPIDER then identifies the functions affected by the patch using *java-diff-utils*[2], a common text *diff* tool. Our system then applies a state-of-the-art AST diffing technique, Gumtree [35], between the original and patched ASTs of the affected functions. Gumtree maps the nodes in the old AST with the corresponding nodes in the new one and identifies nodes that have been moved, inserted, deleted, or updated. A moved node is a node that the patch moved in another position in the AST, but whose content was unchanged, while an updated node is a non-moved node whose content was changed. The differences in the ASTs are also associated to the corresponding nodes in the CFG.

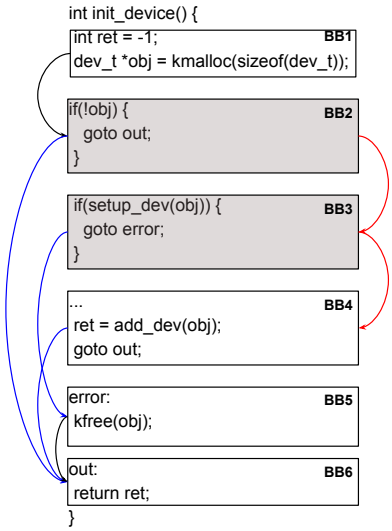


Fig. 3: Control flow annotated listing where the greyed out blocks, i.e., $BB2$ and $BB3$, represent the error-handling basic blocks identified by our approach.

D. Identification of error-handling basic blocks

We use a technique similar to the ones proposed in the works by Kang et al. [42] and Tian et al. [73] in order to identify error-handling basic blocks.

Figure 3 illustrates our approach, where the identified error-handling basic blocks are greyed out. Specifically, we consider a basic block BB to be an error-handling basic block if it satisfies any of the following conditions:

- If BB *forces* the function to `return` a constant negative value or a C standard error code (i.e., one of the constant symbols defined in `errno.h`, e.g., `EINVAL`) prepended by a minus sign or `NULL`. For this, we do a basic reaching definition analysis and check that all paths through the basic block reach a function exit that returns a constant negative value or a C standard error code. This is based on the observation that functions use negative integers or values in `errno.h` or `NULL` to indicate error conditions. For the CFG of our running example in Figure 1, we detect $BB2$ as an error-handling basic block as it causes the function to return a negative integer (`return -1`). Similarly, in Figure 3, $BB2$ causes the function to return the value of the variable `ret`, which is a negative integer (`-1`) set in $BB1$. Hence, $BB2$ will be considered as a BB_{err}.
- If BB ends in a direct jump (a `goto`) to a label that might indicate an error condition. We maintain a set of 15 error-related labels (e.g., `panic`, `error`, `fatal`, `err`), and we check if the BB ends with a `goto error-related-label;` statement. We derived our labels from an existing survey [14] and our experience in working with system code. This is based on the observation that most of the system code, especially operating system kernels [15], use `goto` to handle error conditions [6], [14]. In Figure 3, $BB3$ has the `goto error;` statement, and since `error` is one of our labels, $BB3$ will be considered as a BB_{err}. Note that, $BB3$ also satisfies the first condition, similar to $BB2$, as it can also cause the function to return a negative integer.

Unlike the work by Tian et al. [73], we do *not* consider the post-dominators of a BB_{err} to be BB_{err}s, thus, in Figure 3, the post-dominators of the error-handling basic blocks $BB2$ and $BB3$ ($BB5$ and $BB6$, respectively) will not be considered as BB_{err}s. This conservative approach improves precision by avoiding certain basic blocks to be wrongly identified as BB_{err}s (such as $BB6$). However, we may miss certain error-handling basic blocks ($BB5$). Note that, our approach for improving the precision by missing potential error-handling basic blocks is safe. We refer all the interested readers to Appendix K where we explain this in detail.

```
1  - max_len = strlen(buf);
2  + max_len = strlen(buf) + msg->len;
3  total_mem = max_len;
4  if(max_len < MIN) {
5    total_mem = MIN;
6  }
7  if(total_mem >= MAXMEM) {
8    return -EINVAL;
9  }
10 return send_msg(msg, buf);
```

Listing 3: Patch affecting the control-flow of a function.

To check that our error-block detection approach is accurate, we randomly sampled 100 patches, and we verified that all the error basic blocks that we identified are indeed valid BB_{err}s.

As explained in Section III, SPIDER discards all changes that happen within the identified BB_{err}s. In Appendix I, we run SPIDER in "not ignoring mode" (*NoEB*), in which we do not ignore changes within BB_{err}s and show that the detection rate does not materially change (a 0.79% decrease; see, (Default - *NoEB*) in Table IV). This result shows that discarding the changes within error-handling basic blocks does not significantly influence the effectiveness of SPIDER.

E. Patch Analysis

In the remaining part of this section, we explain how SPIDER identifies *sp*s based on the general technique described in Section III.

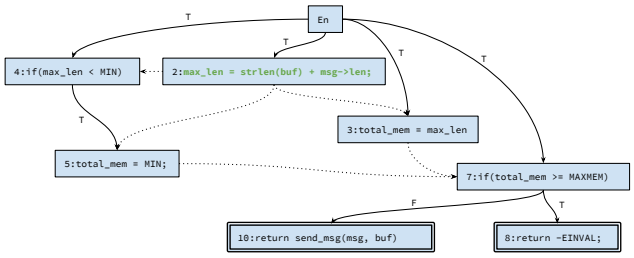


Fig. 4:
Program-Dependency Graph of the *patched* function in Listing 3.

Given the PDG, we remove all the data-dependency and control-dependency cycles in the PDG by removing all the back edges [23]. Given a statement t, we consider an edge to be a back edge if it is originated from a statement that is dominated by t in the PDG. We provide more details in Appendix F, and argue that this technique is safe, if the patch does not contain any directly affected statements within a loop. However, if a patch *directly* modifies any statements within the loop this may not be safe, as proving the condition $C2$ becomes undecidable [46]. To be safe, a patch that directly modifies a statement within a loop will not be considered as an *sp*.

Using the diff-annotated CFG of the patched function, first we find all the directly affected statements. As explained in Section III, these are the statements that are directly modified by the patch.

Second, given the PDG, we follow the edges from the nodes corresponding to the directly affected statements to identify all the statements that are reachable, which represent the indirectly affected statements. The union of the directly and indirectly affected statements is our total affected statements. As mentioned in Section III, we ignore the affected statements that belong to the error-handling basic blocks ($BB_{err}s$).

Verifying non-increasing input space (C1): To verify condition $C1$, we first check if any of the affected statements is a conditional statement. By the definition of *PDG* (Section III-A), these are the nodes that have an outgoing control dependency (solid) edge.

If there are affected conditional statements, then we find all the valid exit points, i.e., the valid `return` statements (or *VEP*). For each statement in the *VEP*, we identify the conditional statements that are part of the path constraint by following the solid edges backward until E_n.

Given a path constraint, we convert each of the conditional in the path constraint into a symbolic expression. As explained in Section III-B4, we start by initializing each of the variables with unique symbolic values in the original and patched function. Therefore, if a variable is *not* modified by the patch, it will have the *same* symbolic value in both the original and patched functions.

Conversion to symbolic expression: For a statement to be converted into symbolic expression, its data dependencies need to be first converted to symbolic expressions as well.

Therefore, given a statement s, we first check if it has any incoming data dependency edges, if this is the case, we go to the parent and try to repeat this process backward in a breadth-first manner until we find all the nodes with no incoming data dependency edges, i.e., the nodes from which all the data-dependency paths are complete (Section III-A).

We call the nodes with no incoming data dependency edges as *free nodes*. We first convert each of the free nodes into symbolic values by following the corresponding instruction semantics (as shown in Table I).

We then forward-propagate the values from the free nodes to the statements along the data dependency edges until we reach s.

To interpret function calls, we create a new symbolic value based on the hash of the function name *and* the symbolic values of its arguments. For instance, for the call `strlen(buf)`, we create a symbolic value with name equal to `hash(strlen, sym(buf))`, where `sym(buf)` is the symbolic value of the variable `buf`.

When multiple definitions of a variable reach an instruction, we use conditional symbolic variables based on the path constraint of the stricter path. For a variable x, if two definitions d_1 and d_2 from statements v_1 and v_2, respectively, reach a statement v_3. Then the symbolic value of x at v_3 would be:

$$v_3(x) = \begin{cases} Ite(PC(v_1),d_1,d_2) & if\ PC(v_1) \rightarrow PC(v_2) \\ Ite(PC(v_2),d_2,d_1) & otherwise \end{cases} \quad (5)$$

Where $Ite(c,a,b)$ represents an `if-then-else` symbolic value, which dictates to use the value a if c is satisfiable else b, PC is the path constraint, and, from the rules of implication, $PC(c_1) \rightarrow PC(c_2)$ indicates that $PC(c_1)$ is a stricter condition than $PC(c_2)$. The Equation 5 correctly handles multiple definitions. We refer all the interested readers to Appendix H, where we explain this in detail.

Consider the statement at line 7 in the *PDG* of Figure 4. Here, multiple definitions of the variable `total_mem` reach line 7. i.e., from line 3 and 5. By using the initial symbolic values, for `strlen(buf)` = $sym1$, `msg->len` = $sym2$, `MIN` = $sym3$, and `MAXMEM` = $sym4$. The definitions of `total_mem` at line 3 and 5 are $sym1 + sym2$ and $sym3$, respectively. The path constraint for line 3 and 5 are $PC(3)$ = (`En==T`) and

$PC(5) = (\texttt{(En==T)} \wedge ((sym1 + sym2) < sym3))$, respectively. We can see that $PC(5) \rightarrow PC(3)$, as $PC(5)$ is a stricter condition, consequently the symbolic value of `total_mem` at line 7 will be: $Ite(PC(5), sym3, (sym1 + sym2))$.

The symbolic expression for the path constraint of the valid `return` (line 10) in the patched function from the *PDG* of Figure 4 would be $(E_n == T \wedge (Ite(PC(5), sym3, (sym1 + sym2)) >= sym4))$, for brevity we did not expand $PC(5)$, but the actual symbolic expression would be only in terms of initial symbolic values.

Following the steps described above, we convert the path constraints of each of the valid `returns` in the patched function to symbolic expressions. Then we obtain Equation 3 by the disjunction of the symbolic expressions. Finally, we convert the disjuncted symbolic expression into a Z3 [34] expression, i.e., $vinputs(f_p)$ (see Section III-B3).

We follow the same steps in the original function to compute $vinputs(f)$, then, using the Z3 tool once again, we verify the implication $vinputs(f_p) \rightarrow vinputs(f)$, thus proving that the patch satisfies condition $C1$.

Verifying output equivalence (C2): Given the list of affected non-control-flow statements, as explained in Section III-B4, we only consider the statements that update the non-local state of the function, i.e., the function output.

Consider the patch in Listing 1, where, although all the statements are affected by the patch, the only statements of interest are at line 17 and 19, as they update the heap and return value.

As explained in Section III-B4 and shown in Table I, we compute the symbolic expressions along each complete data dependency path along with the corresponding path constraints.

Finally, we convert the symbolic expressions into Z3 expression and verify Equation 4 using Z3. This verifies that the function affected by the patch satisfies condition $C2$. Note that the patch showed in Listing 3 changes only local variables and thus the output of the function remains the same as that of the original function for all valid inputs, thus satisfying condition $C2$.

We follow the above steps for each of the functions modified by the patch. We consider a patch to be a *safe patch*, only when $C1$ and $C2$ can be proved by following the steps described above.

Handling library functions: As explained in Section III, we consider patches that have only locally analyzable statements, i.e., patches that do not directly affect function calls and pointers. However, we noticed that there are certain library functions, whose effects can be easily summarized. Such as, `memset`. There are other print and logging library functions, like `printf` and `printk`, that do not affect the output of the patched function.

To handle this, we maintain a few categories of commonly used, well-known library functions (see Appendix G), whose effects can be either summarized or ignored.

V. Assumptions

Our implementation as specified in Section IV-E tries to guarantee that a patch is a safe patch. However, a careful reader might have noticed that there are certain assumptions made by our implementation. In this section, we explicitly describe the assumptions in our implementation:

Non-alias dependencies: As explained in Section IV-E, we use a *PDG* based on variables to compute all the affected statements. However, this ignores the data dependencies that could happen through pointers [19]. Handling this requires precise pointer analysis, which in turn require access to the whole program violating our requirement $R1$.

Pure functions: We consider all functions to be pure functions [75], i.e., the output of a function only depends on the input arguments. In other words, multiple calls to a function with the same arguments results in the same output. Furthermore, reordering function calls without any change to the arguments will also be treated as equivalent. That is, `f1(arg1); f2(arg2);` is equivalent to `f2(arg2); f1(arg1);`. However, there could exist impure functions, whose output could also depend on the global state of the program. Soundly detecting whether a function is impure requires analyzing the function and its callees, which is not scalable and requires resolving function pointers.

Conditional compilation: The preprocessor conditional code directives (e.g., `#ifdef`, `#ifndef`, `#else`, etc.) allow different pieces of code to be be compiled depending on the values of certain preprocessor variables. We use the *unifdef* tool to handle these conditional compilation directives. *unifdef* attempts to obtain maximal code by enabling all preprocessor variables. However, for `#ifdef`-`#else` constructs, to be consistent, it has to select the code either under the `if` or the `else` directive. This could result in certain statements in the patch (which are controlled by preprocessor variables) to be invisible to SPIDER, and, in turn, this could lead to false positives. Handling conditional code compilation precisely requires analyzing the patch under all possible values of preprocessor variables and their combinations. This is not scalable for large codebases like the Linux kernel. To handle this, we allow users to enable the *no preprocessor mode* (*NoPP*). In *NoPP* mode, any patch that affects statements controlled by preprocessor variables will *not* be considered as an *sp*. We show in Appendix I that in *NoPP* mode, the detection rate of SPIDER does not vary much (a decrease of 1.15%, see (Default - *NoPP*) in Table IV). Furthermore, we also allow the user to specify the values of preprocessor variables, which can be used to get the correct source file instead of the conservative *NoPP* mode.

We consider the limitations above to be fundamental implications of our requirements $R1$ and $R2$. Nonetheless, we believe that our system provides a reasonable approach to identify safe patches. Moreover, if these assumptions are considered too strong, it is always possible to fall back to the more conservative Security Patch (SeP) mode (see Section VI for details). Finally, it is also possible to use our system to rank patches and prioritize those identified as safe for manually vetting (and testing).

VI. Security Patch mode

As explained in Section I, there could exist security patches without a corresponding CVE entry. To verify this, we have a configuration of SPIDER called *Security Patch (SeP) mode* that identifies security patches with *no false positives*, i.e., all the patches identified by this configuration are indeed security patches. SeP is based on the intuition that most of the security patches add additional input validation checks. Therefore, in SeP mode, we restrict ourselves to safe patches that affect only control-flow statements. Furthermore, when the commit message is available, we use the technique proposed by Zhou et al. [79] to filter out non-security related fixes. However, there can be false negatives, that is, potential security patches not detected as such.

Note that while SeP mode is more limited in the patches that it considers safe, it does not rely on any of the assumptions discussed

in Section V. We believe that SeP mode of SPIDER is the first step towards a practical solution of automatically identifying security patches that could be easily integrated into any source-control system. We plan to integrate SeP mode of SPIDER into GitHub security alerts [8], which helps both the developers and maintainers to handle security patches. Note that our running example (shown in Listing 1), although being a security patch, is not detected by the SeP mode because it also affects non-control flow instructions.

VII. EVALUATION

We evaluate the effectiveness of SPIDER in three different ways. First, we run it on a large dataset of 341,767 changes (i.e., commits) spanning over 32 repositories, collected from the year 2016 for a total of 32 months, in order to understand if it actually detects *sp*s, according to our definition (see Section VII-A). Second, we run SPIDER on a set of security patches (i.e., CVE patching commits) to evaluate the usefulness of this tool in speeding the propagation of these critical fixes. Third, in Section VII-D, we show a way to use the SeP mode of SPIDER as a vulnerability finding tool by identifying non-CVE security patches that are missing in various active forks of the analyzed projects. Finally, we show in Section VII-E, that there are several non-CVE security patches in the Linux kernel and many of these are still unpatched, at the time of writing, on some of its Android-related forks: this provides real examples where SPIDER can be useful in fixing potential *n-day* vulnerabilities.

The analysis that SPIDER performs, described in Section IV, is an intra-procedural static analysis that does not consider the interaction between different modified functions. For this reason, to isolate the effect of these interactions that represent a possible confounding factor, we evaluate SPIDER only on patches that affect a single C source file (i.e., `.c` format only). All the patches studied in our evaluation are real changes extracted from repositories of widely used open-source projects (see Section VII-A for more details).

Performance: On average SPIDER took 3.4 seconds to analyze a patch on a machine equipped with a two-core 2.40 GHz CPU, and 8GB RAM, demonstrating its speed and scalability.

Active forks: We noticed that most of the forks of repositories are inactive or dead, i.e., there are no new commits made to the repository since they are forked. Considering such inactive forks could exaggerate our results, and, therefore, we considered only active forks. We consider a fork to be active if it has at least ten new commits in the last six months, and using this filter, we were able to eliminate a number of forks. For instance, in the case of Linux kernel (ID 1), we consider only 269 active forks out of 23,854 forks.

A. Large-scale evaluation

We ran SPIDER on a large set of patches: we selected 32 open-source projects widely used by desktop, mobile, and embedded operating systems, and we collected from each of them all the single-C-file commits for the past 32 months from the time of writing (considering merges as single commits). All the details of the projects are shown in Table II.

B. Effectiveness of patch analysis

Table II also shows the number of *sp*s identified by SPIDER in the dataset. Over the total 341,767 commits studied, SPIDER identified 67,408 (19.72%) safe patches (Column 6). Furthermore, 58.72% of these patches are missing in at least one of the active forks (MIAFs).

Checking for patch applicability: We use the following syntactic approach to identify whether a patch of a project is applicable to (or missing from) a fork or other projects. Given a patch, we extract the affected file's source code before the patch (i.e., original file) and compare it to the latest version of the corresponding file in the fork. If the file is present in the fork *and* all the functions affected by the patch do not differ between the original file of the patch and the corresponding latest file in the fork, then it means that the patch can be applied to the fork. To perform the comparison, we use the *git diff* tool, and check that there are no modifications in the targeted functions.

It is interesting to note that, across all repositories, the percentage of *sp*s mostly stays around 20%-25%, without much variation. There are certain projects where the percentage of detected *sp*s is low, such as IDs 15 and 16. This low detection rate is mainly because of the inherently complex code and patches. We refer all the interested readers to Appendix C, where we explain this in detail.

Listing 6 shows a patch identified as an *sp*, where the patch modifies error basic blocks, which are ignored. Also, the patch moves certain function calls `Py_INCREF` and `Py_DECREF`. However, as the arguments to these calls (i.e., `dll` and `ftuple`) are not modified by the patch, the symbolic expressions of the arguments are proved to be equivalent by Z3, resulting in the patch being considered an *sp*. We show a few other *sp*s in Appendix E.

Looking at these results, we argue that SPIDER would be helpful for project maintainers and could be directly used to port the fixes or to prioritize the changes that must be ported.

C. Evaluation on CVEs

We wanted to determine how many security patches are indeed *sp*s, as claimed in Section I. To this end, we collected all the patching commits linked as reference fixes for kernels CVEs from the Android security bulletins [1], and, similar to the large-scale evaluation, we studied only the CVEs that patch a single C file. We also collected all the CVEs for the remaining repositories over the same amount of time. This resulted in the analysis of 809 CVE patches.

Table III shows the results obtained after running SPIDER on these patches, which show that 55.37% of the CVE-patching commits are non-disruptive, while on generic patches (i.e., Table II) the percentage was 19.72%. This finding shows that SPIDER could be useful not only to speed-up the process of selecting and applying a significant number of changes (as shown in Section VII-A) but also to apply more than half of the security patches in a faster way.

Listing 5 shows an example of CVE patching commit from Android security bulletin identified as a *sp* by SPIDER. Listing 5 is also one of the CVEs that we mentioned in Section I, which was patched in Android more than a year after the appearance of the corresponding entry in the database.

Looking at Table III, it is interesting to see that SPIDER performed relatively well with more than 50% success rate in all but OpenSSL and VLC CVEs. This is because the security fixes in these software packages are complex. More details can be found in Appendix D.

D. Security patches missing a CVE number

We used SPIDER in SeP mode on all the commits to identify security patches. We then checked if these patches have an associated CVE number. Listing 6 and 4 show examples of security patches missing CVE entries, which are detected by the SeP mode of SPIDER.

ID	Project	Studied git branch/tag	Commits	Active forks	*sps* (% over commits)		Non-CVE security patches	
					Total (%)	MIAFs (%)	Total	MIAFs (%)
Linux Kernels								
1	Linux kernel mainline[3]	master	102,607	269	20,171 (19.66%)	9,427 (46.74%)	635	297 (46.77%)
2	Linaro ARM Linux kernel[4]	optee	96,990	7	19,172 (19.77%)	6,846 (35.71%)	587	211 (35.95%)
3	Rasberry Pi Linux kernel[5]	rpi-4.14.y	54,585	168	11,250 (20.61%)	10,511 (93.43%)	394	362 (91.88%)
Android Kernels								
4	Qualcomm Msm Android kernel[6]	android-msm-angler-3.10-oreo-r6	472	N/A	128 (27.12%)	N/A	8	0 (0.0%)
5	NVIDIA Tegra Android kernel[7]	android-tegra-dragon-3.18-oreo-r6	297	N/A	88 (29.63%)	N/A	9	0 (0.0%)
6	Xiaomi Android kernel[8]	sagit-o-oss	9,718	85	2,332 (24.0%)	2,332 (100.0%)	94	94 (100.0%)
7	Android x86_64 kernel[9]	android-8.0.0_r0.23	211	N/A	56 (26.54%)	N/A	4	0 (0.0%)
8	Xperia Android kernel[10]	aosp/LA.UM.6.4.r1	10,932	69	2,559 (23.41%)	2,554 (99.8%)	99	99 (100.0%)
9	Base Android Kernel[11]	android-4.9-o	21,927	39	4,967 (22.65%)	4,330 (87.18%)	201	173 (86.07%)
Bootloader and Firmware								
10	LK Embedded kernel[12]	master	30	26	7 (23.33%)	7 (100.0%)	0	N/A
11	Qualcomm LK Kernel[13]	lk.lnx.1.0.r21-rel	219	N/A	42 (19.18%)	N/A	5	0 (0.0%)
Trusted Operating Systems								
12	OP-TEE Trusted OS[14]	master	499	45	96 (19.24%)	71 (73.96%)	4	3 (75.0%)
OpenBSD								
13	OpenBSD[15]	master	8,651	14	1,424 (16.46%)	843 (59.2%)	42	30 (71.43%)
Windows Compatible OS								
14	reactos-Open Source Windows Compatible OS[16]	master	2,936	35	510 (17.37%)	123 (24.12%)	11	1 (9.09%)
Interpreters								
15	Python Interpreter[17]	master	862	207	108 (12.53%)	38 (35.19%)	12	2 (16.67%)
16	PHP Interpreter[18]	master	3,096	289	344 (11.11%)	274 (79.65%)	7	3 (42.86%)
Graphical Subsystems								
17	nautilus-Ubuntu default graphical subsystem[19]	master	591	N/A	129 (21.83%)	N/A	2	0 (0.0%)
18	winfile-Windows File Manager[20]	master	28	23	2 (7.14%)	1 (50.0%)	0	N/A
19	X Windows Subsystem[21]	master	644	1	119 (18.48%)	60 (50.42%)	5	2 (40.0%)
Multimedia								
20	VLC Player[22]	master	6,815	98	1,025 (15.04%)	778 (75.9%)	34	28 (82.35%)
21	FFmpeg-multimedia processing tools[23]	master	6,538	449	697 (10.66%)	573 (82.21%)	42	37 (88.1%)
Distributed Databases								
22	redis-In memory Database[24]	unstable	1,385	659	113 (8.16%)	69 (61.06%)	11	7 (63.64%)
Utilities								
23	Tmux-Terminal Multiplexer[25]	master	543	77	110 (20.26%)	86 (78.18%)	5	4 (80.0%)
24	curl-transfer a URL[26]	master	984	239	100 (10.16%)	70 (70.0%)	1	0 (0.0%)
25	evince-GNOME document viewer[27]	debian/master	202	N/A	51 (25.25%)	N/A	4	0 (0.0%)
26	Git-version control system[28]	master	2,834	526	360 (12.7%)	286 (79.44%)	15	13 (86.67%)
27	GDB-GNU Debugger[29]	master	55	25	13 (23.64%)	9 (69.23%)	1	1 (100.0%)
28	OpenVPN-Open source VPN daemon[30]	master	201	70	18 (8.96%)	11 (61.11%)	0	N/A
29	Systemd System[31]	master	4,815	N/A	1,067 (22.16%)	N/A	29	0 (0.0%)
Libraries								
30	libpng-PNG reference library[32]	libpng16	82	42	1 (1.22%)	1 (100.0%)	0	N/A
31	OpenSSL-Open Source TLS toolkit[33]	master	1,998	290	347 (17.37%)	283 (81.56%)	17	16 (94.12%)
32	glibc-GNU libc[34]	master	20	7	2 (10.0%)	2 (100.0%)	0	N/A
Total			341,767	3,759	67,408 (19.72%)	39,585 (58.72%)	2,278	1,383 (60.71%)

TABLE II: Overall results of our large-scale evaluation. Active forks are computed for only GitHub hosted projects. The percentage for Total *sps* is over commits for the corresponding projects. MIAFs are percentage over total *sps* and non-CVE commits that are missing in at least one active fork.

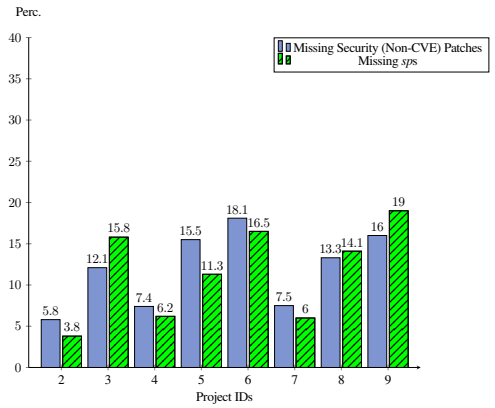


Fig. 5: Distribution of the security (non-CVE) patches (identified by the SeP Mode of SPIDER) and *sp*s in Mainline Linux kernel (Project ID 1) that are missing in other related kernel projects.

```
                error = -EINVAL;
                goto out_put_tmp_file;
        }

+       if (f.file->f_op != &xfs_file_operations ||
+           tmp.file->f_op != &xfs_file_operations) {
+               error = -EINVAL;
+               goto out_put_tmp_file;
+       }
+
        ip = XFS_I(file_inode(f.file));
        tip = XFS_I(file_inode(tmp.file));
```

Listing 4: a security patch identified by SPIDER on the main Linux kernel repository (commit 3e0a3965464505). *which does not have a corresponding CVE ID.*

CVE patches source	*sp*s / CVE
Linux	333 / 611 (54.5%)
Android bulletin	98 / 164 (59.75%)
OpenBSD	5 / 6 (83.33%)
OpenSSL	7 / 21 (33.33%)
Systemd	4 / 4 (100%)
VLC	1 / 3 (33.33%)
Total	448 / 809 (55.37%)

TABLE III: Results of SPIDER on CVE patches.

```
 int check_aboot_addr_range_overlap
      (uint32_t start, uint32_t size)
 {
         /* Check for boundary conditions. */
-        if ((start + size) < start)
+        if ((UINT_MAX - start) < size)
                 return -1;
```

Listing 5: Real integer overflow patch identified as *sp* by SPIDER (CVE-2014-9795 from July 2016 Android security bulletin).

The last two columns of Table II shows the complete results: Overall SPIDER identified 2,278 security patches across all the repositories. After manual verification, we found these results to be correct. This shows that CVE IDs are not always used for security patches, and that relying on them is not an effective way to secure related repositories. The number of patches identified by the SeP mode is smaller compared to the total number of *sp*s (i.e., 2,278 $\ll$ 67,408). This is because the SeP mode, as explained in Section VI, imposes strict requirements. Nonetheless, the SeP mode identified 2,278 security patches missing a CVE number.

Furthermore, 60.71% of these patches are missing in at least one of the active forks denoted as MIAFs. This is alarming, as these cases reveal unpatched security vulnerabilities in the forks, which could be exploited by a motivated attacker monitoring the patches. We observed a considerable number of patches (for example see Listing 10), where the commit message contains the vulnerability-triggering input, further reducing the effort for the attacker.

```
-Py_INCREF(dll); /* for KeepRef */
-Py_DECREF(ftuple);
-if (!_validate_paramflags(type, paramflags))
+if (!_validate_paramflags(type, paramflags)) {
+   Py_DECREF(ftuple);
    return NULL;
+}
self = (PyCFuncPtrObject
      *)GenericPyCData_new(type, args, kwds);
-if (!self)
+if (!self) {
+   Py_DECREF(ftuple);
    return NULL;
+}
...
*(void **)self->b_ptr = address;
+Py_INCREF(dll);
+Py_DECREF(ftuple);
```

Listing 6: A non-CVE security patch (commit d77d97c9a1f) fixing a reference counting vulnerability in the Python interpreter identified by SPIDER. *This patch does not have a corresponding CVE ID.*

E. Missing patches in vendor kernels

To identify missing patches in vendor kernels, we check how many of the Linux Kernel mainline commits identified as *sp*s still have to be applied to one or more of the eight vendor kernel repositories that we studied (i.e., projects 2 - 9 in Table II), at the time of writing. To do that, given a commit identified as an *sp*, we extract the affected file's source code before the change, and we compare it to the same file, if present, in all the listed kernel repositories (Table II show the git branch or tag that we studied) using the `git diff` technique described in Section VII-B.

The stripe bars in Figure 5 shows the percentage of missing *sp*s in different vendor kernels. We found that 9,427 of the 20,171 Linux kernel identified *sp*s (i.e., 46.74%) are still not applied in at least one of the considered vendor kernels. A significant portion of these changes not considered useful by the maintainers (e.g., removals of unused code, small refactoring, etc.), and therefore, not imported. However, we found out that 297 of them are CVE patching commits (i.e., the ones that we linked to the corresponding CVEs, as shown in Section VII-C) that still have to be imported by the maintainers of some repositories: this supports the findings of previous studies [57], [53], [22], [41] that report that vulnerability databases are not always effective in speeding the propagation of security fixes.

Unfixed vulnerabilities in vendor kernels: We also checked the security patches (which do not have a CVE number) identified by the SeP mode in the Linux Kernel mainline that still have to be applied to one or more of the eight Linux Kernel repositories that we studied (i.e., projects 2 - 9 in Table II). The plain bars in Figure 5 show the percentage of missing non-CVE security patches in different vendor kernels. There are in total **229** security patches that do not have a corresponding CVE number and are missing on different kernel repositories, including the ARM Linux kernel main repository (i.e., project 2 in Table II): these can be seen as potential unfixed or *n-day* vulnerabilities. Given their potential severity, we manually verified them to assess their impact. For a few of these vulnerabilities, the impact is less severe because of the variation in kernel configurations. However, we found several missing patches

in critical components like `netfilter`, which applies to all kernel configurations. The snippet of a non-CVE security patch that is missing in the msm kernel (ID 4) is shown in Listing 11, this patch, as mentioned before also contains the triggering input.

We are in the process of reporting all of these patches to the corresponding project maintainers and vendors, and submit all the necessary requests for CVEs.

VIII. Limitations

Along with the assumptions described in Section V, SPIDER comes with several limitations. Specifically,

Small patches: As we can see from Figure 6, the majority (57.1%) of the patches detected as *sp*s are small (0-5 lines). Furthermore, SPIDER cannot verify patches that modify statements within a loop. These limitations are mainly because SPIDER tries to verify a patch to be safe in a sound way. We believe it is important to have a system with no false positives, that provides stronger guarantees, and that can be used by the maintainers safely.

Syntactic approach for patch applicability check: We use a syntactic approach to check for patch applicability in the related repositories. However, a patch although syntactically applicable to a file in a project may not be semantically applicable because the condition fixed by the patch could be impossible to occur in the project [40]. This limitation is induced by our requirement R1, as checking for semantic applicability of patches require sound static analysis techniques which require build environment and access to all source files, thus violating our requirement R1.

Heuristic approach for error-handling basic blocks detection: As explained in Section IV-D, we use a heuristic approach to identify error-handling basic blocks. However, these heuristics may not hold for other projects resulting in cases where a basic block matching our heuristics is not a true error-handling basic block. Consequently, we could have unsafe patches being identified as safe. To handle this, we provide the *NoEB* mode of SPIDER (Appendix I) where we do not ignore the changes in the error-handling basic blocks. This mode provides a safer version of SPIDER, albeit with a slight decrease in detection rate.

Susceptible to adversarial evasion: As a consequence of our assumptions (Section V), SPIDER is susceptible to adversarial evasion. For instance, as we treat macros as function calls, an adversarial developer or contributor could use macro calls to make SPIDER consider otherwise safe patches as unsafe. However, as we explained in Section I, the main use case of SPIDER is for developers and maintainers. Furthermore, we assume developers to be non-malicious users who want to ensure that their applications are as secure as possible.

Tool dependencies: The current implementation of SPIDER works only on C source code; however, the parser that we use should be easily extensible to other languages. The fine-grained diff step is language agnostic, thus, to extend the tool to other languages, we would only need to add language-specific heuristics and preprocessing. A good solution would be to have a configurable front end for different languages, similar to LLVM [47]. As our implementation is based on Joern and Gumtree, we also share the same limitations that these tools have.

IX. Related work

Source code changes and patches as research topics received a lot of attention in the past decade. This section covers a comprehensive portion of prior work on these topics (Section I already covered state-of-the-art code analysis techniques that were used in the field, thus they will not be covered here).

Vulnerability finding and exploitation: In Section VII-E we show how SPIDER can be used to find instances of unpatched code starting from the identified *sp*s, including vulnerabilities. Finding *unpatched code clones* is the focus of most of the prior research on patches in the security field [50], [41], [49]. In this work, we do not look for code clones but for instances where the function affected by a patch is still equal to the unpatched version. Brumley et al. [28], instead, show how to generate exploits for a vulnerability starting from the corresponding patch.

Easing the patching process: Prior research has been very active in designing approaches and building tools to ease and speedup the process of patching [59], [71], [20], However, most of these techniques target only specific bug classes [55].Other studies concentrate on helping developers in applying systematic changes [78], [54]. Long et al. [51], in contrast with the previously mentioned studies, use machine learning to model correct code and generate generic defects fixes, but do not focus on propagating existing patches as we do in this study. Similar to what we do in this work, Kreutzer et al. [43] use AST differencing on changes to extract metrics to help cluster the changes by similarity.

Software evolution: Mining software repositories is a well-known technique to gain insights into the dynamics of software evolution [39], [38]. Perl et al. [61] built VCCFinder, a tool that leverages code metrics and patch features (e.g., keywords in commits) to identify vulnerability-contributing changes. However, in this work, we do not rely on the commit messages, and, instead perform a systematic analysis of the patches.

X. Conclusion and future work

In this work, we designed, implemented, and evaluated SPIDER, a fast and lightweight tool (**R2**) based on our *sp* identification approach that can determine if a patch is safe using only the original and the patched source code of the affected file (**R1**), without the need for external information (e.g., build environment, commit message, etc.). Our large-scale evaluation on 341,767 commits extracted from 32 different open-source repositories, and on 809 CVE patches, demonstrates the effectiveness of SPIDER, and shows that a significant amount of security patches could have been automatically identified (i.e., 55.37%). Furthermore, we show how the SeP mode of SPIDER can be used to find unpatched security issues.

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd Alina Oprea for their valuable comments and input to improve our paper.

This material is based upon work supported by AFRL under Award No. FA8750-19-C-0003, by ONR under Award No. N00014-17-1-2011, and by NAVSEA under Award No. N00024-12-C-6404/0451. This research was also sponsored by DARPA under agreement number HR001118C0060. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the U.S. Government, or the other sponsors.

NOTES

[1] http://dotat.at/prog/unifdef/

[2] code.google.com/archive/p/java-diff-utils/

[3] https://github.com/torvalds/linux.git

[4] https://github.com/linaro-swg/linux.git

[5] https://github.com/raspberrypi/linux.git

[6] https://android.googlesource.com/kernel/msm

[7] https://android.googlesource.com/kernel/tegra

[8] https://github.com/MiCode/Xiaomi_Kernel_OpenSource.git

[9] https://android.googlesource.com/kernel/x86_64.git

[10] https://github.com/sonyxperiadev/kernel.git

[11] https://github.com/aosp-mirror/kernel_common.git

[12] https://github.com/littlekernel/lk.git

[13] https://source.codeaurora.org/quic/la/kernel/lk

[14] https://github.com/OP-TEE/optee_os.git

[15] https://github.com/openbsd/src.git

[16] https://github.com/reactos/reactos.git

[17] https://github.com/python/cpython.git

[18] https://github.com/php/php-src.git

[19] https://gitlab.gnome.org/GNOME/nautilus.git

[20] https://github.com/Microsoft/winfile.git

[21] https://github.com/mirror/xserver.git

[22] https://github.com/videolan/vlc.git

[23] https://github.com/FFmpeg/FFmpeg.git

[24] https://github.com/antirez/redis.git

[25] https://github.com/tmux/tmux.git

[26] https://github.com/curl/curl.git

[27] https://salsa.debian.org/gnome-team/evince.git

[28] https://github.com/git/git.git

[29] https://github.com/bminor/binutils-gdb.git

[30] https://github.com/OpenVPN/openvpn.git

[31] https://github.com/systemd/systemd.git

[32] https://github.com/glennrp/libpng.git

[33] https://github.com/openssl/openssl.git

[34] https://github.com/bminor/glibc.git

REFERENCES

[1] 2016 android security bulletins. source.android.com/security/bulletin/2016.html. Accessed: 2017-02-11.

[2] Android 10 msm kernel. https://android.googlesource.com/kernel/msm/+/refs/tags/android-10.0.0_r0.16. Accessed: 2019-10-28.

[3] Apple's goto fail bug. https://www.imperialviolet.org/2014/02/22/applebug.html. Accessed: 2017-02-13.

[4] The biggest and weirdest commits in linux kernel git history. www.destroyallsoftware.com/blog/2017/the-biggest-and-weirdest-commits-in-linux-kernel-git-history. Accessed: 2017-02-15.

[5] Community reaction to delayed patching. https://twitter.com/RatedG4E/status/760322614912954368. Accessed: 2017-02-13.

[6] Error handling via goto in c. https://ayende.com/blog/183521-C/error-handling-via-goto-in-c. Accessed: 2019-07-13.

[7] [extended version] spider: Enabling fast patch propagation in related software repositories. https://drive.google.com/file/d/1ZXYv6YjXNgj7-_WbyrsrxwF9Y8aKuVth/view?usp=sharing. Accessed: 2019-07-31.

[8] Introducing security alerts on github. https://github.com/blog/2470-introducing-security-alerts-on-github. Accessed: 2017-02-13.

[9] Irb human subject regulations exempt decision charts. https://www.hhs.gov/ohrp/regulations-and-policy/decision-charts/index.html. Accessed: 2017-02-13.

[10] libebml fixed a vulnerability but no cve was assigned. https://twitter.com/wdormann/status/1154138404910768134. Accessed: 2017-07-25.

[11] libembl security fix without a cve id. https://github.com/Matroska-Org/libebml/commit/05beb69ba60acce09f73ed491bb76f332849c3a0. Accessed: 2017-07-25.

[12] Linux kernel configuration. http://www.tldp.org/HOWTO/SCSI-2.4-HOWTO/kconfig.html. Accessed: 2017-02-13.

[13] Openssl bug fix for cve-2016-0703. https://git.openssl.org/?p=openssl.git;a=commit;h=ae50d8270026edf5b3c7f8aaa0c6677462b33d97. Accessed: 2017-02-13.

[14] Use of goto in systems code. https://blog.regehr.org/archives/894. Accessed: 2019-07-13.

[15] Using goto in linux kernel code. https://koblents.com/Ches/Links/Month-Mar-2013/20-Using-Goto-in-Linux-Kernel-Code/. Accessed: 2019-07-13.

[16] Vlc media player affected by a major vulnerability in a 3rd library, libebml. https://hub.packtpub.com/vlc-media-player-affected-by-a-major-vulnerability-in-a-3rd-library-libebml-updating-to-the-latest-version-may-help/. Accessed: 2017-07-25.

[17] J. Admanski and S. Howard. Autotest-testing the untestable. In *Proceedings of the Linux Symposium*. Citeseer, 2009.

[18] F. E. Allen. Control flow analysis. In *ACM Sigplan Notices*, vol. 5, pp. 1–19. ACM, 1970.

[19] W. Amme and E. Zehendner. Data dependence analysis in programs with pointers. *Parallel Computing*, 24(3-4):505–525, 1998.

[20] J. Andersen, A. C. Nguyen, et al. Semantic patch inference. In *Proceedings of the ACM International Conference on Automated Software Engineering (ASE)*. 2012.

[21] G. Antoniol, K. Ayari, et al. Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*. 2008.

[22] A. Arora, R. Krishnan, et al. An empirical analysis of software vendors' patch release behavior: impact of vulnerability disclosure. *Information Systems Research*, 21(1):115–132, 2010.

[23] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. 2005.

[24] G. Bavota. Mining unstructured data in software repositories: Current and future trends. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2016.

[25] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSME)*. 1992.

[26] D. Binkley, R. Capellini, et al. An implementation of and experiment with semantic differencing. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSME)*. 2001.

[27] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In *Proceedings of the Annual International Cryptology Conference (CRYPTO)*. 1998.

[28] D. Brumley, P. Poosankam, et al. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 2008.

[29] R. P. Buse and W. R. Weimer. Automatically documenting program changes. In *Proceedings of the ACM International Conference on Automated Software Engineering (ASE)*. 2010.

[30] C. Cadar, P. Godefroid, et al. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 2011.

[31] J. M. Cardoso and P. C. Diniz. Modeling loop unrolling: Approaches and open issues. In *Proceedings of the International Workshop on Embedded Computer Systems (SAMOS)*. 2004.

[32] F. Chow, S. Chan, et al. A new algorithm for partial redundancy elimination based on ssa form. In *ACM Sigplan Notices*, vol. 32, pp. 273–286. ACM, 1997.

[33] R. Clarke, D. Dorwin, et al. Is open source software more secure? *Homeland Security/Cyber Security*, 2009.

[34] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2008.

[35] J.-R. Falleri, F. Morandat, et al. Fine-grained and accurate source code differencing. In *Proceedings of the ACM International Conference on Automated Software Engineering (ASE)*. 2014.

[36] J. Ferrante, K. J. Ottenstein, et al. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1987.

[37] D. Gao, M. K. Reiter, et al. Binhunt: Automatically finding semantic differences in binary programs. In *Proceedings of the International Conference on Information and Communications Security (ICICS)*. 2008.

[38] E. Giger, M. Pinzger, et al. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 2011.

[39] A. E. Hassan. The road ahead for mining software repositories. In *Proceedings of the IEEE International Conference on Software Maintenance (FOSM)*. 2008.

[40] Z. Huang, D. Lie, et al. Using safety properties to generate vulnerability patches. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 2019.

[41] J. Jang, A. Agrawal, et al. Redebug: finding unpatched code clones in entire os distributions. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 2012.

[42] Y. Kang, B. Ray, et al. Apex: Automated inference of error specifications for c apis. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*. 2016.

[43] P. Kreutzer, G. Dotzler, et al. Automatic clustering of code changes. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 2016.

[44] S. K. Lahiri, C. Hawblitzel, et al. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 2012.

[45] S. K. Lahiri, K. Vaswani, et al. Differential static analysis: Opportunities, applications, and challenges. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER)*. 2010.

[46] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[47] C. Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pp. 1–2. 2008.

[48] F. Li and V. Paxson. A large-scale empirical study of security patches. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017.

[49] H. Li, H. Kwon, et al. A scalable approach for vulnerability discovery based on security patches. In *Proceedings of the International Conference on Applications and Techniques in Information Security (ATIS)*. 2014.

[50] Z. Li, D. Zou, et al. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the Annual Conference on Computer Security Applications (ACSAC)*. 2016.

[51] F. Long, P. Amidon, et al. Automatic inference of code transforms and search spaces for automatic patch generation systems. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*. 2017.

[52] P. D. Marinescu and C. Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*. 2013.

[53] M. A. McQueen, T. A. McQueen, et al. Empirical estimates and observations of 0day vulnerabilities. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS)*. 2009.

[54] N. Meng, M. Kim, et al. Systematic editing: generating program transformations from an example. *ACM SIGPLAN Notices*, 46(6):329–342, 2011.

[55] M. Monperrus. Automatic software repair: a bibliography. *University of Lille, Tech. Rep. hal-01206501*, 2015.

[56] A. Murgia, G. Concas, et al. A machine learning approach for text categorization of fixing-issue commits on cvs. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2010.

[57] A. Nappa, R. Johnson, et al. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 2015.

[58] F. Nielson, H. R. Nielson, et al. *Principles of program analysis*. Springer, 2015.

[59] A. Nistor, P.-C. Chang, et al. Caramel: detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 2015.

[60] M. Payer and T. R. Gross. Hot-patching a web server: A case study of asap code repair. In *Proceedings of the Annual Conference on Privacy, Security and Trust (PST)*. 2013.

[61] H. Perl, S. Dechand, et al. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015.

[62] S. Person, M. B. Dwyer, et al. Differential symbolic execution. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*. 2008.

[63] B. C. Pierce and D. N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2000.

[64] S. Raghavan, R. Rohana, et al. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSME)*. 2004.

[65] S. Rastkar and G. C. Murphy. Why did this code change? In *Proceedings of the International Conference on Software Engineering (ICSE)*. 2013.

[66] B. Ray and M. Kim. A case study of cross-system porting in forked projects. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*. 2012.

[67] E. Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.

[68] E. Rescorla. Security holes... who cares? In *Proceedings of the USENIX Security Symposium (SEC)*. 2003.

[69] E. A. Santos and A. Hindle. Judging a commit by its cover: correlating commit message entropy with build status on travis-ci. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. 2016.

[70] G. Schryen. Security of open source and closed source software: An empirical comparison of published vulnerabilities. In *Proceedings of the Americas Conference on Information Systems (AMCIS)*. 2009.

[71] S. Son, K. S. McKinley, et al. Fix me up: Repairing access-control bugs in web applications. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*. 2013.

[72] Y. Tao, Y. Dang, et al. How do software engineers understand code changes?: An exploratory study in industry. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*. 2012.

[73] Y. Tian and B. Ray. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*. 2017.

[74] D. Votipka, R. Stevens, et al. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 2018.

[75] P. Wadler. The essence of functional programming. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. 1992.

[76] R. Wu, H. Zhang, et al. Relink: Recovering links between bugs and changes. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*. 2011.

[77] F. Yamaguchi, N. Golde, et al. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. 2014.

[78] T. Zhang, M. Song, et al. Interactive code review for systematic changes. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 2015.

[79] Y. Zhou and A. Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE)*. 2017.

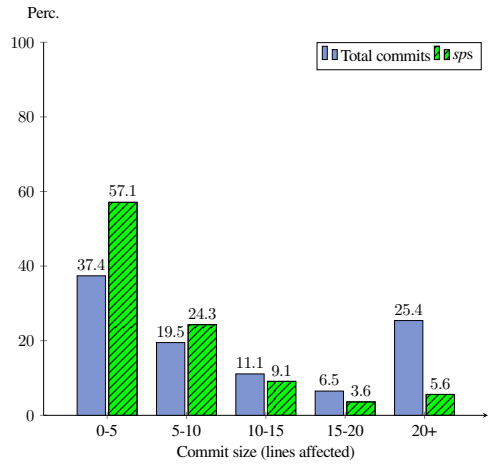


Fig. 6: The distribution of the size of all the commits studied and *sp* identified by SPIDER.

APPENDIX

A. *Shortcomings of the* sp *formalism*

According to our definition in Section II-B, a patch that removes all the functionality as shown in Listing 7 is an *sp*. This is because none of the inputs *execute* through the program or in other words all the inputs will end up in an error basic block. Equation 1 and 2 trivially hold as for all the inputs $(i \hookrightarrow f_p)$ evaluates to `false`.

We ignore all the updates to local variables as they are not part of the function output. Consider the patch as shown in Listing 9, which removes a seemingly useless `memset`. This is valid and commonly known as dead-store elimination [32]. However, on closer inspection, one can recognize that the `memset` may be required as it would potentially clean up some secret data to avoid information leaks. Our current definition of *sp* does not handle these cases.

```
int main(int argc, char **argv) {
+     if(argc > 0) {
+       return -1;
+     }
...
}
```

Listing 7: a patch that removes all the functionalities

1) Patch sizes: The plot in Figure 6 shows the distribution of the size of the studied commits as well as those that SPIDER identified as *sp*s. One can see that 42.9% of the patches identified as *sp*s (i.e., 28,873 of them) affect more than five lines of code (the affected lines are the sum of the lines added and deleted according to the *git diff* tool), and 5.6% of the patches affect more than 20 lines of code. We manually checked a few of these large patches and verified that these are in fact *sp*s. Most of these patches fix a simple issue across multiple functions, resulting in large amounts of affected lines.

2) False negatives: One can imagine that SPIDER could have false negatives i.e., it could classify certain patches as non-*sp*s where as they were in reality *sp*. This is not a major problem, as finding *all sp*s is *not* the goal of SPIDER. However, during our manual investigation, we found certain interesting *sp*s patches which violate our conditions as explained in Section III.

Listing 8 shows a patch which fixes a memory leak and ideally is an *sp* but does not satisfy the Equation 2. Here, among other things, the patch changes the third argument to the function `copy_from_user` to `PAGE_SIZE` (line 2) from `sccb->length` (line 10). Since we consider the arguments to a call to be the output of a function, the patch in Listing 8 changes the output of the function and thus will not be considered as a *sp*. However, a careful analysis of the patched function along with an understanding of the behavior of function `copy_from_user` would clarify that this patch doesn't affect the output (for all valid inputs) and hence is a *sp*. In fact, this patch is a security fix for CVE-2016-6130.

```
1  + copied = PAGE_SIZE -
2  +   copy_from_user
       (sccb, u64_to_uptr(ctl_sccb.sccb), PAGE_SIZE);
3  + if (offsetof(struct sccb_header, length) +
4  +     sizeof
       (sccb->length) > copied || sccb->length > copied) {
5  +   rc = -EFAULT;
6  +   goto out_free;
7  + }
8  - if (sccb->length > PAGE_SIZE || sccb->length < 8)
9  -   return -EINVAL;
10 - if (copy_from_user
       (sccb, u64_to_uptr(ctl_sccb.sccb), sccb->length)) {
11 -   rc = -EFAULT;
12 + if (sccb->length < 8) {
13 +   rc = -EINVAL;
```

Listing 8: Snippet from a *sp* patch (532c34b5fbf1687df63b3fcd5b2846312ac943c6 (fix potential information leak with /dev/sclp)) from mainline linux kernel (ID 1) that violate Equation 2.

This ratifies our intuition that, our definition of *sp* is sufficient but *not* necessary for a patch to be an *sp*.

```
int decrypt(..) {
     char secret_buff[4096];
...
...
-  memset(secret_buff, 0, sizeof(secret_buff));
}
```

Listing 9: an optimizing patch that may induce a security vulnerability.

```
      avpriv_report_missing_feature
          (s->avctx, "Lowres for weird subsampling");
      return AVERROR_PATCHWELCOME;
    }
+   if ((
    AV_RB32(s->upscale_h) || AV_RB32(s->upscale_v)) && s->
    progressive && s->avctx->pix_fmt == AV_PIX_FMT_GBRP) {
+   avpriv_report_missing_feature
    (s->avctx, "progressive for weird subsampling");
+   return AVERROR_PATCHWELCOME;
+   }

    if (s->ls) {
      memset(s->upscale_h, 0, sizeof(s->upscale_h));
      memset(s->upscale_v, 0, sizeof(s->upscale_v));
```

Listing 10: A non-CVE security patch (commit ee1e3ca5eb1) in FFmpeg (ID 21) that has triggering input in the commit message.

```
    parse_exthdrs(struct
          sk_buff *skb, const struct sadb_msg *hdr, void *
        ...
    uint16_t ext_type;
    int ext_len;

+   if (len < sizeof(*ehdr))
+      return -EINVAL;

    ext_len  = ehdr->sadb_ext_len;
```

Listing 11: A non-CVE security patch (commit 4e7REDACTED) in Main kernel (ID 1) that is missing in Qualcomm (ID 4) kernel.

	Commits	*sps* (% over commits)			
		NoEB	***NoPP***	***NoEB* ∪ *NoPP***	**Default**
Total across all projects	341,767	64,682 (18.93%)	63,463 (18.57%)	60,878 (17.81%)	67,408 (19.72%)

TABLE IV: The summary of the number and percentage of *sp*s detected by running SPIDER in various modes across all the projects listed in Table II. Detailed break down across each of the projects can be found in Table IV of our extended version [7].

B. *Control dependency versus control flow*

The concept of control dependency is different from the more commonly-used concept of control flow. Control flow captures *possible flows* of execution, while control dependency captures the *necessary conditions* that must hold for the execution to reach a particular statement. Consider the *PDG* of the patched function in Figure 2b. We can see a control dependency edge from the node (that corresponds to the instruction) at Line 3 to Line 15 with label F. This means that the condition at Line 3 must evaluate to `false` for the execution to reach Line 15. This is correct because if the condition at Line 3 evaluates to *true*, then the execution will immediately return from the function (Line 4). On the other hand, consider the control-flow graph of the patched function in Figure 1. There is no direct edge from $BB1$ (that contains Line 3) to $BB5$ (that contains Line 15). This is because the execution does not flow directly from $BB1$ to $BB5$ as there are other instructions in between (in $BB3$ and $BB4$).

C. *Reasons for low detection rate of* sps

There are certain projects where the percentage of detected *sp*s is low, such as IDs 15 and 16. After manual investigation of the subset of these patches, we found the following reasons:

Complex code: There are certain projects that mostly contains complex functions with data-dependencies inside nested loops. Specifically, the Python (ID 15) and PHP (ID 16) interpreters, and cURL (ID 24). Here, although the patches themselves are simple, the data dependencies increase the complexity of constraints, resulting in SPIDER failing to prove implication for the condition $C1$ (Section III-B3) resulting in a smaller *sp* detection rate.

Complex patches: In projects such as libpng (ID 30) and, OpenVPN (ID 28), the commits tend to be complex as they deal with media file formats and cryptographic protocols. Consequently, SPIDER fails to prove the equivalence for the condition $C2$.

D. *Patches fixing cryptographic issues*

Most of the CVEs in OpenSSL fix security issues related to cryptographic operations that affect the control flow in complex ways. A few OpenSSL CVEs fix cryptographic implementations against time side-channel attacks, which SPIDER is unable to reason about. For instance, the commit hash ae50d8270026edf5b3c7f8aaa0c6677462b33d97 [13] for CVE-2016-0703 of the OpenSSL repository fixes SSLv2 implementation against the Bleichenbacher [27] attack. We fail to identify this as an *sp* because the changes does not satisfy our definition of *sp* (refer Section II-B).

E. *Examples of* sps

The Listing 12 shows a patch identified as *sp*, in this case, the patch just adds an error basic block through a case statement, satisfying conditions $C1$ and $C2$, as the valid input space (matching the `case` statement) is restricted, and the function output does not change for the valid inputs. Listing 13 shows a commit in Redis identified as an *sp*, where there are no conditions affected (condition $C1$ holds) and changes are made only to a local variable thus not affecting the function output (condition $C2$ holds).

```
+ case MEM_AREA_SHM_VASPACE:
+ /* Find VA from PA in dynamic SHM is not yet supported */
+   va = NULL;
+   break;
  default:
    va = map_pa2va(find_map_by_type_and_pa(m, pa), pa);
  }
  ...
  return va;
}
```

Listing 12: An *sp* identified in OPTEE (commit 388302877d413) where the changes just add an error basic block.

```
int HelloRepl1_RedisCommand(RedisModu
..
- RedisModuleCallReply *reply;
..
- reply = RedisModule_Call(ctx,"INCR","c!","foo");
+ RedisModule_Call(ctx,"INCR","c!","foo");
- reply = RedisModule_Call(ctx,"INCR","c!","bar");
+ RedisModule_Call(ctx,"INCR","c!","bar");

RedisModule_ReplyWithLongLong(ctx,0);
return REDISMODULE_OK;
}
```

Listing 13: An *sp* identified in Redis (commit 6798736909b7) where the changes are only to local variables and do not affect the output of the function.

F. *Removing back edges in the* PDG

In this section, we argue that removing back-edges is safe when a patch does not directly modify a statement within a loop.

In principle, removing back-edges in the PDG unrolls [31] the corresponding loop once. The symbolic expression of the values computed inside the loop will be as if the loop is executed once.

If the output of the function does *not* depend on the number of iterations of a loop then unrolling the loop once or multiple times does not affect our output equivalence checking, and hence it is safe.

As explained in Section III-B4, we use symbolic expressions (Table I) to check the output equivalence of the functions. Now, consider the case where the output of the function depends on the number of iterations of a loop, and the symbolic expressions of the output are same in the original and the patched function. This means that the number of iterations of the loop will be the same in the original and patched function, and consequently, the output should be the same.

Hence, our approach of removing the back-edges and using symbolic expressions for output equivalence checking is safe when a patch does *not directly* modify a statement within a loop. However, if the patch directly modifies a statement within a loop, the removal of the back-edges prevents the back-propagation of this information resulting in computation of potentially wrong symbolic output-constraint pairs, thus is not safe.

G. Summarizing library functions

As mentioned in Section IV-E, we maintain a list of well-known library functions which could be easily summarized. The categories of the functions are:

- Print and logging functions (e.g., `printf` (without the `%n` format specifier), `printk`), we ignore its affects as they are used for logging.
- Memory initialization and release functions (e.g., `kmemset`, `memset`, `kfree`, `free`), are considered as writes to the corresponding variables.
- Kernel synchronization function calls (e.g., `spin_lock`, `spin_unlock`, `mutex_lock`, `mutex_unlock`). Similar to the logging functions, these do not affect the output. However, improperly used synchronization functions could cause deadlocks. To avoid this, we need to check that any `*_lock` or `*_unlock` function should have corresponding `*_unlock` or `*_lock` respectively. To do this, for any inserted `*_lock` or `*_unlock` function in a basic block BB, we check that there exists corresponding `*_unlock` or `*_lock` function in one of the post-dominator or pre-dominator basic blocks of BB respectively.
- C security-sensitive function calls (e.g., `strcpy`, `strncpy`, `strlcpy`, `memcpy`, `sprintf`, `sscanf` and their variants). We model these are assignments. For instance, the call`strcpy(dst, src)` will be treated as the assignment `dst=src`.

H. Handling multiple definitions

In principle, there are two basic cases when multiple definitions of a variable can reach a statement. We show in this section that both of the cases are handled by our Equation 5.

The first case is shown below. In this case, the definitions at both line 1 or 3 can reach can the statement at line 5:

```
1  v = d1;
2  if(c) {
3    v = d2;
4  }
5  y = v + x;
```

All the executions that reach line 3 should have executed line 1. In other words, line 3 is guarded by a stricter condition. Furthermore, if the constraint represented by the condition `c` is satisfied, then the value defined at line 3 (i.e., `d2`) will reach line 5. This is captured by the first case of the Equation 5.

Consider the second case, where the definitions at line 2 or 3 can reach line 6:

```
1  if(c) {
2    v = d1;
3  } else {
4    v = d2;
5  }
6  y = v + x;
```

In this case, both statements are mutually exclusive, and, depending on whether the constraint represented by the condition *c* is `true` or `false`, the definitions at line 2 or line 4 reach line 6. This is captured by the second case of the Equation 5.

I. Impact of our assumptions

Although, SPIDER tries to be sound in detecting *sp*s, it has a few assumptions (Section V) and uses heuristics (Section IV-D) which may not be desirable for certain users. In this section, we evaluate the effectiveness of our technique when each of these assumptions or heuristics is disabled. Specifically, we run SPIDER in following modes:

- *NoEB*: As explained in Section IV-D, we ignore the affected statements that are part of a error-handling basic block (BB_{err}). The BB_{err}s are detected using certain heuristics that may not hold for certain projects. Incorrect identification of BB_{err}s might cause certain non-safe patches to be wrongly classified as *sp*s. In *NoEB* mode, we *do not* ignore any affected statements and analyze all the statements even if they belong to an BB_{err}.
- *NoPP*: We use the tool `unifdef` to handle preprocessor directives. However, as explained in Section V, this could cause certain code to ignored if it is part of a `#if-then-else` construct. In *NoPP* mode, if a patch affects a function which has *any* code controlled by a preprocessor directive it will *not* be considered as a *sp*.

Table IV shows the overall effectiveness of SPIDER across all the projects with each of these modes turned on. The detection rate did not vary much across the projects. The detailed breakdown for each project is provided in Table IV of our extended version [7]. The column *NoEB* ∪ *NoPP* shows the results when both the modes are enabled. We also show the effectiveness when all of these modes are turned off, i.e., the Default mode. The detection rate does not vary much across all the modes, which shows that the effectiveness of the techniques used by SPIDER does not largely depend on the assumptions and heuristics.

J. Anonymous survey on the requirement of SPIDER

To get a feeling for the utility of a tool like SPIDER, we performed an anonymous survey of maintainers and developers of various open-source software projects, including Ubuntu, OpenSUSE, Linaro, OpenBSD, and VLC. 82% of those who completed the survey (32 out of 39 participants) agreed that such a tool would be very useful, and they were prepared to use it for their projects. Interestingly, only OpenBSD developers (the remaining 7 participants) expressed concerns, as such a system might also propagate bug-inducing patches like Apple's goto fail [3], but they agree that it could still help expert developers to prioritize their efforts. This anonymous survey is exempted from IRB approval [9], as there is no collection or use of user private information.

K. Impact of not identifying all error-handling basic blocks

It is important to observe that the risky mistakes are those where we (incorrectly) identify non-error basic blocks as error blocks. In such cases, we could falsely identify a patch as an *sp*. It is much less problematic or safer to misidentify an error-handling block as a non-error-handling block. The reason is that such a mistake might cause our system to analyze more statements than necessary which could result in discarding a safe patch as unsafe, but it does not introduce unsafe patches as safe. Thus, we consider our approach safely conservative.