Precisely Characterizing Security Impact in a Flood of Patches via Symbolic Rule Comparison

Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu
University of Minnesota, Twin Cities
{wu000273, he000242}@umn.edu, mccamant@cs.umn.edu, kjlu@umn.edu

Abstract—A bug is a vulnerability if it has security impacts when triggered. Determining the security impacts of a bug is important to both defenders and attackers. Maintainers of large software systems are bombarded with numerous bug reports and proposed patches, with missing or unreliable information about their impact. Determining which few bugs are vulnerabilities is difficult, and bugs that a maintainer believes do not have security impact will be de-prioritized or even ignored. On the other hand, a public report of a bug with a security impact is a powerful first step towards exploitation. Adversaries may exploit such bugs to launch devastating attacks if defenders do not fix them promptly. Common practice is for maintainers to assess the security impacts of bugs manually, but the scaling and reliability challenges of manual analysis lead to missed vulnerabilities.

We propose an automated approach, S\textsuperscript{ID}, to determine the security impacts for a bug given its patch, so that maintainers can effectively prioritize applying the patch to the affected programs. The insight behind S\textsuperscript{ID} is that both the effect of a patch (either submitted or applied) and security-rule violations (e.g., out-of-bound access) can be modeled as constraints that can be automatically solved. S\textsuperscript{ID} incorporates rule comparison, using under-constrained symbolic execution of a patch to determine the security impacts of an un-applied patch. S\textsuperscript{ID} can further automatically classify vulnerabilities based on their security impacts. We have implemented S\textsuperscript{ID} and applied it to bug patches of the Linux kernel and matching CVE-assigned vulnerabilities to evaluate its precision and recall. We optimized S\textsuperscript{ID} to reduce false positives, and our evaluation shows that, from 54K recent valid commit patches, S\textsuperscript{ID} detected 227 security bugs with at least 243 security impacts at a 97% precision rate. Critically, 197 of them were not reported as vulnerabilities before, leading to delayed or ignored patching in derivative programs. Even worse, 21 of them are still unpatched in the latest Android kernel. Once exploited, they can cause critical security impacts on Android devices. The evaluation results confirm that S\textsuperscript{ID}'s approach is effective and precise in automatically determining security impacts for a massive stream of bug patches.

I. INTRODUCTION

Major system programs receive an overwhelming number of bug reports, and dealing with these bug reports is much of the life-cycle cost of the software. For instance, Mozilla developers dealt with almost 300 bugs per day in 2005 [2], and a similar rate of new bugs are received in the Mozilla bug database [30] today. The Linux kernel also experiences this problem. As of August 2019, more than 855K patches have been applied by kernel maintainers [49], and the actual number of submissions examined is even higher because many proposed patches are not applied, or require several rounds of revision. Linux also receives many proposed patches from external contributors. Sometimes, these patches fix important bugs, while other patches fix general bugs or even insignificant bugs. Therefore, maintainers must manually review and prioritize the submitted patches to decide if they should be applied immediately or not. Large-scale commercial software development faces similar challenges with bug reports from internal testers and changes proposed by less-experienced developers. This work is time-consuming and error-prone. For example, Hooimeijer et al. [25] showed that 70% of the total life-cycle cost of software is consumed by maintenance, such as modifying existing code and dealing with bugs.

Given their limited resources, maintainers have to prioritize which bugs to fix by assigning bugs to different priority levels. Highest-priority bugs, such as an obvious security vulnerability, must be fixed immediately. However, lower-priority bugs may be fixed slowly, remain unpatched for a long period of time, or fall through the cracks completely. The common practice is for maintainers to assess the security impacts of bugs manually [39], which is not only challenging and expensive, but also error-prone. This manual classification requires considerable human effort and requires code maintainers to have wide security domain knowledge.

If the security impacts of a critical bug cannot be correctly identified, it will be treated as a lower-priority bug, which will lead to serious security problems. For instance, Arnold et al. [3] described a high-impact compromise of servers for Debian Linux made possible by a Linux kernel vulnerability for which a patch had been available for eight weeks. The Debian administrators had not updated the kernels because the security implications of the patch were not clear until after it was used in a successful exploit. Arnold et al. called this kind of bug a hidden-impact vulnerability: one that is not identified as a vulnerability until after it is made public and potentially exploited by attackers. Their work shows that 32% of vulnerabilities in the Linux kernel were hidden impact vulnerabilities before they were publicized.

Lack of reliable information about the security impacts of bugs is even more critical when open-source software is used in other projects. For example, the Linux kernel is widely used and customized by a large number of platforms such as the Internet of Things (IoT) devices and mobile devices (most prominently Android). A 2018 survey reported by Hall [24] shows that more than 70% of IoT developers use Linux, and
Android had a 75% share of the worldwide mobile OS market as of April 2019 [14]. Given the fragmentation of versions and uses of the Linux kernel, it would be impossible for every patch to be promptly applied to every Linux-based device. Instead, patches must be prioritized based on their severity. For instance, under the Android Security Rewards Program [22], reporters are typically required to demonstrate the reproducibility and impact of the reported bugs; otherwise, the reported patches will likely be declined. Previously, we reported three new NULL-pointer dereference bugs to the Android Security team without mentioning their security impacts or providing a proof-of-concept exploit. We considered these to be vulnerabilities because they can cause DoS, but the Android Security team declined the patches because we did not prove the security impact of the bugs. The empirical results we report also show that bugs that cannot be determined to have security impacts may not be fixed promptly, and thus may introduce serious security problems.

Given the significance of security bugs, many recent papers [5, 19, 23, 50, 53, 54, 65] have attempted to distinguish security bugs from general bugs automatically. Most of these works focus on analyzing textual information, such as a bug description, and their classification of security impacts is mainly based on text-mining techniques. A fundamental limitation is that such classification is highly dependent on the quality of the textual information, which in turn depends on the experience and security-related knowledge of the reporters. Unfortunately, our results indicate that, in many cases, the reporters themselves are not aware of the potential security impacts of the bugs they report. We found that 60.8% of vulnerability patches for the Linux kernel do not mention security impacts in the patch description or subjects. Thus, we cannot expect any classification based on this textual information to reliably classify security bugs. This observation is consistent with recent results by Chaparro et al. [8] which show that many textual bug reports are missing important details and the measurements of Tian et al. [47] which suggest that bug severity ratings are unreliable (i.e., they differ even for duplicate reports of the same bug). To identify security-related patches precisely, we need a more reliable approach to determine the security impacts of bug patches based on code instead of prose.

Existing automated tools also do not provide sufficient support to analyze the security impacts of bugs. Static-analysis tools can warn about code constructs that may have security impacts when misused. However, they generally do not analyze all the factors that affect security impacts. Instead, they make conservative assumptions and thus produce a significant number of false-positive reports that must be filtered out in another step. Providing a proof-of-concept exploit (“PoC”) is strong evidence of security impacts, but it requires every patch to include an exploit, which would be a major burden on bug reporters. Bug-finding tools based on fuzzing [44, 49] or whole-program symbolic execution [42, 50] often create a PoC when detecting a problem, but such tools currently generate only a minority of kernel-bug reports, because of challenges such as state explosion and modeling hardware devices. We would like the process of fixing a vulnerability to be faster than the process of exploiting it if defenders are to stay ahead of attackers. Thus, we need an approach to assess the security impact of a bug that is easier than generating a PoC. For adoption, such a tool must have a low false-positive rate and the ability to relate the results to the specific security impact that is implicated. An analysis tool must be trustworthy to convince developers to take a second look at a patch they would otherwise pass by.

In this paper, we propose an automated system, S1D, to determine the security impacts of bugs, given their patches. Using security rules that capture common security impacts, S1D distinguishes unsafe (rule-violating) and safe (rule-compliant) behaviors of patched and unpatched code, which allows S1D to characterize the security impacts of a bug. S1D employs differential, under-constrained symbolic execution to match a security risk that is fixed by a patch. The intuition is that both security rules and the program behaviors that are feasible in the unpatched and patched versions can be captured precisely with symbolic constraints. By comparing security constraints with code, S1D can reliably determine: (1) if the unpatched code must violate a security rule—the unpatched code has a security problem, and (2) whether the patched code can never violate the same security rule—the patched code has eliminated the security problem present in the unpatched code. If both conditions are evaluated to be true, this is a strong confirmation that the patch will fix a security violation, and thus that the bug will have a security impact. More importantly, the conservativeness of under-constrained symbolic execution ensures the reliability and the scalability of S1D’s determination of security impacts. We use slicing and under-constrained symbolic execution to precisely analyze just the code region that is directly relevant to a patch, making conservative assumptions about interactions with other kernel states. This approach avoids most false positives but without expanding the analysis to the whole kernel or requiring an effort that is equivalent to fuzzing or exploit generation. S1D determines security impacts based on the code semantics instead of textual information. Based on the semantics, S1D can detect the security bugs reliably and provide details about how a bug can be exploited to cause security impacts. This supports developers in formulating an appropriate response to a security bug.

Our priority is for S1D’s reports of security impact to be reliable, i.e., with high precision and few false positives. To achieve this, we are willing to accept false-negative cases where there is a security impact that the current implementation of S1D is unable to recognize. Some causes include unusual types of security impacts that are not captured by S1D’s current security rules and the conservative strategy of under-constrained symbolic execution which may miss some cases. An empirical analysis of S1D’s false-negative results for known vulnerabilities appears in [8]. Further development to reduce false negatives would expand the benefits of S1D, but since the current state of practice does not use automated tools to analyze impact at all, we believe that the best path to adoption and security benefit is to begin with tools whose results developers can easily trust when they signal a security bug.

We have implemented S1D based on LLVM as multiple static analysis passes. One is a data-flow analysis pass, which identifies vulnerable operations and security operations; the other is an under-constrained symbolic execution pass, which precisely reasons about security impacts. We choose the Linux kernel as our experimental target because it is one of the most widely used and actively-maintained open-source system programs. The security of the Linux kernel is also important to many IoT and mobile devices. For evaluation, we selected 66K
recent git commits from the Linux kernel. From these commits, we identified 54K valid commit patches and finally compiled and analyzed 110K LLVM IR files in total. By analyzing these files, S1D successfully found 227 security bugs with a 97% precision rate. These security bugs may introduce security impacts such as out-of-bound access, use-after-free, double-free, uninitialized use, and permission bypass. More critically, we found that 21 of these security bugs are still not patched in Android, which can cause severe security problems for billions of Android devices.

To further confirm that the identified security bugs are vulnerabilities, we analyzed the reachability of the security bugs from attacker-controllable entry points (e.g., system calls) and also reported them to CVE maintainers. As a result, we find that 67.8% of identified security bugs are potentially reachable from entry points. On the other hand, we in total reported 154 security bugs to CVE maintainers and have received 37 responses with 24 new CVEs assigned. The evaluation results show that S1D is effective and precise in automatically determining the security impacts of a large number of bug patches.

We make the following contributions in this paper.

• **A study of security bugs and patches.** The boundary between bugs and vulnerabilities can be unclear. We first study the differences between bugs and vulnerabilities. We then model patches for common security bugs, including missing/wrong bound check, missing pointer nullification, missing initialization, and missing permission check. The modeling enables us to define the security impacts of bugs and thus confirm security bugs.

• **Symbolic rule comparison for determining security impacts.** We propose S1D to determine the security impacts of bugs automatically. The core of S1D is symbolic rule comparison which employs differential and under-constrained symbolic execution to precisely confirm the security impacts that a patch fixes. In addition, S1D also provides details about the security impacts to facilitate bug fixing.

• **Finding of security bugs and unpatched vulnerabilities.** With S1D, we found 227 security bugs in the Linux kernel; 21 of them still remain unpatched in Android, which can be exploited to attack billions of Android devices. Also, 24 new CVEs have been assigned to the identified security bugs. Further, we evaluated the reachability of all the identified security bugs, and found that 67.8% of them are potentially reachable from attacker-controllable entry points.

The rest of this paper is organized as follows. We review background concepts in §II and give an overview of our approach in §III. We then present the design of S1D in section §IV; the implementation of S1D in section §V; the evaluation of S1D in section §VI; limitations and future work in section §VII; related work in §VIII; and the conclusion in §IX.

### II. BACKGROUND

To propose an effective approach to understand the causes and security impacts of bugs and thus to find security bugs, we analyzed some existing patches for vulnerabilities in the Linux kernel. Specifically, we first show differences between general bugs and vulnerabilities. Then, we analyze the common causes and security impacts of vulnerabilities. Based on the statistical results, we summarize the model and components of the vulnerability patches. After that, we define the problem scope and the assumptions of this work.

#### A. General Bugs, Security Bugs, and Vulnerabilities

A bug is a vulnerability if it causes security impacts when triggered. A vulnerability is also called a security-critical bug (or just a security bug), and is distinguished from a general bug. Different kinds of vulnerabilities often differ in security impacts. The example in Figure 1 shows the difference between a general bug and a vulnerability. In this example, missing the check in line 3 is a vulnerability because it leads to an out-of-bound access in line 9. In comparison, missing the check in line 6 will not introduce any security impact; thus, it is just a general bug. More details about the definition and detection for security checks can be found in previous work [51].

![Figure 1](https://example.com/figure1.png)

**Fig. 1:** Differences between a vulnerability and a general bug. Missing the check in line 3 results in a vulnerability while missing the check in line 6 is a general semantic bug.

#### B. Common Security Bugs and Impacts

<table>
<thead>
<tr>
<th>Common security bugs (root cause)</th>
<th>Percent of bugs</th>
<th>Main security impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Missing/wrong bound check</td>
<td>21%</td>
<td>Out-of-bound access</td>
</tr>
<tr>
<td>Missing initialization</td>
<td>9%</td>
<td>Uninitialized use</td>
</tr>
<tr>
<td>Missing permission check</td>
<td>9%</td>
<td>Permission bypass</td>
</tr>
<tr>
<td>Missing NULL check</td>
<td>7%</td>
<td>NULL-pointer dereference</td>
</tr>
<tr>
<td>Missing/wrong locks/unlocks</td>
<td>6%</td>
<td>Use-after-free, double-free, Permission bypass, NULL-pointer dereference</td>
</tr>
<tr>
<td>API misuse</td>
<td>5%</td>
<td>Out-of-bound access, Permission bypass, Uninitialized use</td>
</tr>
<tr>
<td>Missing error-code check</td>
<td>5%</td>
<td>Out-of-bound access, Permission bypass, NULL-Pointer dereference</td>
</tr>
<tr>
<td>Missing pointer nullification</td>
<td>4%</td>
<td>Use-after-free, double-free</td>
</tr>
<tr>
<td>Others such as numerical errors</td>
<td>34%</td>
<td>Uninitialized use, Out-of-bound access, NULL-pointer dereference, Others</td>
</tr>
</tbody>
</table>

**TABLE I:** Common security bugs and security impacts.

In this work, we aim to cover the most common security bugs and their corresponding security impacts. To this end, we first examined recent Linux-kernel vulnerabilities included in the national vulnerability database (NVD). There are nearly 800 vulnerabilities reported in the past three years, but only a small part of them include valid git-commit information of their patches. Thus, we chose to analyze 100 of them across these years.
We then create a patch model that incorporates the components, (The symbol, \(+\)) violating operations (e.g., out-of-bound access and use-after-free). Vulnerable operations are the ones whose (invalid) values or status can lead to security impacts. As such, critical variables are typically targeted by security operations. For example, a checked bound variable is a critical variable.

- **Vulnerable operations** signal the risk of a security bug, often because they can behave unsafely. Based on our study, common vulnerable operations include buffer and array operations, read or write operations, pointer operations, operations involving critical data structures such as inodes or files, and resource-release operations.

### Table I

<table>
<thead>
<tr>
<th>ID</th>
<th>Vulnerable_op(CV, ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>+ Security_op(CV, ...)</td>
</tr>
</tbody>
</table>

#### Table II: The common patch model and the three key components: security operation, critical variable (CV), and vulnerable operation. The security operation is typically added or updated by a patch.

This model shows that patch updates or adds new security operations in the vulnerable code to eliminate the security impacts which are introduced by the vulnerable operations. Most commonly, a security operation is inserted before a vulnerable operation to prevent an unsafe state. Evaluation results in [VI-I] show that about 88% of vulnerabilities in the Linux kernel can precisely or partially fit into this model. Thus, we can use this model to determine security impacts for most patches.

#### D. Problem Scope and Assumptions

In this work, we analyze the bug patches in the Linux kernel. We choose the Linux kernel as the target program for the following reasons. (1) The Linux kernel is a foundational and widely used program. Many other operating systems are based on the Linux kernel, such as Android. Security bugs in the Linux kernel may introduce critical security impacts in all Linux-based systems. Thus, applying patches for security bugs in the Linux kernel is vital. (2) The Linux kernel is an open-source program with a well-maintained patch history, which facilitates patch-based analyses. These reasons motivate us to choose the Linux kernel as the experiment target. However, SiD is general and applicable to other similar software such as FreeBSD and Firefox, as discussed in [VII].

We assume that the provided patches correctly fix actual bugs. We may not correctly obtain the security impacts of vulnerabilities if the patches are incorrect. Based on the statistical results in [Table I], we choose to determine all the common security impacts listed in [Table III]. The current version of SiD does not include NULL-pointer dereference because it is difficult to exploit in the Linux kernel—the zero page is protected against being allocated. However, NULL-pointer dereference can be naturally supported by modeling the “non-NULL” as a constraint and the NULL dereference as a vulnerable operation. [Table XII] shows how to use our model to cover the patches for other common types of vulnerabilities. Also, more details about extending SiD to detect more types of bugs are discussed in [VII].

To determine common security operations related to these impacts, we selected 100 recent vulnerabilities for each security impact in the Linux kernel from NVD. By manually checking the patches of these vulnerabilities, we count the security operations as shown in [Table III]. Based on this result, we choose to cover the most common security operations for each security impact including: (1) bound checks, (2) initialization operations, (3) permission checks, and (4) pointer nullification. In our current implementation, we do not include other security operations such as lock or unlock operations because either they are not common or they predominantly cause non-security impacts such as incorrect results. In total, based on [Table III] by calculating the proportion of these covered security operations against these common security impacts, the current...
TABLE III: Common security operations for fixing common security impacts.

<table>
<thead>
<tr>
<th>Common security impacts (%)</th>
<th>Common security operations (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permission bypass (21.9%)</td>
<td>Permission check (59%)</td>
</tr>
<tr>
<td></td>
<td>Changing permission flags (8%)</td>
</tr>
<tr>
<td></td>
<td>Others (33%)</td>
</tr>
<tr>
<td>Out-of-bound access (16.5%)</td>
<td>Bound check (79%)</td>
</tr>
<tr>
<td></td>
<td>Reset the size of buffer (10%)</td>
</tr>
<tr>
<td></td>
<td>Others (11%)</td>
</tr>
<tr>
<td>Uninitialized use (13.7%)</td>
<td>Initialize the variable (78%)</td>
</tr>
<tr>
<td></td>
<td>Others (22%)</td>
</tr>
<tr>
<td>Use-after-free/double-free (4.3%)</td>
<td>Pointer nullification (32%)</td>
</tr>
<tr>
<td></td>
<td>Lock or unlock operations (25%)</td>
</tr>
<tr>
<td></td>
<td>Others (43%)</td>
</tr>
</tbody>
</table>

implementation of S1D can support about 38% of vulnerabilities. However, S1D’s approach is generic, and covering more types of vulnerabilities requires only extra engineering efforts for modeling and identifying the three patch components. Table XII shows how to support several more types of bugs which can cover 13% more of vulnerabilities. More discussion can be found in §VII.

We further assume that the bug fixed by a patch is triggerable, which means that, by providing specific inputs, the execution can reach the buggy code. Existing techniques, such as guided fuzzing [55, 61] and symbolic execution over untrusted inputs [42, 60], can search for inputs that trigger a bug. However, determining how to trigger a bug can be challenging; if a bug has security impacts, it is usually worthwhile to fix it, even if it is not obvious to be triggerable.

1) Security Rules: Security impacts occur when security rules are violated. To precisely determine if a security impact exists, we also need to define the corresponding security rule. The specific security rules help S1D construct constraints that can be solved. With security rules, the determination of security impacts can be transformed into a constraint-solving problem. For the security impacts shown in Table III, we define the corresponding security rules as follows, which are consistent with the standard definition in CWE [12].

- **Out-of-bound access.** Memory read and write operations should be within the boundary of the current object.
- **Use-after-free and double-free.** An object pointer should not be used after the object has been freed.
- **Uninitialized use.** A variable should not be used until it has been initialized.
- **Permission bypass.** Permissions should be checked before performing sensitive operations such as I/O.

III. OVERVIEW OF S1D

Given a patch and the target program, S1D automatically determines if the patch fixes some security impacts. In this section, we show the approach and workflow of S1D.

A. The Approach of S1D

We now use an example, shown in Figure 3, to illustrate the symbolic rule comparison approach of S1D. This is an out-of-bound access security bug that is fixed by a patch that inserts a bound check in line 6. Given this patch, the goal of symbolic rule comparison is to confirm whether the patch fixes violations of some security rules, e.g., out-of-bound access, that introduce security impacts.

Symbolically analyzing patched code. First, S1D analyzes the patched version to prove that it will never violate a security rule. In function iwl_sta_ucode_activate, S1D identifies the security operation in the patch, bound check in line 6. Then, S1D extracts the critical variable, sta_id, from the security operation. By using data-flow analysis, S1D identifies six potential vulnerable operations located in lines 11, 14, 16, 19, 21, and 23. Each pair of security operation and vulnerable operation defines a slice, and S1D performs under-constrained symbolic execution against each slice. S1D will construct and collect three sets of constraints. The first set of constraints are constructed from the security operation of the patch, e.g., sta_id < IWLAGN_STATION_COUNT, in the example. The second set of constraints is collected from the slice through symbolic execution (e.g., capturing if a variable is modified via arithmetic). After that, S1D artificially constructs the third set of constraints that represent the violation of a security rule, e.g., memory access out of the bound of stations. All the three constraint sets are then merged as the final constraint set. Finally, S1D employs a constraint solver to prove that the final constraint set is unsolvable, which means that the slice will not violate the security rule.

Symbolically analyzing unpatched code. Second, if the patched version never violates a security rule, S1D analyzes the unpatched version to prove that every behavior removed by the patch would violate the security rule. A slice in the patched code is matched with one in the unpatched code based on its critical variables and vulnerable operation, as described in more detail in §V-D. Similar to the analysis for the patched code, S1D constructs and collects three sets of constraints for the unpatched code, however, in a different way. Specifically, the first set corresponds to the security operation in the patch. Since the security operation is not in the unpatched code, S1D will add an artificial constraint that is the opposite of the constraint for the security operation; it is important to note that this captures the behaviors that are blocked by the security operation in the patched code. For example, while the security operation is to ensure sta_id < IWLAGN_STATION_COUNT, S1D will instead add a constraint, sta_id >= IWLAGN_STATION_COUNT. The second constraint set is similarly collected from the slice through symbolic execution. The last constraint set corresponds to the security rule. However, the constraints are constructed in such a way that they satisfy rather than violate the security rule, e.g., in-bound access of stations. Finally, the three constraint sets are merged and proved to be unsolvable, i.e., the unpatched code will violate the security rule in all the cases blocked by the patch. The first and the third constraint sets of the unpatched version are constructed in a opposite way as in the patched version, which allows S1D to leverage the conservative unsolvability of under-constrained symbolic execution to precisely determine the security impacts.

Confirming security impact. At last, if both cases are indeed unsolvable, S1D confirms that the patch is a fix for a security-rule violation. In other words, the corresponding bug is a security bug. Symbolic execution allows S1D to precisely check the match between a patch and the corresponding security
/* Linux: drivers/net/wireless/intel/iwlwifi/dvm/sta.c */

int iwl_sta_ucode_activate(..., u8 sta_id)
{
    if (sta_id >= IWLAGN_STATION_COUNT)
        return -EINVAL;

    if ((priv->stations[sta_id].used ))
        IWL_ERR(priv, "invalid sta_id %u", sta_id);
    return -EINVAL;

    if (priv->stations[sta_id].used)
        IWL_DEBUG_ASSOC(priv, "STA id %u addr %pM already present in uCode"
        sta_id, priv->stations[sta_id].sta.addr);

    if (priv->stations[sta_id].used)
        IWL_DEBUG_ASSOC(priv, "Added STA id %u addr %pM"
        sta_id, priv->stations[sta_id].sta.addr);

    return 0;
}

Fig. 2: Overview of SID. $C_{SO}$ = Constraints from security operations, $C_{SR}$ = Constraints from security rules, $C_P$ = Constraints from paths.

Fig. 3: A missing bound-check bug and its patch (lines 6-9).

rule, while the under-constrained approach makes the analysis
conservative about parts of the code outside the analysis scope.

B. The Workflow of SID

Figure 2 is an overview of SID including the following three
main phases: pre-processing, dissecting patches, and symbolic
rule comparison.

Pre-processing. The pre-processing phase includes two tasks.
First, given a specific git commit, it ensures that it is a bug-
fixing patch. Commits for branch merging, documentation,
and code formatting are eliminated. Second, it identifies the
dependent files of the patch, which are collected using standard
control-dependency analysis against the patched code and taint
analysis against the variables involved in the patch, in both a
forward and backward manner. We found that the dependent
files are a single file containing the patch in most cases. It then
prepares the patched code and unpatched code (by reverting
the patch) and invokes LLVM to compile them into LLVM IR files.

Dissecting patches. In this phase, SID dissects the patch to
identify the key components according to the patch model
(SII-C). SID first identifies the security operations in both the
patched and unpatched versions. SID then extracts involved
critical variables from the identified security operations. Next,
SID applies data-flow analysis against critical variables to
collect vulnerable operations. The data-flow analysis also
helps SID collect slices from security operations to vulnerable
operations for the critical variables. It is worth noting that
security operations and vulnerable operations have a many-
to-many mapping, which means that a single patch may have
multiple slices. After obtaining these slices, SID further employs
symbolic execution to test the feasibility of the slices by
testing if the path conditions of the slices are satisfiable. If
the conditions are unsatisfiable, the path is infeasible, and is
immediately discarded.

Symbolic rule comparison. For the feasible slices, SID then
performs a symbolic rule comparison to confirm the security
impacts of the bug with the approach described in SII-A

IV. DESIGN OF SID

In this section, we present the design of SID. In particular,
we focus on the static analysis for dissecting patches and
symbolic rule comparison for determining security impacts.

A. Static Analysis for Dissecting Patches

Given a patch and the target program, SID employs static
analysis to (1) identify security operations, critical variables,
and vulnerable operations and (2) construct slices from security
operations to vulnerable operations for the critical variables.
According the the patch model (SII-C), we summarize the patch
patterns for each class of vulnerability in Table IV

- **Out-of-bound access**
  - 1. Security_ck(Bound);
  - 1. ret = Perm_func(CV, ...);
  - 1. Security_check(ret);
  - 2. Vulnerable_op(Bound, ...);
  - 3. Vulnerable_op(CV, ...);
- **Use-after-free or double-free**
  - 1. free(Pointer);
  - 1. Initialize(CV);
  - 2. Pointer = NULL;
  - 3. Vulnerable_op(Pointer, ...);
  - 2. Vulnerable_op(CV, ...);

TABLE IV: Patch patterns for different classes of vulnerabilities and
the key components in the patches. + denotes the security operation in
the patches, and Vulnerable_op represents some vulnerable operations.

Identifying security operations. First, SID analyzes the
patches to identify security operations. As described in SII-D

6
SID identifies four kinds of security operations: permission check, bound check, initialization operation, and pointer nullification. Based on the statistical results in Table III, we summarize common patterns of patches in Table IV. These patterns describe how security operations fix the corresponding security impacts caused by the vulnerable operations. For out-of-bound access vulnerabilities, the patches typically insert the security operation, bound check, to make sure in-bound access against a memory object. For permission bypass vulnerabilities, the patches insert permission check as the security operation. The permission checks are usually done by checking the return value of permission functions. For use-after-free and double-free vulnerabilities, the patches often insert pointer nullification as the security operation. Since, in the Linux kernel, NULL check is typically enforced before using a pointer, nullification becomes a common way for fixing use-after-free. For uninitialized-use vulnerabilities, the patches instead initialize a memory object before it is being used. §V will present further details on analyzing the code to identify the security operations.

Extracting critical variables. Next, SID extracts the targets of the identified security operations as critical variables. For nullification, initialization, and bound-check cases, critical variables can be easily identified by extracting the involved variables (not constants). However, the critical variables in permission-check cases can be challenging to identify. The method SID uses to identify such critical variables is based on permission functions (e.g., ns_capable()) which include both critical variable and capability numbers as parameters. Non-constant parameters (e.g., objects such as files, inodes, or subjects such as users) used in such permission functions are typically sensitive resources whose accesses require permission checks. Therefore, SID identifies these parameter variables as critical variables.

Slicing to find vulnerable operations. After extracting the critical variables, SID then uses data-flow analysis to find vulnerable operations using extracted critical variables, i.e., slicing to find vulnerable operations. An operation is regarded as a vulnerable operation if it may introduce security impacts via the critical variables. We do the backward or forward data-flow analysis against the critical variables to match the vulnerable operations, according to the corresponding patterns shown in Table IV. With that, we also obtain slices for critical variables that involve both security operations and vulnerable operations. Because vulnerability operations and security operations are many-to-many mappings, one vulnerable operation or security operation can be in multiple slices. More details about collecting vulnerable operations for different kinds of vulnerabilities can be found in §V-C.

Pruning slices. After extracting these slices, SID removes slices with the following cases. (1) The critical variables are newly introduced in the patch. In this case, the unpatched code will never use them. (2) The vulnerable operations exist only in the patched version but not in the unpatched version, which means that the vulnerable operations are also newly introduced by the patch, and corresponding security impacts will not exist in the unpatched version.

Removing infeasible slices. Finally, SID removes infeasible slices using symbolic execution. SID performs the under-constrained symbolic execution for each slice to collect path constraints. When reaching the end of the slice, SID tries to solve the constraints. If these constraints are unsolvable, SID will discard this slice. Removing unsolvable slices will reduce false positives and make sure that the unsolvability in symbolic rule comparison must be related to security-rule violations (not vacuous), ensuring the effectiveness of SID.

B. Symbolic Rule Comparison

SID further performs a symbolic rule comparison to determine security impacts and identify security bugs. SID determines that a patch is for a security bug if the patched and unpatched versions satisfy both of the following requirements.

- The patched version never violates a security rule.
- The unpatched version always violates the security rule in the situations excluded by the patch.

The checking against the absolute requirements is possible because SID uses under-constrained symbolic execution, which is conservative [41]. By combining both requirements, it is intuitive to determine that the patch prevents violation of a security rule. To realize the checking against the requirements, SID first constructs and collects the constraint sets from patches, security rules, and the slice paths for the patched and unpatched code separately. If both constraint sets for the patched and unpatched code are unsolvable, SID determines the security impact.

1) Constructing and Collecting Constraints: SID constructs and collects three sets of constraints for both the patched version and the unpatched version. These constraints come from three sources—security operations from the patch, the code path of each slice, and security rules. We now describe how SID collects or constructs these constraints.

<table>
<thead>
<tr>
<th>Security operations</th>
<th>Constraints from security operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer nullification</td>
<td>$\text{FLAG}<em>{CV} \geq 1 \quad \text{FLAG}</em>{CV} \leq 0$</td>
</tr>
<tr>
<td>Initialization</td>
<td>$\text{FLAG}<em>{CV} = 1 \quad \text{FLAG}</em>{CV} = 0$</td>
</tr>
<tr>
<td>Permission check</td>
<td>$\text{FLAG}<em>{CV} = 1 \quad \text{FLAG}</em>{CV} = 0$</td>
</tr>
<tr>
<td>Bound check</td>
<td>$CV &lt; \text{UpBound}$, or $CV \geq \text{UpBound}$, resp.</td>
</tr>
<tr>
<td></td>
<td>$CV &gt; \text{LowBound}$, or $CV \leq \text{LowBound}$</td>
</tr>
</tbody>
</table>

TABLE V: Constraints for security operations from patches. $\text{FLAG}_{CV}$: Flag symbol; $CV$: critical variable; UpBound: checked upper bound; LowBound: checked lower bound.

Constructing constraints from security operations. The constraints from security operations are used to capture the “effects” of them in preventing security impacts. We define the constraints for each class of security operation for the patched and unpatched code, respectively. Table V shows our rules for constructing constraints for different security operations. Besides out-of-bound access, constraints for other security operations are used to indicate whether the security operations are present. Therefore, we use a binary-flag symbol to represent the constraint. Specifically, $\text{FLAG}_{CV} = 1$ indicates the status of the corresponding critical variables, in terms of the presence of the security operations. $\text{FLAG}_{CV} = 0$ means that the security operation has been enforced against critical variable $CV$. By contrast, $\text{FLAG}_{CV} = 0$ indicates the absence of the security
operation for CV. The constraints for bound-check cases are more complicated. The constraints are used to limit the upper bound and/or the lower bound of a memory object. In addition to indicating the presence of the security operation, we also need to know the specific value range of the value of the critical variables. Thus, we use symbolized critical variable to represent the value.

To check against the requirements for the patched and unpatched code, we must construct these constraints differently (in an opposite way). For permission bypass, use-after-free, double-free, and uninitialized use, SID inserts constraints $FLAG_{CV} = 1$ for the security operations in the patched version while inserting $FLAG_{CV} = 0$ in the unpatched version because the security operations are missing. For out-of-bound access vulnerabilities, SID adds the constraints $CV < UpBound$ and/or $CV > LowBound$ on the symbolized critical variable in the patched version. For example, in Figure 3, SID inserts the constraint from the security operation in the patch is $sta_id < IWLAGN\_STATION\_COUNT$. The values of $UpBound$ and $LowBound$ are determined based on the specific bound-check security operations. In the unpatched version, SID instead inserts the constraints, $CV >= UpBound$ or $CV <= LowBound$, which are opposite to the ones in the patched version. This is to prove that, without the security operation, an out-of-bound access problem will occur in the unpatched version. For the example in Figure 3, SID will insert the constraint $sta_id >= IWLAGN\_STATION\_COUNT$ for the patched version.

Collecting constraints from slice paths. The constraints from paths (from a security operation to a vulnerable operation) are collected from two parts. The first part is the same as the one in removing infeasible slices (see Section IV-A). These constraints are collected from path conditions that are checked to make sure the slice itself is feasible. The second part is to collect manipulations against critical variables. For example, in the uninitialized use case, for an initialization against the critical variable in the slice path, SID will add a constraint, $FLAG_{CV} = 1$. 

Constructing constraints from security rules. The last set of constraints SID constructs are from the security rules. These constraints are important to evaluate if a security-rule violation may occur. We develop multiple rules for constructing these constraints of different security rules, as shown in Table VI. SID also constructs opposite constraints for the patched and unpatched versions.

For the patched version, we want to prove that, with the protection of the security operations, it is impossible to violate the security rules. Therefore, SID inserts rule-violating constraints and hopes that they are unsolvable, i.e., $FLAG_{CV} = 0$, $CV >= MAX$ and/or $CV <= MIN$. SID will first employ static analysis to figure out the size of the memory object in use. In the example in Figure 3, SID can easily find that the buffer, stations[], is on the stack with a fixed length 16. For other cases, e.g., the buffer is on the heap, SID will use backward data-flow analysis to find the allocation site to determine its size. If the size is a constant, the value will be used; otherwise, for a variable, SID instead symbolizes it. After knowing the buffer size, SID then inserts an out-bound constraint (e.g., $sta_id >= 16$) for the patched version. For the unpatched version, we want to prove that, without the security operations, it always violates the security rules. In order to achieve this, the constraints in the unpatched version will be opposite to the constraints in the patched version, which instead represents the compliance of the security rules. Here, we hope to prove that the constraints are unsolvable, so the compliance of security rules is impossible. For example, in Figure 3, SID inserts the constraint, $sta_id < 16$, for the unpatched version.

<table>
<thead>
<tr>
<th>Security rules</th>
<th>Patched version</th>
<th>Unpatched version</th>
</tr>
</thead>
<tbody>
<tr>
<td>No use after free</td>
<td>$FLAG_{CV} = 0$</td>
<td>$FLAG_{CV} = 1$</td>
</tr>
<tr>
<td>Use after initialization</td>
<td>$FLAG_{CV} = 0$</td>
<td>$FLAG_{CV} = 1$</td>
</tr>
<tr>
<td>Permission check before</td>
<td>$FLAG_{CV} = 0$</td>
<td>$FLAG_{CV} = 1$</td>
</tr>
<tr>
<td>sensitive operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In-bound access</td>
<td>$CV \geq MAX$, and/or $CV &lt; MIN$. resp.</td>
<td>$CV &gt; MIN$</td>
</tr>
</tbody>
</table>

TABLE VI: Rules for constructing constraints from security rules. MAX: maximum bound of the buffer; MIN: minimum bound of the buffer.

2) Solvability for each slice: To know the solvability of each slice, SID merges the three sets of constraints as the final ones for the patched and unpatched versions, respectively. SID then uses SMT solver, Z3, to solve the constraints. In the example in Figure 3, for the patched version, the final constraint set is $sta_id < IWLAGN\_STATION\_COUNT \&\& sta_id >= 16$, which are generated from the security operation and security rules. Similarly, for the unpatched version, the final constraint set is $sta_id >= IWLAGN\_STATION\_COUNT \&\& sta_id < 16$. Both final constraint sets for the patched and unpatched versions are unsolvable, because $IWLAGN\_STATION\_COUNT$ is 16.

3) Comparison against symbolic execution: Finally, after solving these constraints, SID compares the results of solvability to determine security impacts. A patch is determined to fix a security impact if the constraints for both the patched and unpatched versions are unsolvable, which means that the patched version must not violate the security rule, and the unpatched version must violate the security rule. Therefore, the patch fixes the violation against the security rule. Thanks to the conservativeness of under-constrained symbolic execution, the determination is precise. In the example in Figure 3, SID finds both constraints in the patched and unpatched versions unsolvable, so this patch fixes an out-of-bound access problem. If either constraint set is solvable, SID disqualifies the bug fixed by the patch as a security bug.

V. IMPLEMENTATION

We have implemented SID on top of LLVM with multiple passes for finding security checks, symbolic execution, and data-flow analysis. SID in total contains 5.6K line C++ code and 1.2K line Python code. The rest of this section presents important implementation details of SID, including preparing analysis environment, collecting vulnerable and security operations, matching information between the patched version and the unpatched version, and the symbolic-execution engine.

A. Preparing Analysis Environment

Since not all of the patches fix bugs, we use script code to eliminate common non-bug commits such as branch merging,
Preparing LLVM IR. Since SID is based on LLVM, the patches should be compiled into LLVM IR with a patched version and an unpatched version. Because the corresponding vulnerable operations can be in other source files than the one containing the patch, we employ static analysis to extract the dependent files from the patch automatically. We first extract the patch code and variables involved in the patch. Then, we employ standard control-dependency analysis against the patched code and taint analysis against the involved variables, in a both forward and backward manner, to identify all dependent files. Interestingly, we found that, in most cases, the file containing the patch also contains the vulnerable operations. The unpatched version is obtained simply by checking out the git commit right before the one for the patch. Finally, we invoke clang to compile these files, for both the patched version and unpatched version, into LLVM IR.

To mitigate the path-explosion problem in the under-constrained symbolic execution, we unrolled loops in the LLVM IR level by treating them as if statements, which is a common practice adopted by recent techniques [56] [57]. To identify indirect-call targets, we take recent advances that use struct types to match the function targets [18] [31] [62].

B. Identifying Security Operations

As shown in [3IV] in dissecting patches, SID first identifies security operations. By analyzing the patch code in the git log, SID can tell if a patch contains security operations. However, the security operations can be intended for new variables introduced in the patch. In this case, the security operations are not aimed at fixing bugs in the unpatched code. To eliminate these cases, SID ensures that the critical variables are also in the unpatched code. Further details are presented in [3V-D]. For different types of security impacts, the corresponding security operations are different. We identify security operations as follows.

• **Bound checks.** We regard bound checks as security operations. We use two criteria to identify bound checks. (1) A bound check uses a conditional statement such as if statement. The operator of the comparison instruction should be =, >, <, >=, or <=, and both of the operands of bound check should be of integer type such as int or unsigned. (2) One branch of the conditional statement should result in error handling (e.g., returning an error code) when a bound check fails while other branches continue normal execution. This is similar to the check definition in [31, 35].

• **Pointer nullification.** NULL checks are typically enforced before using pointers. Based on the common patch patterns, we regard pointer nullification as the security operation against use-after-free. Nullification can be easily identified when NULL is assigned to a pointer.

• **Initialization.** We regard initialization operations as the security operations against uninitialized use. Initialization is either a store instruction that assigns 0 to a variable or a call to memset() that takes 0 as the value argument.

• **Permission checks.** Permission check is the security operation against permission bypass. By looking into how permission bypass is commonly patched, we first empirically collect the common permission functions such as afs_permission() and ns_capable(). Then, we identify a conditional statement (e.g., if statement) as a permission check if it is a security check [31] against the return value of these permission functions.

C. Identifying vulnerable operation

SID then identifies vulnerable operations. To do that, SID first extracts the critical variables from the identified security operations, as described in [3IV-A]. Based on the uses of the critical variables, SID employs data-flow analysis (i.e., slicing) to identify the vulnerable operations, according to the rules in [3I-C]. Here, we have extracted the critical variables from security operations. With that, we present some implementation details on the identification of vulnerable operations that use the critical variables.

• **Out-of-bound access.** We first identify the instructions that access an array or buffer using the critical variables (i.e., size variables) as vulnerable operations. We also identify common read or write functions (e.g., memcpy()) that take as input the critical variables as vulnerable operations.

• **Use-after-free and double-free.** We conservatively identify all pointer dereference operations that target the critical variables as vulnerable operations.

• **Uninitialized use.** We identify the common operations on the uninitialized variables as vulnerable operations. These common operations include pointer dereference, function calls, memory access, and binary operations such as arithmetic operation. Also, these operations must also target the critical variables.

• **Permission bypass.** Based on the extracted the critical variables in permission checks, which take struct types such as kuid_t, inode, file or corresponding pointer types, we conservatively identify operations against critical variables as vulnerable operations.

D. Mapping Operations in Patched and Unpatched Versions

A patch may involve multiple security operations and vulnerable operations, thus multiple slices. To perform the symbolic rule comparison, we need to map the corresponding slices in the patched and unpatched versions. SID pairs slices relying on various types of information such as function name and control flow. Specifically, to pair the slices, SID first extracts the vulnerable operation of the slices for both the patched and unpatched versions. The vulnerable operations must exactly match. If the vulnerable operations are matched, SID further employs control-flow comparison to make sure that the two slices are the same except the parts introduced by the patch. With these two steps, SID can map the slices between the patched version and the unpatched version.

E. The Under-Constrained Symbolic-Execution Engine

SID uses under-constrained symbolic execution to analyze the code for patched and unpatched versions. Similar to UC-KLEE [41], the symbolic execution of SID can start from any point in a function. Specifically, SID only executes on the slices collected from the static analysis during dissecting
patches. Since the collected constraints in these individual slices are not complete, they are under-constrained, which may lead to false negatives. However, S\textsuperscript{ID}'s main goal is to determine security impacts with a low false-positive rate. We will discuss how to collect more constraints beyond the slices in §VII.

**Avoiding path explosion.** Path explosion is a general problem in symbolic execution, which is fortunately mitigated in S\textsuperscript{ID}. Different from the whole-program symbolic execution, S\textsuperscript{ID} only works on the slices collected by the static analysis. Most of the security operations are near the vulnerable operations, so most slices are short, involving a single module. As such, the path-explosion problem, in most cases, does not occur. However, we did observe the path-explosion problem in some instances. To completely avoid path explosion, S\textsuperscript{ID} chooses to discard slices with more than 150 basic blocks. This threshold number is carefully selected based on our statistical study—more than 98.8\% of slices cover less than 150 basic blocks. The heuristic is also used in previous works, such as UC-KLEE [41], to alleviate path explosion. By using this method, slices can be symbolically analyzed quickly without encountering the path-explosion problem.

**VI. Evaluation**

We evaluate the accuracy, effectiveness, and scalability of S\textsuperscript{ID}, and also present new findings regarding characteristics of security bugs. We chose the Linux kernel as the target program, and collected more than 66K git commits in recent years. During the pre-processing, 11,433 non-bug commits are eliminated, which finally returns us 54,651 valid patches. For these valid patches, we generated 110,136 LLVM IR bit code files for both the patched and unpatched versions. The experiments were performed on Ubuntu 18.04, 64-bit OS with LLVM-8.0. The machine has a 32GB RAM and is equipped with six cores Intel (R) Xeon (R) W-2133 CPU @ 3.60GHz. It is worth noting that the following measurement uses a single thread without parallel computing.

**Efficiency and scalability.** For each patch, S\textsuperscript{ID} analyzes both the patched version and the unpatched version. The analysis takes an average of 0.415 seconds (median 0.056s, max 15s) for each version, which means that, for every patch, the analysis costs 0.83 seconds on average. During the analysis, the detection of out-of-bound read or write vulnerabilities patches takes 63\% of the total time, while all other cases take only 37\% of the total time. Processing patches for out-of-bound access vulnerabilities requires more time since some more slices and constraints that are more complex, requiring more time for symbolic execution. The results indicate that S\textsuperscript{ID} is efficient enough to handle a massive amount of patches precisely.

A. **False Positives of S\textsuperscript{ID}**

We use precision, \(|TP| / |TP+FP|\), to evaluate the false positives of S\textsuperscript{ID}, where TP and FP are the numbers of true positives and false positives. To calculate the precision, we manually checked all the results generated by S\textsuperscript{ID}. In order to precisely confirm that these bugs are true security bugs, we look into the patch description (comments), the patch code, and the involved source code. If the comments have already mentioned the same security impacts as S\textsuperscript{ID} found, we regard them as a security bug because both Linux maintainers and reporters have confirmed the security impact. Otherwise, we manually review the patch code and the involved source code to check (1) if the vulnerable operations found by S\textsuperscript{ID} indeed introduce security impacts in the unpatched version and (2) if these security impacts are eliminated by the security operations in the patch. If both of these conditions are true, we confirm the security impacts and the security bug. Finally, we confirmed 227 security bugs with 8 false-positive cases. As a result, the precision rate of S\textsuperscript{ID} is 97\%. We investigated these false positives and summarized the reasons as follows.

**Missing constraints in preventive-patching cases.** S\textsuperscript{ID} employs under-constrained symbolic execution to analyze only the slices that start from the security operations to the vulnerable operations. As such, earlier constraints that are before the security operations will be missed. In general, if the "earlier" constraints have already been able to prevent a security impact, the constraints in the patch are unnecessary. However, we did find five cases in which the patches are preventive and enforce redundant constraints. Developers enforce the preventive patches because the “earlier” constraints can be changed in future code. In these cases, S\textsuperscript{ID} will identify them as patches fixing security bugs, leading to false positives. Eliminating all these preventive patches is a hard problem, which requires a more complete constraints set. However, we would like to mention that these five cases violate S\textsuperscript{ID}'s threat model—the provided patches correctly fix actual bugs.

**Inaccurate static analysis.** During dissecting patches, S\textsuperscript{ID} employs static data-flow analysis to find the slices from security operations to vulnerable operations. Due to the inaccuracy of the static analysis and the incompleteness of identifying security or vulnerable operations, many slices are infeasible. Although S\textsuperscript{ID} further employs symbolic execution to validate the slices, because the symbolic execution is under-constrained, the resulting slices may still be infeasible, leading to false positives. The remaining three false positives are caused by such inaccuracy. In the future, we plan to improve the under-constrained symbolic execution by collecting more constraints, as discussed in §VI.

B. **False Negatives of S\textsuperscript{ID}**

By design, S\textsuperscript{ID} aims to ensure less false positives by allowing more false negatives. In this section, we evaluate the false negatives of S\textsuperscript{ID} and investigate the causes. Generally, the evaluation of false-negative cases for static analysis tools is not as easy as a precision evaluation because it requires a ground-truth set. To this end, we use S\textsuperscript{ID} to detect known vulnerabilities in the Linux kernel to evaluate how many of them are missed by S\textsuperscript{ID}. Specifically, like selecting recent patches in §II-B, we chose patches from 100 recent vulnerabilities, which violate at least one of our security rules. We used S\textsuperscript{ID} to analyze the corresponding patches. It turns out that S\textsuperscript{ID} found 47 vulnerabilities out of the 100. Therefore, S\textsuperscript{ID} missed 53\% of vulnerabilities. After manually checking the false-negative cases, we found the following main causes. The corresponding solutions to reducing false negatives are discussed in §VII.

**Under-constrained symbolic execution.** S\textsuperscript{ID} uses under-constraint symbolic execution in both patched code and unpatched code to determine security impacts conservatively. In some cases, even when the patched code can never violate
a security rule or the unpatched code must violate the security rule, the conservative approach may not be able to prove it. The conservative approach is mainly to reduce false positives, which, however, introduces a significant number of false negatives. This problem causes 17 cases.

**Incomplete coverage for security and vulnerable operations.** Intuitively, the incompleteness of the coverage will result in false negatives in confirming security impacts. In the current implementation, we collected the most common operations based on our statistical study (§IV-C). For example, for out-of-bound read or write bugs, we only collected the vulnerable operations such as array operations, common read and write functions; however, vulnerable operations using custom functions would be missed. This problem causes 31 cases. Also, similar to the causes of false positives, there are 5 cases caused by inaccurate static analysis.

**C. The Trade-off between False Positive and False Negative**

Handling partial-fix patches. To distinguish the root difference between patches for general bugs and security-related bugs, we introduce the concept of safety-state transition. First, we say that the program is in an unsafe state if it violates at least one security rule; we say the program is in a safe state if the violations are eliminated. Figure 4 shows all the transition states. Most commonly, a patch fixes a security bug if the unpatched version is in the unsafe state, and the patched version is in the safe state. In this case, we say that the patch blocks all the security impacts of the bug. However, in some corner cases, a patch only relieves a security bug, for which both patched and unpatched versions are in the unsafe state, and the patched version has fewer security-rule violations than the unpatched version does. Thus, we call them partial-fix patches. In this project, since $\text{S}^\text{ID}$ is designed to cover the patches that correctly and completely fix a security bug, it may miss security bugs with partial-fix patches. Table VII summarizes how partial-fix can happen for the covered types of bugs. In the course of our evaluation, we only found two partial-fix cases—incompletely initializing memory, which shows that partial-fix patches are not common. However, in the future, it is possible to extend $\text{S}^\text{ID}$ to detect such partial-fix cases by analyzing the security operations in a finger-grained manner (e.g., which bytes have been initialized) and relaxing the symbolic rules (i.e., does not require the block of all violations).

Relaxing symbolic rules. A unique strength of $\text{S}^\text{ID}$ is using the conservativeness of under-constrained symbolic execution to precisely determine security impacts. Specifically, if the under-constrained symbolic execution is unsolvable, it is truly unsolvable; however, if the under-constrained symbolic execution is solvable, it can be a false positive due to missing constraints.

<table>
<thead>
<tr>
<th>Bug types</th>
<th>Partial fix on security operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out-of-bound access</td>
<td>Incomplete bound check</td>
</tr>
<tr>
<td>Use-after-free/double-free</td>
<td>N/A</td>
</tr>
<tr>
<td>Uninitialized use</td>
<td>Incomplete initialize</td>
</tr>
<tr>
<td>Permission bypass</td>
<td>Incomplete perm check</td>
</tr>
</tbody>
</table>

**TABLE VII: Possible partial-fix patches.**

Therefore, we design strict rules—the patch should block all violations (against a security rule) in the unpatched version by proving the opposite constraints (see §IV-B) unsolvable.

Readers may wonder whether we can relax the symbolic rules—determining security impacts as long as some violations have been blocked—to reduce false negatives of $\text{S}^\text{ID}$. A critical issue with rule relaxing is that, with the relaxed rules, we cannot construct the opposite constraints to prove the unsolvability. This will prevent $\text{S}^\text{ID}$ from benefiting from the conservativeness of under-constrained symbolic execution because the detected blocks of violations will likely be false positives due to missing constraints, rendering the security-impact determination highly imprecise.

**D. Security Evaluation for Identified Security Bugs**

Every patch identified by $\text{S}^\text{ID}$, besides the false positives (3%), fixed at least one security impacts; therefore, we believe that the corresponding bugs are security bugs. To further validate that the identified security bugs are real vulnerabilities, we conduct two evaluations: (1) requesting CVEs and (2) analyzing reachability.

Vulnerability confirmation for CVE. Surprisingly, out of 227 security bugs found by $\text{S}^\text{ID}$, only 31 of them have been already assigned with CVE numbers. The remaining 196 security bugs were not reported and were improperly treated as non-security bugs. To confirm vulnerabilities, we request CVEs for the remaining security bugs in two phases. In the first phase, we requested CVEs for 40 security bugs individually. Due to that requesting CVE is a time-consuming process; in the second phase, we requested CVEs in a single batch.

The 40 security bugs submitted in the first phase come from two sets; the first set contains 21 bugs that are patched in the Linux kernel but still unpatched in the Android kernel. We believe that this set represents less-likely security bugs because the Android team might have confirmed them as non-security bugs thus did not patch them. The second set includes the other 19 detected security bugs that are randomly selected but cover different types of security impacts. For these 40 cases, which were submitted individually, we have received responses for 37 of them. In particular, we have obtained 24 CVEs for 23 security bugs (one bug was assigned with two CVE IDs), and 9 of these bugs are from the unpatched set in Android. 14 security bugs will not be assigned with CVEs due to non-technical or controversial reasons: (1) the security bug is in pre-release versions (5 cases); (2) information-leak (memory disclosure to userspace) bugs in obsolete kernel code (5 cases); (3) patch commits do not mention security impacts (4 cases). For the first reason, we believe that most code will

\[\text{Response: “CVE IDs are not required for information leak to userspace in various obsolete kernel code from approximately 2013.”}\]
be released, and the corresponding security bugs will qualify CVEs. For the second reason, we would disagree—memory disclosures to userspace are security-critical because they break ASLR [26, 29] and leak sensitive data; also, the involved code still exists in the latest and released versions. The third reason shows that manually confirming the security impacts of bugs without commits mentioning security issues is hard.

Because maintainers would not assign CVEs for bugs in the pre-release (i.e., release candidate (RC)) versions, in the second phase, we filtered out 42 such bugs and reported 114 security bugs in a single batch. We still have not received the responses yet because the review of the reports for these bugs requires significant manual work for CVE maintainers. In comparison, individual reports receive responses much more timely. In summary, we totally reported 154 security bugs to CVE maintainers. We have received 37 responses with 24 new CVEs assigned. This means that, including the previously confirmed CVEs, 54 out of 227 identified bugs have been assigned with CVEs. Note that none of the rejected cases is due to misidentifying security impacts. Table XIII show the details of the security bugs and CVE. These results indicate that S1D is effective in determining security bugs from massive general bugs.

**Reachability analysis for security bugs.** Since a bug becomes a vulnerability when it can be triggered and cause security impacts, we also evaluate the reachability (from attacker-controllable entry points) of the identified bugs. The identified security bugs were detected through either fuzzers or other techniques such as static analysis. Clearly, if a bug was found by fuzzers such as Syzkaller [46], we can directly confirm its reachability. In particular, by checking the git commits, which would mention the corresponding fuzzers if a bug was found through fuzzing, we found that 28 (12.3%) of identified security bugs were found by fuzzers, thus are reachable from attackers.

The remaining 199 security bugs were mainly found through static analysis; confirming their reachability from attacker-controllable entry points has been a challenging and open problem [20]. Therefore, in this evaluation, we focus on finding the reachable call-chain between attacker-controllable functions and the functions containing the vulnerable operations. To this end, we first identified entry points—functions in the kernel that can be arbitrarily called by attackers. Based on how the kernel interacts with external entities, we empirically identify the following entry points.

- **System calls.** They are the most commonly used interface between user-space and kernel-space, which are also widely targeted by kernel fuzzers.
- **Driver-specific I/O-control handlers.** These handler functions are registered and can be called through the ioctl system call. By setting specific parameters, attackers can control the handlers. The previous work, DIFUZE [10], also fuzzes these handlers to find bugs in the kernel drivers.
- **Interrupt (IRQ) handlers in drivers.** Malicious hardware can invoke such handler functions by triggering the interrupt and prepare their parameter; therefore, they are also controllable to attackers. PeriScope [45] also fuzzes kernel drivers and regards these IRQ handlers as entry points.

We first identify the 338 system calls in the Linux kernel based on the system-call list [1]. Then, following the method of DIFUZE [10], we identify a set of structures that can be used to register ioctl handler by drivers, and based on these structures, we find 603 ioctl handlers [11]. To identify IRQ handlers, PeriScope [45] shows that drivers can register their own IRQ using multiple types of APIs, and tasklest is one of the most commonly used software interrupts (softirq). Therefore, based on the declarations of tasklest and IRQ-related keywords in drivers, we finally find 126 IRQ handlers.

The idea of the evaluation is to traverse the global call-graph of the kernel to collect the shortest call-chain path between the entry points and functions containing the vulnerable operations. We employ Dijkstra’s Shortest Path (DSP) algorithm [16] to find the paths. Given a bug, we say it is reachable from attacker-controllable entry points if we find such paths. To minimize false positives and false negatives, we employ the state-of-the-art techniques—using struct types to match functions [28, 31, 62]—to precisely identify indirect-call targets. Table VIII shows the number of bugs that are reachable from different types of entry points. In particular, we found that 133 security bugs are reachable from systems, and 154 are reachable from the three classes of entry points.

<table>
<thead>
<tr>
<th>Entry points</th>
<th>Num of reachable bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamically confirmed bugs (fuzzers)</td>
<td>28</td>
</tr>
<tr>
<td>System calls</td>
<td>133</td>
</tr>
<tr>
<td>I/O control handlers</td>
<td>148</td>
</tr>
<tr>
<td>Interrupt handlers</td>
<td>131</td>
</tr>
<tr>
<td>Total</td>
<td>154 (67.8%)</td>
</tr>
</tbody>
</table>

**TABLE VIII:** Number of bugs that can be reached from different kinds of entry points.

**E. Generality of S1D’s Patch Model**

<table>
<thead>
<tr>
<th>Num of key components</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three components</td>
<td>77%</td>
</tr>
<tr>
<td>Two or one component</td>
<td>11%</td>
</tr>
<tr>
<td>Other cases</td>
<td>12%</td>
</tr>
</tbody>
</table>

**TABLE IX:** The generality of S1D’s patch model. It shows the numbers of components the vulnerabilities have.

S1D’s patch model includes three key components of patches: security operation, vulnerable operation, and critical variables. To evaluate the generality of the model, we analyzed the most recent 100 vulnerabilities in the Linux kernel that were disclosed in 2019, Table IX shows the statistical results of this evaluation.

We can find all of the three components in 77 vulnerabilities; therefore, S1D can support the security-impact determination for them. Furthermore, 11 vulnerabilities only have one or two of these key components. For example, pointer usage in an incorrect order can introduce use-after-free vulnerabilities, and the corresponding patches just change the pointer reference order. In this case, the security operation is not modeled and thus will be missed. In addition, 12 cases involve code removal as the fix or multiple patches, which cannot be clearly represented by S1D’s current model. For example, there are five patches that only delete some redundant code, such as deallocation functions. Some vulnerabilities were fixed by more than one patches or complex patches.
STD’s model is, in fact, conceptual and general—while a vulnerability typically has vulnerable operations, the patch performs security operations to prevent them, and both kinds of operations often target variables. In the future, we can certainly extend the model to support more cases. For example, even for memory leaks where there is no explicit vulnerable operation at all, we can artificially model “object pointer never being released” as the vulnerable operation, which can be realized by analyzing the operations against the pointer (critical variable).

F. New and Important Findings

Patching-time window for security and general bugs. In order to show the importance of STD, we would like to know how Linux maintainers treat security bugs and general bugs differently. To this end, we measure and compare the patching time window for them—the time from the submission/report of a patch to the application of the patch. We tested 8,000 patches for general bugs, 1,339 patches for vulnerabilities, and all the security bugs that are found by STD but do not have CVE ID. We use the cumulative distribution function (CDF) to show the statistical patching-time window in Figure 5. We find that the patching-time window for CVE-assigned vulnerabilities (5.8 days) is shorter than security bugs (8.6 days) found by STD. This means that the maintainers have not treated these security bugs as important as vulnerabilities.

We also find that the patching time window of security bugs identified by STD is shorter than other general bugs. We believe one reason is that the patches for these security bugs have fewer code changes, and the bugs have clearer patterns, which is also reported by Li et al. [27].

More statistical results are shown in Table X, from which we can find that security patches still take a long time to be applied. Nearly 10% of security bugs are patched more than one month after they have been reported. This significant time window gives attackers much time to craft critical exploits, not to mention that the reported patches are visible to attackers. Thus, an automated tool that can determine the security impacts of bugs is demanded.

<table>
<thead>
<tr>
<th>Type</th>
<th>Average (Days)</th>
<th>Median (Days)</th>
<th>Maximum (Days)</th>
</tr>
</thead>
<tbody>
<tr>
<td>General patches</td>
<td>15.8</td>
<td>3</td>
<td>1240</td>
</tr>
<tr>
<td>Patches of security bugs</td>
<td>8.6</td>
<td>2</td>
<td>111</td>
</tr>
<tr>
<td>Patches of vulnerabilities</td>
<td>5.8</td>
<td>1</td>
<td>974</td>
</tr>
</tbody>
</table>

TABLE X: Statistics on patch-time window.

Delayed disclosure of security impacts of existing vulnerabilities. We found that the disclosure of the security impacts of existing vulnerabilities is commonly delayed, which is also known as hidden impact vulnerability [53]. To measure the delaying, we define the delayed time as the time window from the patch date to the CVE-release date. We collected 1,339 vulnerabilities in the Linux kernel from the CVE database [13] [38] and analyzed the delayed time. The CDF for the delayed time is shown in Figure 5. The results show that only 23.9% of them are identified as vulnerabilities less than two weeks (14 days) after they have been patched. The other 75% are reported as vulnerabilities after two weeks. The average and median delayed time period for these vulnerabilities is 191 and 45 days. The longest case is more than 12 years, which was identified as a security bug by STD and assigned with a CVE (CVE-2007-6762) after we reported it.

Security bugs threatening derivative software. Many programs are derived from other open-sourced programs. For example, the Android kernel is a modified version of the Linux kernel, and the Android kernel is further customized into thousands of versions [52] running on tens of thousand device models [64]. This problem is known as Android fragmentation [52] [64]. Manufactures are unable to fix all bugs timely, due to a large number of derivative programs. Instead, they prioritize security-critical ones and postpone or even ignore non-security ones. To understand the severity of the problem, we test more than 5K bug patches in the Android kernel of version 4.14-p-release and perform two evaluations.

The first evaluation is to check how many security bugs (identified by STD) remain unpatched in Android. Specifically, we manually checked the security bugs found by STD in the latest (as of the experiment) Android kernel release version, Android-4.14-p-release, which was ported from the Linux kernel 4.14 on November 12, 2017 [48]. As such, patches applied before the date in Linux will also be available in Android. Therefore, we analyzed the security bugs found by STD that were introduced before November 12, 2017, but patched in Linux after the date. We found that 39 such security bugs were reported after November 12, 2017, in the Linux kernel and were not assigned with a CVE; 11 of them do not affect the Android code anymore; thus, they are excluded. For the remaining 28 security bugs, we found that only seven are patched in the Android kernel, and 21 (75%) remain unpatched. Details can be found in appendix (Table XIII). These security bugs may pose serious security risks to Android.

The second evaluation is to measure the bug-fixing time windows of CVE-assigned bugs and non-security bugs in Android. Specifically, non-CVE bugs are fixed with an average of 44.6 days (30 days following Linux kernel patches), while CVE-assigned bugs are fixed with an average of 27.8 days (only 14 days following the Linux kernel patches). For security bugs, because most of them are not identified as vulnerabilities, they will be treated as general bugs and would not be fixed in time. The average time window is 44.6 days. For instance, Figure 6 is a missing bound check security bug in the Linux kernel. After our report, this security bug has a CVE ID, CVE-2019-15926, with CVSS score 9.4. This bug was introduced from the Linux kernel 3.0 but still has not been patched in the Android 4.14-p-release until the submission of this paper.

Multi-impacts per bug. The semantics of vulnerable operations often decides security impacts. Intuitively, when a critical variable is used in multiple vulnerable operations, it may have multiple security impacts. In particular, for the 227 security bugs, STD actually found 243 security impacts, as shown in Table XI. For example, some out-of-bound access cases are also caused by uninitialized use when the uninitialized variable is used as a size variable in memory access.

Characterizing bugs and vulnerabilities. In addition to the security impacts, we also characterize other differences between general bugs and security bugs. First, we analyzed the differences in the number of changed lines in patches for 1,350 randomly selected general bugs, the security bugs...
Fig. 5: Statistical findings. CDF: cumulative distribution function; (1) CDF for time windows from bug report date to bug fix date; (2) CDF for time window from the patch date to the CVE release date; (3) CDF for the number of changed lines for different kind of bugs. In (2), about 1% of vulnerabilities are assigned with a CVE before the bugs are actually patched.

Fig. 6: An out-of-bound access vulnerability in Android 4.14

<table>
<thead>
<tr>
<th>Security impacts</th>
<th>#</th>
<th>Security impacts</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninitialized use</td>
<td>85</td>
<td>Use-after-free/Double-free</td>
<td>67</td>
</tr>
<tr>
<td>Out-of-bound access</td>
<td>65</td>
<td>Permission bypass</td>
<td>13</td>
</tr>
<tr>
<td>NULL-pointer dereference</td>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE XI: Security impacts for the identified security bugs

Another finding is that among the git commits for the 1,339 vulnerabilities, 814 (60.8%) of them did not mention any security impacts such as use-after-free, double-free, etc. This result implies that one cannot reliably determine the security impacts solely based on the textual information in patch commits.

VII. DISCUSSION

The generality of Sid. Sid can be extended to determine the security impacts of patches for other well maintained open-source programs such as Firefox, Chrome, and FreeBSD. Our patch model is general, which can be used to describe the security operations and vulnerable operations in a program-agnostic manner. To extend Sid to other programs, only the pre-processing part requires new manual effort. For example, for out-of-bound access vulnerabilities, for different target programs, we need to collect functions for memory access and bound checks.

The extensibility of Sid. In our work, Sid is only used to support the common classes of vulnerabilities. However, other classes of vulnerabilities can also be supported by specifying the security rules for them—how these vulnerabilities violate the security rules. In addition, the rules for constructing constraints from security operations and security rules should also be specified. Table XII shows that we can naturally extend Sid to support more classes of bugs by modeling the three components of their patches. For example, numerical-error vulnerabilities such as divide-by-zero can be supported. The security rule is that the divisor cannot be zero. Correspondingly, the vulnerable operation is division, and the security operation can be a zero-check for the divisor. Similarly, NULL-pointer dereference also fits Sid’s model. The security rule is that a dereferenced pointer cannot be NULL. Therefore, the vulnerable operation is pointer dereferencing, and the security operations can be a NULL check. After including these types, Sid can cover at least 51% of vulnerabilities (13% more). In the future, we would like to support more classes of vulnerabilities.

In addition, the current implementation of Sid considers only simple patching patterns for vulnerabilities because we find that the average distance between a security operation and a vulnerable operation is 6.6 lines of code, and the longest distance is 65 lines of code. This result is consistent with the finding of SPIDER [32]—94.4% of safe patches affect less than 20 lines of code. Therefore, the under-constrained symbolic execution can handle most of them efficiently. In the future, Sid can also be extended to support complicated patterns. For example, some patches use multiple security operations together to fix a vulnerability. These cases can be described using complex security rules and represented with multiple
Reducing false negatives. First, the conservativeness of the under-constrained symbolic execution indeed introduces a significant number of false negatives, because in the current implementation, the constraints for the slice paths are collected only from the security operation to the vulnerable operation. Therefore, the constraints are under-constrained. As an improvement, in the future, we can extend the constraint collection beyond the security operation—backwardly collecting as many constraints as possible from the security operation and adding them to the final constraint set. This method could reduce at most 17% of false negatives in our evaluation.

Second, the incompleteness of the security and vulnerable operations sets also causes false negatives. To reduce them, we can collect more custom functions for security and vulnerable operations. Such functions can be modeled based on dynamic analysis [63] and wrapper-function analysis [21, 62]. Covering more security and vulnerable operations can reduce at most 31% of false negatives. For example, if we model the lock/unlock operations for use-after-free, we can additionally cover 25% of use-after-free vulnerabilities in the evaluation. But to do so, more manually analysis work on patches and engineering efforts are needed. Therefore, we put these works in future works.

VIII. RELATED WORK

Mining security-critical vulnerabilities from bugs. Wijayaekara et al. [53] show the hidden impact vulnerabilities that were first identified as non-security bugs and publicized and later were identified as vulnerabilities due to exploits. In addition, previous work [5, 13, 23, 50, 54, 65] has used supervised and unsupervised learning techniques to classify the vulnerabilities and general bugs based on the textual information of the patches. Tsy [50] showed that the Naive Bayes and Support Vector Machine classifiers always have the best performance. However, such work cannot handle the patches without descriptions or if they have incomplete/inaccurate descriptions. Moreover, these works focused only on differentiating vulnerabilities from general bugs, which cannot determine the specific security impacts of the bugs or pinpoint the vulnerable operations resulting in the security impacts. Recent work, SPIDER [32], identifies fixes as security fixes as long as they do not disrupt the intended functionalities. This assumption does not hold for some patches, such as the ones only improving code readability or replacing equivalent APIs. In contrast to these papers, Std aims to automatically determine the security impacts of a patched bug and provide details on the vulnerable operations even when the commit description is not available. Moreover, Std is not based on the assumptions used by SPIDER.

Testing the exploitability of bugs. Several prior studies have attempted to test the exploitability of a particular class of bugs. Specifically, Lu et al. [50] showed how to exploit uninitialized-use bugs using symbolic execution and fuzzing in the Linux kernel. Xu et al. [58] presented a memory conflict strategy to exploit the use-after-free vulnerabilities in the Linux kernel. You et al. [61] presented SEMFuzz, which guides the automatic generation of proof-of-concept exploits for vulnerabilities. Thanassis et al. [3] proposed Fuzz, a symbolic execution-based automatic exploit generation tool that can automatically exploit memory-corruption bugs such as buffer overflows. Wu et al. [55] presented FUZE, which can automatically exploit use-after-free vulnerabilities in the Linux kernel. Unlike these studies, Std does not focus on the exploitability of a specific class of vulnerability. Instead, Std aims to automatically determine the security impacts of a bug once it is triggered. Moreover, Std is not limited to a specific vulnerability class.

Bug-severity assessment. Mell et al. [33] presented the common vulnerability scoring system (CVSS), which is the most widely used vulnerability scoring system. CVSS requires manual scoring of the severity of vulnerabilities based on their confidentiality, integrity, and availability. However, Munuah et al. [47] showed that CVSS is often biased in determining the severity. For example, it does not treat code execution and privilege escalation as important factors when analyzing the severity of vulnerabilities. Most of the severity-analysis techniques [9, 34, 40, 43] are based on bug reports, which also cannot handle the bug patches without a description or if they have incomplete descriptions.

Symbolic execution. Symbolic execution has been used for decades. Cadar et al. [7] proposed a symbolic execution method to generate inputs to trigger bugs in real code automatically. Later, Cadar et al. developed KLEE [6], which is a widely-used symbolic execution engine. Both of these tools need the complete constraints in the program; thus, they can only symbolically execute from the entry of a program. Such symbolic execution does not scale well to large programs. Under-constrained symbolic execution [17, 41], implemented in UC-KLEE, lifts this limitation by treating symbolic values coming from unexecuted parts of the code especially. This approach makes symbolic execution much more flexible and expands the possible applications. Because under-constrained symbolic execution is unaware of properties of data established by the unexecuted code, it can still produce false-positive error reports that would not occur when executing a complete program. Similar to UC-KLEE, Std also uses under-constrained symbolic execution to execute from an arbitrary point in a
function symbolically. However, SID minimizes false positives by combining the constraints from security rules and differential analysis. Also, different from UC-KLEE, which detects bugs introduced by new patches, SID determines the security impacts of patches.

IX. CONCLUSION

Maintainers of large software programs are bombarded with a large number of bug reports without a reliable description of security impacts. With limited resources, maintainers have to prioritize the patching for bugs with security impacts, which is however challenging, and de-prioritizing a security-critical bug will lead to critical security problems. This paper presented SID, an automated approach to determining the security impacts for a massive number of bug patches. The core of SID is the symbolic rule comparison mechanism that employs differential, under-constrained symbolic execution to precisely confirm the security impacts of a bug. SID can further automatically classify vulnerabilities based on their security impacts. We have implemented SID and applied it to determine the security impacts of Linux-kernel bugs. As a result, SID have found 227 security bugs from 54K valid commits patches in the Linux kernel, and 21 of them remain unpatched in the latest Android kernel (version 4.14), which may cause critical security problems to Android devices. Many of the identified security bugs have been assigned with a CVE ID and a high CVSS score. The evaluation results show the precision and effectiveness of SID in automatically determining security impacts.

X. ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful feedback. This research was supported in part by the NSF awards CNS-1815621 and CNS-1931208. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES


<table>
<thead>
<tr>
<th>Git commit</th>
<th>BF</th>
<th>ST</th>
<th>SI</th>
<th>CSS</th>
<th>ET</th>
<th>Git commit</th>
<th>BF</th>
<th>ST</th>
<th>SI</th>
<th>CSS</th>
<th>ET</th>
</tr>
</thead>
<tbody>
<tr>
<td>7fe6d93e9e5eb19</td>
<td>R/U</td>
<td>MI</td>
<td>UAF</td>
<td>SII</td>
<td>MI</td>
<td>4.9</td>
<td>MN</td>
<td>R</td>
<td>N</td>
<td>MBC</td>
<td>OBA</td>
</tr>
<tr>
<td>6dd93e9e5eb19e15882b6c6caf</td>
<td>R/U</td>
<td>MI</td>
<td>UAF</td>
<td>SII</td>
<td>MI</td>
<td>1.9</td>
<td>MN</td>
<td>R</td>
<td>N</td>
<td>MBC</td>
<td>OBA</td>
</tr>
<tr>
<td>5d6751eaff672</td>
<td>R/D</td>
<td>MN</td>
<td>UAF</td>
<td>SII</td>
<td>MI</td>
<td>7.2</td>
<td>MN</td>
<td>R</td>
<td>N</td>
<td>MBC</td>
<td>OBA</td>
</tr>
<tr>
<td>0d012b9866249</td>
<td>R/D</td>
<td>MN</td>
<td>UAF</td>
<td>SII</td>
<td>MI</td>
<td>7.5</td>
<td>MN</td>
<td>R</td>
<td>N</td>
<td>MBC</td>
<td>OBA</td>
</tr>
</tbody>
</table>

**TABLE XIII:** List of security bugs detected by Sfd, BF: bug type; ST: security impact; SI: security impact; UI: entry point type; MBC: missing initialization; NPD: NULL-pointer dereference; UU: uninitialized use; MN: missing nullification; SII: system-call & IO count handler & IRQ handlers; MI: system-call & IO count handler & IRQ handlers; PE: profiled dynamically (by Fuzzier); CSS: common vulnerability scoring system.