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, SID, 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 SID 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. SID incorporates rule comparison, using under-constrained symbolic execution of a patch to determine the security impacts of an un-applied patch. SID can further automatically classify vulnerabilities based on their security impacts. We have implemented SID and applied it to bug patches of the Linux kernel and matching CVE-assigned vulnerabilities to evaluate its precision and recall. We optimized SID to reduce false positives, and our evaluation shows that, from 54K recent valid commit patches, SID 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 SID'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 [36] 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

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24419
www.ndss-symposium.org

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, 59] or whole-program symbolic execution [42, 60] 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, SID, to determine the security impacts of bugs, given their patches. Using security rules that capture common security impacts, SID distinguishes unsafe (rule-violating) and safe (rule-compliant) behaviors of patched and unpatched code, which allows SID to characterize the security impacts of a bug. SID 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, SID 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 SID's determination of security impacts. We use slicing and *under-constrainted* 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. SID determines security impacts based on the code semantics instead of textual information. Based on the semantics, SID 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 SID'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 SID is unable to recognize. Some causes include unusual types of security impacts that are not captured by SID's current security rules and the conservative strategy of under-constrained symbolic execution that may miss some cases. An empirical analysis of SID's false-negative results for known vulnerabilities appears in §VI-B. Further development to reduce false negatives would expand the benefits of SID, 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 SID 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, SID 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 SID 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 SID to determine the security impacts of bugs automatically. The core of SID is *symbolic rule comparison* which employs differential and under-constrained symbolic execution to precisely confirm the security impacts that a patch fixes. In addition, SID also provides details about the security impacts to facilitate bug fixing.
- **Finding of security bugs and unpatched vulnerabilities.** With SID, 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 SID in section §IV; the implementation of SID in section §V; the evaluation of SID 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].

```
1  int Bug_Vuln(unsigned int Type)  {
2      char colors[4] = {'r','g','b','-'};
3      if (Type > 3)
4          return -1;
5
6      if (Type == 3)
7          return 0;
8
9      printf("Color Type: %c", colors[Type]);
10     return 0;
11 }
```

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

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

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.

Table I presents the analysis results. The most common causes for security bugs in the Linux kernel are missing or wrong security checks (bound check, permission check, NULL check, etc.), missing initialization, missing or incorrect locks/unlocks, API misuse, and missing nullification. The following results also show the relationship between the root causes of security bugs and their security impacts: (1) missing/wrong bound check typically leads to out-of-bound access; (2) missing initialization often leads to uninitialized use; (3) missing permission check leads to permission bypass; (4) missing NULL check commonly leads to NULL-pointer dereference; and (5) missing pointer nullification leads to use-after-free and double-free.

In the study, we differentiate the bugs (i.e., the root causes) from their security impacts, which are often mixed up in traditional vulnerability classification (e.g., in NVD). While traditional vulnerability classification tends to focus on security impacts, they are not the root causes. For example, missing a bound check is the bug; however, the out-of-bound access caused by the missing-check bug is the security impact. As such, bug patches typically fix the root causes and only indirectly prevent security impacts. Therefore, to determine security impacts, we need to analyze the "effects" of patches. Moreover, we define security impacts based on the security-rule violating operations (e.g., out-of-bound access and use-after-free) instead of the resulting exploits such as information leaks or control-flow hijacking. This is consistent with the goal of SID—determining how a bug results in security-rule violating operations. How these operations can be exploited for an attack is out of our scope.

C. Patch Model and Components

To determine the security impacts with a given patch, we first need to identify the components in the patch that are related to security impacts and to build a patch model. Based on our empirical analysis of existing patches for vulnerabilities, we identify three key components in determining security impacts. We then create a patch model that incorporates the components, as shown in Table II. In this model, the three components are security operations, critical variables, and vulnerable operations. (The symbol, +, indicates security operations introduced by the patch).

- **Security operations** are used in patches to fix at least one security impact. Table I shows that missing or wrong security operations are the most common root causes for security bugs. From those statistical results, we summarize the common security operations: security checks (e.g., bound checks, permission checks), initialization operations, lock or unlock operations, and pointer nullification.
- **Critical variables** 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.

```
1.+ Security_op(CV, ...)
...
2. Vulnerable_op(CV, ...)
```

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-E 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

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

TABLE III: Common security operations for fixing common security impacts.

implementation of SID can support about 38% of vulnerabilities. However, SID'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 SID 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 SID

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

A. *The Approach of* SID

We now use an example, shown in Figure 3, to illustrate the *symbolic rule comparison* approach of SID. 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, SID analyzes the patched version to *prove that it will never violate a security rule*. In function `iwl_sta_ucode_activate`, SID identifies the security operation in the patch, bound check in line 6. Then, SID extracts the critical variable, `sta_id`, from the security operation. By using data-flow analysis, SID 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 SID performs under-constrained symbolic execution against each slice. SID 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, SID *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, SID 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, SID 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, SID 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, SID 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`, SID 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 SID 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, SID 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 SID to precisely check the match between a patch and the corresponding security

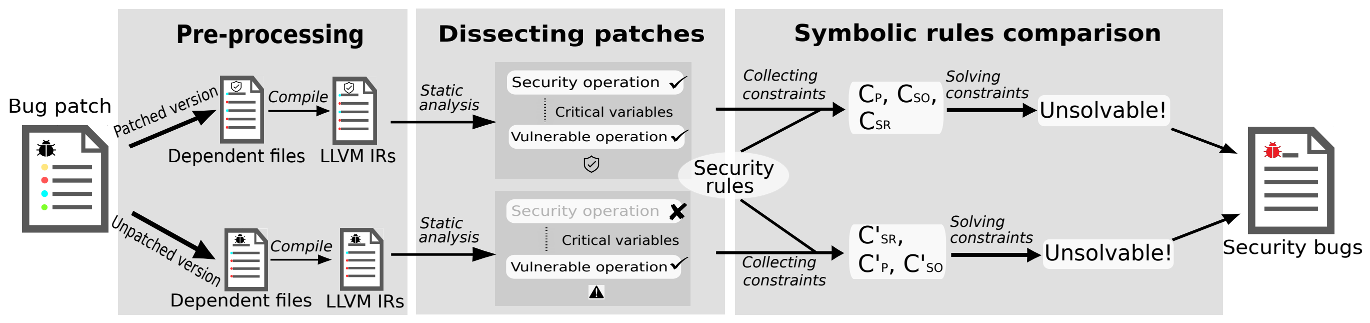


Fig. 2: Overview of SID. C_{SO} = Constraints from security operations, C_{SR} = Constraints from security rules, C_P = Constraints from paths.

```
1  /* Linux: drivers/net/wireless/intel/iwlwifi/dvm/sta.c
2   * CVE-2012-6712 */
3
4  int iwl_sta_ucode_activate(... , u8 sta_id)
5  {
6  +   if (sta_id >= IWLAGN_STATION_COUNT) {
7  +       IWL_ERR(priv, "invalid sta_id %u", sta_id);
8  +       return -EINVAL;
9  +   }
10
11     if (!(priv->stations[sta_id].used ))
12         IWL_ERR(priv,"Error active station id %u "
13             "addr %pM\n",
14             sta_id, priv->stations[sta_id].sta.sta.addr);
15
16     if (priv->stations[sta_id].used) {
17         IWL_DEBUG_ASSOC(priv,
18                 "STA id %u addr %pM already present in uCode"
19                 sta_id, priv->stations[sta_id].sta.sta.addr);
20     } else {
21         priv->stations[sta_id].used |= IWL_STA_UCODE_ACTIVE;
22         IWL_DEBUG_ASSOC(priv, "Added STA id %u addr %pM\n",
23                 sta_id, priv->stations[sta_id].sta.sta.addr);
24     }
25
26     return 0;
27 }
```

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 (§II-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 §III-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 (§II-C), we summarize the patch patterns for each class of vulnerability in Table IV.

Out-of-bound access	**Permission bypass**
`1.+ Security_ck(Bound);`	`1.+ ret = Perm_func(CV, ...);`
...	`2.+ Security_check(ret);`
...	...
`2. Vulnerable_op(Bound, ...);`	`3. Vulnerable_op(CV, ...);`
Use-after-free or double-free	**Uninitialized use**
`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 §II-D, 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 vulnerable 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.

Security operations	**Constraints from security operations**	
	Patched version	**Unpatched version**
Pointer nullification	$\texttt{FLAG}_{CV}$ = 1	$\texttt{FLAG}_{CV}$ = 0
Initialization	$\texttt{FLAG}_{CV}$ = 1	$\texttt{FLAG}_{CV}$ = 0
Permission check	$\texttt{FLAG}_{CV}$ = 1	$\texttt{FLAG}_{CV}$ = 0
Bound check	CV < UpBound, or CV > LowBound	CV $\geqslant$ UpBound, resp. CV $\leqslant$ LowBound

TABLE V: Constraints for security operations from patches. $\texttt{Flag}_{CV}$: 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, $\texttt{FLAG}_{CV}$ indicates the status of the corresponding critical variables, in terms of the presence of the security operations. $\texttt{FLAG}_{CV}$ = 1 means that the security operation has been enforced against critical variable `CV`. By contrast, $\texttt{FLAG}_{CV}$ = 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 $\texttt{FLAG}_{CV}$ `= 1` for the security operations in the patched version while inserting $\texttt{FLAG}_{CV}$ `= 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, 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 unpatched 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 §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, $\texttt{FLAG}_{CV}$ `= 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., $\texttt{Flag}_{CV}$ `= 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, from the security rule of in-bound access, SID inserts the constraint, `sta_id < 16`, for the unpatched version.

Security rules	**Patched version**	**Unpatched version**
No use after free	$\texttt{FLAG}_{CV}$ `= 0`	$\texttt{FLAG}_{CV}$ `= 1`
Use after initialization	$\texttt{FLAG}_{CV}$ `= 0`	$\texttt{FLAG}_{CV}$ `= 1`
Permission check before sensitive operations	$\texttt{FLAG}_{CV}$ `= 0`	$\texttt{FLAG}_{CV}$ `= 1`
In-bound access	`CV` $\geqslant$ `MAX`, and/or `CV` $\leqslant$ `MIN`	`CV < MAX`, resp. `CV > MIN`

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 rules: 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, documentation, and indentation updating. In total, 17.4% of git commits are classified as non-bug commits and thus eliminated, which improves the efficiency of SID. For the remaining patches, SID continues to generate LLVM IR and dissect them.

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 §IV, 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 §V-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 §IV-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 §II-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 unpathed 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, SID'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 SID. Different from the whole-program symbolic execution, SID 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, SID 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 SID, 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, SID 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 SID is efficient enough to handle a massive amount of patches precisely.

A. *False Positives of* SID

We use precision, `|TP| / |TP+FP|`, to evaluate the false positives of SID, 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 SID. 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 SID 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 SID 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 SID is 97%. We investigated these false positives and summarized the reasons as follows.

Missing constraints in preventive-patching cases. SID 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, SID 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 SID's threat model—the provided patches correctly fix actual bugs.

Inaccurate static analysis. During dissecting patches, SID 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 SID 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 §VII.

B. *False Negatives of* SID

By design, SID aims to ensure less false positives by allowing more false negatives. In this section, we evaluate the false negatives of SID 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 SID to detect known vulnerabilities in the Linux kernel to evaluate how many of them are missed by SID. 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 SID to analyze the corresponding patches. It turns out that SID found 47 vulnerabilities out of the 100. Therefore, SID 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. SID 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 execution 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 (see §V-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

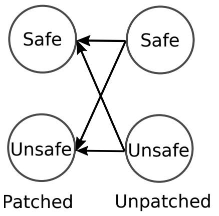


Fig. 4: The safety-state transition diagram from unpatched version to patched version.

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 SID 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 SID 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 SID 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.

Bug types	**Partial fix on security operations**
Out-of-bound access	Incomplete bound check
Use-after-free/double-free	N/A
Uninitialized use	Incomplete initialize
Permission bypass	Incomplete perm check

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 SID. 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 SID 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 SID, 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 SID, 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 [1] (5 cases); (3) patch commits do not mention security impacts (4 cases). For the first reason, we believe that most code will

[1]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 SID 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.

Entry points	**Num of reachable bugs**
Dynamically confirmed bugs (fuzzers)	28
System calls	133
I/O control handlers	148
Interrupt handlers	131
Total	154 (67.8%)

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

E. Generality of SID*'s Patch Model*

Num of key components	**Percent**
Three components	77%
Two or one component	11%
Other cases	12%

TABLE IX: The generality of SID's patch model. It shows the numbers of components the vulnerabilities have.

SID'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, SID 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 SID'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.

SID'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 SID, 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 SID 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 SID. 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 SID 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.

Type	Average (Days)	Median (Days)	Maximum (Days)
General patches	15.8	3	1240
Patches of security bugs	8.6	2	111
Patches of vulnerabilities	5.8	1	974

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 SID and assigned with a CVE (CVE-2007-6762) after we reported it.

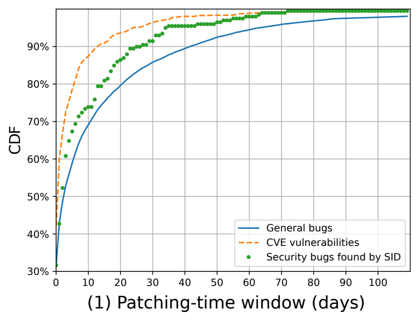
Security bugs threatening derivative software. Many programs are derived from other open-sourced programs. For example, the Android kernel is a modified version of the Linux kernel, and the Android kernel is further customized into thousands of versions [52] running on tens of thousand device models [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.

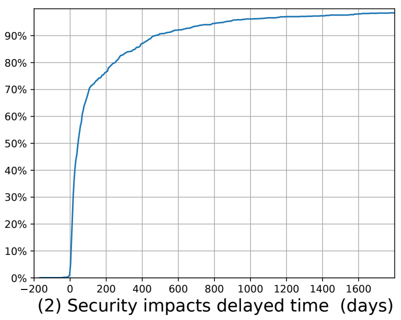
The first evaluation is to check how many security bugs (identified by SID) remain unpatched in Android. Specifically, we manually checked the security bugs found by SID in the latest (as of the experiment) Android kernel release version, Android-4.14-p-release, which was ported from the Linux kernel 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 SID 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, thus 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, SID 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





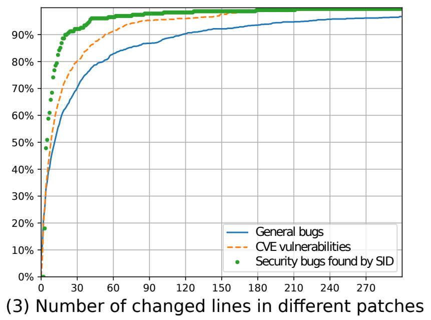


Fig. 5: Statistical findings. CDF: cumulative distribution function; (1) CDF for time windows from bug report date to bug fix date; (2) CDF for time window from the patch date to the CVE release date; (3) CDF for the number of changed lines for different kind of bugs. In (2), about 1% of vulnerabilities are assigned with a CVE before the bugs are actually patched.

```
/* CVE-2019-15926, CVSS 9.4
 * drivers/net/wireless/ath/ath6kl/wmi.c */
static int ath6kl_wmi_pstream_timeout_event_rx(
                struct wmi *wmi, u8 *datap,
                int len) {
        ...
        ev = (struct wmi_pstream_timeout_event *) datap;
+       if (ev->traffic_class >= WMM_NUM_AC) {
+           ath6kl_err("invalid traffic class: %d\n",
+                   ev->traffic_class);
+           return -EINVAL;
+       }

        ...
        wmi->stream_exist_for_ac[ev->traffic_class] = 0;
        ...
}
```

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

Security impacts	#	Security impacts	#
Uninitialized use	85	Use-after-free/Double-free	67
Out-of-bound access	65	Permission bypass	13
NULL-pointer dereference	13		

TABLE XI: Security impacts for the identified security bugs

found by SID, and 1,339 CVE-assigned vulnerabilities. The statistical results are shown in Figure 5. We can find that security bugs and vulnerabilities are highly similar in terms of the number of changed lines in their patches—both have a smaller number then general bugs patches have. The finding is consistent with the results found by Li et al. [27]. Also, the patches for vulnerabilities and security bugs tend to change fewer files than patches for general bugs. Patches for general bugs changed 3.0 files on average, which in contrast to 2.3 files for patches of vulnerabilities and security bugs found by SID.

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

Security bugs (by root cause)	**Security impact**	**Security operation**	**Vulnerable operation**	**Critical Variable**	**SR in PV or SO in UPV**	**SR in UPV or SO in PV**
Missing release	Memory leak	Release operation	Allocation operation	Allocated pointer	$\texttt{FLAG}_{CV}$ = 0	$\texttt{FLAG}_{CV}$ = 1
Missing NULL check	NULL dereference	NULL check	Pointer dereference	Checked pointer	$\texttt{FLAG}_{CV}$ = 0	$\texttt{FLAG}_{CV}$ = 1
Missing zero check	Divide by zero	Zero check	Use as divisor	Divisor	$\texttt{FLAG}_{CV}$ = 0	$\texttt{FLAG}_{CV}$ = 1
Missing/wrong locks/unlocks	Race condition (UAF/DF and etc.)	Lock/unlock	Operations in critical section	Lock variable	$\texttt{FLAG}_{CV}$ = 0	$\texttt{FLAG}_{CV}$ = 1

TABLE XII: The key components of patches and constraints modeling for more types of bugs. SO = security operations; SR = security rules; PV = patched version; UPV = unpatched version.

constraints.

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. Wijayasekara 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, 15, 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. Tyo [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, SID can precisely determine the security impacts of a patched bug and provide details on the vulnerable operations even when the commit description is not available. Moreover, SID 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. [30] showed how to exploit uninitialized-use bugs using symbolic execution and fuzzing in the Linux kernel. Xu et al. [58] presented a memory collision 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. [4] proposed AEG, a symbolic execution–based automatic exploit generation tool that can automatically exploit memory-corruption bugs such as buffer overflow. Wu et al. [55] presented FUZE, which can automatically exploit use-after-free vulnerabilities in the Linux kernel. Unlike these studies, SID does not focus on the exploitability of a specific class of vulnerability. Instead, SID aims to automatically determine the security impacts of a bug once it is triggered. Moreover, SID 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, Munaiah et al. [37] 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, SID 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

[1] Linux syscall reference, 2019. https://syscalls.kernelgrok.com/.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange, ETX*, pages 35–39, Oct. 2005.

[3] J. Arnold, T. Abbott, W. Daher, G. Price, N. Elhage, G. Thomas, and A. Kaseorg. Security impact ratings considered harmful. In *Proceedings of HotOS'09: 12th Workshop on Hot Topics in Operating Systems*, May 2009. https://www.usenix.org/conference/hotos-xii/security-impact-ratings-considered-harmful.

[4] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[5] D. Behl, S. Handa, and A. Arora. A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf. In *Optimization, Reliabilty, and Information Technology (ICROIT), 2014 International Conference on*, pages 294–299. IEEE, 2014.

[6] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[8] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 396–407. ACM, 2017.

[9] K. Chaturvedi and V. Singh. Determining bug severity using machine learning techniques. In *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*, pages 1–6. IEEE, 2012.

[10] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2017.

[11] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. Fuzzer for linux kernel drivers, 2019. https://github.com/ucsb-seclab/difuze.

[12] M. Corporation. Common weakness enumeration (cwe), 2019. https://cwe.mitre.org/data/definitions/1000.html.

[13] M. Corporation. Common vulnerabilities and exposures, 2019. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=linux+kernel.

[14] S. Counte. Mobile operating system market share worldwide, 2019. http://gs.statcounter.com/os-market-share/mobile/worldwide.

[15] D. C. Das and M. R. Rahman. Security and performance bug reports identification with class-imbalance sampling and feature selection. In *2018 Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*, pages 316–321. IEEE, 2018.

[16] E. W. Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[17] D. R. Engler and D. Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–4, July 2007.

[18] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 179–194. IEEE, 2016.

[19] M. Gegick, P. Rotella, and T. Xie. Identifying security bug reports via text mining: An industrial case study. *The 7th IEEE Working Conference on Mining Software Repositories (MSR'10)*, 2010.

[20] A. Y. Gerasimov, L. V. Kruglov, M. Ermakov, and S. P. Vartanov. An approach to reachability determination for static analysis defects with the help of dynamic symbolic execution. *Programming and Computer Software*, 44(6):467–475, 2018.

[21] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for c programs. In *ACM SIGPLAN Notices*, volume 36, pages 47–58. ACM, 2001.

[22] Google. Android security rewards program rules, 2019. https://www.google.com/about/appsecurity/android-rewards/.

[23] K. Goseva-Popstojanova and J. Tyo. Identification of security related bug reports via text mining using supervised and unsupervised classification. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 344–355. IEEE, 2018.

[24] C. Hall. Survey shows linux the top operating system for internet of things devices, 2018. https://www.itprotoday.com/iot/survey-shows-linux-top-operating-system-internet-things-devices.

[25] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 34–43. ACM, 2007.

[26] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee. From zygote to morula: Fortifying weakened aslr on android. In *2014 IEEE Symposium on Security and Privacy*, pages 424–439. IEEE, 2014.

[27] F. Li and V. Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215. ACM, 2017.

[28] K. Lu and H. Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[29] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 280–291. ACM, 2015.

[30] K. Lu, M.-T. Walter, D. Pfaff, S. Nümberger, W. Lee, and M. Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *NDSS*, 2017.

[31] K. Lu, A. Pakki, and Q. Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1769–1786, Santa Clara, CA, Aug. 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/lu.

[32] A. Machiry, N. Redini, E. Cammellini, C. Kruegel, and G. Vigna. Spider: Enabling fast patch propagation in related software repositories. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[33] P. Mell, K. Scarfone, and S. Romanosky. Common vulnerability scoring system. *IEEE Security & Privacy*, 4(6):85–89, 2006.

[34] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *2008 IEEE International Conference on Software Maintenance*, pages 346–355. IEEE, 2008.

[35] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[36] Mozilla. Bugzilla main page, 2019. https://bugzilla.mozilla.org/home.

[37] N. Munaiah and A. Meneely. Vulnerability severity scoring and bounties: why the disconnect? In *Proceedings of the 2nd International Workshop on Software Analytics*, pages 8–14. ACM, 2016.

[38] nluedtke. linux_kernel_cves, 2019. https://github.com/nluedtke/linux_kernel_cves.

[39] M. Ohira, Y. Kashiwa, Y. Yamatani, H. Yoshiyuki, Y. Maeda, N. Limsettho, K. Fujino, H. Hata, A. Ihara, and K. Matsumoto. A dataset of high impact bugs: Manually-classified issue reports. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 518–521. IEEE, 2015.

[40] W. Y. Ramay, Q. Umer, X. C. Yin, C. Zhu, and I. Illahi. Deep neural network-based severity prediction of bug reports. *IEEE Access*, 7:46846–46857, 2019.

[41] D. A. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, 2015.

[42] M. J. Renzelmann, A. Kadav, and M. M. Swift. SymDrive: Testing drivers without devices. In *10th USENIX Symposium on Operating Systems Design and Implementation, (OSDI)*, pages 279–292, Oct. 2012.

[43] N. K. S. Roy and B. Rossi. Towards an improvement of bug severity classification. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 269–276. IEEE, 2014.

[44] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[45] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*, 2019.

[46] Thgarnie. Syzkaller, 2019. https://github.com/google/syzkaller.

[47] Y. Tian, N. Ali, D. Lo, and A. E. Hassan. On the unreliability of bug severity data. *Empirical Software Engineering*, 21(6):2298–2323, 2016.

[48] L. Torvalds. Linux kernel 4.14, 2017. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tag/?h=v4.14.

[49] L. Torvalds. Linux kernel source tree, 2019. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/.

[50] J. P. Tyo. Empirical analysis and automated classification of security bug reports. 2016.

[51] W. Wang, K. Lu, and P.-C. Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1899–1913. ACM, 2018.

[52] L. Wei, Y. Liu, and S.-C. Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 226–237, 2016. ISBN 978-1-4503-3845-5.

[53] D. Wijayasekara, M. Manic, J. L. Wright, and M. McQueen. Mining bug databases for unidentified software vulnerabilities. In *2012 5th International Conference on Human System Interactions*, pages 89–96. IEEE, 2012.

[54] D. Wijayasekara, M. Manic, and M. McQueen. Vulnerability identification and classification via text mining bug databases. In *IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society*, pages 3612–3618. IEEE, 2014.

[55] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou. {FUZE}: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 781–797, 2018.

[56] T. Xie, N. Tillmann, J. De Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 359–368. IEEE, 2009.

[57] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[58] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 414–425. ACM, 2015.

[59] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019.

[60] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. R. Engler. Automatically generating malicious disks using symbolic execution. In *2006 IEEE Symposium on Security and Privacy (S&P 2006)*, pages 243–257, May 2006.

[61] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154. ACM, 2017.

[62] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang. Pex: a permission check analysis framework for linux kernel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1205–1220, 2019.

[63] Z. Zhang, Y. Wang, and Z. Fan. Similarity analysis between scale model and prototype of large vibrating screen. *Shock and Vibration*, 2015, 2015.

[64] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 409–423, 2014. ISBN 978-1-4799-4686-0.

[65] Y. Zhou and A. Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 914–919. ACM, 2017.

Appendix

Git commit	BT	SI	ST	CVSS	ET
bf703c3f19934	MBC	OBA	N		
74415a36767d9	MBC	OBA	N		
9ed87fd34c97a	MBC	OBA	N		SII
494264379d186	MBC	OBA	N		SII
3d0ccd021b23c	MBC	OBA	N		
ec3cbb9ce241d	MBC	OBA	R		
82033bc52abeb	MBC	OBA	N		SII
b56fa1ed09615	MBC	OBA	N		SII
62c8ba7c58e41	MBC	OBA	N		IO
5d60122b7e30f	MBC	OBA	R		SII
a9ae4692eda4b	MBC	OBA	R		SII
b9f62ffe05e40	MBC	OBA	R		SII
d3c2155ce5889	MBC	OBA	R		SII
12f4543f5d681	MBC	OBA	R		SII
9a07826f99034	MBC	OBA	R		SII
e1718d97aa88e	MBC	OBA	R		SII/F
49521b13cbc02	MBC	OBA	R		SII
ef6ff8f47263b	MBC	OBA	R		SII
f716abd55d1e1	MBC	OBA	R		SII
8986a11978373	MBC	OBA	R		SII
f658f17b5e0e3	MBC	OBA	R		SII
952e5daa2565f	MBC	OBA	U/R/D		
8513027a73c2f	MBC	OBA	N		
3a63e44420932	MBC	OBA	R/D		SII
b4810773754fe	MBC	OBA	R/D		
ac57245215696	MBC	OBA	R/D		SII
6e893ca25e9ea	MBC	OBA	R/D		SII
1e7eb89ba936f	MBC	OBA	N		SII
5270041d342de	MBC	OBA	N		SII
32fb5f06dbb6c	MBC	OBA	N		SII
42d8644bd77dd	MBC	OBA, NPD	R		SII
a0a74e45057cc	MBC	OBA, NPD	N		SII
8dd2c9e3128a5	MBC	OBA, NPD	N		
518ff04fd8429	MBC	OBA, NPD	N		SII
bb1553c800227	MBC	OBA, NPD	N		SII
55e8dba1acc2e	MBC	OBA, NPD	R		
60ad768933ec1	MBC	OBA, NPD	N		SII
0b1d250afb8eb	MBC	OBA, NPD	N		SII
eef08e5350618	MBC	OBA, NPD	N		SII
221be106d75c1	MBC	OBA, NPD	R/U		SII
ef4b4856593fc	MBC	OBA, NPD	R		SII
2a2f11c227bdf	MBC	OBA	R	7.5	SII
0031c41be5c52	MBC	OBA	R	7.5	SII
0926f91083f34	MBC	OBA	R	7.5	SII
12f09ccb46127	MBC	OBA	R	7.5	
2da424b0773ce	MBC	OBA	R	7.5	SII/F
43622021d2e2b	MBC	OBA	N	6.2	SII/F
78214e81a1bf4	MBC	OBA	N	4.7	SII/F
41df7f6d43723	MBC	OBA	N	4.7	F
0fb6bd06e0679	MBC	OBA	N	4.7	SII/F
297502abb32e2	MBC	OBA NPD	N	5.4	SII/F
1fa2337a315a2	MBC	OBA	R	7.5	SII
9d47964bfd471	MBC	OBA	R	4.6	
193e87143c290	MBC	OBA	R	7.5	SII
780e982905bef	MBC	OBA	R	4.6	
b550a32e60a49	MBC	OBA NPD	N	7.8	
dad5ab0db8dea	MBC	OBA NPD	N	7.2	SII
db7683d7deb25	MN	UAF	R		
6d6340672ba3a	MN	UAF	R		SII/F
25524288631fc	MN	UAF	R		
e9777ad4399c2	MN	UAF	R/U/D		
11e40f5c57762	MN	UAF	R/U		SII/F
ae6ccb0f8153f	MN	UAF	R		SII
c278c253f3d99	MN	UAF	R	4.4	SII/F
36e4ad0316c01	MN	UAF	R	6.1	SII
c9fbd7bbc23db	MN	UAF	R/U	4.6	SII
54648cf1ec2d7	MN	UAF	R/U	4.6	

Git commit	BT	SI	ST	CVSS	ET
f4351a199cc12	MBC	OBA	R/U	7.2	SII
d7ac3c6ef5d8c	MBC	OBA	R/U	4.7	SII
04f25edb48c44	MBC	OBA	R/U	7.2	SII
5d6751eaff672	MBC	OBA	R/U	9.4	SII
de591dacf3034	MI	UU	N		SII
325fb5b4d2603	MI	UU	N		SII/F
2fc2111c27294	MI	UU	R/D		IO
ed77ed6112f2d	MI	UU	R/D		
ce384d91cd7a4	MI	UU	R/D		IO
1a8b7a67224eb	MI	UU	R/D		SII
b5f15ac4f89f8	MI	UU	R/D		SII
cccbe5ef85284	MI	UU	R/D		SII
eca67aaeebd6e	MI	UU	R/D		IO
a0c5a3944ce12	MI	UU	R/D		
5b919f833d9d6	MI	UU	R/D		
938abd8449c27	MI	UU	R/D		
144ce879b057c	MI	UU	N		
99b0d365e5ade	MI	UU	N		SII
9561f7faa45cb	MI	UU	R/D		IO
21dba24481f70	MI	UU	R/D		SII
e7332691de2f9	MI	UU	R/D		
02745f63443c0	MI	UU	R/D		
47966e9779528	MI	UU	R/D		IO
5ffedc6ed3d06	MI	UU	R/D		
5dbd5068430b8	MI	UU	R		
282c4c0ecce9b	MI	UU	R/D		IO
dc43376c26cef	MI	UU	R/D		
b08e1ed9cfcf7	MI	UU	R/D		SII
b09c74ae1263e	MI	UU	R/D		SII
6ea437a3639b1	MI	UU	R/D		SII
d14df339c72b6	MI	UU	R/D		
bffbbc0a2ccb9	MI	UU	R/D		IO
3ca9e5d36afb5	MI	UU	R/D		IO
ee7ff5fed2571	MI	UU	R/D		SII
c9889803e3ba6	MI	UU	R/D		SII
7e8631e8b9d4e	MI	UU	R/D		SII
a5f6fc28d6e6c	MI	UU	R/D		
81907478c4311	MI	UU	R/D		SII
a44b0f5edfc63	MI	UU	R/D		
5899f0478528b	MI	UU	R/D		SII
12b055662ac62	MI	UU	R/D		SII
79b568b9d0c7c	MI	UU	R/D		SII
aee177ac5a422	MI	UU	R/D		SII
62d494ca27735	MI	UU	R/D		SII
ab73ef46398e2	MI	UU	R/D		SII
0b857b44b5e44	MI	UU	N		SII
9cd70e80f7f0d	MI	UU	R/D		
7307616245bab	MI	UU	N		
02a9079c66341	MI	UU	R/D		
d69bb92e402ff	MI	UU	R/D		IO
0f4bbb233743b	MI	UU	R/D		IO
5b0907407e7f2	MI	UU	R/D		SII
e15882b6c6caf	MI	UU	R/D		SII/F
72ccc471e13b8	MI	UU	R/D		SII/F
b51456a6096eb	MI	UU	R/D		
f7a6cb7b38c68	MI	UU	R/D		
6ce14f6416c84	MI	UU	R/D		SII/F
4c5009c5256d0	MI	UU	R/D		SII
df7e40425813c	MI	UU	R/U		
d0ea2b1250054	MI	UU	R/U		SII/F
2d93913e22013	MI	UU	R/D		
5540fbf438458	MI	UU	R/U		
40f7090bb1b4e	MI	UU	R/U		
81114baa835b5	MI	UU	R/U		SII
58796e67d5d52	MI	UU	R/U		
6d084ac27ab4b	MI	UU	R/U		SII
e4818d615b58f	MI	UU	N		SII
e55449e71aade	MI	UU	N		F
84ce4d0f9f55b	MI	UU	N		SII
354d0fab649d4	MI	UU	R/U		SII
982f7c2b2e6a2	MI	UU	N	1.9	SC
abd39c6ded9db	MN	UAF	R/U	4.9	SII
4397f04575c44	MN	DF	R/U	7.2	SII
d024206133ce2	MPC	PE	R		
d2c2b11cfa134	MPC	PE	R		
ed82571b1a14a	MPC	PE	R		
f2b20f6ee8423	MPC	PE	R/D		
e3a2b93dddad3	MPC	PE	R		SII
03ce7b1d23498	MPC	PE	R/U		SII
41bdc78544b8a	MPC	PE	N	2.1	SII
c0ca3d70e8d3c	MPC	PE	N	4.9	

Git commit	BT	SI	ST	CVSS	ET
3af54c9bd9e6f	MI	UU	N	1.9	SC
97e69aa62f8b5	MI	UU	N	1.9	IO
c4c896e1471ae	MI	UU	N	1.9	
8d03e971cf403	MI	UU	N	1.9	
792039c73cf17	MI	UU	N	1.9	
9344a972961d1	MI	UU	N	1.9	
e862f1a9b7df4	MI	UU	N	1.9	
1f86840f89771	MI	UU	N	1.9	SII
c88e739b1fad6	MI	UU	N	4.3	
96b340406724d	MI	UU	N	1.7	
8e3fbf870481e	MI	UU	N	1.9	
b6878d9e03043	MI	UU	N	2.1	IO
eda98796aff0d	MI	UU	N	1.9	
681fef8380eb8	MI	UU	N	2.1	IO
342ffc26693b5	MI	UU	R	2.1	IO
0625b4ba1a5d4	MI	UU	R/U	2.1	SII
3b7d2b319db0b	MN	DF	N		SII
1cfafab965198	MN	DF	R		SII
266e8ae37daa0	MN	DF	R		
f5c4441cd8012	MN	DF	R		SII
1c963bec3534b	MN	DF	N		SII
1eb8f7a7da6d3	MN	DF	R		SII
1adb2e2b5f850	MN	DF	R		
f69ae770e74df	MN	DF	N		SII
eb1716af88737	MN	DF	N		
c654ecbbfefbe	MN	DF	R		SII
6dd93e9e5eb19	MN	DF	R		SII
1e7bac1ef754b	MN	DF	R		SII
6cae6d3189ef3	MN	DF	R		
b1214e4757b7d	MN	DF	R		SII
32b8544296b94	MN	DF	R		F
7d78874273463	MN	DF	R		SII
c1b03ab5e8867	MN	DF	R		SII
d7426c69a1942	MN	DF	R		SII/F
b3b51417d0af6	MN	DF	R		
f683c80ca68e0	MN	DF	R		SII/F
7dc4a6b5ca942	MN	DF	R		SII
23418dc131464	MN	DF	R/U/D		
7fafcfdf6377b	MN	DF	R		SII
ef2a7cf1d8831	MN	DF	R/U		SII/F
dc035d4e934e5	MN	DF	R		SII
4d45e21867bee	MN	UAF	N		SII
f3429545d03a5	MN	UAF	N		
e04ca626baee6	MN	UAF	N		
e3e14de50dff8	MN	UAF	N		
3fc98b1ac0366	MN	UAF	N		SII
8f68ed9728193	MN	UAF	N		
6cf9e995f91e5	MN	UAF	N		SII
ba54238552625	MN	UAF	R		SII/F
715252d419129	MN	UAF	N		SII
ece1d77ed73b3	MN	UAF	N		SII
f276795627045	MN	UAF	R		SII
0d012b9866249	MN	UAF	R		
e8243f32f2550	MN	UAF	R		SII
fc09149df6e20	MN	UAF	R		
29322d0db98e5	MN	UAF	R		
400ffaa2acd72	MN	UAF	R		SII
44aa91ab2bb86	MN	UAF	R		SII/F
a723bab3d7529	MN	UAF	R		SII
c7de572630762	MN	UAF	R		SII
fa6114d4bde70	MN	UAF	R		SII
a7a7aeefbca29	MN	UAF	R		IO/F
c5540a0195ec6	MN	UAF	R		
8667f515952fe	MN	UAF	R		SII
741b8b832a574	MN	UAF	R		
031e5896dfdc2	MN	UAF	R		
3587cb87cc44c	MN	UAF	R		
87fc030231b11	MN	UAF	R		SII
5bfd37b4de5c9	MN	UAF	R		SII/F
6818caa4cdc95	MN	UAF	N		SII/F
073931017b49d	MPC	PE	N	3.6	SII
497de07d89c14	MPC	PE	N	3.6	F
bfc81a8bc18e3	MBC	OBA	N	7.2	SII/F
0c319d3a144d4	MBC	OBA	R	7.5	SII
89c6efa61f570	MBC	OBA	R	4.6	SII/F
6acb47d1a318e	MBC	OBA	R	4.6	SII
4da62fc70d7cb	I	UU, PE, OBA	R/D		
49c37c0334a9b	I	UU	N	4.9	
fd02db9de73fa	I	UU	N	1.9	IO

TABLE XIII: List of security bugs detected by SID. BT: bug type; SI: security impact; ET: entry point types; MBC: missing/wrong bound check; OBA: out-of-bound access; MI: missing initialization; NPD: NULL-pointer dereference; UU: uninitialized use; MN: missing nullification; MPC: missing permission check; DF: double-free; UAF: use-after-free; PE: permission bypass; U: unpatched bug in Android 4.14-p; R: requested CVE ID; D: CVE request declined by CVE maintainers; N: not request CVE ID; SC: system-call; IRQ: IRQ handlers; IO: I/O control handlers; SII: system-call & I/O control handler & IRQ handlers; F: confirmed dynamically (by Fuzzer); CVSS: common vulnerability scoring system.