# On the Feasibility of Stealthily Introducing Vulnerabilities in Open-Source Software via Hypocrite Commits

Qiushi Wu and Kangjie Lu
*University of Minnesota*
{wu000273, kjlu}@umn.edu

***Abstract*—Open source software (OSS) has thrived since the forming of Open Source Initiative in 1998. A prominent example is the Linux kernel, which has been used by numerous major software vendors and empowering billions of devices. The higher availability and lower costs of OSS boost its adoption, while its openness and flexibility enable quicker innovation. More importantly, the OSS development approach is believed to produce more reliable and higher-quality software since it typically has thousands of independent programmers testing and fixing bugs of the software collaboratively.**

**In this paper, we instead investigate the insecurity of OSS from a critical perspective—the feasibility of stealthily introducing vulnerabilities in OSS via hypocrite commits (i.e., seemingly beneficial commits that in fact introduce other critical issues). The introduced vulnerabilities are critical because they may be stealthily exploited to impact massive devices. We first identify three fundamental reasons that allow hypocrite commits. (1) OSS is open by nature, so anyone from anywhere, including malicious ones, can submit patches. (2) Due to the overwhelming patches and performance issues, it is impractical for maintainers to accept preventive patches for "immature vulnerabilities". (3) OSS like the Linux kernel is extremely complex, so the patch-review process often misses introduced vulnerabilities that involve complicated semantics and contexts. We then systematically study hypocrite commits, including identifying immature vulnerabilities and potential vulnerability-introducing minor patches. We also identify multiple factors that can increase the stealthiness of hypocrite commits and render the patch-review process less effective. As proof of concept, we take the Linux kernel as target OSS and safely demonstrate that it is practical for a malicious committer to introduce use-after-free bugs. Furthermore, we systematically measure and characterize the capabilities and opportunities of a malicious committer. At last, to improve the security of OSS, we propose mitigations against hypocrite commits, such as updating the code of conduct for OSS and developing tools for patch testing and verification.**

## I. Introduction

Open source software (OSS) shares its source code publicly, and allows users to use, modify, and even distribute under an open-sourcing licence. Since the forming of the Open Source Initiative in 1998, OSS has thrived and become quite popular. For example, as of August 2020, GitHub was reported to have over 40 million users and more than 37.6 million public repositories [19] (increased by 10 million from June 2018 [18]). It was also reported that everyone uses OSS [50] while 78% of companies run OSS [60].

OSS is praised for its unique advantages. The availability and low costs of OSS enable its quick and wide adoption. Its openness also encourages contributors; OSS typically has thousands of independent programmers testing and fixing bugs of the software. Such an open and collaborative development not only allows higher flexibility, transparency, and quicker evolution, but is also believed to provide higher reliability and security [21].

A prominent example of OSS is the Linux kernel, which is one of the largest open-source projects—more than 28 million lines of code used by billions of devices. The Linux kernel involves more than 22K contributors. Any person or company can contribute to its development, e.g., submitting a patch through git commits. To make a change of the Linux kernel, one can email the patch file (containing `git diff` information) to the Linux community. Each module is assigned with a few maintainers (the list can be obtained through the script `get_maintainer.pl`). The maintainers then manually or employ tools to check the patch and apply it if it is deemed valid. Other popular OSS, such as FreeBSD, Firefox, and OpenSSL, also adopts a similar patching process.

Because of the wide adoption, OSS like the Linux kernel and OpenSSL has become attractive targets for high-profile attacks [9, 15]. While adversaries are incentivized, it is not always easy to find an exploitable vulnerability. Popular OSS is often extensively tested by developers and users in both static and dynamic ways [63]. Even a bug was found, it may not manifest the exploitability and impacts as the adversaries wish. Thus, finding ideal exploitable vulnerabilities requires not only advanced analyses and significant efforts, but also a bit of luck.

In this paper, we instead investigate the insecurity of OSS from a critical perspective—the feasibility of a malicious committer stealthily introducing vulnerabilities such as use-after-free (UAF) in OSS through hypocrite commits (seemingly beneficial minor commits that actually introduce other critical issues). Such introduced vulnerabilities can be critical, as they can exist in the OSS for a long period and be exploited by the malicious committer to impact a massive number of devices and users. Specifically, we conduct a set of studies to systematically understand and characterize hypocrite commits, followed by our suggestions for mitigation.

We first identify three fundamental reasons that allow the hypocrite commits.

- *OSS openness.* By its nature, OSS typically allows anyone

from anywhere to submit commits to make changes, and OSS communities tend to not validate the committer identity. Consequently, even a malicious committer can submit changes directly or by impersonating reputable contributors or organizations.

- *Limited maintenance resources and performance concerns.* The maintenance of OSS is mainly voluntary and has limited resources. Due to the overwhelming patches, it is impractical for OSS communities to accept preventive patches for "immature vulnerabilities" where not all vulnerability conditions (e.g., a UAF has three conditions: a free, a use, and the use is after the free) are present yet. The Linux community explicitly states that an acceptable patch "must fix a real bug [26]". This is actually understandable. Immature vulnerabilities are not real bugs, and even bug detectors would not report them. On the other hand, applying preventive patches would incur runtime performance overhead.
- *OSS complexity.* OSS can be highly complex. For example, the Linux kernel has 30K modules developed by 22K contributors (according to the Git log), and is full of hard-to-analyze factors such as indirect calls, aliasing pointers, and concurrency. As a result, when the vulnerability conditions are introduced by patches, it can be hard to capture the formed vulnerabilities.

We then systematically study the capabilities of a potential malicious committer. We show how to identify immature vulnerabilities by analyzing common vulnerability conditions and how their absent conditions can be introduced through minor code changes. We also develop multiple tools that automatically identify possible placements for hypocrite commits. Further, we identify multiple factors that can increase the stealthiness of hypocrite commits to evade the review process, including involving concurrency, error paths, indirect calls, aliases, and modules developed by other programmers. Such cases pose significant challenges to even automated analyses. As a proof-of-concept, we safely demonstrated that introducing UAF bugs in the Linux kernel by submitting hypocrite commits is practical. *Note that the experiment was performed in a safe way—we ensure that our patches stay only in email exchanges and will not be merged into the actual code, so it would not hurt any real users (see §VI-A for details).*

To understand and quantify the risks, we further conduct a set of measurements. First, our tools identify a large number of *immature vulnerabilities* and placement opportunities for hypocrite commits. For example, we identified more than 6K immature UAF vulnerabilities in the Linux kernel that can potentially be turned into real vulnerabilities. Then, we qualitatively and quantitatively measure the stealthiness factors. We statistically show that the factors indeed increase the stealthiness. We also define and calculate the *catch rate* for each factor. In particular, we find that involving concurrency has the lowest catch rate and thus is the stealthiest—Linux maintainers catch only about 19% introduced UAF vulnerabilities when the patches involve concurrency.

The intentionally introduced vulnerabilities are critical because (1) they are stealthy and can exist in the OSS for a long time, (2) they can be readily exploited by the malicious committer who knows how to trigger them, and (3) they impact a large number of users and devices due to the OSS popularity. The hypocrite commits may also be abused to get rewards from bug bounty programs [1, 33].

Although OSS communities have known that bug-introducing patches are not uncommon [28, 67], hypocrite commits incur new and more critical risks. To mitigate the risks, we make several suggestions. First, OSS projects would be suggested to update the code of conduct by adding a code like "By submitting the patch, I agree to not intend to introduce bugs." Second, we should call for more research efforts for developing techniques and tools to test and verify patches. Third, if possible, OSS communities could adopt means (e.g., OSS-contribution tracker [56]) to validate committers and pay particular attention to patches sent from unrecognized contributors. Forth, the communities could proactively accept certain preventive patches for high-risk immature vulnerabilities. Last but not least, OSS maintenance is understaffed. We should very much appreciate and honor maintainer efforts, and increase potential incentives if possible to encourage more people to join the OSS communities. We also reported our findings to the Linux community and summarized their feedback.

We make the following research contributions in this paper.

- **A new vulnerability-introducing method.** We discover that a malicious committer can stealthily introduce vulnerabilities in OSS via seemingly valid hypocrite commits. We identify three fundamental causes of the problem. This is critical due to the natures of OSS, and the method is unfortunately practical. We hope that the finding could raise the awareness of that attackers may construct stealthy new vulnerabilities at their will, without having to find existing vulnerabilities.
- **A study of malicious-committer capabilities.** We systematically study the capabilities of a malicious committer. We identify common conditions of vulnerabilities and show how they can be introduced through minor patches, which may be abused to turn immature vulnerabilities into real ones. We also develop tools that automatically identify possibilities of such minor patches.
- **Identification of factors increasing stealthiness.** Through an empirical study of existing maintainer-evading patches, we identify factors that increase the stealthiness of introduced vulnerabilities. Such factors pose significant challenges to the manual review and even automated analysis techniques.
- **Proof-of-concept, measurements, and suggestions.** We safely confirm the practicality of the problem with a proof-of-concept. We also measure the opportunities of a malicious committer and the stealthiness of each factor. We finally provide suggestions to mitigating the risks.

The rest of this paper is organized as follows. We present the overview of the vulnerability-introducing method in §II, the cause study in §III, the details of the method in §IV, the stealthy methods in §V, the proof-of-concept in §VI, measurements in §VII, and suggested mitigations in §VIII. We present related work in §IX, conclusions in §X, and acknowledgment in §XI.

## II. Approach Overview

In this section, we provide the threat model of hypocrite commits, and then introduce the vulnerability-introducing method, new studies and analysis techniques.

### A. Threat Model

The adversaries can be anyone from anywhere. Their goal is to stealthily introduce vulnerabilities in a target OSS program like the Linux kernel, so that they can exploit the vulnerabilities to compromise the program or use them for other purposes like getting rewards from bounty programs. In this model, we have the following assumptions.

- **The OSS.** We assume the OSS management allows the submission of patches from any contributors who may not have submitted any patch before; however, the submitted patches have to go through the patch-review process before being merged. This assumption is valid for most popular OSS we are aware of, such as Google Open Source and the Linux projects. Also, the OSS is relatively complex, which has multiple modules and is developed in unsafe programming languages such as C/C++.
- **Patch-review process of the OSS.** Maintainers are benign and would not intentionally accept bad commits. Further, the OSS has limited computing and human resources. We assume that the review process may employ both manual analysis and automated tools. Our interactions with maintainers for OSS like Linux, FreeBSD, Firefox, and OpenSSL indicate that they may use some tools like Smatch [7], Coccinelle [10] to check and test patches.
- **Adversaries.** We assume that adversaries are not the developers or maintainers, so cannot directly insert functional or large changes into the OSS. They can only submit small changes (e.g., several lines of code) that fix minor issues, such as improving the error handling and coding styles, to introduce vulnerabilities. Adversaries have the source code of the OSS and may use analysis tools to identify minor issues, *immature* vulnerabilities, and placement opportunities for small patches.

### B. Definition of Hypocrite Commits and Minor Patches

We define a hypocrite commit as one that submits a small patch to fix a minor issue, such as improving error handling or fixing trivial bugs, so it seemingly improves the code. However, it actually introduces the remaining conditions of an immature vulnerability, resulting in a *much more critical* issue (e.g., UAF) than the fixed one. The introduced vulnerability can be stealthy and thus be persistently exploited by the malicious committer. Compared with the larger commits which change hundreds of lines-of-code (LOC) and alter the functionality of programs, minor patches involve small changes to fix bugs or minor issues. The statistical results of previous works [35, 64] show that more than 70% of bug-fixing patches change only less than 30 LOC, and as the number of LOC grows, the covered proportion increases much slower. Therefore, in this project, we regard a patch as a minor patch if it changes less than 30 LOC and does not introduce new features or functionalities.

### C. The Vulnerability-Introducing Method

We discover that, under the threat model, an adversary can stealthily introduce vulnerabilities in an OSS program through hypocrite commits. The idea is based on the fact that a vulnerability is formed by multiple conditions that are located in different code pieces in the program, and the program may have many immature vulnerabilities in which some vulnerability conditions (not all) are already present. By introducing the remaining conditions via hypocrite commits, adversaries can successfully form the vulnerability. Because the complexity of the OSS may cover up the introduced conditions and existing conditions, the introduced vulnerability can be stealthy (although the patch code is simple). For example, Figure 1 shows a simple patch that balances the refcount upon an error, which is a common error-handling strategy and seemingly valid because it avoids a potential refcount leak. The inserted `put_device()` gives up the last reference to the device and will trigger the refcount mechanism to *implicitly* free `bus` when the refcount reaches zero [11]. However, for this particular case, a dedicated function `mdiobus_free()` is also called when `mdiobus_register()` fails, which uses `bus` and frees it again, leading to a UAF. This introduced vulnerability is stealthy because the patch looks reasonable, and involves hardly tested error paths and an implicit free. As a result, the vulnerability existed in the Linux kernel for five years.

```
1  /*Introducing: CVE-2019-12819*/
2  int __mdiobus_register(...) {
3      ...
4      err = device_register(&bus->dev);
5      if (err) {
6          pr_err("mii_bus %s failed to register\n",
7                                  bus->id);
8  +       put_device(&bus->dev);
9          return -EINVAL;
10     }
11 }
```

**Fig. 1:** A stealthy use-after-free vulnerability introduced by a patch that seems to fix a refcount bug. Its latent period is five years.

Figure 2 shows an overview of the vulnerability-introducing method. Given the source code of an OSS program, the method first identifies immature vulnerabilities and their corresponding absent conditions. In §IV-A, we will provide a set of conditions for common vulnerabilities such as UAF and uninitialized uses, and show how to identify them. The method then tries to construct a set of minor patches that can introduce the absent conditions. In §IV-B, we will show different kinds of such minor patches. After that, the method analyzes the program to identify stealthy opportunities for placing the hypocrite commits. By empirically studying maintainer-evading patches, we identify multiple factors that increase the stealthiness of hypocrite commits, as will be presented in §V. We also develop analysis tools to identify the stealthy placement opportunities from the code. At last, the adversaries decide the hypocrite commits and submit them to the OSS community with a random account. In addition to the method, we further safely provide a proof-of-concept in the Linux kernel, and conduct a

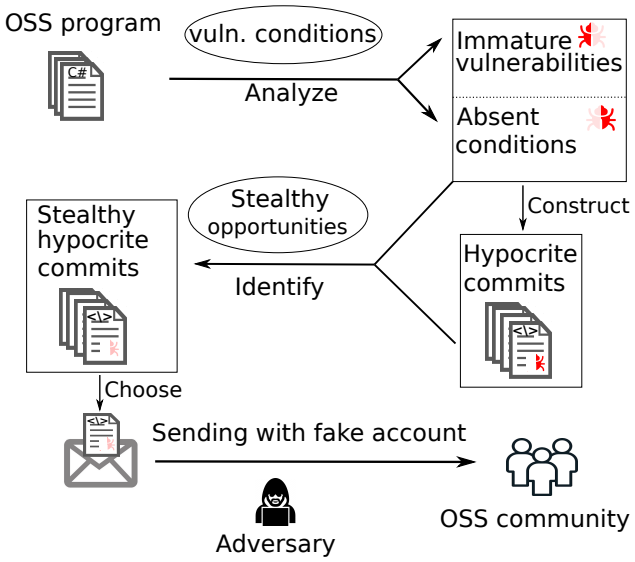


**Fig. 2:** An overview of the vulnerability-introducing method.

set of studies and measurements to understand the criticality, commonness, and practicality of hypocrite commits.

## III. Issues with the OSS Patching Process

We have extensive interaction experience with OSS communities—reporting bugs, submitting patches, and analyzing patches. We found that vulnerability-introducing patches are not uncommon; even experienced maintainers may (unintentionally) submit vulnerability-introducing patches. We believe that when the committers are malicious, the issue would aggravate. To understand the causes of the problem, we first conduct a study on vulnerability-introducing patches that evaded maintainers. Note that in these cases, the committers did not intentionally introduce the vulnerabilities.

**A dataset of vulnerability-introducing patches.** We choose the Linux kernel as an example of the OSS program because it is foundational and widely used—billions of Android and Internet of things (IoT) devices are also based on the Linux kernel. We use script code to analyze all the CVE-assigned vulnerabilities in the Linux kernel whose introducing and fixing commits are available. As defined in §II-B, we regard the patches with less than 30 LOC as minor patches. We found that 9.8% of vulnerabilities were caused by minor patches, which are used to fix non-functional issues of the program, such as memory-leak fixes. In total, we collected 138 CVE-assigned vulnerabilities of different types, which are introduced by minor patches. We also collected their corresponding introducing and fixing patches.

**Immature vulnerabilities.** In general, the forming of vulnerabilities requires multiple conditions. When not all vulnerability conditions are present, the vulnerability is not formed yet. We call such a potential vulnerability as an *immature vulnerability*. Our study (see §VII-A) shows that OSS tends to have a large number of immature vulnerabilities, which poses hidden insecurity to OSS. First, immature vulnerabilities are not real vulnerabilities, so bug detectors (both static and dynamic ones) may not report them at all. Second, as will be shown in §III-A, OSS communities typically refuse to accept preventive patches for immature vulnerabilities. The Linux community explicitly state that they will not accept preventive patches [26]. Third, when the remaining conditions are introduced, due to the complexity of code (see §III-B) and other reasons, the newly introduced vulnerabilities often slip through the patch review, such as the example in Figure 1. In the following, we discuss three factors that allow a malicious committer to turn an immature vulnerability into a real one.

### A. OSS Maintenance Philosophy—No Preventive Patches

Maintenance strategies vary for different open-source projects. These strategies are important as they decide how patches would be accepted or rejected.

**Rejecting preventive patches.** Widely used OSS projects, especially large ones that are implemented in unsafe programming languages, e.g., the Linux kernel and Firefox, are receiving overwhelming bug reports and patches. For example, on average the Linux community receive more than 400 email bug reports [27] per day. However, OSS projects have limited maintainers and resources to review the patches. On the other hand, such projects put especial care on performance, and thus try to make their code as concise as possible. As a result, preventive patches that stop immature vulnerabilities (i.e., potential future vulnerabilities) are not welcome and would likely be rejected.

**Rejecting patches for bugs without PoC.** Some OSS projects may even refuse to accept patches without a PoC (i.e., the exploitability is not confirmed with a test case), although the bug is manually or statically confirmed. For example, although the Android Open Source Project does not explicitly state so, it generally requires a PoC. We reported multiple patches without a PoC for real bugs, and none of them was accepted. We believe such a case is common for OSS projects running bug bounty programs, i.e., the rewards typically require a PoC.

**The Linux patch guidance.** The Linux documentation explicitly lists their rules for accepting patches into stable versions. We summarize the important ones in Table I. In particular, Linux will not accept preventive patches, large patches with more than 100 lines, trivial patches fixing issues like white-space, and patches for theoretical race conditions.

| Rules for patches accepted into "-stable" tree |
|---|
| Must be obviously correct and tested |
| Cannot be bigger than 100 lines |
| Must fix only one thing |
| Must fix a real bug |
| No "theoretical race condition" issues |
| Cannot contain "trivial" fixes (e.g., white-space cleanup) |

**TABLE I:** Common rules for patches to be accepted into the stable tree, provided by the Linux kernel documentation [26].

### B. Complexity and Customization of Code

By analyzing the vulnerability-introducing patches, we summarize general reasons for the review process failing to catch the introduced vulnerabilities.

**Complexity and carelessness.** The OSS code can be highly complicated, so maintainers may fail to understand all of the semantics, and most of these vulnerabilities are caused by complexity such as pointer alias, indirect calls, callback functions, etc. Also, according to our statistical result, 8% of vulnerabilities in the Linux kernel are caused by concurrent issues, which are hard to analyze. On the other hand, we believe that some vulnerabilities (9.4%) are caused by carelessness because they are actually quite obvious to catch.

**Customization and diversity.** Both the committers and maintainers may not clearly understand or be familiar with the usage of customized or special functions. For example, the most well known and broadly used release function in the Linux kernel is `kfree()`. However, other functions such as customized release functions or the refcount mechanism (an object is automatically released when the refcount reaches zero) can also release objects implicitly, and committers and even maintainers may not know that.

### C. Openness of OSS

By its nature, an OSS project typically allows anyone to contribute, which also raises security concerns. Patches from third parties may accidentally or even maliciously introduce bugs and vulnerabilities. Some previous works [4, 6] also show that third-party contributors tend to make more bugs because they have less experience with the OSS program. Although OSS projects might have different policies regarding bug patches and reports, we found that most OSS projects, especially widely used ones, accept bug patches and reports from third parties.

**Accepting bug patches.** Large OSS projects we are aware of accept bug patches from third parties but will enforce the patch review. Prominent examples include the Linux kernel, Android, FreeBSD, Firefox, Chromium, OpenSSL, etc. Communities of such OSS projects typically receive many bug reports; to efficiently fix the reported bugs, they encourage reporters to also submit patches.

**Accepting bug reports only.** Smaller OSS projects may only accept bug reports but not patches from third parties. Examples include `libjpeg` and `readelf`. Maintainers of such projects would typically receive a small number of bug reports, and they can patch the bugs by themselves.

## IV. Introducing Vulnerabilities via Hypocrite Commits

In this section, we present how hypocrite commits can introduce the remaining conditions to turn an immature vulnerability to a real one. We study the same vulnerability set described in §III to understand vulnerability conditions, condition-introducing commits and their placements.

**Characterizing contributors of minor patches.** Before we present vulnerability conditions, we first characterize the contributors of general patches and vulnerability-introducing patches, which we believe is of interest to readers. We analyzed the email domains of the contributors for recent 500 minor patches in the Linux kernel. As shown in Table II, 47.8% of minor patches are from maintainers, 38.0% from for-profit organizations such as CodeAurora, 1% from researchers in academia, 3.8% from non-profit organizations, and 9.4% from individuals. In comparison, we also checked the contributors for minor patches introducing CVE-assigned vulnerabilities. Different from the general minor patches, only 37.3% of vulnerability-introducing patches are from kernel maintainers, and the remaining cases are all from third-party contributors. This result indicates that third-party contributors tend to introduce more vulnerabilities.

| Type of contributors | General patches | Vulnerable patches |
|---|---|---|
| Academia | 1.0% | 1.5% |
| Company | 38.0% | 47.0% |
| Maintainer | 47.8% | 37.3% |
| Personal | 9.4% | 11.1% |
| Organization | 3.8% | 3.0% |

**TABLE II:** A study of contributors.

### A. Vulnerability Conditions

We manually study the collected CVE-assigned vulnerabilities introduced by minor patches to identify common vulnerability conditions. By differentially checking the code before and after the minor patch and referencing to the commit messages, we are able to understand why the vulnerability was formed, i.e., which vulnerability conditions were introduced. For example, a common case is that a minor patch introduces a use of a freed pointer, from which we can conclude that a use is a condition of UAF.

| Vuln. conditions | (%) | Common vulnerability types (state) |
|---|---|---|
| With a state | 36.4% | NULL dereference (nullified)<br>Use-after-free (freed) |
| Without a state | 36.4% | Uninitialized use (initialized)<br>NULL dereference (initialized)<br>Out-of-bound access (bounded)<br>Access-control error (privileged)<br>Integer overflow (bounded) |
| A use | 21.6% | Use-after-free<br>Uninitialized use<br>Access-control error<br>NULL dereference<br>Out-of-bound access |
| A temporal order | 5.7% | Use-after-free<br>NULL dereference |

**TABLE III:** A study of conditions for different types of vulnerabilities introduced by minor patches.

Table III summarizes the results of our study. We find that vulnerability conditions can be generally classified into four categories. The first condition category is an object (variable or code) *with a specific state*. Common cases include the `nullified` state for NULL-pointer dereferences and the `freed` state for UAF vulnerabilities. The second condition category is an object *without a specific state*. In particular, common cases include uninitialized use, NULL-pointer dereference, out-of-bound access, access-control error, and integer overflow. Their absent states are `initialized`, `initialized`, `bounded`, `privileged`, and `bounded`, respectively. The third condition category is *a use* of an object, such as the use of an uninitialized

variable (i.e., uninitialized use). The last condition category is *a specific temporal order*, which describes the execution order of specific operations. For example, UAF occurs only when the use of a pointer is after the free. The commonness of each category is also listed in Table III.

### B. Introducing Vulnerability Conditions

To introduce vulnerabilities, the idea is to introduce the absent conditions of immature vulnerabilities through hypocrite commits. To this end, adversaries need to construct minor patches that fix some minor (less-critical) issues while introducing vulnerability conditions. In this section, we show how each category of the vulnerability conditions listed in Table III can be introduced.

**Introducing a specific state for an object.** A common condition of a vulnerability is for an object to have a specific state, e.g., the `freed` state for UAF. Such states can be introduced by *inserting specific function calls or operations*. For the common cases listed in Table III, an adversary can call resource-release functions against the objects or nullify the pointers. In complex OSS programs, many functions can explicitly or implicitly introduce a freed or nullified state for an object. For example, using the patterns of release functions defined in [8], we find 457 memory-release functions in the Linux kernel, and more than half of them do not even have keywords like `dealloc` or `release` in the names, thus can stealthily introduce the freed state. Also, refcount `put` functions can implicitly cause an object to be freed when the refcount reaches zero, as shown in Figure 1. Introducing the `nullified` state is straightforward. Figure 3 (CVE-2019-15922) shows an example. The patch is seemingly valid because it nullifies `pf->disk->queue` after the pointer is released. However, some functions such as `pf_detect()` and `pf_exit()` are called after this nullification, and they would further dereference this pointer without checking its state, leading to NULL-pointer dereference (i.e., crashing).

```
1  static int pf_detect(void) {
2      ...
3      for (pf = units, unit = 0;
4                   unit < PF_UNITS; pf++, unit++) {
5  +       blk_cleanup_queue(pf->disk->queue);
6  +       pf->disk->queue = NULL;
7  +       blk_mq_free_tag_set(&pf->tag_set);
8          put_disk(pf->disk);
9      }
10 }
```

**Fig. 3:** A minor patch introducing the `nullified` state to form a NULL-pointer dereference vulnerability (CVE-2019-15922).

**Removing a state for a variable.** Another common vulnerability condition is that an object should not have a specific state, e.g., an uninitialized use requires that an object does not have the `initialized` state when being used. We found three methods for introducing such a condition. (1) Removing an operation against an object. An adversary can remove the corresponding operations (e.g., initialization and bound check) to directly remove the states. (2) Invalidating an operation related to a state. For example, inserting the second fetch following a check would invalidate the check and cause double-fetch [65]. (3) Creating a new variable without the required state. An adversary can introduce new variables without the states. Note that introducing a new variable may not necessarily introduce new functionalities if it does not introduce new semantics against the variable, so this is still in the threat model. Figure 4 (CVE-2013-2148) is an example of a minor patch that introduces a new variable (`metadata.reserved`) without an initialized state. Because this uninitialized variable was sent to the userspace through `copy_to_user()`, the patch caused information leak.

```
1  struct fanotify_event_metadata {
2          ...
3  +       __u8 reserved;
4  }
5  static ssize_t copy_event_to_user(...) {
6      ...
7      if (copy_to_user(buf, &metadata, FAN_EVENT_METADATA_LEN))
8          goto out_close_fd;
9  }
```

**Fig. 4:** A minor patch introducing an uninitialized use in the Linux kernel by adding a new variable without an initialized state (CVE-2013-2148).

**Introducing a use of an object.** Introducing a use condition is straightforward and has many choices. It can be realized through a function call or an operation. Common cases include dereferencing a pointer, using a variable in memory access (e.g., indexing), using a variable in checks (`if` statements), used as a parameter of a function call, etc. In particular, we find that the prevalent error-handling paths offer many opportunities for introducing uses because it is typically plausible to call an error-message function that could stealthily use freed objects, such as the hypocrite commit shown in Figure 11.

**Introducing a specific temporal order.** Temporal vulnerabilities such as use-after-free, use-after-nullification, and uninitialized uses require operations to happen in a specific temporal order. We found two methods for introducing the condition. (1) Leveraging the non-determinism of concurrency. The execution order of concurrent functions is non-deterministic and decided by the scheduler, interrupts, etc. For example, if a free of a pointer and a dereference of a pointer are in concurrent code, the order—use after the free—will have a possibility to occur. (2) Removing synchronization operations. By removing the synchronization, such as lock/unlock and refcount inc/dec, the execution order of the code also becomes non-deterministic. Figure 5 shows a patch that removed the "superfluous" `get/put_device()` calls which counts references as well as serves as synchronization. However, these get/put functions are useful in maintaining the lifecycle of device variables; removing them will cause the free of `ir` to happen at an early time before its uses (lines 4-5), which causes UAF.

### C. Placements of Minor Changes

We also find that the acceptance rate of vulnerability-introducing patches is related to where the code changes are placed. Based on our study of existing vulnerability-introducing

```
1  static void lirc_release(struct device *ld) {
2      struct irctl *ir = container_of(ld, struct irctl, dev);
3  -   put_device(ir->dev.parent);
4      if (ir->buf_internal) {
5          lirc_buffer_free(ir->buf);
6      }
7  }
8  int lirc_register_driver(struct lirc_driver *d) {
9  -   get_device(ir->dev.parent);
10 }
```

**Fig. 5:** Removing the get/put functions causes the free of `ir` to occur before its uses in lines 4-5.

commits, we summarize three factors increasing the acceptance rate. (1) Being minor. The changes should be minor and focus on less-critical issues. Large or critical changes would delay the patch review or alert communities to pay extra attention. In particular, the Linux communities explicitly require the changes to have less than 100 lines [26]. (2) Fixing real issues. When the patches indeed fix real issues, although they are minor, maintainers are more willing to accept the patches. The fixes can be improving readability, fixing error handling and memory leak, etc. (3) Being stealthy. The introduced conditions should form the vulnerabilities in a stealthy way. For example, error-handling paths are often complicated and less tested, making the review process less likely to capture the introduced vulnerable conditions. In the next section, we will present in detail factors that improve the stealthiness of hypocrite commits.

In particular, error paths often offer ideal placement opportunities for minor changes. First, it is plausible to print out an error message using a format function in which we can introduce a use of freed objects, and to clean up resources (e.g., freeing objects and decrementing refcounts), which introduces specific states. Second, error paths are hard to test and draw less attention from the review process. In §VI, we will further show how we develop LLVM pass to find such placement opportunities in the Linux kernel.

## V. Increasing the Stealthiness

Hypocrite commits fix minor issues but at the same time introduce more critical vulnerabilities. To induce the maintainers to accept the hypocrite commits, the introduced vulnerabilities should be stealthy. In this section, we present multiple factors that increase the stealthiness of hypocrite commits.

To discover the stealthy factors, we conduct a study on the previously introduced vulnerabilities, which evaded the patch-review process. In this study, we again use the same dataset as we used in §III, which includes 138 CVE-assigned vulnerabilities in the Linux kernel. In particular, given a CVE-assigned vulnerability, we collect the patch introducing it and the patch fixing it. By understanding the patches, we identify the introduced conditions, the original conditions, as well as the corresponding code contexts and features. We then empirically summarize the reasons why maintainers failed to capture the introduced vulnerabilities, which include concurrency, error paths, implicit operations, pointer alias, indirect calls, and other issues that involve complex code semantics. Note that these factors do not guarantee the stealthiness but increase it. In §VII, we will evaluate their stealthiness. In the next sections, we will present the details of each factor.

**Concurrency.** Concurrency is inherently hard to reason about. As shown in many research works [2, 13, 16, 17], concurrency issues are prevalent but hard to detect. On the one hand, it is hard to know which functions or code can be executed concurrently due to the non-determinisms from the scheduling, interrupts, etc. On the other hand, most concurrency issues like data races are considered harmless [16, 17] or even intended [12], and fixing the concurrency issues itself is error-prone and requires significant maintainer efforts [67]. As a result, many bugs stem from concurrency issues, and developers are willing to live with them and tend to not fix them unless they are proven to be harmful.

```
1  static int iowarrior_release(...) {
2      mutex_lock(&dev->mutex);
3      if (!dev->present) {
4          mutex_unlock(&dev->mutex);
5          iowarrior_delete(dev);
6      }
7  }
8  static void iowarrior_disconnect(...) {
9  +   dev->present = 0;
10     mutex_lock(&dev->mutex);
11 -   dev->present = 0;
12     if (dev->opened)
13         ...
14     mutex_unlock(&dev->mutex);
15 }
```

**Fig. 6:** A patch introducing concurrent UAF; CVE-2019-19528 introduced by Linux commit `c468a8aa790e`.

In the scenario of patches related to concurrency, maintainers often fail to precisely understand how threads can actually interleave, thus could not catch introduced vulnerabilities when they involve concurrency. Therefore, involving concurrency increases the stealthiness of hypocrite commits. Figure 6 shows a UAF caused by a patch that introduced concurrency issues. The functions `iowarrior_release()` and `iowarrior_disconnet()` can run concurrently. The temporal order was originally protected by the Mutex lock. However, the patch incorrectly moved the operation `dev->present = 0` out of the protected critical section. Consequently, `iowarrior_releae()` may free `dev` through `iowarrior_delete(dev)`, and `iowarrior_disconnet()` still uses it, leading to UAF.

**Error paths.** We also found that introducing vulnerability conditions in error paths can be a reliable way for increasing the stealthiness. First, error paths are hard to test (thus less tested) and can be complex. Testing the error paths in driver code is particularly hard because it requires the hardware devices and the errors to trigger the execution [24]. Second, people tend to pay less attention to error paths, as they will not be triggered at all during the normal execution. More importantly, calling cleanup functions (e.g., free) or error-printing functions is a common error-handling strategy [37, 49]; such functions often free and use objects, so introducing vulnerability conditions in error paths may not arise suspicions. For example, in our case studies (§VI), we place three minor patches in error paths to (safely) demonstrate that introducing UAF vulnerabilities can be practical and stealthily.

**Implicit operations.** Functions may have implicit effects that introduce vulnerability conditions, and the effects are not explicitly reflected in the function names. Common cases include functions triggering the release of objects and nullification of pointers. We use object release as an example to explain implicit operations. There are two common implicit object-release cases. First, some functions, whose names do not indicate a release, may actually release objects. For example, `jbd2_journal_stop()` (CVE-2015-8961) implicitly releases a passed-in parameter, which is still used afterwards, leading to a UAF. We also found some functions that take a pointer (which is a field of a struct) as a parameter, and use `container_of()` to get the pointer of the parent struct and aggressively release it. Such release operations are out of the user expectation, and thus people may not be aware of them. Second, the refcount mechanism may implicitly trigger the release of an object, which we believe is a very interesting case and has actually caused many UAF vulnerabilities. Once the refcount field reaches zero, the object containing the refcount field will be *automatically* released, and people are often unaware of it. As shown in Figure 1, `put_device()` causes the refcount of `bus` to reach zero early, which triggers the implicit release of `bus`. However, `bus` is still used after the function returns, leading to UAF.

**Pointer aliases.** The pointer aliases are prevalent and have posed significant challenges to static analysis [62] and manual review. Alias analysis remains an open problem, and analysis techniques used in practice are often imprecise and intra-procedural. Therefore, complex pointer aliases, especially inter-procedural ones, can also impede the review process from revealing introduced vulnerabilities. For example, in our case study (Figure 9), we leverage the stealthiness inter-procedural aliases (`pointerA` is an alias of `pointerC` from the caller) to introduce a UAF.

**Indirect calls.** Large programs commonly use indirect calls to improve the scalability and flexibility. Statically identifying the target functions of an indirect call is a hard problem [36, 38, 48]. Existing techniques either completely stop tracking indirect calls [65] or use imprecise matching [48]. In particular, most of the tools used by Linux communities do not support indirect calls. Therefore, when vulnerability conditions span over indirect calls, it is hard to connect them together and to reveal the introduced vulnerabilities. Figure 7 shows an example of introduced double-free vulnerability involving an indirect call. `sgmii->close()` is an indirect call to `emac_sgmii_close()` which internally frees `adpt->phy->irq`. However, `adpt->phy->irq` can be already freed before the call of `emac_shutdown()`, leading to double-free. In this case, during the patch-review process, it is hard to know which exact `close` function is called and if it frees `adpt->phy->irq`.

**Other possible issues.** Beyond the aforementioned stealthy ways, in general, other factors that involve complicated code semantics and contexts can also potentially evade the patch-review process, such as involving modules developed by different programmers, using specialized functions or functions that do not follow coding conventions.

```
1  static void emac_shutdown(struct platform_device *pdev)
2  {
3      struct net_device *netdev = dev_get_drvdata(&pdev->dev);
4      struct emac_adapter *adpt = netdev_priv(netdev);
5  ★   if (netdev->flags & IFF_UP)
6  +       sgmii->close(adpt);
7  }
```

**Fig. 7:** Introduced double-free involving indirect-call (Linux commit `03eb3eb4d4d5`).

## VI. Proof-of-Concept: UAF in Linux

As a proof-of-concept, in this section, we use UAF as an example to show how adversaries can introduce vulnerabilities into the Linux kernel. The Linux kernel is one of the most widely used OSS; it is used by a large number of devices including smartphones and IoT devices. On the other hand, UAF is one of the most severe and common memory-corruption vulnerabilities.

**Experiment overview.** In this experiment, we leverage program-analysis techniques to prepare three minor hypocrite commits that introduce UAF bugs in the Linux kernel. The three cases represent three different kinds of hypocrite commits: (1) a coding-improvement change that simply prints an error message, (2) a patch for fixing a memory-leak bug, and (3) a patch for fixing a refcount bug. We submit the three patches using a random `Gmail` account to the Linux community and seek their feedback—whether the patches look good to them. The experiment is to demonstrate the practicality of hypocrite commits, and it will not introduce or intend to introduce actual UAF or any other bug in the Linux kernel.

### A. Ethical Considerations

**Ensuring the safety of the experiment.** In the experiment, we aim to demonstrate the practicality of stealthily introducing vulnerabilities through hypocrite commits. Our goal is not to introduce vulnerabilities to harm OSS. Therefore, we safely conduct the experiment to make sure that the introduced UAF bugs will not be merged into the actual Linux code. In addition to the minor patches that introduce UAF conditions, we also prepare the correct patches for fixing the minor issues. We send the minor patches to the Linux community through email to seek their feedback. Fortunately, there is a time window between the confirmation of a patch and the merging of the patch. Once a maintainer confirmed our patches, e.g., an email reply indicating "looks good", we immediately notify the maintainers of the introduced UAF and request them to not go ahead to apply the patch. At the same time, we point out the correct fixing of the bug and provide our correct patch. In all the three cases, maintainers explicitly acknowledged and confirmed to not move forward with the incorrect patches. All the UAF-introducing patches stayed only in the email exchanges, without even becoming a Git commit in Linux branches. Therefore, we ensured that none of our introduced UAF bugs was ever merged into any branch of the Linux kernel, and none of the Linux users would be affected.

**Regarding potential human research concerns.** This experiment studies issues with the patching process instead of individual behaviors, and we do not collect any personal information. We send the emails to the Linux community and seek their feedback. The experiment is not to blame any maintainers but to reveal issues in the process. The IRB of University of Minnesota reviewed the procedures of the experiment and determined that this is not human research. We obtained a formal IRB-exempt letter.

The experiment will not collect any personal data, individual behaviors, or personal opinions. It is limited to studying the patching process OSS communities follow, instead of individuals. All of these emails are sent to the communities instead of individuals. We also prevent the linkage to maintainers. In particular, to protect the maintainers from being searched, we use a random email account, and three cases presented in §VI-C are redacted.

Bug-introducing patch is a known problem in the Linux community [28, 67]. We also informed the community that malicious committers could intentionally introduce bugs, and they acknowledged that they knew patches could introduce further bugs, and they will do their best to review them. Before the paper submission, we also reported our findings to the Linux community and obtained their feedback (see §VIII-D).

**Honoring maintainer efforts.** The OSS communities are understaffed, and maintainers are mainly volunteers. We respect OSS volunteers and honor their efforts. Unfortunately, this experiment will take certain time of maintainers in reviewing the patches. To minimize the efforts, (1) we make the minor patches as simple as possible (all of the three patches are less than 5 lines of code changes); (2) we find three real minor issues (i.e., missing an error message, a memory leak, and a refcount bug), and our patches will ultimately contribute to fixing them.

### B. LLVM-Based Tools for Hypocrite Commits

To facilitate the construction of hypocrite commits, we develop multiple LLVM-based tools. First, we write an LLVM pass to identify dereferenced pointers that we can potentially introduce `free` operations before the dereferences. To reduce false positives, the LLVM pass applies two selective rules to report the candidate cases: (1) the memory object may have a potential memory leak or refcount leak, and (2) there are error paths between the allocation and the dereference. In the current implementation, we only consider commonly used functions for allocation, deallocation, and refcount operations. To identify if a path is an error path, we check if it returns a standard error code or calls a standard error handling function (e.g., `pr_err()`) [37]. Our data-flow analysis is flow-, context-, and field-sensitive. As a result, our LLVM pass reports in total 921 candidate cases.

Second, we write another LLVM pass to identify freed (but non-nullified) pointers that we may potentially introduce further uses. To reduce false positives, the pass also applies multiple selective rules: (1) the pointer is inter-procedural (passed in from parameters) and involves aliases, and (2) there are error paths after the `free` but within the liveness scope of the pointer. These cases may allow to stealthily insert uses of the freed pointers in the error paths. Finally, the LLVM pass reports 4,657 cases.

### C. The Selected Immature UAF Vulnerabilities

We then try to select both types of immature UAF vulnerabilities from the results reported by our LLVM passes described in §VI-B. To also demonstrate how we use three different stealthy methods to introduce UAF vulnerability conditions (see §VI-D), we choose to select three immature UAF vulnerabilities. In particular, we manually looked into 100 reported cases for each type and selected three cases that we believe are relatively complicated and would allow for stealthy hypocrite commits. All these experiments including the manual analysis were finished within one week by one researcher. Note that to prevent the cases from being re-identified in the code, we simplify the code, and redact names of functions and variables.

**UAF case 1 (error message).** In this case (Figure 8), `devA` is a pointer that refers to a shared memory object among multiple concurrent functions, and it is released in line 5. However, it may be further dereferenced because it involves concurrency, and it is not nullified after being released. Therefore, we identify it as an ideal immature UAF and will try to introduce a use in one error path of its concurrent functions—using `devA` in error-message printing.

```
1      B_device *devB = get_data_from_devA(devA);
2      ...
3      // devA will be freed below
4  *   kfree(devB);
5  *   release_device(devA);
```

**Fig. 8:** Immature UAF case 1. "*": immature UAF point. The code is simplified and redacted.

**UAF case 2 (memory leak).** In this case (Figure 9), `pointerA` is a pointer allocated in line 1. It is reported by Syzkaller+KMemleak as a leaked pointer in the error path—line 6 (note that line 5 is inserted by our hypocrite commit). However, `pointerA` is an alias of pointer `pointerC`, which comes from the caller and will be further used, and thus if we freed `pointerA` improperly in an earlier location, a UAF would occur. Therefore, we identify it as an ideal immature UAF and will try to introduce a release early in the error path.

```
1      pointerA = pointerC = malloc(...);
2      ...
3      pointerB = malloc(...);
4  *   if (!pointerB) {
5  +       kfree(pointerA);
6  *       return -ENOMEM;
7  *   }
```

**Fig. 9:** Immature UAF case 2, as well as our hypocrite commit, which stealthily introduces a UAF by fixing a memory leak in an error path. "*": immature UAF case, "+": the code added by our hypocrite commit, which introduces the vulnerability condition. The code is simplified and redacted.

**UAF case 3 (refcount bug).** This case is a refcount-leak bug shown in Figure 10. `devA` is an object managed by the refcount

mechanism. The code in line 1 increments its refcount, so the error path (lines 5-6) should decrement the refcount. Since the original code failed to do so (note that line 4 is inserted by our hypocrite commit), it is a refcount bug. On the other hand, line 5 uses `devA`, and the caller may also use it. For this case, we will trigger an implicit free of `devA` through the refcount mechanism to cause a UAF—decrementing the refcount on the error path before a use of the object.

```
1       get_device(devA);
2       ...
3       if (ret < 0) {
4   +       put_device(devA);
5   ★       dev_err(&devA->dev, "error message.");
6   ★       ...
7       }
```

**Fig. 10:** Immature UAF case 3, as well as our hypocrite commit, which improperly fixes a refcount bug by inserting line 4. It introduces a UAF through the implicit free operation of the refcount mechanism. "*": immature UAF case, "+": added code by our hypocrite commit. The code is simplified and redacted.

### D. Stealthy Conditions and Our Hypocrite Commits

We now present how to use stealthy methods to introduce the remaining conditions for the selected immature UAF vulnerabilities described in §VI-C. The general idea is to make sure that the introduced conditions and the existing conditions involve some hard-to-analyze factors such as concurrency.

**UAF case 1.** To introduce a UAF in the first case, we can introduce a use (dereference) of the freed pointer `devA`. As mentioned in §VI-C, `devA` involves concurrency as well as error paths in its concurrent functions. Therefore, we will adopt two stealthy methods: *concurrency* and *error paths*. Specifically, we first find its concurrent functions that have error paths, and then try to print an error message, which uses the freed `devA`.

```
1       err = dev_request(devA);
2       if (err) {
3           disable_device(devA);
4   +       dev_err(&devA->dev, "Fail to request devA!\n");
5           return err;
6       }
```

**Fig. 11:** The first hypocrite commit which just adds an error message that uses a potentially freed object. The code is simplified and redacted.

Figure 11 shows the hypocrite commit. This code piece can be executed concurrently with the code shown in Figure 8 because they are handler functions of asynchronous events. Such concurrency may be hard to notice during review process. If `devA` is already freed in Figure 8, the error-printing code has a UAF. Also, because the added code is in the error path, it is hard to be covered by dynamic testing.

**UAF case 2.** This case is a memory-leak bug, and thus the hypocrite commit pretends to improve the code by fixing the memory leak. In this case, we call `kfree(pointerA)` when the allocation of `pointerB` failed. This is seemingly logical—we clean up the resources upon an error. However, due to `pointerA` points to shared memory, this release would be effective to the caller function, and thus a UAF will occur when the caller tries to use or release the object pointed by `pointerA`. Due to this bug is in the error path, the general dynamic analysis is hard to trigger it; also, this leak is reported by Syzkaller, maintainers would likely trust this report. Moreover, using the alias, `pointerC`, of the released pointer inter-procedurally further makes it harder to be discovered. The concrete code is listed in Figure 9. A correct fix should instead add the release in a later stage in the caller.

**UAF case 3.** The third immature UAF vulnerability involves a refcount-managed object, `devA`. The use is already present, so we can stealthily introduce a free before the use. Specifically, we adopt the *implicit operation* and *concurrency* stealthy methods to introduce a free. Figure 10 shows the hypocrite commit which calls `put_device()` to fix the potential refcount leak bug in the error path. After the refcount reaches zero, object `devA` will be automatically released through a callback function inside `kobject_put()`, which is called by `put_device()`. However, `devA` is further used in line 5 and the caller, which leads to a UAF. A correct fix should place `put_device()` after the dereference is done. In fact, according to our study, previous patches have already caused multiple such cases, even when the committers are not malicious. For example, commit `52e8c38001d8`, which intended to fix a memory leak, also introduced a UAF because of the implicit free of the refcount mechanism.

### E. Increasing the Impacts of Introduced Vulnerabilities

This section will discuss several factors that will influence the potential impacts of the introduced vulnerabilities in the Linux kernel. First, stable release versions are widely used by end users. The Linux kernel includes the release candidate (`rc`) versions, release versions, and the stable-release versions. The `rc` versions would not affect most users, which are mainly aimed for kernel developers. Most of the software and hardware is based on the stable-release versions. If adversaries target these versions, most of the related software and hardware would be influenced directly. Second, to have a higher exploitability, adversaries may use fuzzers or symbolic execution to confirm the triggerability. The three introduced UAF vulnerabilities in our experiment may not be exploitable. The purpose of the experiment is to demonstrate the practicality instead of the exploitability, so exploitation of them is out of the scope.

## VII. Measurements and Quantification

In this section we again use UAF and the Linux kernel as the experimental targets. We will first measure the immature UAF vulnerabilities and the potential placement opportunities for hypocrite commits. Further, we will measure and quantify the stealthiness of each method described in §V.

### A. Immature Vulnerabilities and Condition-Introducing Opportunities

Given an OSS program, the first step is to identify immature UAF vulnerabilities by analyzing the present and absent UAF conditions. In §VI, we developed LLVM passes to find immature UAF using selective rules. In this section, we will

provide a more systematic measurement. A UAF has three conditions, a free of a pointer (without nullification), a use of the pointer, and a specific temporal order—the use is after the free. We first statistically count immature UAF vulnerabilities, in which one condition is absent. Then, corresponding to the different types of absent vulnerable conditions, we measure the opportunities for introducing the conditions.

**Dereferenced pointers without release operations.** In this case, a pointer is dereferenced, but the release operations before the dereferences are absent. Pointer uses are widespread in programs written in low-level languages, and thus nearly every function in the Linux kernel would dereference local or global pointers. Therefore, this type of immature vulnerabilities is highly prevalent. Adversaries can introduce the release through memory-release functions or refcount-decrementing operations.

In fact, refcount leak is a fairly common problem. By searching the Git history in the past two years (from August 2018 to August 2020), we in total found 353 patches for fixing refcount-leak bugs. As such, we believe that adversaries have many opportunities to invoke refcount-decrementing functions to trigger implicit releases. Similarly, we also measure the opportunities for introducing memory release. The Git history in the past two years shows 1,228 memory-leak bugs were fixed, indicating that memory leak is also very common. Adversaries can readily employ existing tools like Syzkaller [57] and Hector [53] to find memory leaks. For example, we ran Syzkaller on the latest version of the Linux kernel with a regular desktop for only one week, and it successfully found 12 memory leaks. These results confirm that it is not hard for adversaries to find opportunities to introduce object releases.

**Freed pointers without nullification and redefinition.** In this case, the present vulnerable condition is a released pointer that is not nullified or redefined (i.e., assigned with another valid pointer), and the absent conditions are uses following the release. This case is prevalent in the Linux kernel. By developing an LLVM pass, we collect freed pointers that are not nullified or redefined. In this analysis, we only consider `kfree()` as the release function. The analysis reports more than 30K cases. Therefore, adversaries have a large number of potential opportunities to introduce a use to form a UAF. As shown in §VI, if we refine the results by only counting cases that can be further used in error paths, we still have 4,657 candidate cases.

**The use-after-free temporal order.** In this situation, a pointer is both used and freed. However, their order is uncertain or can be changed. To measure the commonness of such cases, we attempt to count the objects that can be used and freed in concurrent functions. Identifying concurrent functions itself is a challenging problem. Fortunately, we contacted the authors of a recent work for concurrent UAF detection [2] and were able to obtain the list of concurrent functions. The list contains 43.9K concurrent function pairs. Note that this list is not exhaustive because the work only considers common lock/unlock functions but not customized ones. We found that 1,085 objects are referenced in concurrent functions, i.e., they can be used and freed concurrently. Since these objects are likely protected with locks (otherwise, they are already UAF), adversaries can attempt to remove the locks or move the uses out of the critical sections to introduce the use-after-free temporal order (e.g., the example shown in Figure 6).

### B. Measurement of Stealthiness

In §V, we have empirically discussed several methods for increasing the stealthiness of hypocrite commits. To measure the stealthiness of the methods, we conduct two experiments using two datasets: (1) the first set contains introduced UAF vulnerabilities that have been merged into the Linux kernel branches; (2) the second set contains *blocked* UAF vulnerabilities, which were identified by maintainers during review and not committed into the Linux kernel branches. All of these cases are in the recent six years (from January 2015 to August 2020)

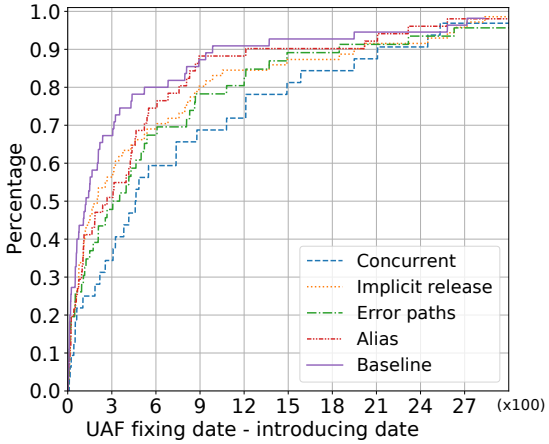


**Fig. 12:** Distribution of UAF vulnerabilities involving stealthy methods, based on their latent period in days (from introducing date to fixing date).

**Qualitative analysis based on latent period.** Precisely quantifying the stealthiness is hard because we lack metrics. Therefore, we first qualitatively compare the stealthiness of the methods against the baseline. The idea is that if a UAF was introduced by a patch involving a stealthier method, the UAF tends to exist in the code for a longer time—a longer latent period. The latent period is calculated by counting the days between the fixing date and the introducing date. The date information is available in the commits. Also, according to our study, during the patch-review process, the Linux community typically verify the patches manually or by using some simple code analysis tools. Therefore, to imitate the typical patch-review process, the baseline is the UAF vulnerabilities in first dataset that are identified through manual approach or static analysis. That is, UAF vulnerabilities found by fuzzing are excluded. We further classify the selected UAF vulnerabilities based on which stealthy method they involve. Note that, in this measurement, we exclude the `indirect-call` stealthy method because we failed to collect enough cases involving indirect calls (we collected only 9 cases), and manually tracing the call

chain to identify indirect calls from the free to the use is a challenging and time-consuming task—it requires to look into numerous functions and structs in an inter-procedural way.

Figure 12 shows the cumulative distribution function (CDF) for latent periods of the baseline and the UAF vulnerabilities involving each stealthy method. In this figure, we use the solid (purple) line to indicate the baseline. The lines below the baseline mean the vulnerabilities tend to have a longer latent period, which indicates that the corresponding method has a higher stealthiness. From this figure, we can find that all these stealthy methods (concurrent, implicit release, error paths) indeed increase the stealthiness. More specifically, the stealthiness of these four methods can be ranked as:

`Concurrency > error paths > implicit release > alias.`

**Quantifying catch rate.** To further quantify the stealthiness of these methods, we introduce the concept of catch rate, which is defined as follows.

$$\texttt{Catch rate} = \frac{\texttt{NC}}{\texttt{NC + NE}} \tag{1}$$

NC denotes the number of vulnerabilities caught by maintainers (i.e., the blocked ones) during a time period, and NE denotes the number of evading vulnerabilities (i.e., the merged ones) during the same time period. This equation is used to estimate the possibility of a hypocrite commit to be caught by maintainers. Since it is impossible to collect all evading UAF vulnerabilities, and many are not found yet, we conservatively assume that the found ones are all of the UAF. As such, the calculated catch rate is an upper bound; that is, the actual rate would be lower. Table IV shows the catch rate for patches involving each stealthy method. It is worth noting that, due to there are not enough cases for the indirect-call case, we list the concrete numbers instead of calculating its catch rate. From the results, we find that hypocrite commits with concurrent issues are the hardest to catch, and all of these cases have a relatively low catch rate. Note that even when an introduced vulnerability patch is caught and blocked, a malicious committer can just try a different one to have a final success.

| Conditions | Catch rate(%) |
|---|---|
| Concurrent issue | 19.4% |
| Implicit release | 36.3% |
| UAF in error-paths | 42.0% |
| Alias | 38.4 % |
| Indirect call | 5/9 |
| Baseline | 56.6% |

**TABLE IV:** Comparison of the catch rate of each stealthy method.

## VIII. Mitigation Against the Insecurity

In this section, we discuss how OSS communities could mitigate the risks from hypocrite commits. We discuss the mitigations from three main perspectives: committers, assisting tools, and maintainers.

### A. Committer Liability and Accountability

By its nature, OSS openly encourages contributors. Committers can freely submit patches without liability. We believe that an effective and immediate action would be to update the code of conduct of OSS, such as adding a term like "by submitting the patch, I agree to not intend to introduce bugs." Only committers who agreed to it would be allowed to go ahead to submit the patches. By introducing the liability, the OSS would not only discourage malicious committers but also raise the awareness of potential introduced bugs for benign committers.

Verifying the identities of committers, i.e., introducing the accountability, is also an effective mitigation against hypocrite commits. This would not only reduce malicious committers but also help with the attribution of introduced vulnerabilities. In particular, OSS projects may only allow reputable organizations or individuals to contribute to the changes, or require certificates from the committers. However, previous works [14, 42] show that checking the identity over online social networks is a challenging problem.

### B. Tools for Patch Analysis and Testing

**Advanced static-analysis techniques.** OSS code can be highly complex; however, patch review is largely manual. As confirmed by our study and the Linux community, the manual review inevitably misses some introduced vulnerabilities. Although communities also use source-code analysis tools to test patches, they are very limited, e.g., using patterns to match simple bugs in source code. Therefore, we believe that an effective mitigation is to employ advanced static-analysis tools. For example, incremental symbolic execution [20, 66] specifically analyzes the code changes and audits patches. Such tools should also support alias analysis [62], concurrency analysis [2, 16], indirect call analysis [36, 48], and common bug detection such as double-free. Also, static analysis is particularly useful for analyzing drivers when the hardware devices are not available. Although static analysis tends to suffer from false positives, it can serve as a preliminary check and provides results for maintainers to confirm.

**High-coverage, directed dynamic testing.** We also suggest to use high-coverage or directed dynamic testing (e.g., fuzzers) to test changed code. In particular, fuzzing, together with sanitizers, can precisely identify bugs. In fact, Linux communities have been using `Syzbot` to test the kernel code, which is an automated system that runs the Syzkaller fuzzer on the kernel and reports the resulting crashes. Directed dynamic testing techniques [5, 61] would be particularly useful in testing the patches. However, we still need to overcome two significant challenges in kernel fuzzing: testing drivers without the hardware devices and testing the error-handling code. In the latest version of the Linux kernel (v5.8), 62% of the code is driver code, counted with tool `cloc`. Existing fuzzers could employ emulators [3, 39], which however only support a limited number of drivers because the emulation is still a manual work. According to [24], existing fuzzers have a very low

coverage rate for error paths; fortunately, we can employ fault injection [24, 47] to effectively improve the coverage. One downside is that the bugs triggered by injected faults can be false positives.

### C. The Maintainer Side

OSS maintainers are mainly volunteers. Our interaction with the OSS communities indicate that OSS maintaining is understaffed. We should appreciate and honor maintainer efforts, and increase potential incentives if possible to encourage more people to join the maintaining.

**Accepting certain preventive patches.** It is understandable that communities tend to reject preventive patches—bugs are not really present, and maintainers are already bombarded with massive patches. However, to mitigate the risks from vulnerability-introducing patches, we highly recommend communities to accept preventive patches for high-risk immature vulnerability if *the remaining vulnerability conditions are likely to be introduced in the future*.

**Raising risk awareness.** It is also important to raise the awareness of hypocrite commits. Our interactions with Linux communities show that they would assume that all contributors are benign and vulnerability-introducing patches are rare mistakes. Through the findings of this paper, we hope that OSS communities would become more aware of potential "malicious" committers who can intentionally introduce vulnerabilities with incentives. On the other hand, we hope our findings also raise the awareness of the common cases of hypocrite commits and the stealthiness factors that may prevent maintainers from finding the introduced vulnerabilities.

**Public patch auditing.** A patch is typically submitted by emailing to a limited number of maintainers. Although a patch is carbon-copied to a public mailing-list, typically only the key maintainers respond and accept the patch. Therefore, a general suggestion is to invite more people to engage in the auditing of the patches. For instance, `get_maintainer.pl` can be more inclusive in deciding maintainers; anyone who has ever contributed to the module or are subscribed to the Linux development mailing-lists can be included.

### D. Feedback of the Linux Community

We summarized our findings and suggestions, and reported them to the Linux community. Here we briefly present their feedback. First, the Linux community mentioned that they will not accept preventive patches and will fix code only when it goes wrong. They hope kernel hardening features like KASLR can mitigate impacts from unfixed vulnerabilities. Second, they believed that the great Linux community is built upon *trust*. That is, they aim to treat everyone equally and would not assume that some contributors might be malicious. Third, they mentioned that bug-introducing patches is a known problem in the community. They also admitted that the patch review is largely manual and may make mistakes. However, they would do their best to review the patches. Forth, they stated that Linux and many companies are continually running bug-finding tools to prevent security bugs from hurting the world. Last, they mentioned that raising the awareness of the risks would be hard because the community is too large.

## IX. Related Work

**OSS maintenance and security.** Quite a number of research works investigate the maintenance and security of OSS. Koponen et al. [30] presented the maintenance process framework of OSS. Midha et al. [45] discussed the participation management and responsibility management for OSS. Thompson [58] discussed the code trustworthiness. This work concludes that people cannot trust code that did not totally create by themselves, and source-level verification would not protect people from using untrusted code. Midha et al. [46] discussed the approaches for improving the maintenance of OSS, and Koponen et al. [29] and Kozlov et al. [31] evaluated the OSS maintenance framework. However, none of them discussed the security issues caused by malicious committers. Hansen et al. [22] found that opening the source code can increase the trustworthiness of the program, but it is not a panacea for security issues. Hoepman et al. [23], Schryen et al. [54], and Witten et al. [63] show similar findings that although opening the source code will first increase the exposure to risks, in a long term, it can actually increase its security because many third parties can find and fix bugs collaboratively. However, these risk-evaluation works did not consider the situation in which the adversaries can intentionally introduce vulnerabilities into the OSS programs. Meneely el al. [43] analyzed the correlations between the known vulnerabilities and the developer activities for the open-source Red Hat Enterprise Linux 4 (RHEL4) kernel. They concluded that files involved in 9 or more developers were likely to have 16 times of vulnerabilities as files changed by less than 9 developers, which means that the changing code involving more developers can have a detrimental effect on the security of software. Shihab el al. [55] concluded that software developers are mostly accurate in identifying code changes that introduce bugs. However, it focuses on patches from benign committers.

**Bug-introducing patches.** Previous works [6, 44, 59] show that non-experienced developers have a strong relationship with the likelihood of bug-introducing patches. More specifically, Bosu et al. [6] analyzed vulnerabilities and code commits for different open-source projects, and showed that the code changes sent by new developers introduce 1.8 to 24 times vulnerabilities as the ones from experienced developers. Bird et al. [4] also believed that the removal of low-expertise contributions can reduce contribution-based defect-prediction performance. Unlike this radical idea, Rahman et al. [51] showed that the author experienced in a specific file is more important than the general experience, which can be asked to control the quality of code and submitted patches. After analyzing the incorrect bug-fixing patches in the Linux, OpenSolaris, and FreeBSD, Yin et al. [67] showed that concurrency bugs are the most difficult to fix correctly, and 39% of concurrency bug fixes are incorrect. This result is similar to our conclusion that the concurrency issue is one of the stealthiest methods that can be used by attackers to hide

their vulnerability conditions. This paper also shows that 27% of the incorrect patches are made by fresh developers who have never touched the source code files before. Similar to hypocrite commits, Paul A. [25] discussed Trojan Horses in general; an inherent difference is that hypocrite commits themselves do not contain malicious functionalities or vulnerabilities, but introduce vulnerability conditions. Most of these existing works focused on the statistical analysis of the previous (unintentional) bug-introducing patches and summarizing the introduction reasons. However, to our knowledge, none of them studied the risks from hypocrite commits.

**Analysis and testing of patches.** Many previous works focused on analyzing and testing the correctness of patches to eliminate bug-introducing patches. Zhou et al. [68] used natural language processing and machine learning techniques to identify the security issues of patches, by analyzing the commit messages of bug reports. However, the hypocrite commits constructed by adversaries typically would also fix some minor issues in the program, and the commit messages would not have differences from other general patches. Beyond simple commit message analysis, previous works also equipped techniques like incremental symbolic execution [32, 40] and semantic patch analysis. Marinescu et al. [41] proposed KATCH, a static analysis tool, which combines symbolic execution and heuristics to test the patches of software. Le et al. [34] present MVICFG, which can analyze the patches by comparing the control-flow commonalities and differences for different versions of the program. However, adversaries can leverage the stealthy methods introduced in §V to hinder and confuse these static analysis tools. Bosu et al. [52] concluded that peer code review can identify and remove some vulnerabilities in the project and increase software security.

## X. Conclusion

This paper presented hypocrite commits, which can be abused to stealthily introduce vulnerabilities in OSS. Three fundamental reasons enable hypocrite commits: the openness of OSS, which allows anyone including malicious committers to submit patches; the limited resources of OSS maintaining; and the complexity of OSS programs, which results in the manual review and existing tools failing to effectively identify introduced vulnerabilities. We then systematically characterized immature vulnerabilities and studied how a malicious committer can turn immature vulnerabilities into real ones. We also identified multiple factors that increase the stealthiness of the introduced vulnerabilities, including concurrency, error paths, aliases, indirect calls, etc. Furthermore, we provided a proof-of-concept to safely demonstrate the practicality of hypocrite commits, and measured and quantified the risks. We finally provided our suggestions on mitigating the risks of hypocrite commits and hope that our findings could motivate future research on improving the patching process of OSS.

## XI. Acknowledgment

We would like to thank Linux maintainers for reviewing our patches and the anonymous reviewers for their helpful suggestions and comments. We are also grateful to the Linux community, anonymous reviewers, program committee chairs, and IRB at UMN for providing feedback on our experiments and findings. 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] G. Android. Android Security Rewards Program Rules, August 2020. https://www.google.com/about/appsecurity/android-rewards/.

[2] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 255–268, 2019.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[4] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14, 2011.

[5] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[6] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 257–268, 2014.

[7] S. Cai and K. Knight. Smatch: an evaluation metric for semantic feature structures. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 748–752, 2013.

[8] X. Chen, A. Slowinska, and H. Bos. Who Allocated My Memory? Detecting Custom Memory Allocators in C Binaries. In *WCRE*, Sept. 2013. URL http://www.cs.vu.nl/~herbertb/papers/membrush_wcre13.pdf. Best Paper Award.

[9] Y. Chen and X. Xing. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1707–1722, 2019.

[10] C. contributors. Coccinelle, Dec 2019. https://coccinelle.gitlabpages.inria.fr/website/.

[11] W. contributors. Reference counting, August 2020. https://en.wikipedia.org/wiki/Reference_counting.

[12] J. E. Cook and A. L. Wolf. Event-based detection of concurrency. *ACM SIGSOFT Software Engineering Notes*, 23(6):35–45, 1998.

[13] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 166–177. IEEE, 2015.

[14] T. DuBois, J. Golbeck, and A. Srinivasan. Predicting trust and distrust in social networks. In *2011 IEEE third international conference on privacy, security, risk and trust and 2011 IEEE third international conference on social computing*, pages 418–424. IEEE, 2011.

[15] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[16] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *ACM SIGOPS operating systems review*, 37 (5):237–252, 2003.

[17] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective

data-race detection for the kernel. In *OSDI*, volume 10, pages 1–16, 2010.

[18] GitHub. Repository search for public repositories - Retrieved June 5, 2018, showing 28,177,992 available repository results, 2018. https://github.com/search?q=is:public.

[19] I. GitHub. Repository search for users, 2020. https://github.com/search.

[20] S. Guo, M. Kusano, and C. Wang. Conc-ise: Incremental symbolic execution of concurrent software. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 531–542, 2016.

[21] I. Guseva. Bad Economy Is Good for Open Source, march 2009. http://www.cmswire.com/cms/web-cms/bad-economy-is-good-for-open-source-004187.php.

[22] M. Hansen, K. Köhntopp, and A. Pfitzmann. The open source approach—opportunities and limitations with respect to security and privacy. *Computers & Security*, 21(5):461–471, 2002.

[23] J.-H. Hoepman and B. Jacobs. Increased security through open source. *Communications of the ACM*, 50(1):79–83, 2007.

[24] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu. Fuzzing error handling code using context-sensitive software fault injection. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[25] P. A. Karger. Limiting the damage potential of discretionary trojan horses. In *1987 IEEE Symposium on Security and Privacy*, pages 32–32. IEEE, 1987.

[26] T. kernel development community. Everything you ever wanted to know about Linux -stable releases, August 2016. https://www.kernel.org/doc/html/v4.10/process/stable-kernel-rules.html.

[27] T. kernel development community. Linux kernel mailing list, August 2020. https://lkml.org.

[28] T. kernel development community. Submitting patches: the essential guide to getting your code into the kernel, 2020. https://www.kernel.org/doc/html/v4.10/process/submitting-patches.html.

[29] T. Koponen. Evaluation framework for open source software maintenance. In *2006 International Conference on Software Engineering Advances (ICSEA'06)*, pages 52–52. IEEE, 2006.

[30] T. Koponen and V. Hotti. Open source software maintenance process framework. In *Proceedings of the fifth workshop on Open source software engineering*, pages 1–5, 2005.

[31] D. Kozlov, J. Koskinen, J. Markkula, and M. Sakkinen. Evaluating the impact of adaptive maintenance process on open source software quality. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 186–195. IEEE, 2007.

[32] T. Kuchta, H. Palikareva, and C. Cadar. Shadow symbolic execution for testing software patches. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 27(3):1–32, 2018.

[33] A. Kuehn and M. Mueller. Analyzing bug bounty programs: An institutional perspective on the economics of software vulnerabilities. In *2014 TPRC Conference Paper*, 2014.

[34] W. Le and S. D. Pattison. Patch verification via multiversion interprocedural control flow graphs. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1047–1058, 2014.

[35] 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, 2017.

[36] 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.

[37] K. Lu, A. Pakki, and Q. Wu. Automatically identifying security checks for detecting kernel semantic bugs. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS)*, Luxembourg, Sept. 2019.

[38] K. Lu, A. Pakki, and Q. Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[39] D. Maier, B. Radtke, and B. Harren. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *13th {USENIX} Workshop on Offensive Technologies ({WOOT} 19)*, 2019.

[40] P. D. Marinescu and C. Cadar. High-coverage symbolic patch testing. In *International SPIN Workshop on Model Checking of Software*, pages 7–21. Springer, 2012.

[41] P. D. Marinescu and C. Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 235–245, 2013.

[42] A. M. Meligy, H. M. Ibrahim, and M. F. Torky. Identity verification mechanism for detecting fake profiles in online social networks. *Int. J. Comput. Netw. Inf. Secur.(IJCNIS)*, 9(1):31–39, 2017.

[43] A. Meneely and L. Williams. Secure open source collaboration: an empirical study of linus' law. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 453–462, 2009.

[44] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 65–74. IEEE, 2013.

[45] V. Midha and A. Bhattacherjee. Governance practices and software maintenance: A study of open source projects. *Decision Support Systems*, 54(1):23–32, 2012.

[46] V. Midha, R. Singh, P. Palvia, and N. Kshetri. Improving open source software maintenance. *Journal of Computer Information Systems*, 50(3): 81–90, 2010.

[47] R. Natella, D. Cotroneo, and H. S. Madeira. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)*, 48(3):1–55, 2016.

[48] B. Niu and G. Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 577–587, 2014.

[49] A. Pakki and K. Lu. Exaggerated Error Handling Hurts! An In-Depth Study and Context-Aware Detection. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Virtual conference, Nov. 2020.

[50] M. Pittenger. The state of open source in commercial apps: You're using more than you think, 2016. https://techbeacon.com/security/state-open-source-commercial-apps-youre-using-more-you-think.

[51] F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500, 2011.

[52] S. Ristov and M. Gusev. Security evaluation of open source clouds. In *Eurocon 2013*, pages 73–80. IEEE, 2013.

[53] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[54] G. Schryen and R. Kadura. Open source vs. closed source software: towards measuring security. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 2016–2023, 2009.

[55] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[56] J. Siri. OSS-Contribution-Tracker, July 2020. https://github.com/amzn/oss-contribution-tracker.

[57] Thgarnie. Syzkaller, 2020. https://github.com/google/syzkaller.

[58] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[59] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering*, pages 1039–1050, 2016.

[60] S. J. Vaughan-Nichols. It's an open-source world: 78 percent of companies run open-source software, April 2015. https://www.zdnet.com/article/its-an-open-source-world-78-percent-of-

companies-run-open-source-software/.

[61] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.

[62] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, 2004.

[63] B. Witten, C. Landwehr, and M. Caloyannides. Does open source improve system security? *IEEE Software*, 18(5):57–61, 2001.

[64] Q. Wu, Y. He, S. McCamant, and K. Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *Network and Distributed Systems Security Symposium, NDSS*, 2020.

[65] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 661–678. IEEE, 2018.

[66] G. Yang, S. Person, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(1):1–42, 2014.

[67] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 26–36, 2011.

[68] 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, 2017.