Strengthening Supply Chain Security with Fine-grained Safe Patch Identification

Changhua Luo
The Chinese University of Hong Kong
Hong Kong SAR, China
chluo@cse.cuhk.edu.hk

Wei Meng
The Chinese University of Hong Kong
Hong Kong SAR, China
wei@cse.cuhk.edu.hk

Shuai Wang
HKUST
Hong Kong SAR, China
shuaiw@cse.ust.hk

ABSTRACT

Enhancing supply chain security is crucial, often involving the detection of patches in upstream software. However, current security patch analysis works yield relatively low recall rates (*i.e.*, many security patches are missed). In this work, we offer a new solution to detect safe patches and assist downstream developers in patch propagation. Specifically, we develop SPatch to detect fine-grained safe patches. SPatch leverages fine-grained patch analysis and a new differential symbolic execution technique to analyze the functional impacts of code changes.

We evaluated SPatch on various software, including the Linux kernel and OpenSSL, and demonstrated that it outperformed existing methods in detecting safe patches, resulting in observable security benefits. In our case studies, we updated hundreds of functions in modern software using safe patches detected by SPatch without causing any regression issues. Our detected safe security patches have been merged into the latest version of downstream software like ProtonVPN.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Supply Chain Security; Fine-grained Patch Analysis; Differential Symbolic Execution;

ACM Reference Format:
Changhua Luo, Wei Meng, and Shuai Wang. 2024. Strengthening Supply Chain Security with Fine-grained Safe Patch Identification. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639104

1 INTRODUCTION

Modern software commonly depends on upstream code on the supply chain. Recent works have shown that vulnerabilities in dependency code are common and often remain unfixed for extended periods [29, 45]. The significant patch propagation delays leave a large attack window for experienced hackers [43]. For example, the Android kernel maintainers had not patched a severe use-after-free vulnerability[1], which had been first discovered and fixed in Linux kernel one year ago, until that vulnerability had been finally exploited on most Android devices [12].

Identifying security patches in upstream software is critical for promptly addressing vulnerabilities in dependency code of the downstream software. Security patches refer to the code changes that fix security vulnerabilities. Existing approaches to identifying security patches include rule-based techniques [34, 40, 45] and deep learning [31, 33, 36]. Despite continuous efforts, they often suffer from high false negatives, *i.e.*, missing many security patches. For example, SID can only identify specific types of security patches [40]. Other deep learning-based tools such as GraphSPD [33] and PatchRNN [36] can achieve only a recall rate of lower than 50%.

In addition to identifying security patches, prior studies also proposed to identify code changes that do not alter the program's semantics to help update dependency code [9, 24]. Specifically, Upgradvisor utilized static analysis and selective tracing to help maintainers determine whether the dependency updates affect application semantics and minimize the efforts maintainers invest in dependency updates [9].

Spider introduced an automated method to identify code changes that (likely) preserve application functionality [24]. It considered code changes safe to port (referred to as safe patches) if they satisfy certain predefined conditions. The paper's user survey showed that 82% of software maintainers (*e.g.*, those of Ubuntu, *etc.*) considered using safe patches for their projects. The remaining 18% of participants agreed that safe patches helped maintainers prioritize their patch efforts.

We adopt Spider's concept of safe patches as it was demonstrated to be useful in the user survey [24]. However, we take a step further by considering security features when detecting safe patches. The rationale behind this is that fixing vulnerabilities is an important motivation for software maintainers to update dependency code. In our preliminary study, we find that Spider can identify only 38 out of 71 (53.5%) safe security patches as safe. The rest 46.5% safe security patches would be incorrectly excluded. Upon further analysis, we summarize two reasons for its false negatives. First, it employs an imprecise intra-procedural data flow analysis that cannot reason about the functional impact of code changes spanning multiple functions. Second, it does not differentiate security updates from functional updates in the same commits. Patch propagation remains a challenge due to these limitations.

We aim to develop better techniques to address the aforementioned patch propagation issues in software supply chain security. Specifically, for the security updates that do not alter software

ICSE '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639104

[1]CVE-2019-2215

functionality, we can decouple them from the irrelevant functional updates in the same commits. Therefore, we can generate and provide downstream maintainers with more safe security patches that improve their software's security while ensuring functionality consistency. To achieve this, unlike prior patch analysis works that conduct analysis at the commit level [24, 33, 36], our approach performs *inter-procedural analysis* to infer functional impacts at a *finer-grained* level—independent code changes in one commit. We target detecting *partially-safe commits*, which consist of safe code changes (*i.e.*, safe patches) and possibly some irrelevant unsafe code changes. By identifying and porting the safe security-relevant code changes in such commits, we can fix more vulnerable downstream programs without breaking the existing code.

However, we encounter several challenges in identifying partially-safe commits. First, we must strike the right granularity when splitting a commit into independent code change groups to avoid losing patch meaning (too fine-grained) or being ineffective (like Spider, too coarse-grained). In our solution, to split a commit, we consider both edit actions [10] and data-flow effects. Specifically, we group code changes into a change unit (CU) if they belong to one edit action or affect the same variable. A CU serves as the basic unit in safe patch identification and preserves the data-flow effects of independent code changes. However, due to the imprecise nature of dependency analysis, some CUs cannot be successfully applied (*e.g.*, it uses undeclared variables). Generating compilable patched code is a prerequisite for the subsequent differential symbolic execution (DSE [27]) analysis (which determines if a CU is safe). Accordingly, we perform a compatibility test on each CU. Different software versions often require different building environments, making it challenging to test patch compatibility. For example, a compiler used for an older version of the Linux kernel might not work for more recent versions. To tackle this issue, we propose a solution to test the compatibility of a CU as long as its code changes do not employ language features unsupported by the testing environment. Specifically, we identify and utilize compilation commands associated with that CU rather than the `Makefile` utility for testing its compatibility.

The second challenge we encounter is the complexity involved in analyzing the functional impacts of code changes in modern applications. While Spider attempts to address this challenge, it suffers from issues with stability and feature completeness. For example, it lacks support for C/C++ macros and the analysis of modifications to function calls. To overcome this challenge, we implement C/C++ DSE [27], a technique that computes and compares symbolic expressions in original and patched functions to infer a CU's effects. Our tool is robust as it handles various language features by leveraging a mature compiler wrapper called `scan-build` [21]. However, we represent symbolic expressions using program variables rather than LLVM-IR (though it is used by mainstream symbolic execution tools like KLEE [17]), as this enables a precise differential analysis of original and patched functions. We further implement an under-constrained analysis and propose an *on-demand* DSE analysis, which includes selectively symbolic interpretation of function calls and statically filtering out statements irrelevant to the patches. All these enhancements mitigate the path explosion problem during the DSE phase.

We implemented all the techniques in a system called SPatch and conducted a comprehensive evaluation of SPatch on complex software projects including the Linux kernel and OpenSSL. SPatch detected 12,140 safe patches (including security patches and non-security safe patches) among 17,513 commits. Our analysis indicates that the number of partially-safe commits is almost equivalent to that of commits consisting solely of safe patches (which we refer to as *safe commits*). We conducted an ablation study to analyze the effectiveness of SPatch's components. We also emulated Spider [24] following its design, as its source code has not been released yet. The results confirmed the significance of both the consideration of partially-safe commits and our proposed DSE technique for detecting safe patches. To evaluate the security benefits of SPatch, we manually analyzed 1,100 randomly selected commits, which included 52 security patches. Among them, SPatch detected 40 as safe security patches, achieving a recall rate of 76.93% for all security patches. In contrast, Spider achieved a recall rate of 40.48%. We conducted end-to-end patch porting on popular open-source software—Redis [28] and ProtonVPN [5]. We updated hundreds of functions in their dependencies using the safe patches detected by SPatch. The updated codes could be successfully built and they passed all regression tests. Maintainers have merged five safe security patches we submitted via pull requests. Additionally, we were rewarded with a bounty for fixing vulnerabilities in the outdated dependency code of ProtonVPN.

In summary, this paper makes the following contributions:

- We developed SPatch, a tool for comprehensively detecting fine-grained safe patches using DSE.
- Our evaluation showed that SPatch brought clear security benefits compared to existing tools without incurring regression issues.
- SPatch and the artifacts will be publicly available at https://github.com/cuhk-seclab/SPatch.

2 BACKGROUND

In this section, we introduce code reuse and the vulnerabilities in dependency code (§2.1), the existing works on identifying security patches (§2.2) and safe patches (§2.3).

2.1 Supply Chain Vulnerabilities

Supply chain vulnerabilities are prevalent. Past works have shown that many vulnerabilities in dependency code remain unpatched for a long time even if they are fixed in the upstream programs [18, 29, 43]. For instance, Reid *et al.* [29] found that over half of their studied projects had a common upstream dependency and thus derived the same vulnerability from the upstream project, which fixed it three years ago. Zhang *et al.* [43] revealed that nearly half of the CVEs were not patched on the OEM devices until 200 days or more after the initial patches in upstream code were publicly committed. The significant patch propagation delays leave a large attack window for attackers and could cause severe security issues [15, 43, 45].

2.2 Security Patches

Security patches are code changes that fix vulnerabilities. Identifying security patches in upstream software is crucial as it helps downstream developers fix vulnerabilities in their own dependency code. Consequently, numerous approaches have been proposed to detect security patches, including rule-based methods [13, 40] and deep learning techniques [33, 36, 44]. Detecting security patches requires some prior knowledge. Rule-based approaches rely on predefined code patterns to detect security patches, whereas deep learning methods depend on high-quality and diverse datasets to train models. Both approaches face a trade-off between precision and recall. For example, the state-of-the-art security patch detection tool GraphSPD [33] employs a GNN model, surpassing those using RNN [36, 44]. Its evaluation achieves high precision yet a relatively lower recall rate of 43.5% (*i.e.*, many false negatives).

2.3 Patches Preserving Functionality

In addition to detecting patches fixing security vulnerabilities, existing works also detect patches that preserve application functionality to help downstream developers update dependency code.

2.3.1 Upgradvisor. David *et al.* proposed Upgradvisor to identify safe updates, *i.e.*, code changes that preserve application functionality, in dependencies [9]. To achieve this, it utilizes static analysis to identify the code changes that might be reachable from the application code. Subsequently, the code is selectively traced. The tracing is implemented with minimal overhead, allowing Upgradvisor to capture the effects of code changes in a production environment. Upgradvisor requires manual efforts to examine the tracing results and evaluate whether the executed code changes impact application semantics. The evaluation on Python projects revealed that many previously blocked dependencies were safe to update.

2.3.2 Spider. Machiry *et al.* defined a concept called *safe patches* (abbreviated to *SP*s) [24]. SPs are the code changes that satisfy two conditions:

- Non-increasing input space (C1): the code changes *do not* increase the valid input space. The valid inputs denote the inputs whose executions do not trigger the error-handling code.
- Output equivalence (C2): for all valid inputs, the updated function has the same function outputs as the original function. The function outputs denote the variables that can be accessed outside the function scope, including the global variables, return values, *etc.*

Machiry *et al.* proposed Spider [24] to identify SPs. They implemented C/C++ symbolic execution from scratch to check C1 and C2. Due to the challenges of implementing C/C++ symbolic execution, Spider is open to false negatives, *i.e.*, potentially classifying SPs as unsafe. For example, Spider considers any code changes involving function calls unsafe. Its evaluation shows that only 19% of commits are safe and can be ported to downstream.

Compared to manually examining the functional impacts of some patches (as in Upgradvisor), detecting SPs offers an *automated* way to pinpoint (presumably) portable patches. According to the user surveys in [24], developers plan to use SPs or prioritize investigating SPs. Based on this observation, this work uses the definition of SPs in Spider to infer the functional impacts of code changes. Unlike Spider, we detect partially-safe commits and SPs involving function calls, using *differential symbolic execution* (DSE). The design is driven by insights from our patch study in §3.1, which underscores the importance of a fine-grained inter-procedural analysis in detecting safe security patches.

3 PROBLEM STATEMENT

In this section, we motivate the work with an example in §3.1, and present our research scope and research goals in §3.2.

3.1 A Motivating Example

Listing 1 shows a commit that includes a security patch fixing a vulnerability[2] in the Linux kernel. We choose this example as it represents a typical security patch and is relatively easy for readers to comprehend. The security patch fixes a use-after-free (UAF) by validating `page` before using it (lines 12-19). To detect this patch using security patch detection tools, it is necessary to model UAF vulnerabilities or include similar UAF patches in the training dataset. In the following, we show how we can detect it by identifying (fine-grained) SPs and the limitations of Spider.

The patch involves modifications to function calls. It also introduces a new `if` condition (line 15), which could lead to the error handling code (lines 16-17). By inlining the callee functions of these modified calls and analyzing the control flow and data flow, we can conclude that the patch reduces the valid input space (satisfying C1) and does not alter function outputs (satisfying C2). Thus, the code changes in lines 12-19 are SPs. Additionally, the commit introduces other code changes.

The security patch may seem simple, but understanding its functional impacts is not straightforward. Spider fails to detect this patch for two reasons. First, Spider cannot understand the effects of function calls, leading to a missed identification of the safe security patch. Second, it identifies patches at a commit granularity, causing the SP (partially-safe commit) in lines 12-19 to be conservatively excluded once again.

We investigate the ability of Spider to analyze the functional impacts of *security patches*. Specifically, we randomly selected 100 commits that include security patches from patchDB [35], and manually pinpointed the locations of security patches. We found that ① 31 security patches involved modifications to function calls (that invoked custom functions like input validation functions, lock/unlock functions, clean functions, *etc.*); ② 38 commits included the simultaneous implementation of safe security patches and other code changes. There were occurrences where ① and ② overlapped. Seventeen safe security patches, which included modifications to function calls, were committed alongside other code changes. Our manual analysis showed that 71 security patches were also safe patches, while Spider could detect 38 among them. In contrast, SPatch identified all SPs detected by Spider and 27 additional ones, achieving a 91.55% recall rate for safe security patches and a 65% recall rate for all security patches. We further discuss the reasons for false negatives of SPatch in detecting safe security patches in §6.3.

3.2 Research Goals and Scope

This work improves SP detection techniques by taking into account the following *security patch features*. We specifically emphasize

[2]CVE-2019-11487

```
***
2 files changed, 49 insertions(+), 12 deletions(-)

diff --git a/mm/gup.c b/mm/gup.c
index 75029649baca..81e0bdefa2cc 100644
--- a/mm/gup.c
+++ b/mm/gup.c
(*\textbf{@@ -157,8 +157,12 @@ retry:}*)
    goto retry;
  }

- if (flags & FOLL_GET)
-   get_page(page);
+ if (flags & FOLL_GET) {
+   if (unlikely(!try_get_page(page))) {
+     page = ERR_PTR(-ENOMEM);
+     goto out;
+   }
+ }
  if (flags & FOLL_TOUCH) {
    if ((flags & FOLL_WRITE) &&
        !pte_dirty(pte) && !PageDirty(page))

... // other code changes.
```

Listing 1: A commit fixes a *use-after-free* vulnerability in Linux kernel.

the consideration of security patch features in SP detection due to the important role of patching security vulnerabilities in updating dependency code. First, instead of performing patch analysis at a commit level like many prior works did, we aim to detect which code changes of a commit are SPs. We do this because a commit may contain both security updates and functionality updates. Second, we aim to design a precise and powerful DSE approach for complex C/C++ software. Specifically, our analysis should be inter-procedural and support various C/C++ language features that Spider might overlook. These capabilities are important. As revealed in an exhaustive study conducted by Wang *et al.* [35], a significant proportion (24.4%) of the security patches involved modifications to function calls. Language features like macro calls and C/C++ directives (*e.g.*, #ifdef) are also widely used in modern software.

We follow Spider's SP concept, aiming to comprehensively detect code changes satisfying C1 and C2. In §7, we will compare this approach with Upgradvisor, which employs manual analysis to evaluate the functional impacts of code changes. Besides, we only analyze code changes in the C/C++ files that are compiled into executable. Finally, we aim to assist patch propagation but do not directly fix vulnerabilities in downstream software. As downstream developers usually reuse code with diverse syntax modifications [39], generating new patches for different programs is orthogonal to this work.

4 DESIGN

We develop SPatch to detect SPs in the upstream software repository. In this section, we first give an overview of SPatch in §4.1 and then present its detailed component-wise design.

4.1 Overview

SPatch employs several techniques to achieve its research goals. First, it adopts a fine-grained approach to analyze SPs in each commit. Instead of considering whole commits, SPatch groups code changes of a commit into Change Units (CUs) and identifies CUs that independently qualify as SPs. This approach allows for more precise patch analysis. Second, to comprehensively analyze each CU, SPatch performs on-demand DSE. It compiles (and symbolically

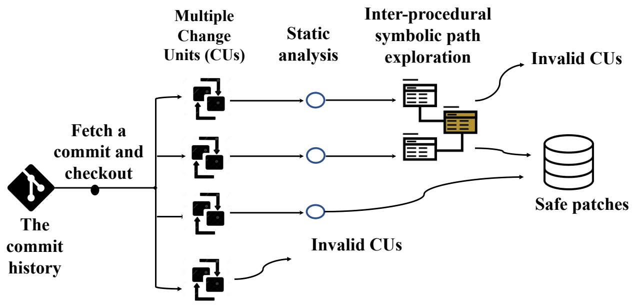
Multiple Change Units (CUs)
Static analysis
Inter-procedural symbolic path exploration
Invalid CUs
Fetch a commit and checkout
The commit history
Safe patches
Invalid CUs

Figure 1: Overview of SPatch.

executes) the updated program components to analyze the effects of their code changes. To handle modifications to function calls, SPatch selectively inlines callee functions that might influence the identification of SPs. These strategies avoid exploring unrelated program paths in DSE.

The workflow of SPatch is depicted in Figure 1. Since not all commits update the source code, SPatch filters out some irrelevant commits. It first (temporarily) excludes merge commits which only include a lot of repetitive code changes. The merge commits are later analyzed based on the detection results of non-merge commits. Furthermore, not all C/C++ files are compiled into binaries. SPatch also filters out the C/C++ files that are not utilized in the software building process, as their source code and patches are never reachable.

SPatch then groups code changes of a commit into CUs. Multiple code changes are grouped into one CU if they belong to one edit action or have intra-procedural data dependencies with the same program variable (§4.2). Each CU within a patched function is independent of the others. SPatch can later *separately* verify them to determine which ones are SPs. SPatch applies each CU and compiles the updated code to determine whether or not the code causes compatibility issues. It excludes the CUs that result in compatibility issues and feeds the rest into the subsequent analysis.

SPatch employs a two-stage program analysis technique, including coarse-grained static analysis and on-demand DSE analysis, to detect SPs among CUs (§4.3). Static analysis allows SPatch to efficiently pinpoint some SPs, *i.e.*, the CUs that have no dependencies with the variables (referred to as *critical* variables in the subsequent sections) used in path constraints (C1) or function outputs (C2). It also facilitates the symbolic execution phase by providing necessary information, such as the function calls that require symbolic interpretation. The DSE phase analyzes the CUs that cannot be determined by static analysis. SPatch compares the symbolic expressions in the original and patched functions to identify if a CU is an SP.

4.2 Grouping Code Changes

In this subsection, we introduce how we split a commit into multiple CUs (the basic units of SP identification). A CU is atomic, *i.e.*, the code changes in one CU occur entirely or not at all.

```
1  int foo() {
2  -    int a=1;
3  +    int a=0;
4  -    int b=1/a;
5  +    int b=1/(a+1);
6  +    int c;
7  }
```

Listing 2: An example that demonstrates how SPatch identifies CUs.

4.2.1 Identifying CUs. SPatch merges the code changes that belong to one edit action into a CU. An edit action is an atomic modification made to the code. We use `Gumtree` [10] to identify four categories of edit actions, namely add, delete, update, and move. In addition to edit actions, we also consider data dependencies among code changes. Applying partial code changes in a commit may lead to issues, as will be illustrated in Listing 2. To avoid potential issues, we merge the code changes into one CU when they have data-flow influence on the same variable(s).

We use Listing 2 to demonstrate how SPatch identifies CUs from code changes. There are three edit actions, which are *update* (lines 2 and 3), *update* (lines 4 and 5), and *add* (line 6). SPatch merges code changes having data dependencies, which are code changes in line 2 and line 4 (in the original function), line 3 and line 5 (in the updated function). This avoids potential issues, *e.g.*, updating a to 0 but not updating expressions using a. Code changes in lines 2, 3, 4, and 5 are merged into one CU as they either belong to one edit action or have data dependencies. Finally, SPatch identifies two CUs in Listing 2, which are ① changes in lines 2-5 and ② changes in line 6.

It should be noted that inter-procedural data dependencies are not being considered at this time. Failure to consider inter-procedural data dependencies may lead to issues. For instance, if we discard a CU and apply another one that uses the variables defined in the discarded CU, the applied CU will be invalid and thus may not be successfully ported to the vulnerable code. We discuss how we solve this problem in §4.2.2.

4.2.2 Identifying SP Candidates. Similar to other patch analysis works [24, 33], we exclude certain types of code changes from further analysis. Since SPatch aims to generate a *portable* patch, it further filters out the CUs that cause compatibility issues. We view the remaining CUs as potential *SP candidates* for further analysis.

The following CUs are *initially* excluded by SPatch. SPatch first discards any CUs that involve global scope edits. Updating global scope statements usually requires analyzing the whole program [19]. For instance, updating a global scope data structure requires analyzing all the code using the data structure. They are beyond our analysis scope. Second, SPatch excludes the CUs that 1) directly update loop statements or the number of iterations of loop statements that affect function outputs or 2) write to unknown memory locations pointed by pointers. This is because handling pointers requires the whole program analysis [23], and updating statements in loops renders the data analysis results undecidable [24].

After discarding these CUs, SPatch proceeds to identify and eliminate any incompatible CUs caused by using identifiers or data types defined in the discarded CUs. To this end, we apply a CU and compile the updated code to test if it causes any compatibility issues (*e.g.*, using undefined variables). Compiling the software is time-consuming and fragile due to different environment requirements. We generate a minified program that includes the patched function affected by a CU to test its compatibility. As a function typically employs legacy identifiers or data types that are defined outside the scope of the function (*e.g.*, `FOLL_WRITE` in Listing 1), compiling the minified program alone might incur compilation errors caused by these (legacy) identifiers, which are not directly related to the CU itself. We discuss how SPatch addresses this issue in §5.3.

4.3 Identifying SPs

In this subsection, we perform static analysis and DSE to identify SPs among SP candidates. Static analysis not only detects the SPs that do not influence the critical variables but assists the following DSE (*e.g.*, by providing updated function call arguments). DSE enables SPatch to identify SPs even though they affect critical variables, as having data dependencies with critical variables does not necessarily indicate updating their values.

4.3.1 Static Analysis. In this step, SPatch performs light-weight static analysis to quickly identify some SPs.

Error-handling Code Although some code changes in error-handling code might affect functionality, we adopt the approach in Spider and consider that modifications to the error-handling code do not influence the detection results of SPs. We use the state-of-the-art tool ErrDoc [32] to identify error-handling code. However, it shares some limitations (*e.g.*, false positives) with other tools. We leave the optimization as future work. In our subsequent analysis, we omit the path constraints that result in the execution of identified error-handling code (lines 16-17 in Listing 1) and function outputs in error-handling code.

Data-flow Analysis SPatch performs light-weight intra-procedural data-flow analysis to pinpoint the SP candidates that do not affect path constraints and function outputs. To this end, it performs slicing in a forward direction using the SP candidates as criteria. If the data-flow slicing does not contain statements written to critical variables used in path constraints or function outputs, the candidates are identified as SPs directly.

The data-flow analysis also facilitates DSE. First, SPatch has been designed to exclude modeling complex statements that are determined to remain unaffected by SP candidates. This improves accuracy as SMT has limitations in modeling intricate expressions [6]. Besides, SPatch identifies the function calls affected by the candidates. The results are utilized when SPatch performs selectively symbolic path exploration in §4.3.2.

4.3.2 On-demand DSE. SPatch employs DSE to analyze whether the data dependencies involving crucial variables lead to modifications in their values. Although software typically contains a substantial amount of code, only a minor portion of it, such as the statements that write to global variables, determines whether the candidates are SPs. Therefore, we perform DSE on demand. In particular, we symbolically execute only the updated functions, and selectively interpret function calls to mitigate the path explosion problem.

Symbolic Execution at Source Code Level SPatch implemented DSE at a source-code level. Most C/C++ symbolic execution tools are based on LLVM-IR, and we are unable to directly use these tools as comparing symbolic expressions represented with IR variables cannot be done precisely. For instance, the same variable in original and patched functions might be represented with distinct IR variables. Aligning these symbols and constraints unavoidably incurs errors [22].

We built SPatch using `Juxta` [25], a mature tool based on scan-build [21], enabling source-code-level symbolic execution. `Juxta` runs the compiler on the source code and intercepts the compiler output (*e.g.*, control flow graphs) to symbolically explore the program paths.

Under-constrained Analysis Efficiency is a primary concern in DSE due to millions of patched functions that need to be analyzed. We initially leveraged the *incremental updating* mechanism and recompiled the software upon analyzing a new commit. The compiler, as well as `Juxta`, would then only focus on the code that has changed since the last analysis. This approach is not practical for the following reasons. First, the compiler may encounter several problems during software building, such as missing dependencies or incompatible compiler versions. If the building process fails, `Juxta` cannot conduct symbolic execution. Second, even though we could successfully compile the software, it is difficult to interpret the symbolic execution results. As an example, building OpenSSL produces more than a million program paths. Pinpointing the paths that a CU updates is difficult.

Our approach is to reuse the minified program generated in compatibility testing, *i.e.*, we compile and perform DSE on the original and patched functions only. However, no source-code-level symbolic execution tools, including `Juxta`, support under-constrained analysis. We discuss the implementation detail of under-constrained analysis on `Juxta` in §5.4.

Selectively Symbolic Interpretation SPatch selectively inlines the callee functions to analyze the modifications to function calls. Specifically, SPatch selectively interprets function calls if two conditions are met. First, if the function calls are updated (*i.e.*, the arguments are updated, or the call sites are added or deleted) by the SP candidates. Second, if the callee function is not classified as one of the logging functions (*e.g.*, `printk()`). Function calls that do not meet the two conditions are not interpreted. Since their behaviors remain unchanged before and after applying the code changes, they should not have any influence on the detection of SPs.

The callee functions could also contain call sites. SPatch follows the same strategies to interpret the updated function calls in callee functions. Specifically, it symbolically interprets the nested call sites whose arguments are data-dependent on the modified arguments (of the callee function). Note that the commit might update callee functions simultaneously. SPatch does not consider SPs made to callee functions when analyzing the caller functions, as the function outputs of callee functions should not be changed by applying SPs produced by SPatch.

SPatch might be unable to symbolically analyze a few SP candidates. It conservatively discards some candidates if they satisfy the following conditions. First, to symbolically interpret a function call, SPatch needs to statically identify and include the callee function (see more details in §5.4). If SPatch cannot identify a unique callee function (*e.g.*, of an indirect call), it considers any modifications on them unsafe. Second, SPatch considers an SP candidate unsafe if the modified arguments of function calls write to unknown pointers in callee functions. Finally, SPatch considers an SP candidate unsafe upon reaching the predefined threshold (five nested call sites in our implementation) or encountering recursive calls. Similar to Juxta, SPatch unrolls loops once during symbolic execution. SPatch terminates symbolic execution in the aforementioned cases and can analyze 71.91% (16,123 out of 22,420) of CUs in our evaluation.

4.3.3 *Detecting SPs in Non-merge Commits.*

In this step, SPatch detects SPs among the SP candidates.

Comparing Symbolic Expressions SPatch utilizes the `z3` solver to detect SPs by comparing the symbolic expressions of original and updated functions. To ensure 1) the equivalence of function outputs between the original and patched functions and 2) the code changes do not expand the valid input space, SPatch formulates and solves two expressions. The first expression, $O_o{=}{=}O_u$, checks if the function outputs of the original (O_o) and updated (O_u) functions are equal. The second expression, $Implies(PC_u, PC_o)$, checks if the path constraints in the patched function (PC_u) imply those in the original function (PC_o). SPatch then identifies the statements where the violation of C1 or C2 occurs, which are referred to as $Stmt_{invalid}$. $Stmt_{invalid}$ can either be the 1) (expanded) control-dependent statements or 2) return statements or assignments to (updated) function outputs.

Detecting SPs SPatch finally selects SPs from the SP candidates by excluding those that have data dependencies with $Stmt_{invalid}$ identified in the previous step. Since the SP candidates are independent of each other, it could safely exclude arbitrary ones without any concerns. In this way, SPatch separates functionality updates from SPs and answers which code changes are SPs, rather than determining whether all code changes made in a commit are SPs.

4.3.4 *Detecting SPs in Merge Commits.*

We propose an approach to exclusively handle merge commits. As a merge commit (C_m) has multiple parent commits, SPatch first identifies one parent commit (C_p) the merge commit compares with. SPatch then obtains SPs from C_p to C_m by aggregating SPs introduced in other parent commits of C_m. In this way, SPatch avoids repeatedly analyzing code changes that are introduced by non-merge commits and later reappear in merge commits.

However, the code changes in a merge commit are not always equivalent to the aggregated changes of all its parent commits. A typical example is when merge conflicts occur. Resolving these conflicts will require additional adjustments to ensure compatibility. SPatch ignores parents' SPs (*e.g.*, those writing to one file and merging into one commit) that lead to conflicts by not aggregating them, although this causes these SPs to be finally missed.

5 IMPLEMENTATION

We implemented SPatch with 3,830 LoC in Python, Scala, and Java. We discuss some important implementation details below.

5.1 Static Analysis

SPatch performs dependency analysis for the original and patched functions using `Joern` [42]. It computes path constraints by collecting the statements on which the return statements are control-dependent, and performs data-dependency analysis when grouping code modifications into CUs (§4.2) and verifying SPs (§4.3).

SPatch also needs to infer target functions of function calls. To this end, it constructs a call graph for the software using `MLTA` [20]. `MLTA` operates on LLVM-IR code, which is time-consuming to generate for each commit. Therefore, SPatch generates a call graph for a base commit and reuses that call graph when analyzing call relationships of the same program in the subsequent commits.

5.2 Cross-function Mapping

SPatch merges code modifications if they have data dependencies with the same variables. The code modifications and their respective data-dependent statements are located in both the original and patched functions. As a result, identifying the overlapping data-dependent statements becomes challenging since they are distributed across two functions.

To solve this problem, an approach is to combine the deleted and added statements into one function like GraphSPD did [33]. For example, we could produce a new function that includes lines 12-19 when analyzing Listing 1. We found that this approach might not work when the code modifications alter the control structures. For instance, some security patches update a single *if* statement ending with `{`. Applying both the deleted *if* and added *if* statements would produce an invalid function on which `Joern` fails to perform control- and data-dependency analysis.

To synchronize the positions of the statements, SPatch maintains two versions of the functions. We substitute each removed statement (lines 12-13 in Listing 1) with a new line in the updated version and each added statement (lines 14-19) with a new line in the original version. We then locate the overlapped data-dependent statements based on the line numbers.

5.3 Compiling Minified Programs

SPatch compiles a minified program that includes the patched function to validate CUs (§4.2.2) and collect symbolic path constraints (§4.3.2). However, a single function is usually not compilable as it might use identifiers defined outside the function scope. Thus, it is important to include the necessary dependency code before compiling it. Prior works [40] performed control-dependency analysis and taint analysis to identify the dependent code of a function. These approaches have limitations because of the imprecise static analysis.

Our observation is that a C/C++ program typically includes the dependency files using the `#include` directives. Therefore, SPatch starts by locating the file (*e.g.*, `FF.c`) where `func()` is located. It then includes the header files and global identifiers inside `FF.c` to compile `func()`. The challenge of this approach is to provide the compiler with the concrete paths of any custom header files. For instance, the header file `linux/init.h` in the Linux kernel corresponds to the path `linux_dir/include/linux/init.h`. To do so, we run the *make* utility and log the executed command lines for compiling each source file (*e.g.*, `gcc -I./include/linux ... -c FF.c`). We then re-compile the function using the compile options (which specify the locations of dependency files) under the same directory. Note that we only need to log the compilation commands once and reuse them when compiling the functions in other commits.

5.4 Differential Symbolic Execution

SPatch performs DSE on the original function and updated function. Our DSE is built upon `Juxta`, a source-code-level symbolic execution tool that is further built upon `scan-build`. `Juxta` traverses the control flow graph (CFG) from the *entry functions* of a software project and inlines callee functions until its analysis reaches a predefined threshold. The entry functions are the functions that are not called by other functions in the program.

We implemented under-constrained analysis and selective symbolic interpretation analysis upon `Juxta` [25]. We perform under-constrained analysis by reusing the minified program that includes the patched function and its dependency files. Specifically, we generate CFG for the minified program using Clang. When compiling the program, `Juxta` updates the path constraints and symbolic values following its CFG paths. To symbolically interpret the function calls in a patched function, we include the callee functions in the minified program. We then modify `Juxta` to only interpret the function calls we specify. The callee functions would then be inlined when the target function is symbolically executed. Since the callee function might use identifiers that are not defined in the caller function's scope, we also include the header files of the inlined callee functions. Finally, we compare the symbolic expressions in original and patched functions to perform DSE.

6 EVALUATION

In this section, we evaluate SPatch on a large set of commits in real-world software. we first describe the experiment setups used for identifying SPs (§6.1). We then present the evaluation results, *e.g.*, the number of partially-safe commits (§6.2). Next, we discuss the security implications of SPatch by analyzing real-world vulnerabilities and security patches (§6.3), and further showcase its benefits with one end-to-end case study (§6.4). Finally, we discuss the analysis performance (§6.5).

6.1 Experiment Setups

We evaluated SPatch on a set of Git commits. Due to the unavailability of an evaluation dataset for safe patch analysis, we conducted a random selection from Git history of popular software, resulting in a total of continuous 45,296 commits from 11 widely-used code repositories. The dataset spans from 2015 to 2023, ensuring that it includes a diverse representation of Git commits over the years. Our dataset includes a diverse range of 11 software. For instance, we included the Linux kernel that was reused by the Android operating system and the OpenSSL library as it served as the foundation for numerous software applications. The software names and the number of analyzed commits are listed in the first and second columns of Table 1, respectively. We tend to include a higher number of commits in our dataset for projects with a longer commit history. In our experiments, we built the software using the default compilation options on a computer running Debian stretch, equipped with Intel Xeon W-2123 4-core 3.6GHz Processor and 16 GB RAM. We evaluated SPatch's performance on the same computer.

6.2 SPs in Commits

In this subsection, we present the overall results and how each component in SPatch contributes to the results in §6.2.1, then

Table 1: The final results and intermediate results of each stage. $Commits_{total}$, $Commits_{sing}$, and $Commits_{ana}$ denote the number of total, non-merge, and analyzed commits. SP_{cand} denotes the number of SP candidates. SP_{df} and SP_{se} denote the number of SPs that are identified with data-flow analysis and symbolic execution. SP_{it} and SP_{intra} denote the number of SPs identified with and without interpreting call sites. CU_{no} denotes the number of CUs that are identified as non-SPs and $CU_{unknown}$ denotes the number of CUs that remain undetermined.

Software	$Commits_{total}$	$Commits_{sing}$	$Commits_{ana}$	Functions	Edit actions	CUs	SP_{cand}s	SP_{df}s	SP_{se}s		Non SPs	
									SP_{it}s	SP_{intra}s	CU_{no}s	$CU_{unknown}$s
Linux kernel	12,235	11,107	3,824	6,155	11,933	9,326	6,103	488	1,380	873	1,404	1,958
python interpreter	7,111	7,095	1,613	2,935	6,543	4,595	2,721	410	626	250	854	581
PHP interpreter	3,179	2,002	505	812	1,698	1,181	929	155	271	49	344	110
OpenSSL	5,559	5,559	2,099	4,147	9,532	6,668	3,216	353	772	579	514	998
FFmpeg	1,647	1,647	1,116	1,745	3,549	2,440	999	80	117	355	201	246
libpng	3,657	3,597	1,859	3,028	7,275	3,876	3,400	581	358	751	1,317	393
lua	3,065	3,064	2,080	3,534	6,598	5,352	2,549	247	1,011	102	270	919
binutils-gdb	3,212	3,212	2,118	3,987	5,096	4,439	3,013	476	426	183	1,411	517
git	3,183	2,301	1,102	1,762	3,556	2,512	1,549	98	378	107	491	475
libarchieve	1,655	1,282	723	896	1,265	1,052	752	226	115	137	208	76
openVPN	793	792	474	537	708	555	378	75	49	62	168	24
Sum.	45,296	41,658	17,513	29,538	57,753	41,996	25,609	3,189	5,503	3,448	7,172	6,297

provide an ablation study and compare SPatch's performance with a recent work Spider in §6.2.3.

6.2.1 Results. Table 1 provides the final results (*i.e.*, SPs) and intermediate outcomes at each stage (*e.g.*, SP_{df} shows the number of SPs identified in the static analysis phase). SPatch examined 45,296 commits, of which 41,658 (91.97%) were non-merge commits. Among the non-merge commits, 17,513 (38.66% of the total) updated one or more C/C++ functions that were used in the software building process and were subsequently analyzed by SPatch. We checked the commits that were not covered by SPatch and found they were mainly the ones that 1) implemented functionalities (*ext3* in the Linux kernel) that were not enabled by default, 2) were tailored to other computer architectures, or 3) were written in languages other than C/C++ (*e.g.*, text files). Additionally, SPatch did not analyze certain commits even though they updated C/C++ files compiled during the software building process, as these updates only involved global scope statements and not functions.

The fifth column of Table 1 lists the numbers of functions updated in the 17,513 commits. In total, 29,538 C/C++ functions were updated and SPatch extracted 57,753 edit actions from them. After merging the edit actions that had data dependencies into CUs (§4.2), we obtained 41,996 CUs. SPatch filtered out 6,063 CUs that wrote to unknown pointers or updated loop statements, and 10,324 CUs that caused compilation errors. The remaining 25,609 CUs (60.98% of the total CUs) were the SP candidates for the following static and DSE analysis. SPatch detected 3,189 SPs from all these SP candidates using static analysis. For the remaining 22,420 SP candidates, SPatch successfully conducted DSE for 16,123 (71.91%) cases and terminated DSE for the rest (reasons are detailed in §4.3.2). Among the 16,123, SPatch identified 8,951 SPs, resulting in a total of 12,140 SPs.

6.2.2 False Positives and False Negatives. As there is no ground truth dataset, we randomly selected 100 CUs and identified if they were SPs by manually checking C1 and C2. In this way, we estimate SPatch's false positives and false negatives. Some CUs are complex.

Table 2: Ablation study results. Spider represents the baseline, SPatch$_p$ denotes the number of safe and partially-safe commits identified with intra-procedural analysis, and SPatch$_{it}$ denotes the number of safe commits identified with inter-procedural analysis.

Software	Spider	SPatch$_p$	SPatch$_{it}$	SPatch
Linux kernel	736	802	1,007	1,752
python interpreter	243	416	290	792
PHP interpreter	91	152	129	233
OpenSSL	363	401	475	807
FFmpeg	102	179	104	243
libpng	322	453	420	891
lua	374	457	522	631
binutils-gdb	137	377	206	886
git	97	116	154	209
libarchieve	238	260	251	296
openVPN	109	163	127	219
total	2,812	3,776	3,684	6,959

We were able to accurately analyze 92 CUs and found that 29 CUs were SPs (the ground truth). SPatch detected 24 SPs with no false positives and 5 false negatives. We emphasize the importance of no false positives (no unsafe code changes are detected as safe) as this enhances SPatch's potential usability.

6.2.3 Ablation Study. In this subsection, we present the ablation study of each component in SPatch. We specifically evaluate the benefits of considering partially-safe commits and analyzing modifications to external calls. We also use Spider, a state-of-the-art tool for analyzing the functional impacts of commits, as the baseline for comparison. Despite our attempts to obtain the code and dataset from the authors, we have not been successful in acquiring them. Nonetheless, our ablation study allows us to replicate Spider's design. We performed *intra-procedural* symbolic execution on all updated functions within a commit. By counting the number of commits where *all* code modifications to C/C++ files are SPs, we obtained the results of Spider. Note that we did not include Upgradvisor as it 1) analyzed Python projects rather than C/C++

projects and 2) involved manual examination and was difficult to scale to the level of many (17,513) commits in our dataset.

The results of Spider are listed in the second column of Table 2. In total, our implementation of Spider (*i.e.*, the baseline) identified 2,812 (16.06% of) commits as SPs (or safe commits). In contrast, SPatch identified 6,959 commits (2.47× of Spider) that partially or solely included SPs, as demonstrated in the fifth column of Table 2.

To better understand SPatch's performance, we first evaluate the benefits of analyzing modifications to function calls. In the fourth column, we listed the results of SPs detected by SPatch$_{it}$. SPatch$_{it}$ identified more (1.31× than Spider) SPs because of its ability to symbolically interpret function calls in these commits.

In addition to inter-procedural analysis, SPatch also conducted a fine-grained identification of SPs. This allowed SPatch to identify partially-safe commits. SPatch$_p$ in Table 2 denotes the sum of partially-safe commits and safe commits identified by SPatch using intra-procedural analysis. In addition to 2,812 safe commits, the fine-grained analysis alone enabled SPatch$_p$ to detect 964 (34.28% of safe commits) partially-safe commits.

Our ablation study stressed the importance of combining the two techniques—considering partially-safe commits and symbolically interpreting function calls—to effectively identify SPs. SPatch detected more (partially-)safe commits (1.84 times of SPatch$_p$ and 1.90 times of SPatch$_{it}$) compared to using a single technique. The results indicated that many SPs involving modifications to function calls were code snippets committed alongside other code modifications. Listing 1 is such an example.

6.3 Security Implications

In this subsection, we evaluate the security benefits of SPatch by comprehensively detecting SPs. We also compare our approach with Spider regarding the security implications of SPs.

6.3.1 Security Patches. We evaluate the security benefits of SPatch. To this end, we randomly selected 100 non-merge commits from each software project in the evaluation dataset, resulting in 1,100 commits. We used SPatch to identify SPs in those commits and counted the number of security patches among the SPs.

Since there is no ground truth of security patches in commits, we analyzed each commit to pinpoint the included security patches if any. We first identified the commits that fixed vulnerabilities by checking if 1) they were the patches for fixing known CVEs, 2) they were in the security patch database [35], or 3) the commit messages explicitly mentioned that the commits fixed security vulnerabilities. We then located security patches in those commits based on the vulnerability information, *e.g.*, if the code sanitized variables that might cause overflow. A commit might fix multiple vulnerabilities (*e.g.*, it adds the same input validation check in multiple vulnerable functions). We consider multiple fixes in one commit as one security patch, and SPatch is considered to have found an incomplete security patch if it detects at least one but not all fixes in that commit. In total, we identified 52 security patches. We believe the volume of security patches used as our dataset is sufficiently large, as related works on security patch detection, such as [33], use similar numbers.

We list the analysis results in Table 3. Overall, SPatch determined 40 safe security patches (including 5 incomplete security patches) among 52 (the ground truth of total security patches), achieving a recall rate of 76.92% for all security patches. Yet Spider only pinpointed 21 (40.38%) safe security patches due to the false negatives in its design. Upon further investigation, we found that among the security patches detected by SPatch but missed by Spider, 15 involved modifications to function calls, and 7 were implemented as partially-safe commits (3 of which were partially-safe commits involving modifications to function calls).

Table 3: Evaluation of security patches on 1,100 commits. Spider$_{SP}$ denotes the number of safe commits identified by Spider. SPatch$_{SP}$ denote the number of safe and partially-safe commits identified by SPatch. Spider$_{sec}$ and SPatch$_{sec}$ denote the number of commits that include security patches among Spider$_{SP}$ and SPatch$_{SP}$, respectively. T$_{sec}$ denotes the total number of commits that include security patches.

Software	Spider$_{SP}$	Spider$_{sec}$	SPatch$_{SP}$	SPatch$_{sec}$	T$_{sec}$
Linux kernel	13	2	33	4	5
python interpreter	7	1	17	2	3
PHP interpreter	12	4	23	7	10
OpenSSL	12	2	20	3	5
FFmpeg	8	1	14	1	2
libpng	5	0	16	0	1
lua	18	3	25	6	6
binutils-gdb	21	2	27	5	7
git	12	1	27	4	4
libarchieve	11	2	19	2	2
openVPN	9	3	29	6	7
total	128	21	250	40	52

We also investigated the reasons for the security patches that SPatch failed to detect. First, we did not detect some security patches that involved global scope edits or directly updated the statements in loops. Second, a few security patches were not SPs. For instance, the patch to CVE-2016-10087 fixed a NULL pointer dereference by updating one of the function outputs. These security patches might affect functionalities and whether deploying them or not became a trade-off for maintainers.

6.4 End-to-end Case Studies

In this subsection, we demonstrate SPatch's capabilities in assisting patch propagation with two case studies.

6.4.1 Redis. Redis is a popular in-memory database. It currently uses Lua v5.1 as one of its dependencies. We updated 298 functions in Lua using the detected SPs from version v5.1 to the latest version, aligning the syntax modifications in Redis as necessary. Our updated software successfully passed *all* test suites (89 tasks in total). Note that we failed to build Redis by updating Lua to the latest version. The compatibility issues of using different versions of Lua were observed in other downstream projects like Wireshark [4].

We investigated the security benefits of applying SPs to update Lua. SPatch identified SPs that fixed nine CVEs, while Spider, by design, only found three of them. Although Redis developers ported five, four were still not propagated to Redis. We submitted those four safe patches. The maintainers' investigation revealed that three patches were associated with Lua components that were not utilized in Redis. They prioritized merging one into the latest version.

6.4.2 ProtonVPN. ProtonVPN is a VPN service software project with more than 10 million downloads on Google Play [5]. We found that the software was built upon OpenVPN, which utilized an outdated version of OpenSSL. It might not be realistic to upgrade the OpenSSL library because of the potential compatibility issue [26]. We employed SPatch to identify the SPs in OpenSSL. We updated in total 167 functions in ProtonVPN by merging SPs in 2,268 commits. After manually porting these SPs, we built the software successfully and used it to connect to a VPN server without meeting any issues. All four detected safe security patches have been merged into ProtonVPN, and we received a bounty for addressing vulnerabilities in the outdated dependency code of ProtonVPN.

6.5 Performance

In the subsection, we discuss the analysis time of SPatch from two perspectives, at the commit level and the CU level.

6.5.1 Commit-level Performance. SPatch spent a total of 144.05 hours detecting 12,140 SPs among the 45,296 commits, covering the entire process from fetching these commits to obtaining SPs. The time used in different commits varied a lot, depending on the code changes actually analyzed by SPatch. Further details on this variation and performance breakdown are discussed in §6.5.2.

Next, we describe the time used for analyzing merge commits and non-merge commits. Merge commits, especially in projects like the Linux kernel, could include code changes spanning over thousands of C/C++ files. Performing program analysis (*e.g.*, generating and analyzing CUs) on these commits was time-consuming. For example, we ran program analysis on one commit updating over 5,000 C/C++ files in the Linux kernel, and the analysis was not finished within 1 hour. This motivated the methods described in §4.3.4, where we could obtain SPs in a merge commit C_m quickly by merging the SPs in C_m's other parent commits. However, certain non-merge commits can also update many C/C++ files. To handle such complex non-merge commits, we have set a limit, allowing the analysis of a maximum of 500 updated functions per commit (after excluding the functions not used in the compilation). In cases where the significant code changes within a non-merge commit exceeded this threshold, SPatch moved on and fetched the subsequent commit without completing the analysis of the current commit. Our experiments indicated that this was uncommon (< 1%), occurring only during evaluating a few major updates.

6.5.2 CU-level Performance. Since we analyzed SPs on a per CU basis, the number of CUs directly affected the analysis time. Following our design, we break down the analysis time into two phases, generating CUs and detecting SPs from CUs.

It took SPatch 35.92 hours to generate 41,996 CUs from all commits. The average time to generate a CU was 3.1 seconds, which included the time used to generate edit actions (<1 second) and static data-flow analysis (>2 seconds). During the data-flow analysis, parsing a C/C++ function into Code Property Graphs (CPGs) took more than 1 second. SPatch was able to identify SP candidates among CUs in negligible time. It reused CPGs to perform the data-flow analysis needed in §4.2.2. Note that during compatibility testing, SPatch used the GCC compiler instead of Juxta.

SPatch spent 108.13 hours detecting SPs among the 25,609 candidates (15.20 seconds per candidate on average). The time used for different candidates also varied. For the candidates that did not go through symbolic execution, SPatch could quickly identify if they were SPs. For each remaining candidate, the analysis time involved two steps: 1) compiling the original and updated functions using the modified Juxta and 2) using the z3 solver to solve constraints (5 seconds as the time limit). Using our on-demand symbolic execution, computing symbolic expressions of path constraints and function outputs for a function often took less than 7 seconds. This duration included: 1) the compilation phase where symbolic execution took place and 2) scanning the symbolic execution results produced by the modified Juxta and stored in files.

7 DISCUSSION

Threats to Validity In this work, we have employed techniques to identify partially-safe commits, allowing us to detect more SPs. One may question the security benefits of partially-safe commits as they may not completely fix vulnerabilities. For instance, a security patch commit may address vulnerabilities in multiple functions, but SPatch only identifies safe patches in some functions, leaving the vulnerabilities unfixed in the remaining functions. We report that among the 40 safe security patches identified in §6.3.1, five were incomplete patches due to this reason. Nevertheless, we advocate the usage of SPatch given applying its uncovered patches generally reduces the attack surface to a large extent.

Comparison with Upgradvisor SPatch shares similar goals (*i.e.*, porting patches that preserve functionality to downstream) as Upgradvisor, but focuses on different aspects. Upgradvisor reduces human overload to investigate the functional impacts of patches. In contrast, SPatch represents an automated technique to infer the patches likely to preserve functionality. As an automated method, SPatch cannot guarantee that the detected SPs preserve functionality as expected by humans. Nevertheless, it has the benefit of scaling to a large number of commits without requiring manual analysis of runtime traces. Additionally, SPatch and Upgradvisor operate in distinct domains (C/C++ and Python, respectively). Therefore, we believe that the two works have advantages in different scenarios and distinct application scopes.

8 RELATED WORK

Supply Chain Security. Many works studied the security issues in supply chain systems. Xie *et al.* [11] and Reid *et al.* [29] found that many projects reused out-of-date code containing security vulnerabilities. Ishio *et al.* [14] and Woo *et al.* [39] further revealed that most (95% of) open-sourced components were reused with some modifications in downstream. A few works also detected (*e.g.*, [16, 38, 41]) and exploited ([7, 8]) the vulnerabilities in the unpatched code. For example, VUDDY detected vulnerabilities introduced by code clones [16]. MVP [41] detected recurring vulnerabilities by matching vulnerability and patch signatures. VulScope [8] aligned PoCs to exploit vulnerabilities in unpatched versions.

Maintaining third-party libraries requires non-trivial human effort. In this work, we comprehensively detect and port safe patches to address security issues in outdated dependency code.

Patch Analysis. Some works analyze the code changes in upstream software to assist patch propagation. Many researchers distinguished the security patches from other code changes [31, 33, 40]. For example, GraphSPD [33] and PatchRNN [36] trained a deep learning model to detect security patches in C/C++ programs, and Tian *et al.* [31] utilized textual features to identify security patches in Linux. Wang *et al.* [37] leveraged random forest to classify security patches into specific types. Soto *et al.* [30] conducted a large-scale study on Java security patches and provided insights into automated code repair in Java programs. In contrast, Spider [24] prioritizes functionality over security. It enables downstream developers to use or prioritize investigating some portable patches.

9 CONCLUSION

This work demonstrates the importance of detecting fine-grained SPs for efficient patch porting. To achieve this, we have developed SPatch, which incorporates several techniques. First, SPatch analyzes code changes at a CU granularity, distinguishing itself from many prior works focusing on commits. Second, we propose a new DSE technique that efficiently analyzes the functional impacts of code changes. Our evaluation results show that SPatch provides more security benefits compared to prior tools. The two case studies further demonstrate that SPatch brings observable security benefits without incurring regression issues.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful suggestions and comments. The work described in this paper was partly supported by a grant from the Research Grants Council of the Hong Kong SAR, China (Project No.: CUHK 14209323). Shuai Wang was supported in part by contracts RMGS23CR17 and RMGS24EG02.

REFERENCES

[1] 2020. *Proceedings of the 29th USENIX Security Symposium (Security).* Virtual Event.
[2] 2022. *Proceedings of the 43nd IEEE Symposium on Security and Privacy (Oakland).* San Francisco, CA, USA.
[3] 2022. *Proceedings of the 44th International Conference on Software Engineering (ICSE).* Pittsburgh, PA, USA.
[4] 2023. Lua version used for wireshark dissectors. https://github.com/o-gs/dji-firmware-tools/issues/153.
[5] Proton AG. 2023. Proton VPN. https://play.google.com/store/apps/details?id=ch.protonvpn.android&utm_campaign=ww-all-2a-vpn-int_site-g_acq-apps_links_free_vpn_page&utm_source=protonvpn.com&utm_medium=link&utm_content=free_vpn_page&utm_term=android&pli=1.
[6] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* Sacramento, CA, USA.
[7] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *2008 IEEE Symposium on Security and Privacy (sp 2008).* Oakland, CA, USA.
[8] Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. 2021. Facilitating vulnerability assessment through poc migration. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS).* Virtual Event, Korea.
[9] Yaniv David, Xudong Sun, Raphael J Sofaer, Aditya Senthilnathan, Junfeng Yang, Zhiqiang Zuo, Guoqing Harry Xu, Jason Nieh, and Ronghui Gu. 2020. {UPGRADVISOR}: Early Adopting Dependency Updates Using Hybrid Program Analysis and Hardware Tracing. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* Carlsbad, CA, USA.
[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering.* 313–324.
[11] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. 2017. Some from here, some from there: Cross-project code reuse in github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR).* IEEE, 291–301.
[12] Google. 2023. Bad Binder: Android In-The-Wild Exploit. https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html.
[13] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using safety properties to generate vulnerability patches. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland).* San Francisco, CA, USA.
[14] Takashi Ishio, Yusuke Sakaguchi, Kaoru Ito, and Katsuro Inoue. 2017. Source file set search for clone-and-own reuse analysis. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR).* IEEE, 257–268.
[15] Zheyue Jiang, Yuan Zhang, Jun Xu, Xinqian Sun, Zhuang Liu, and Min Yang. 2022. AEM: Facilitating Cross-Version Exploitability Assessment of Linux Kernel Vulnerabilities, See [2].
[16] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland).* San Jose, CA, USA.
[17] Klee. 2023. KLEE Symbolic Execution Engine. KLEESymbolicExecutionEngine.
[18] Nir Kshetri and Jeffrey Voas. 2019. Supply chain trust. *IT Professional* 21, 2 (2019), 6–10.
[19] Yuxiang Lei and Yulei Sui. 2019. Fast and precise handling of positive weight cycles for field-sensitive pointer analysis. In *Static Analysis: 26th International Symposium, SAS 2019, Porto, Portugal, October 8–11, 2019, Proceedings 26.* Springer, 27–47.
[20] Kangjie Lu and Hong Hu. 2019. 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.* 1867–1881.
[21] Lua. 2015. Potential arithmetic overflow in Lua. https://clang-analyzer.llvm.org/scan-build.html.
[22] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE).* Hong Kong.
[23] Yunlong Lyu, Yi Fang, Yiwei Zhang, Qibin Sun, Siqi Ma, Elisa Bertino, Kangjie Lu, and Juanru Li. 2022. Goshawk: Hunting Memory Corruptions via Structure-Aware and Object-Centric Memory Operation Synopsis, See [2].
[24] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. 2020. Spider: Enabling fast patch propagation in related software repositories. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland).* San Francisco, CA, USA.
[25] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP).* Monterey, CA, USA.
[26] openvpn. 2023. Compatiblity issues in OpenVPN. https://forums.openvpn.net/viewtopic.php?t=35028.
[27] Suzette Person, Matthew B Dwyer, Sebastian Elbaum, and Corina S Pǎsǎreanu. 2008. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering.* 226–237.
[28] Redis. 2023. Redis. https://github.com/redis/redis/blob/unstable/deps/README.md.
[29] David Reid, Mahmoud Jahanshahi, and Audris Mockus. 2022. The extent of orphan vulnerabilities from code reuse in open source software, See [3].
[30] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. 2016. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *Proceedings of the 13th International Conference on Mining Software Repositories.* 512–515.
[31] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying linux bug fixing patches. In *Proceedings of the 34th International Conference on Software Engineering (ICSE).* Zurich, Switzerland.
[32] Yuchi Tian and Baishakhi Ray. 2017. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 11th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* Paderborn, Germany.
[33] Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. 2022. GraphSPD: Graph-Based Security Patch Detection with Enriched Code Semantics, See [2].
[34] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2020. An empirical study of secret security patch in open source software. *Adaptive Autonomous Secure Cyber Systems* (2020), 269–289.
[35] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. 2021. Patchdb: A large-scale security patch dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE, 149–160.

[36] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, Sushil Jajodia, Sanae Benchaaboun, and Frank Geck. 2021. Patchrnn: A deep learning-based system for security patch identification. In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*. IEEE, 595–600.

[37] Xinda Wang, Shu Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2020. A machine learning approach to classify security patches into vulnerability types. In *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 1–9.

[38] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. 2022. {MOVERY}: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified {Open-Source} Software Components. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*. Los Angeles, CA, USA.

[39] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. 2022. CENTRIS: A precise and scalable approach for identifying modified open-source software reuse, See [3].

[40] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. 2020. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[41] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, and Feng Li. 2020. MVP : Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures, See [1].

[42] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA, USA.

[43] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. 2020. An Investigation of the Android Kernel Patch Ecosystem., See [1].

[44] Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. 2021. Spi: Automated identification of security patches via commits. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–27.

[45] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. 2022. {SyzScope}: Revealing {High-Risk} Security Impacts of {Fuzzer-Exposed} Bugs in Linux kernel. In *Proceedings of the 31st USENIX Security Symposium (Security)*. Boston, MA, USA.