Backporting Security Patches of Web Applications:
A Prototype Design and Implementation on Injection Vulnerability Patches

Youkun Shi[1,¶], Yuan Zhang[1,¶], Tianhan Luo[1], Xiangyu Mao[1], Yinzhi Cao[2], Ziwen Wang[1], Yudi Zhao[1], Zongan Huang[1], and Min Yang[1]

[1]*School of Computer Science, Fudan University, China*
[2]*Department of Computer Science, Johns Hopkins University, USA*
[¶]*co-first authors*

Abstract

Web vulnerabilities, especially injection-related ones, are popular among web application frameworks (such as WordPress and Piwigo), which can lead to severe consequences like user information leak and server-side malware execution. One major practice of fixing web vulnerabilities on real-world websites is to apply security patches from the official developers of web frameworks. However, such a practice is challenging because security patches are developed for the latest version of a web framework, but real-world websites often run an old version due to legacy reasons. A direct application of security patches on the old version often fails because web frameworks, especially the code around the vulnerable location, may change between versions.

In this paper, we design a security patch backporting framework and implement a prototype on injection vulnerability patches, called SKYPORT. SKYPORT first identifies safely-backportable patches of injection vulnerabilities and web framework versions in theory and then backports patches to corresponding old versions. In the evaluation, SKYPORT identifies 98 out of 155 security patches targeting legacy injection vulnerabilities, which can be backported to 750 old versions of web application frameworks. Then, SKYPORT successfully backported all of the aforementioned backportable patches to corresponding old versions to correctly fix vulnerabilities. We believe that this is a first-step towards this important research problem and hope our research can draw further attention from the research community in backporting security patches to fix unpatched vulnerabilities in general beyond injection-related ones.

1 Introduction

Patching is a common practice to apply software code differences between two versions for an update. While patching continues to be the major tactics of fixing web application vulnerabilities, the major challenge in the real-world is practical deployment. For example, after three months since the patch release of CVE-2018-7600, an arbitrary code execution vulnerability of Drupal, researchers still found that around 115,000 websites were unpatched, making them vulnerable to any attackers on the web [2].

The reasons for leaving vulnerable websites unpatched are complicated. Putting aside these human-related issues, one major problem is that the patch cannot be directly applied to vulnerable websites due to a web application version mismatch. Specifically, web application developers, in most cases, just come up with a patch for the latest application version, but website maintainers often run an old version due to legacy reasons, e.g., the application does not have an automated update interface or the website has some customized code tailored made for the old version. Therefore, one practical problem facing the website maintainer is to apply the patch targeting the latest version upon their own website in an old version. According to several prior studies [65, 66], they are often reluctant of doing so due to lacking technical experience and being afraid of breaking website functionalities.

Backporting security patches. The practical issue on cross-version patch deployment can be formalized as a research problem, i.e., how to backport a given security patch for a particular vulnerability to a target, old version of the web application, or for short, we call it *security patch backporting*. The general goals of security patch backporting are to maintain two important properties of web application: (i) security, i.e., defending against the vulnerability-related attacks, and (ii) backward compatibility, i.e., incurring no functionality issues.

Although the security patch backporting problem is important, surprisingly there is not much work in the literature to solve the problem. Specifically, there is a patent [51] on bug patch backporting, but it can only fix earlier bugs if no patching conflicts exist. A Ph.D. dissertation [69] from Singapore Management University discusses patch backporting, but the process is still largely manual and without security or functionality guarantees.

While the specific problem to backport security patches is not thoroughly studied in the past, people have studied the general problem of vulnerability detection and patch-

ing. Based on our study, none of them can be used to backport security patches. One popular research direction is the detection of zero-day security vulnerabilities via code similarity [34,37,44,45]. For example, existing works propose patch-enhanced vulnerability detection [74], but they cannot pinpoint the accurate vulnerability location for patching let alone ensure the security and functionality of backporting. Another research direction is automatic fixes of existing vulnerabilities [26,27,31,36,43,47,48,73]. For example, many researches can automatically generate patches given exploit code [31] or many normal test cases [36,48,73], which may not exist for the backporting problem. For another example, hot-patching frameworks [18,19,56] aim to apply the patching semantics to the vulnerable code without requiring users to explicitly updating their software, while the backporting problem does not assume the existence of a patch for an old version.

Let us describe what a security patch is and why backporting is feasible for many old versions of web applications. Intuitively, a security patch is to update the vulnerable logic for given inputs in a web application with a safe logic. Then, backporting is possible because the same vulnerable logic may also exist in a target old version of the web application. Therefore, we can apply the safe logic from the security patch to the old version and replace the vulnerable logic.

While security patch backporting is intuitively simple, the challenges come from two aspects, the patch, and the target old version. On one hand, the patch may not just update the vulnerable logic but also introduce other new functionalities, e.g., adding new inputs. Such new functionalities may not work on the target old version, leading to compatibility issues. More importantly, they also bring difficulties in pinpointing the exact code location for applying the patch on the target old version. On the other hand, the target version may not contain exactly the same vulnerable logic as the patch aims to fix. For example, although the target version is also vulnerable, some later versions added new inputs to the vulnerable logic and thus the safe logic provided by the patch is not applicable to the target version.

In this paper, we propose two novel concepts, called a *safely-backportable patch* (SBP) and a *safely-backportable version* (SBV), to tackle the aforementioned two challenges. An SBP is a newly-generated patch that only contains safe, deterministically-computable safe logics extracted from the original patch to replace the vulnerable logics; and an SBV is a special target version that has the same vulnerable logics targeted by the SBP. The combination of SBP and SBV achieves the backporting goals of security and backward compatibility, because an SBP removes vulnerability-unrelated fixes and an SBV ensures that the vulnerable logic in the target is either exactly the same as the one targeted by the original patch.

Specifically, our proposed backporting with SBP and SBV is a three-pronged approach. First, it determines whether the original patch is backportable and then generates an SBP from the post-patch version with the original patch. Second, it checks the target version and ensures that the version is an SBV. Lastly, it pinpoints the vulnerability location in the target version and applies the SBP.

SKYPORT: Injection vulnerability patch backporting framework. While the three-pronged backporting is intuitive and effective, the major challenge is how to represent both the safe and vulnerable logic of a web application in a formal, comparable way. Fortunately, a popular type of web application vulnerabilities, i.e., injection-related ones, has one explicitly-defined dangerous function called the sink, such as `echo` for cross-site scripting (XSS) and `move_uploaded_file` for arbitrary file upload. More importantly, the same sink function usually exists in all three related versions, i.e., pre-patch, post-patch, and target. This practical observation enables our design of SKYPORT, a general framework to automate security patch backporting for injection vulnerabilities in legacy web applications.

Given this practical observation, SKYPORT represents the safe and vulnerable logic of web applications as a concept called sink capability. Specifically, a sink capability is a set of pairs that represents all possible data-flows to the vulnerable parameter of the sink function. The first item of a pair is a symbolic expression of one data-flow to the sink function parameter and the second item is the control-flow constraint set of the data-flow. The major advantage of this representation of vulnerable and safe logic as a sink capability is an easy comparison among different versions.

We implemented a prototype of SKYPORT and evaluated it using 155 patches against real-world CVEs from ten CMS. SKYPORT successfully verifies that 111 original patches can be exported to SBPs and that 750 versions are SBVs, which map back to 98 out of 111 SBPs. Then, SKYPORT successfully applies all 98 SBPs on 750 SBVs with backward compatibility and minor performance overhead. As a comparison, we find that the original patches can only be applied on 455 target versions directly without any code conflict. Furthermore, only 39 versions can be fixed by upgrading the website using the official upgrade API with PHP requirement guarantee. These results clearly demonstrate the benefits of SKYPORT.

In all, this paper makes the following major contributions.

- We propose *safely backportable patch* as a new perspective to deploy security patches on web applications. Our solution supports patching across multiple versions, provides the strong guarantee of security and functionality, and requires minimal deployment efforts.
- We design SKYPORT, which can verify whether a security patch can be transformed to an SBP and whether it can be applied to a given version, called SBV.
- We evaluate SKYPORT with 155 real-world vulnerabilities, and generates 98 SBPs that can be applied to 750 SBVs. The results show that SBPs effectively prevents vulnerability exploitation with minor performance overhead and does not affect the normal execution.

```
1    <?php
2  + require_once("../globals.php");
3  + require_once($GLOBALS['srcdir'] . "/patient.inc");
4
5    use OpenEMR\Core\Header;
6
7  - include_once("../globals.php");
8  - include_once($GLOBALS['srcdir'] . "/patient.inc");
9    $template_dir = $GLOBALS['OE_SITE_DIR'] . "/letter_templates";
10   ...
11 - $fh = fopen("$template_dir/".$_GET['template'], 'r');
12 + $fh = fopen("$template_dir/" .
                 convert_safe_file_dir_name($_GET['template']), 'r');
13   ...
14   ?>
```

a) Official Patch for CVE-2018-10572 on OpenEMR 5.0.0.6

```
1  function convert_safe_file_dir_name($label){
2    return preg_replace('/[^A-Za-z0-9_.-]/', '_', $label);
3  }
4
5  function safe_fopen($bp_globals_oe_site_dir, $bp_get_template){
     $template_dir = $bp_globals_oe_site_dir . "/letter_templates";
6    return fopen("$template_dir/" .
7            convert_safe_file_dir_name($bp_get_template), 'r');
   }
```

b) Safely Backportable Patch for CVE-2018-10572 (OpenEMR)

```
1    <?php
2  + $bp_get_template = $_GET['template'];
3  + $bp_globals_oe_site_dir = $GLOBALS['OE_SITE_DIR'];
4    $sanitize_all_escapes  = true;
5    $fake_register_globals = false;
6
7    include_once("../globals.php");
8    include_once($GLOBALS['srcdir'] . "/patient.inc");
9    $template_dir = $GLOBALS['OE_SITE_DIR'] . "/letter_templates";
10   ...
11 - $fh = fopen("$template_dir/".$_GET['template'], 'r');
12 + $fh = safe_fopen($bp_globals_oe_site_dir, $bp_get_template);
13   ...
14   ?>
```

c) Deployed Safely Backportable Patch on OpenEMR 5.0.0.5

Figure 1: Motivating Example of Patching CVE-2018-10572.

2 Overview

In this section, we first illustrate a motivating example and then describe the threat model of SKYPORT.

2.1 A Motivating Example

Figure 1 (a) shows an arbitrary file read vulnerability (CVE-2018-10572) in OpenEMR before version 5.0.1 [5] and its original patch is shown as the diff format. Line 11 is the vulnerable location, which reads a file with user inputs `$_GET['template']` without sanitization and Line 12 is the key patched code of the vulnerability by sanitizing the user input. Note that the developer only fixes the latest version but not prior ones. Here we explain why it is challenging to backport this security patch. First, the patch not only contains the code to fix the vulnerability (i.e., Line 12) but also other vulnerability-irrelevant code (i.e., the addition of Lines 2 and 3 and the deletion of Lines 7 and 8), which may bring backward compatibility issues. Second, it is challenging to apply the patch to a target version 5.0.0.5 as shown in Figure 1 (c), because Line 5 in Figure 1 (a) for pinpointing the patch location has not been introduced in the target version.

Table 1: Dataset for CVEs with their vulnerability types.

Vulnerability Type	# of CVEs
Server-Side XSS	96
SQL Injection	21
Arbitrary File Read/Write/Delete/Include	14
Command/Code Injection	7
Open Redirect	4
Directory Traversal	4
Executable File Upload	4
Server-Side Request Forgery (SSRF)	4
PHP Object Injection	1
Total	155

Next, we describe how SKYPORT backports the patch from OpenEMR 5.0.0.6 to 5.0.0.5. First, SKYPORT generates a safely-backportable patch (`SBP`) as shown in Figure 1 (b), which contains the entire safe logic including the `convert_safe_file_dir_name()` function and removes vulnerability-irrelevant code. Second, SKYPORT checks that the version 5.0.0.5 is a safely-backportable version (`SBV`) because the vulnerable logics of both versions, i.e., those represented as sink capabilities at Line 11 of Figure 1 (a) and (c), are the same. Lastly, SKYPORT applies the generated `SBP` to `SBV` in Figure 1 (c) by adding two backup code (Lines 2 and 3) and replacing Line 11 with Line 12.

2.2 Threat Model

In this subsection, we describe the threat model of SKYPORT. SKYPORT assumes that the victim is a website running an old-version web application without a targeted patch for a vulnerability. The adversary is a normal user of the web application that sends attack requests to exploit the vulnerability. Broadly speaking, security patch backporting could cover any server-side web vulnerabilities. At the same time, as a pioneer work, we consider *injection vulnerabilities* as in-scope, which include the following two factors.

- *Sink function.* A sink function is a server-side dangerous function. Take server-side XSS for example. The sink function is `echo` because if its input contains user-controlled contents and is not sanitized, arbitrary HTML and JavaScript code may be outputted to the client.
- *Critical parameter.* A critical parameter is a parameter of the sink function that may lead to dangerous behavior. The first and only parameter of `echo` is a critical parameter.

For a better understanding, we also list all the studied injection vulnerabilities and the corresponding total number of historical CVEs in Table 1.

3 Backporting Security Patches

In this section, we describe how to represent vulnerable logics of web applications and then the three-pronged backporting. In each subsection, we first generalize our approach on any security patches and then narrow them down to injection ones.

3.1 Vulnerable Logic Representation

The vulnerable logic of a web application, from a high level, can be represented as all the control- and data-flows that are related to given user inputs. That is, a well-crafted, malicious input may cause execution of the vulnerable program following specific control- and data-flow paths, leading to a malicious consequence. Note that the representation of all control- and data-flow paths for given inputs is a generally hard problem of static analysis, and that is why we focus on injection vulnerabilities as a start of backporting.

3.1.1 Vulnerable Logic for Injection Vulnerabilities

In this part, we describe vulnerable logic representation for injection vulnerabilities. Because injection vulnerabilities have a sink and its corresponding critical parameter, all the control-flows related to the vulnerable logic end up with the sink and all the data-flows end up with the critical parameter. If we call all control-flow paths leading to the sink function as sink flows and represent them as $(flow_1, flow_2, ...)$, we can simplify the general vulnerable logic representation with the following two definitions for a given sink flow $flow_k$.

- *Reaching condition (RC).* A RC_{flow_k} is a set of all the control-flow conditions involved in a $flow_k$.
- *Data-flow expression (DE).* A DE_{flow_k} is a symbolic expression of the critical parameter involved in a $flow_k$.

Then, we put all *RC*s and *DE*s of different sink flows together and define the union set ($\{< RC_{flow_1}, DE_{flow_1} >, < RC_{flow_2}, DE_{flow_2} >, ...\}$) as a sink capability, which represents the vulnerable logic of an injection vulnerability.

Let us consider a simple code snippet below as an example in illustrating sink capability.

```
if ($condition) $value = $input1 else $value = $input2
sink_func($value + 1)
```

There are two sink flows: one from the *if* branch ($flow_1$) and the other the *else* branch ($flow_2$). RC_{flow_1} is `{$condition}` and DE_{flow_1} is `$input1 + 1`; and RC_{flow_2} is `{!$condition}` and DE_{flow_2} is `$input2 + 1`. Then, we can come up with the sink capability based on all the *RC*s and *DE*s, which is an efficient, easy-to-compare representation of vulnerable logic.

3.2 Three-pronged Backporting

In this subsection, we describe our three-pronged backporting approach. We start from two novel concepts, `SBP` and `SBV`.

- A Safely Backportable Patch (`SBP`) is a patch that contains the entire, deterministically-computable safe logic by *restricting* the original vulnerable logic without adding new functionalities. We denote the property of `SBP` as P_{SBP}, which ensures backward compatibility of the web application with backported patches.
- A Safely Backportable Version (`SBV`) is a target old version that contains exactly the *same* vulnerable logic as the original official patch is targeting. We denote the property of `SBV` as P_{SBV}, which ensures the security of the web application with backported patches.

Now, let us focus on injection vulnerabilities and describe how to use sink capability to check both P_{SBP} and P_{SBV} in our three-pronged approach. We have three versions here: a pre-patch (*pre*), a post-patch(*post*), and a target old (*target*).

Step I: `SBP` Verification and Generation. In this step, we need to check whether the original, official patch is backportable and convert it into an `SBP`. Specifically, we check the following sub-properties based on sink capability.

- P_{SBP}*-a:* $RC^{post}_{flow_k}$ *is a subset of* $RC^{pre}_{flow_k}$ *for every* $flow_k$. This property prevents `SBP`s from introducing unknown control-flows to the post-patch version, because it may break the functionality of the target version.
- P_{SBP}*-b:* $\{x \mid x = DE^{post}_{flow_k}\}$ *is a subset of* $\{x \mid x = DE^{pre}_{flow_k}\}$ *for every* $flow_k$. This property prevents `SBP`s from introducing unknown data expressions to the post-patch version, because it may also break the functionality of the target version.
- P_{SBP}*-c: Both* $RC^{post}_{flow_k}$ *and* $DE^{post}_{flow_k}$ *are deterministically computable for every* $flow_k$. This property ensures the functionality of the target version when `SBP` includes the entire safe logics in recomputing all *RC* and *DE*. In other words, this property excludes functions that may return non-deterministic values. e.g. `time()` and `rand()`.

These three sub-properties guarantee that the deployment of the an `SBP` does not affect the normal functionality of the target application, because it does not introduce new inputs to the sink function.

Step II: `SBV` Verification. In this step, we check whether the target version is an `SBV` using the following sub-properties based on sink capability.

- P_{SBV}*-a:* $RC^{pre}_{flow_k}$ *is the same as* $RC^{target}_{flow_k}$ *for every* $flow_k$. This property ensures that there are no additional vulnerable control-flows in the target version.
- P_{SBV}*-b:* $DE^{pre}_{flow_k}$ *is the same as* $DE^{target}_{flow_k}$ *for every* $flow_k$. This property ensures that there are no additional vulnerable data expressions in the target version.

These two sub-properties guarantee the security when applying an `SBP` upon an `SBV`, because the vulnerable logics between two versions are exactly the same.

Step III: Patch Deployment. This step is to automatically deploy `SBP` upon `SBV` for backporting. Although this is hard for general web vulnerabilities, the deployment for injected vulnerabilities boils down to pinpoint the target vulnerable sink function based on sink capability matching.

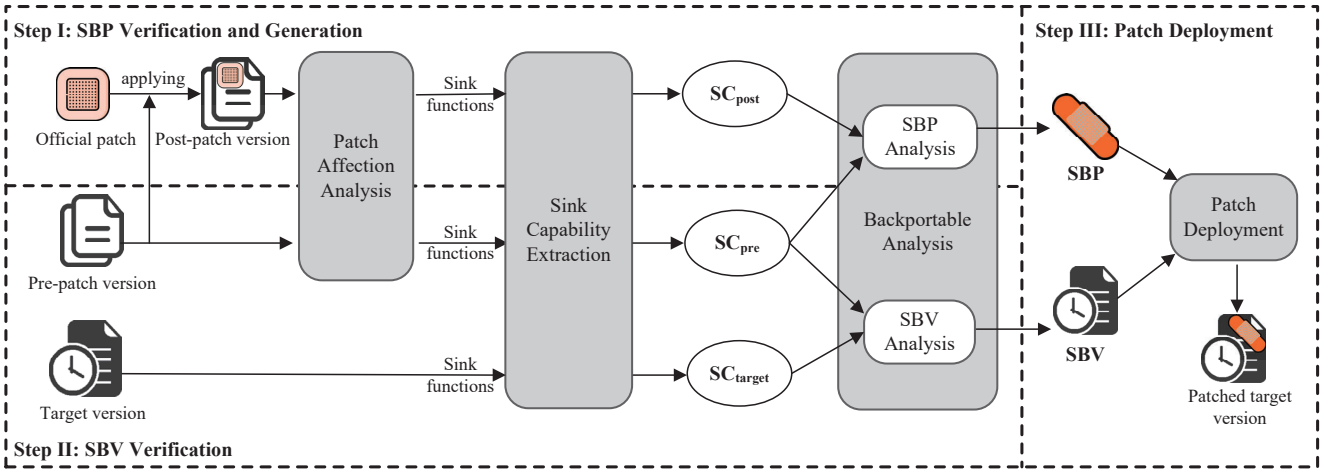

Figure 2: Workflow of SKYPORT. ❶ *patch affection analysis* locates the affected sink functions by an official path and then *sink capability extraction* generates their sink capabilities. Next, *SBP analysis* determines whether the sink functions are backportable and generates SBP. ❷ SKYPORT locates the sink functions of a target version and generates sink capabilities. Then, *SBV analysis* checks whether it is an SBV. ❸ *Patch deployment* patches the SBV with the generated SBP.

4 SKYPORT Design

In this section, we present the design of SKYPORT in Figure 2 with three steps clearly separated and described. The detailed design of SKYPORT is based on the code property graph [12] (CPG). Now, instead of describing the three steps of SKYPORT, we organize the section via four important components of SKYPORT, which sometimes span across multiple steps with different inputs and outputs. Here is a brief overview of these components:

- *Patch Affection Analysis* extracts sink functions that are affected by the official patch from the pre-patch version, i.e., those vulnerable ones that might be fixed by the patch.
- *Sink Capability Extraction* accepts a list of sink functions and a given version of the web application, performs symbolic tracking, and outputs sink capabilities (i.e., reaching conditions and data-flow expressions) for all the sink functions.
- *Backportable Analysis* checks SBPs and SBVs based on sink capabilities and generates SBPs if the check passes.
- *Patch Deployment* applies SBP upon SBV and fixes the vulnerability on the target old version.

4.1 Patch Affection Analysis

The purpose here is to extract sink functions affected by the official patch, thus being candidates for belonging to the target vulnerability of the patch. The key here is a forward taint analysis that starts from patch changes, extracts patch-affected code, and then identifies affected sink functions. We describe the analysis via two steps.

First, SKYPORT performs forward taint analysis from the patch changes and iteratively locates the affected lines via querying the program dependency graph (PDG) with affected variables in affected lines. The analysis includes three types of statements that may affect the reaching condition or the data-flow expression of a sink function and excludes other unrelated statements. Specifically, these three types of statements include assignment (affecting data-flow expression), conditional (affecting reaching condition), and exit (affecting reaching condition).

Second, SKYPORT identifies sink functions from the affected lines extracted from the previous step. Specifically, the identification is based on a human-created map (as shown in Table 2) between sink functions and vulnerability types, e.g., `fopen()` vs. arbitrary file read. It is worth noting that some sink functions may be defined or customized by a developer. In such cases, SKYPORT performs code reachability analysis to finds the final sink function. Besides, due to the conservativeness of static analysis, SKYPORT may have false positives in extracting sink functions. Fortunately, these false positives can be removed in the following steps because they have the same sink capability in the pre-patch version and the post-patch version.

4.2 Sink Capability Extraction

Sink Capability Extraction accepts a list of sink functions and a given version of web application, and extracts the sink capability (*SC*) of the given version for all the provided sink functions. Specifically, SKYPORT collects all the control-flow paths to the sink function, calculates the sink's Reaching Conditions (*RC*) and Data-flow Expressions (*DE*) along each path via symbolic tracking, and then unions them as *SC*.

Let us describe these steps in details. First, SKYPORT performs forward path exploration to follow all control-flow edges from either an entry point or a caller function until a provided sink function. During the exploration, SKYPORT also checks whether there exists an exit statement, such as `die()`, and stops the exploration. If SKYPORT encounters a loop, it unrolls loop once to avoid path explosion. Second,

SKYPORT calculates both *RC* and *DE* via symbolic tracking upon Tree Address Code (TAC) formulas. Specifically, SKYPORT first collects all the *RC* and *DE*, and identifies three types of symbolic variables, which are global variables, sink function caller's parameters, and external variables defined in other PHP files for the calculation. Lastly, SKYPORT union *RC* and *DE* for all the control-flow paths of a sink function as the *SC* for the sink function.

4.3 Backportable Analysis

In this subsection, we first describe three building blocks of the backportable analysis of SKYPORT and then present how to perform `SBP` and `SBV` analysis using these building blocks.

4.3.1 Building Blocks of Backportable Analysis

The key of backportable analysis is to compare different *SC*s of different sink functions belonging to web application versions and ensure their determinism. That is, naturally we have two building blocks, one for *SC* comparison and one for determinism.

Sink Capability Analysis In this part, we describe one building block, i.e., our sink capability analysis. We first present the analysis of reaching condition and data-flow expression and then the combination of these two in sink capability analysis.

- Reaching Condition (*RC*) Analysis. We model *RC* of a given sink function as Equation 1.

$$RC_{sink} = RC_{flow_1} \vee RC_{flow_2} \vee ... \vee RC_{flow_n} \quad (1)$$

  where $flow_i$ is a control-flow path leading to the sink, RC_{flow_i} is the logic `and` operation of all the conditions along $flow_i$, i.e., $RC_{flow_i} = C_1 \wedge ... \wedge C_k$, and C_k is one condition along $flow_i$. Then, the Reaching Condition Analysis boils down to Boolean Algebra and the check of whether a certain condition holds. For example, the equivalence of two *RC* is a check of $RC_a = RC_b$, and whether RC_a is a subset of RC_b equals to a check of $RC_a = RC_a \wedge RC_b$.
- Data-flow Expression (*DE*) Analysis. SKYPORT performs a conservative Data-flow Expression Analysis by checking whether common sanitization functions, such as `trim()` and `intval()`, exist between two *DE*s. If yes, SKYPORT considers one *DE* is a subset of another. We understand that it may bring false negatives, but in practice this is very rare.

Now we describe how SKYPORT compares *SC*s using *RC* and *DE* analysis. Say there exists two *SC*s, i.e., SC_a and SC_b. SKYPORT first performs *RC* analysis by comparing each *RC* of a sink flow in SC_a with all *RC*s in SC_b and finding exact matches. If an exact match cannot be found for a RC_i in SC_a, SKYPORT finds a *RC* in SC_b that is a subset of RC_i. After SKYPORT finds matches for all *RC*s in SC_a and SC_b, SKYPORT performs *DE* analysis to determine whether corresponding *DE*s of matched *RC*s have a subset relationship.

Expression Re-computable Analysis. In this part, we describe the other building block, i.e., Expression Re-computable Analysis. SKYPORT determines whether all data-flow expressions in the sink capability are the same if being recomputed again. The reason for this analysis is that SKYPORT includes the entire safe logic in `SBP`, which recomputes the safe logic from user inputs. Specifically, SKYPORT checks whether *DE*s contain four types of non-deterministic operations, which are network-related operations (e.g., `curl_exec()`) , database manipulation operations (e.g., `mysql_query()`, `pg_query()`), file operations (e.g., `fwrite()`, `unlink()`) and operations that return dynamic values (e.g., `time()`, `rand()`). If none of the operations exists, SKYPORT considers the target *DE* as re-computable.

4.3.2 `SBP` and `SBV` Analysis

In this part, we describe how to perform `SBP` and `SBV` analysis using the two building blocks described in §4.3.1.

`SBP` Analysis. The purpose of `SBP` analysis is to check whether the original patch is backportable and then generate an `SBP` if so. First, SKYPORT compares the sink capabilities of those output by Patch Affection Analysis between pre-patch and post-patch versions (i.e., SC_{pre} and SC_{post}). If SC_{post} is a subset of SC_{pre} as determined by the Sink Capability Analysis, i.e, satisfying both P_{SBP}-*a* and P_{SBP}-*b*, SKYPORT considers it as a candidate. Second, SKYPORT performs Expression Re-computable Analysis to ensure that data-flow expressions in all sink capabilities are deterministic, i.e., satisfying P_{SBP}-*c*. If so, SKYPORT determines that the original patch is backportable.

Next, SKYPORT generates a safe sink function (e.g., `safe_fopen()` in Figure 1 (c)) and replaces the original sink as the `SBP`. The safe sink function includes all the sink flow paths in SC_{post} into a safe-sink function and recomputes the critical parameter(s) for each sink flow. If some reaching conditions are filtered in the original official patch, SKYPORT uses a `die()` function to directly abort the execution in `SBP`. One thing worth noting here is the handling of loop because a loop is only unrolled once during Sink Capability Extraction. That is, SKYPORT does not recover the original loop structure when generating `SBP`. Our high-level idea is to recalculate the sink capability of the post-patched version by encapsulating the loops as functions (sink irrelevant statements are not included) and use the recalculated SC_{post} for `SBP` generation. Specifically, there are three cases based on whether patch modification lines (called patch below for short) and a target sink are inside or outside a loop:

- Patch [inside] and Sink [inside]. SKYPORT treats the loop block as a function and calculates *SC* inside the function.
- Patch [outside] and Sink [inside]. SKYPORT treats the loop block as the sink function and replaces the original sink in calculating *SC*.

- Patch [inside/outside] and Sink [outside]. SKYPORT encapsulates the loop as a function to calculate *SC*.

SBV Analysis. The purpose of SBV analysis is to check whether a given target old version is backportable. First, SKYPORT locates the sink functions at the target version according to their function and file names, and generates sink capabilities for them. Second, SKYPORT compares the sink capabilities of all the sink functions of the pre-patch and target versions using the Sink Capability Analysis. Specifically, if every sink function has a match in two versions and SC_{pre} and SC_{target} are exactly the same, i.e., satisfying both P_{SBV}-*a* and P_{SBV}-*b*, SKYPORT considers this target version as backportable and outputs this version as SBV.

4.4 Patch Deployment

Patch Deployment applies an SBP upon an SBV by a direct modification upon the web application source code. There are two steps. First, SKYPORT replaces the sink function with the generated safe sink function at a target version. There are two types of replacement: (i) function replacement and (ii) parameter replacement. Function replacement is the default, which replaces the sink function with a new, safe function; Parameter replacement is an alternative to function replace when the sink function cannot be replaced, e.g., being a PHP keywords like `include` and `return`. If so, SKYPORT replaces the critical parameter with a safe function, e.g., `include safe_include()`. Note that SKYPORT will check name conflicts to avoid function names being used by the web application.

Second, SKYPORT backups variables related to the sink function, i.e., particularly global variables, sink function caller's parameters, and external variables, at the beginning of the vulnerable code and then uses them in the generated safe sink function. Consider our motivating example in Figure 1 again. SKYPORT backups both `$_GET['template']` and `$GLOBALS['OE_SITE_DIR']`s at the beginning of the PHP file, i.e., Lines 2 and 3 in Figure 1 (b), and then uses them in the safe sink function.

5 Implementation and Dataset

In this section, we first describe our prototype implementation and then present the datasets for patch backporting.

5.1 Prototype Implementation

We implemented a prototype of SKYPORT for PHP applications. Our *patch affection analysis* module and *sink capability extraction* module are based on PHPJoern [12] and Neo4j [4]. Table 2 shows the mapping between vulnerability type and sink function used in the patch affection analysis. Our *patch deployment* implementation relies on the source code and CPG mapping provided by PHPJoern so that SKYPORT can locate the exact line number of source code based on CPG node. Note that because the current implementation of PHPJoern does not resolve the targets for dynamic calls with non-unique function names, we implemented a class hierarchy analysis [29, 33, 78] to optimize the call graph construction of such dynamic functions and improve the accuracy in locating sink functions.

Table 2: Mapping between Vulnerability Types and Sink Functions.

Vulnerability Type	Potential Sink Function
Server-Side XSS	echo, print, print_r
SQL Injection	pg_query, pg_send_query, pg_prepare, mysql_query, mysqli_prepare, mysqli_query, mysqli_real_query
Arbitrary File Read	file, file_get_contents, readfile, fopen
Arbitrary File Write	file_put_contents, fopen, fwrite
Arbitrary File Delete	unlink, rmdir
Arbitrary File Include	include, include_once, require, require_once
Command Injection	exec, passthru, proc_open, system, shell_exec, popen, pcntl_exec
Code Injection	fopen, file_get_contents, fwrite, fputs, eval, create_function, assert, array_map
Directory Traversal	fopen, dir, dirname, opendir, scandir
Executable File Upload	copy, fopen, move_uploaded_file, rename
SSRF	curl_exec, file_get_contents, fsockopen
Open Redirect	header
PHP Object Injection	unserialize

Other than the major functionality improvement, we also fix three major bugs of PHPJoern for SKYPORT.

- `or throw` syntax. We added parsing and code property graph (CPG) generation supports for the `or throw` syntax, which is used by some web vulnerabilities, e.g., CVE-2015-5078.
- Constant condition in `switch` statement. We fixed the incorrect control-flow edges generated by PHPJoern, when the condition of `switch` is a constant value.
- Data-flows for `foreach`'s iterable_expression. We added data-flow paths between the `foreach`'s iterable_expression and the internal variables in `foreach` block.

Other than PHPJoern, we also reused and extended the TAC (Three-Address Code) representation and the related transformation code from NAVEX [10]. Our extension of NAVEX includes new AST node types (e.g., `AST_MAGIC_CONST`) and new AST operation symbols (e.g., `BINARY_SHIFT_RIGHT` for >>, `BINARY_SHIFT_RIGHT` for <<). We used Wolfram-Script [7] for bool computation in the *backportable analysis* module.

5.2 Dataset

In this subsection, we describe our dataset used in the evaluation, which has 155 patches (i.e., 155 pre-patch and post-patch versions) and 1,526 vulnerable target versions. We break down all the patches based on the CVE types in Table 1: Server-side XSS is the most popular vulnerability and PHP Object injection is the least. We also break down all the patches based on web application types in Table 3: WordPress,

Table 3: Dataset of target CVEs and PHP applications.

CMS Name	# CVEs	# Versions	# <CVE,Version>
WordPress	34	187	430
PHPMyAdmin	29	108	257
PrestaShop	11	34	101
RoundcubeMail	8	48	76
Mantisbt	24	74	198
Piwigo	11	37	108
OpenEMR	11	20	70
phpipam	3	6	13
MISP	9	55	118
LimeSurvey	15	82	155
Total	155	651	1,526

Table 4: The reasons for excluding some CVEs.

Reason for Exclusion	Count
Vulnerability type not in scope	88
No patch file	18
Does not fit patch model	8
Total	114

Table 5: Overall Experiment Results.

	Patches	Target Versions	Success Rate
Phase 0: Dataset	155	1,526	N/A
Phase 1: SBP Verif.&Gen.	111	1,137	71.61
Phase 2: SBV Verif.	98	750	65.96
Phase 3: SBP→SBV	98	750	100.00

one popular web application, has the most number of CVEs and corresponding vulnerable target versions.

We now describe how we manually generate the dataset via two phases. The dataset generation takes two authors 20 human hours in total. First, we collect patches and CVEs by selecting ten most popular PHP projects on Github with >1K starts as shown Table 3 and their corresponding CVEs and patch commits from 2014 to 2020. In total, we obtain 269 CVEs and patches. Then, we manually check all the 269 CVEs and patches and exclude 114 CVEs because they are either out-of-scope (i.e., not injection vulnerabilities), with incorrect patch commit URLs, or outside our patch model (e.g., removing sink functions or modifying configuration like *.htaccess*). A detailed breakdown can be found in Table 4. Second, we collect vulnerable target versions via two methods: (i) checking the vulnerability information (e.g., the declared affected version range) in *NVD* [3] and *MITRE* [1], and (ii) manually analyzing the application source code of affected versions and removing those with errors in the database, e.g., without corresponding files of the sink functions.

5.2.1 Dataset Statistics

In this part, we illustrate some statistics of our dataset and further explain why security patch backporting is generally challenging.

First, we find that 98 out of 155 patches contain vulnerability irrelevant modifications. Some modifications may lead to a backward compatibility issue and some may lead to a patch deployment issue. We summarize them into three categories:

- *Functionality Modifications.* Security patches may change existing functionality or add new functionality while fixing vulnerabilities. For instance, the security patch for CVE-2014-7146 not only fixes the arbitrary PHP code vulnerability but also adds the function to retrieve the file contents.
- *Variable or Function Name Modifications.* Some security patches modify variable or function names. For example, the security patch for CVE-2014-1609 changes many variable names, such as `$result` to `$t_result` and `$t_users` to `$t_count`.
- *Miscellaneous Modifications.* Some security patches modify version numbers, add comments or adjust the indentation when fixing vulnerabilities, such as CVE-2018-15139, CVE-2017-16510, and CVE-2019-16220.

Second, let us look at the target versions. We find that 1,071 of 1,526 target versions have code location changes around the patch, which lead to a failure when directly applying the original patch. At the same time, we also find that 563 of 1,526 target versions do not have exactly the same vulnerable logic as the version that the original patch is targeting. Note that these target versions are not SBVs, thus not being backportable.

6 SKYPORT Evaluation

In this section, we present the evaluation of SKYPORT from three aspects: backporting results, comparison with state of the art, and performance overhead.

6.1 Results

In this subsection, we first give an overview of the results and then break them down into details by different steps.

6.1.1 Results Overview

Table 5 shows an overview of SKYPORT's results in each step. The first metric is the success rate, which indicates the percentage of correctly-verified SBPs, SBVs, and patch deployment. The result shows that SKYPORT successfully achieves its tasks in each step with 100% rate, i.e., the number in each step matches our manual analysis of SBPs and SBVs with no false positives and negatives. Here are the numbers of remaining patches and target versions after each step. In the first step, SKYPORT verifies that 111 out of 155 security patches are backportable; in the second step, SKYPORT verifies that 750 target versions are SBVs, which further reduces the number of SBPs to 98; in the third step, SKYPORT successfully deploys all 98 SBPs upon on 750 SBVs and fixes the corresponding vulnerabilities.

Note that the evaluation results on success rates of SBPs and SBVs are performed by five analysts that are independent of the paper authors. The total inspection takes about 175 human hours. The inspection of each SBP and SBV involves at least two analysts and if they disagree on the results, we will include a third analyst and let them have a discussion to reach a consensus.

pre-patch

```
if ( 0 == $t_type || empty( $t_matches['script'] ) ||
    3 == $t_type && preg_match( '@(?:[^:]*)?://@', $t_url ) > 0 ) {
  return ($p_return_absolute ? $t_path . '/' : '') . 'index.php';
}
...
```

post-patch

```
if ( 0 == $t_type || empty( $t_matches['script'] ) ||
    3 == $t_type && preg_match( '@(?:[^:]*)?:/*@', $t_url ) > 0 ) {
  return ($p_return_absolute ? $t_path . '/' : '') . 'index.php';
}
...
```

a) no subset relationship between reaching conditions

pre-patch

```
echo '<a href="https://' . $_SERVER['HTTP_HOST']
                         . $_SERVER['REQUEST_URI'] . '" title="' . '...';
```

post-patch

```
echo '<a href="https://' . $_SERVER['HTTP_HOST']
          . $this->createUrl("admin/globalsettings/sa") . '" title="' . '...';
```

b) no subset relationship between symbolic expressions

Figure 3: Patches that do not meet P_{SBP}.

6.1.2 Results Breakdown

In this part, we break down the results via different steps and give some statistics and case studies on each step.

Step I Results: `SBP` Verification and Generation. In this step, SKYPORT extracts 280 vulnerable sink functions for 155 patches and verifies that 111 patches with 197 sink functions are backportable. Let us start from describing these backportable patches and then these that cannot be backported. First, all 111 patches restrict the input space of 197 sink functions, i.e., limiting their sink capabilities. Specifically, 94 sink functions have new control-flow constraints (limiting reaching conditions), 52 sink functions have new data-flow sanitization (i.e., limiting data-flow expressions), and the rest 51 have both new control-flow constraints and data-flow sanitization.

Second, here is a breakdown of the rest 44 patches with 83 sink functions.

- *8 patches fail on P_{SBP}-a.* Figure 3 (a) gives an example to illustrate such patches. Specifically, the pre-patch version adopts a regular expression `"@(?:[^:]*)?://@"` in the condition check of a branch statement opposed to another regular expression `"@(?:[^:]*)?:/*@"` in the post-patch version. That is, there are no subset relations between the reaching conditions of the two versions.
- *35 patches fail on P_{SBP}-b.* Figure 3 (b) illustrates such an example that fails on P_{SBP}-b. Specifically, one variable, i.e., `$_SERVER["REQUEST_URL"]`, in the sink function `echo` of the pre-patch version is replaced with `$this->createUrl("admin/globalsetting/sa")` in the post-patch version. Therefore, there are no subset relations between data-flow expressions of the pre-patch and post-patch versions.
- *1 patch fails on P_{SBP}-c.* In this patch, there exists a database read operation `query()` in the data-flow expression of critical parameters. Since it is difficult to judge whether the results of the two executions are consistent,

```
1  <?php
2  $utquery = "UPDATE {{tokens_$surveyid}}\n";
3  if (isTokenCompletedDatestamped($thissurvey)) {
4    if (isset($usesleft) && $usesleft <= 1) {
5  -   $utquery.="SET usesleft=usesleft-1, completed='$submitdate'\n";
6  +   $utquery .= "SET usesleft=usesleft-1, completed=" .
                                              dbQuoteAll($submitdate);
7      } else {
8        $utquery .= "SET usesleft=usesleft-1\n";
9      }
10   } else {
11     ...
12   }
13   $utquery .= "WHERE token=" . dbQuoteAll($_POST['token']);
14   $utresult = dbExecuteAssoc($utquery);
15   ?>
```

a) Official Patch for CVE-2015-5078 on LimeSurvey 2.06_plus_150612

```
1  <?php
2  $utquery = "UPDATE {{tokens_$surveyid}}\n";
3  if (isTokenCompletedDatestamped($thissurvey)) {
4    if (isset($usesleft) && $usesleft <= 1) {
5      $utquery.="SET usesleft=usesleft-1, completed='$submitdate'\n";
6    } else {
7      $utquery .= "SET usesleft=usesleft-1\n";
8    }
9  } else {
10   ...
11 }
12 $utquery .= "WHERE token='" . $_POST['token'] . "'";
13 $utresult = dbExecuteAssoc($utquery);
14 ?>
```

b) LimeSurvey 2.00_plus_121002

Figure 4: A case that violates P_{SBV}-b.

SKYPORT conservatively recognizes this operation as non-deterministic.

Step II Results: `SBV` Verification. In this part, SKYPORT verifies that 750 out of 1,137 versions are `SBV`s. Now let us describe why the rest versions are not backportable and break the reasons down.

- *13 versions without the same sink function as the patch.* There are two major reasons: i) the sink functions are introduced at higher versions when implementing new functionalities, and ii) lower versions use other functions that are similar to the sink function. For example, the sink function for the patch of CVE-2017-5608 in Piwigo is `single_insert()`, while version 2.2.2 uses `mass_insert()` instead. These two sink functions have different though similar semantics.
- *263 versions violating P_{SBV}-a.* We consider CVE-2016-10083 of Piwigo as an example. The pre-patch version, i.e., Piwigo 2.8.3, and an older version have different control-flow constraints in an `if` statement. Specifically, the regular expression is `/^[\w-]+$/` in the pre-patch version as opposed to `/^\w+$/` in the older version.
- *111 version violating P_{SBV}-b.* Taking Figure 4 as an example, the pre-patch version sanitizes `$_POST['token']` at Line 13 in (a), while the target, old version in (b) does not, which violates P_{SBV}-b.

Table 6: Patch Result Comparison for 1,526 Target Versions among Auto-upgrade, Direct Patch, and SKYPORT.

	Direct patch	Strict upgrade	Lazy upgrade	SKYPORT
Success	455	39	149	750
Failure	1,071	1,487	1,377	776

Step III Results: Patch Deployment (SBP→SBV). SKYPORT successfully deploys all 98 SBPs upon 750 affected safely backportable versions. For example, the patch for CVE-2014-9281 (an XSS vulnerability of MantisBT) is developed at 1.2.17, while SKYPORT successfully deploys the generated SBP on the version 1.0.4. There are 38 versions between these two spanning over 3,009 days.

6.2 Comparison with State of the Art

In this subsection, we compare SKYPORT with three state-of-the-art approaches in backporting security patches.

- Direct Patch Application. This approach directly applies the official patch upon the target version using the "patch" command.
- Strict Auto-upgrade. This approach uses auto-upgrade APIs provided by the web application framework if the supported PHP versions between two web application versions are the same. Note that we use PHPLint [6] to check PHP version compatibilities of web applications.
- Lazy Auto-upgrade. This approach is a relax of strict auto-upgrade, which considers an auto-upgrade succeeds if the PHP version conflicts only exist for test or demo files.

Table 6 shows the overall results among four approaches including SKYPORT. Clearly, SKYPORT outperforms all state-of-the-art approaches in fixing vulnerabilities in target versions of web applications.

6.2.1 Breakdown of Direct Patch Application Failures

In this part, we break down all the failure cases of direct patch applications.

- *Code conflict (1,049 versions).* When the target version has a different patching context to the pre-patch version, the "*patch*" command raises a "*code conflict*" error. Since there might exist many code changes between the target and the pre-patch versions, such conflict is very common in direct patch application.
- *No such file or directory (5 versions).* When files modified by the official patch do not exist in the target version, a "*no such file or directory*" error is reported. It is worth noting that none of these files are relevant to the vulnerability fix. Therefore, our approach does not meet these problems.
- *Reversed patch detected (17 versions).* In some scenarios, the code lines modified by the patch are the same as the ones in the post-patch version, which will raise a "*reversed patch detected*" error.

Table 7: Performance (Seconds) of SKYPORT Broken-down by Different Modules in Average, Medium and Medium Absolute Deviation (MAD).

SKYPORT Module	Average	Medium	MAD
Patch Affection Analysis	666.49s	40.28s	44.92s
Sink Capability Extraction for SBP	695.32s	72.00s	88.96s
Sink Capability Extraction for SBV	4,064.82s	527.50s	715.36s
Backportable Analysis	129.09s	17.00s	23.72s
Patch Deployment	4.80s	1.00s	0.00s
End-to-end Total	6,459.75s	895.47s	1,181.09s

6.2.2 Breakdown of Auto-upgrade Failures

In this part, we break down all the 1,487 versions that auto-upgrade failed to backport and fix vulnerabilities.

- *Versions without Auto-upgrade APIs (624 versions).* Not all web application frameworks provide auto-upgrade APIs. In our datasets, five frameworks (Piwigo, WordPress, LimeSurvey, PrestaShop and MISP) do provide and five do not, which contribute to 624 target old versions .
- *Parsing Errors in Core Files (753 versions).* In these versions, when using PHPLint with required compatible PHP versions to analyze, PHP parse errors or PHP fatal errors will be reported in some core files, such as `wp-includes/canonical.php` of WordPress 5.8.1 in PHP 5.4, or `Core/Domain/Currency/ValueObject/Precision.php` of PrestaShop 1.7.8.0 in PHP 7.0.
- *Parsing Errors in Non-core Files (108 versions).* In these versions, the PHP errors occur only in non-core files (e.g., demo or test files). For example, `tools/test_piwigo.php` of Piwigo 11.5.0 in PHP 5.4).
- *Parsing Warnings (2 versions).* In these versions, only PHP warnings will appear, which will not affect the normal execution of the program. For instance, PHP Strict Standards will be reported, when using PHPLint with PHP 5.6 to analyze `wp-includes/sodium_compat/src/Core/Base64/Common.php` in WordPress 5.8.1.

6.3 Performance

In this subsection, we evaluate the performance of SKYPORT and break down the performance by different modules of SKYPORT in Table 7. First, both Patch Affection Analysis and Sink Capability Extraction modules are heavyweight, because they rely on static analysis to follow each sink path, collect reaching conditions and generate data-flow expressions. Note that Sink Capability Extraction for SBP is faster than the one for SBV, because our Patch Affection Analysis filters many irrelevant sink functions and the number of analyzed sinks for SBP is much smaller than SBV. Second, Backportable Analysis and Patch Deployment modules are relatively lightweight. Backportable Analysis mainly relies on the analysis of boolean operations and Patch Deployment on locating and replacing PHP code.

Table 8: Breakdown of Test Cases Generated by Each Method in the Functionality Test (x in x/y means the number of test cases triggering `SBP` modified code and y in x/y means the total number of generated test cases).

App	CVE-ID	Version	Breakdown (# of SBP-related / # of Total)				
			Tutotrial	Crawlers	Monkey	Scanner	Manual
LimeSurvey	CVE-2020-11456	4.0.0	11 / 1,182	1 / 599	0 / 543	0 / 823	6 / 6
		4.0.1	11 / 1,182	1 / 488	0 / 1,425	0 / 1,389	6 / 6
	CVE-2020-25798	3.1.0	2 / 1,037	0 / 1,470	0 / 11,853	0 / 936	6 / 6
		3.14.4	2 / 1,067	0 / 1,609	0 / 4,947	0 / 551	6 / 6
OpenEMR	CVE-2018-10571	5.0.0	2 / 973	4 / 1,198	0 / 1,178	0 / 482	6 / 6
		5.0.0.5	2 / 979	2 / 1,190	0 / 1,370	0 / 1,607	6 / 6
		4.1.1	2 / 912	1 / 616	0 / 596	0 / 368	6 / 6
	CVE-2018-10572	5.0.0	3 / 826	1 / 1,125	0 / 1,291	0 / 1,136	6 / 6
		5.0.0.5	3 / 854	2 / 925	0 / 493	0 / 372	6 / 6
		4.1.1	3 / 819	2 / 1,031	0 / 579	0 / 295	6 / 6
	CVE-2018-15139	5.0.0	2 / 812	2 / 1,188	0 / 1,341	0 / 273	6 / 6
		5.0.0.5	2 / 821	0 / 708	0 / 1,381	0 / 871	6 / 6
		4.2.0.3	2 / 943	1 / 1,335	1 / 728	0 / 486	6 / 6
MantisBT	CVE-2017-12061	1.3.10	1 / 873	0 / 328	0 / 1,622	0 / 341	4 / 4
		2.5.0	1 / 895	0 / 215	0 / 606	0 / 672	4 / 4
	CVE-2017-12062	2.2.4	4 / 1,006	0 / 194	58 / 1,483	0 / 1,536	4 / 4
		2.5.0	4 / 991	48 / 433	0 / 407	0 / 688	4 / 4
	CVE-2014-9281	1.2.10	1 / 556	0 / 252	0 / 868	0 / 292	4 / 4
		1.0.4	2 / 615	0 / 262	0 / 1,785	0 / 286	4 / 4
	CVE-2014-9270	1.2.9	8 / 486	0 / 283	0 / 467	0 / 329	4 / 4
		1.2.15	7 / 520	0 / 299	0 / 473	0 / 320	4 / 4
Piwigo	CVE-2017-10682	2.9.0	3 / 247	0 / 2,285	0 / 6,240	0 / 588	3 / 3
		2.6.5	3 / 215	0 / 1,494	0 / 246	0 / 331	3 / 3
		2.4.5	3 / 283	4 / 1,323	0 / 1,482	0 / 304	3 / 3
	CVE-2017-17824	2.9.0	1 / 247	0 / 1,241	1 / 3,698	0 / 404	3 / 3
		2.6.5	1 / 215	1 / 758	0 / 645	0 / 330	3 / 3
		2.4.5	1 / 315	1 / 440	0 / 1,935	0 / 334	3 / 3

Table 9: Breakdown of 1% Sampling of Test Cases Generated by Each Method in the Functionality Test (x in x/y means the number of sampled test cases and y in x/y means the total number of generated test cases that do not trigger `SBP` modified code).

App	CVE-ID	Version	Breakdown (# of 1% sampling / # of Total)			
			Tutotrial	Crawlers	Monkey	Scanner
LimeSurvey	CVE-2020-11456	4.0.0	12 / 1,171	6 / 598	5 / 543	8 / 823
		4.0.1	11 / 1,171	6 / 487	14 / 1,425	14 / 1,389
	CVE-2020-25798	3.1.0	10 / 1,035	14 / 1,470	121 / 11,853	9 / 936
		3.14.4	10 / 1,065	16 / 1,609	49 / 4,947	5 / 551
OpenEMR	CVE-2018-10571	5.0.0	9 / 971	12 / 1,194	12 / 1,178	5 / 482
		5.0.0.5	10 / 977	12 / 1,188	13 / 1,370	16 / 1,607
		4.1.1	9 / 910	6 / 615	6 / 596	3 / 368
	CVE-2018-10572	5.0.0	8 / 823	11 / 1,124	12 / 1,291	11 / 1,136
		5.0.0.5	8 / 851	9 / 923	5 / 493	3 / 372
		4.1.1	8 / 816	10 / 1,029	6 / 579	3 / 295
	CVE-2018-15139	5.0.0	8 / 810	12 / 1,186	13 / 1,341	3 / 273
		5.0.0.5	8 / 819	7 / 708	15 / 1,381	9 / 871
		4.2.0.3	9 / 941	13 / 1,334	7 / 727	5 / 486
MantisBT	CVE-2017-12061	1.3.10	8 / 872	3 / 328	16 / 1,622	3 / 341
		2.5.0	9 / 894	2 / 215	6 / 606	7 / 672
	CVE-2017-12062	2.2.4	10 / 1,002	2 / 194	15 / 1,425	15 / 1,536
		2.5.0	10 / 987	4 / 385	4 / 407	7 / 688
	CVE-2014-9281	1.2.10	6 / 555	2 / 252	9 / 868	3 / 292
		1.0.4	6 / 613	3 / 262	20 / 1,785	3 / 286
	CVE-2014-9270	1.2.9	5 / 478	3 / 283	5 / 467	3 / 329
		1.2.15	5 / 513	3 / 299	5 / 473	3 / 320
Piwigo	CVE-2017-10682	2.9.0	2 / 244	24 / 2,285	64 / 6,240	6 / 588
		2.6.5	2 / 212	16 / 1,494	2 / 246	3 / 331
		2.4.5	3 / 280	13 / 1,319	15 / 1,482	3 / 304
	CVE-2017-17824	2.9.0	2 / 246	13 / 1,241	38 / 3,697	4 / 404
		2.6.5	2 / 214	8 / 757	6 / 645	3 / 330
		2.4.5	3 / 314	4 / 439	19 / 1,935	3 / 334

7 SKYPORT-patched Web App Evaluation

In this section, we evaluate SKYPORT-patched web apps. Because the evaluation involves manual works, such as real web application environment setup and attack input collection, our evaluation is based upon a subset of web applications patched by SKYPORT. Our criterion is to select vulnerabilities with exploits and then those belonging to the top four web frameworks. In total, we select 11 patches and 27 versions as shown in Table 10. All the experiments in this section are performed on a Linux machine with 2 Intel Xeon E7-4820 processors and 378 GB RAM. The evaluation answers the following three Research Questions (RQs):

- RQ1 [Security]: Can SKYPORT-patched web applications prevent attacks against the patch-targeted vulnerability?
- RQ2 [Functionality]: Are SKYPORT-patched web applications still functioning correctly as they do before patching?
- RQ3 [Performance]: What is the performance overhead introduced by SKYPORT on the target version comparing with the official patch on the pre-patch version?

An overview of the answers to all three questions can be found in Table 10. We also describe them in detail below.

7.1 RQ1: Security

In this Research Question, we evaluate the security of SKYPORT-patched web applications, i.e., whether they are still vulnerable to attacks targeting the original vulnerability. Our methodology is as follows. We collect exploits targeting these CVEs in Table 10 from online locations, e.g., the CVE database, and then verify the effectiveness of these exploits by testing them on the pre-patch and target versions. If an exploit can attack the pre-patch and the history versions, we also apply it on the SKYPORT-patched versions. The "Security Test Pass Ratio" in Table 10 shows the results, i.e., SKYPORT-patched web applications can defend against the original exploit according to our evaluation.

7.2 RQ2: Functionality

In this Research Question, we evaluate the functionalities of SKYPORT-patched web applications. Here are how we collect test cases for SKYPORT-patched web applications. First, we follow prior work [11] to collect test cases via tutorials, spider, monkey testing and vulnerability scanner. Second, we also read CVE descriptions and manually construct test cases that can trigger the vulnerable sink on SKYPORT-patched web applications. Third, from the previous two steps, we generate two sets of our own test cases:

Table 10: Evaluation Results of the Safely Backportable Patch in its Security, Functionality and Performance.

Application	CVE-ID	Pre-patched Version	Test Target Versions	Security Test Pass Ratio	Functionality Test Pass Ratio		Performance Overhead	
					SBP-related	1% sampling	Normal Input (O/S)[1,2]	Attack Input (O/S)[1,2]
LimeSurvey	CVE-2020-11456	4.1.11	4.0.0, 4.0.1	2 / 2	36 / 36	76 / 76	18.65%/21.78%	7.26%/12.51%
	CVE-2020-25798	3.21.1	3.14.4, 3.1.0	2 / 2	16 / 16	234 / 234	5.41%/7.66%	6.91%/12.72%
OpenEMR	CVE-2018-10571	5.0.0.6	5.0.0, 5.0.0.5, 4.1.1	3 / 3	31 / 31	113 / 113	7.71%/11.94%	7.42%/8.23%
	CVE-2018-10572	5.0.0.6	5.0.0, 5.0.0.5, 4.1.1	3 / 3	32 / 32	94 / 94	1.86%/2.69%	0.99%/1.58%
	CVE-2018-15139	5.0.1.3	5.0.0, 5.0.0.5, 4.2.0.3	3 / 3	28 / 28	109 / 109	2.23%/3.40%	-3.86%/-2.66%
MantisBT	CVE-2017-12061	1.3.11	1.3.10, 2.5.0	2 / 2	10 / 10	54 / 54	3.46%/9.31%	11.31%/17.23%
	CVE-2017-12062	2.5.1	2.2.4, 2.5.0	2 / 2	122 / 122	67 / 67	0.29%/0.82%	1.86%/4.80%
	CVE-2014-9281	1.2.17	1.2.10, 1.0.4	2 / 2	11 / 11	52 / 52	10.47%/22.65%	5.48%/13.32%
	CVE-2014-9270 [3]	1.2.17	1.2.15, 1.2.9	2 / 2	23 / 23	32 / 32	2.93%/4.62%	6.55%/12.58%
Piwigo	CVE-2017-10682	2.9.1	2.9.0, 2.6.5, 2.4.5	3 / 3	22 / 22	153 / 153	6.13%/10.95%	-49.57%/-48.10%
	CVE-2017-17824	2.9.2	2.9.0, 2.6.5, 2.4.5	3 / 3	15 / 15	105 / 105	2.43%/3.20%	-97.39%/-97.14%
Total	11	11	27	27 / 27	346 / 346	1,089 / 1,089	-	-

[1] O(fficial Patch) = the overhead that is introduce by the official patch on the pre-patched version.
[2] S(afely Backportable Patch) = the overhead that is introduced by the SBP on the pre-patched version.
[3] The SBP generated by SKYPORT involves a loop statement.

- SBP-related Test Cases. We run all the collected test cases and record a test case if it triggers the SBP. A breakdown of this set of test cases is shown in Table 8. Interestingly, SBP modifications and the vulnerability are often very hard to trigger, i.e., only a small number of test cases can trigger them. This is understandable: When a vulnerability is a corner case, it makes the vulnerability harder to discover.
- 1% Sampling of Collected Test Cases. We randomly select 1% of test cases (which do not trigger the SBP) in each collection methods. A brekdown of this 1% sampling dataset is shown in Table 9.

Next, we feed our two sets of the test cases into the SKYPORT-patched versions and record the requests in terms of external and global variables. We have to replay the requests in terms of external and global variables to avoid non-determinism. Then, we replay these recorded requests in the target version and determine whether both responses (including HTTP responses, databases, and other external requests if they exist) are consistent.

Note that we sample 1% of total collected test cases, because many CMSes have deployed anti-replay techniques (e.g., the adoption of random tokens) and the reply involves external and global variables. Thus, our replay involves much manual work and a full test of the entire collected cases is not scalable given the amount of human work.

Single Patch Results. In this part, we only patch a target version with a single SBP and evaluate the functionality. The "Functionality Test Pass Ratio" column in Table 10 shows the evaluation results for both the SBP-related and 1% sampling test cases. The responses for all the test cases are the same between SKYPORT-patched and target versions no matter the test cases trigger the SBP or not.

Multiple Patch Results. In this part, we patch a target version with multiple SBPs and evaluate its functionality. The "CVEs Patched via SKYPORT" column in both Table 11 and Table 12 shows the sequence of CVEs and corresponding SBPs that SKYPORT generates. For each target version, SKYPORT applies all the SBPs in sequence and then tests SKYPORT-patched version's functionality. The "Test Case Breakdown" in both tables shows a breakdown of all the test cases and the results are in the "Test Pass Ratio" column. SKYPORT preserves web applications' functionalities even when multiple patches are backported and deployed on a target version.

7.3 RQ3: Performance

In this Research Question, we evaluate the performance overhead of SKYPORT-patched web applications. We measure the execution time of the vulnerable logic in both the pre-patch and target versions and the time of the safe logic in both post-patch and SKYPORT-patched versions with both normal and attack test cases. Then, we calculate and compare both performance overheads of the original patch and SBP. Specifically, we instrument the source file with the sink to calculate the gap between the load time of the file and the finish time of the sink or the exit time of the current file, and use this time gap as the execution time. Each execution time is calculated as the average of the time in 1,000 tests.

Table 7 shows that SBPs usually introduce a little higher overhead than the official patches. This is because SBPs re-execute the code from the function/file entry-point to the sink function to enforce the extracted patch semantics. We also observe that, under some malicious test cases, both the official patches and the SBPs introduce negative overhead. After a further investigation, we find that the patch semantics abort the current execution while the pre-patched version needs to finish the sink function execution. As a summary, our SBPs introduce negligible overhead when comparing with the official patches.

8 Discussion

In this section, we discuss the limitations of our approach, including some future work.

Table 11: Functionality Evaluation with Multiple SBPs Deployed on an SBV (SBP-related test cases).

Application	Version	CVEs Patched via SkyPort	Test Case Breakdown (# of SBP-related / # of Total)					Test Pass Ratio
			Tutorial	Crawlers	Monkey	Scanner	Manual	
OpenEMR	5.0.0	CVE-2018-10571, CVE-2018-10572, CVE-2018-15139	7 / 816	2 / 1,174	0 / 1,270	0 / 602	18 / 18	27 / 27
	5.0.0.5	CVE-2018-10571, CVE-2018-10572, CVE-2018-15139	7 / 865	2 / 950	0 / 1,102	0 / 935	18 / 18	27 / 27
	4.1.1	CVE-2018-10571, CVE-2018-10572	5 / 863	1 / 809	0 / 583	0 / 327	12 / 12	18 / 18
MantisBT	2.5.0	CVE-2017-12061, CVE-2017-12062	5 / 901	48 / 340	0 / 506	0 / 674	8 / 8	61 / 61
Piwigo	2.9.0	CVE-2017-10682, CVE-2017-17824	4 / 297	0 / 1,735	1 / 5,013	0 / 496	6 / 6	11 / 11
	2.6.5	CVE-2017-10682, CVE-2017-17824	4 / 244	1 / 1,104	0 / 447	0 / 339	6 / 6	11 / 11
	2.4.5	CVE-2017-10682, CVE-2017-17824	4 / 300	2 / 890	0 / 1,721	0 / 320	6 / 6	12 / 12

Table 12: Functionality Evaluation with Multiple SBPs Deployed on an SBV (1% sampling of non SBP-related test cases).

Application	Version	CVEs Patched via SkyPort	Test Case Breakdown (# of 1% sampling / # of Total)				Test Pass Ratio
			Tutorial	Crawlers	Monkey	Scanner	
OpenEMR	5.0.0	CVE-2018-10571, CVE-2018-10572, CVE-2018-15139	8 / 809	11 / 1,172	13 / 1,270	6 / 602	38 / 38
	5.0.0.5	CVE-2018-10571, CVE-2018-10572, CVE-2018-15139	9 / 858	10 / 948	11 / 1,102	9 / 935	39 / 39
	4.1.1	CVE-2018-10571, CVE-2018-10572	9 / 858	8 / 808	5 / 583	3 / 327	25 / 25
MantisBT	2.5.0	CVE-2017-12061, CVE-2017-12062	9 / 896	3 / 292	5 / 506	7 / 674	24 / 24
Piwigo	2.9.0	CVE-2017-10682, CVE-2017-17824	3 / 293	18 / 1,735	50 / 5,012	5 / 496	76 / 76
	2.6.5	CVE-2017-10682, CVE-2017-17824	2 / 240	11 / 1,103	4 / 447	4 / 339	21 / 21
	2.4.5	CVE-2017-10682, CVE-2017-17824	3 / 296	9 / 888	17 / 1721	3 / 320	32 / 32

Sink Capability Comparison. One limitation of our approach is that our current implementation conservatively identifies subset relations between two reaching conditions and data-flow expressions if a string comparison exists. Although such cases are very rare in practice, they may lead to false negatives. In the future, we plan to enhance string solvers [24, 79] and incorporate them into SkyPort.

SBP Management. Another potential limitation is that SBPs will lead to more fragmented versions of web applications, causing potential management issues. Even so, we would like to note that web application fragmentation is an existing issue because many website developers add their own code. More importantly, Table 11 shows that the application of multiple SBP can still preserve the functionality of web applications while fixing corresponding vulnerabilities. Meanwhile, as a long-term solution, if upgrade APIs exist for certain web applications, we still recommend administrators upgrade web applications to avoid further version fragmentations.

Adaption to non-PHP Web Applications. Our prototype of SkyPort is designed for PHP applications; at the same time, the general approach on backporting security patches is applicable to other non-PHP (e.g., Node.js and Java) web applications. For the adaptation, we need to port all four modules with the static analysis tools in other languages (e.g., Soot [70] for Java).

9 Related Work

In this section, we present the most related work to our paper.

Web Application Vulnerability. Web application vulnerabilities remain to be one of the major security threats today. Previous works usually use program analysis tools to detect web vulnerabilities. Due to the popularity, plenty of works focus on detecting PHP application vulnerabilities [21, 35, 62, 75], while recently there are also several works that detect vulnerabilities of Node.js applications [20, 41, 42, 54, 63] and Java applications [46, 52, 60]. To analyze PHP applications, Backes *et al.* designed the PHPJoern [12] tool which borrows the concept of the code property graph [77] to PHP code analysis. Based on PHPJoern, Alhuzali *et al.* [9, 10] proposed the navigation graph to model relationships among several HTTP requests, which help to detect inter-request vulnerabilities.

Though a general vulnerability detection tool (e.g., PHPJoern [12], NAVEX [10]) can detect multiple types of vulnerabilities (e.g., cross-site scripting and SQL injection), some works mainly focus on detecting a specific type of vulnerabilities. SQLCIV [71] is designed to detect SQL injection vulnerabilities, while [22, 39] considered the second-order SQL injection vulnerabilities. To detect file upload vulnerabilities, UChecker [30] uses symbolic execution and features a vulnerability-oriented locality analysis to reduce the workload of symbolic execution, while FUSE [40] uses fuzzing techniques to mutate normal upload requests and bypass content-filter checks. Dahse *et al.* [23] presented an automatic technique to detect and exploit PHP object injection (POI) vulnerabilities by chaining code gadgets.

Compared with the vulnerabilities that have explicit sink functions (e.g., XSS [17] and SQLi), logic flaws are more difficult to detect. Existing works mine behavioral specifications [28] or behavioral patterns [58] from normal operations and network traces to detect common logic flaws in web applications. Besides, Sun *et al.* [68] used invariant analysis to detect the specific payment logic flaws in e-commerce web applications. Access control vulnerability is a traditional logic flaw for websites. Sun *et al.* [67] proposed inferring privileged pages from per-role sitemaps and identifying missed access controls before these pages, while MACE [53] identifies privilege escalation vulnerabilities through authorization state inconsistency checking. To detects cross-site request forgery (CSRF) vulnerabilities, Deemon [59] models the execution traces, data-flows, and architecture tiers in a unified, comprehensive property graph and detects such vulnerabilities by querying the graph. Execution After Redirect (EAR) vulnerability is a new logic flaw discovered by Doupé *et al.* [25] and Payet *et al.* [57] further performed a large-scale measurement of EAR vulnerabilities in the real world.

Given so many researches on detecting web vulnerabilities, it is quite important to mitigate and prevent web vulnerabilities [14–16]. Diglossia [61] prevents code injection attacks with precision and efficiency, and SQLBlock [32] mitigates SQL injection on legacy web applications. To mitigate XSS vulnerabilities, Content Security Policy (CSP) [13, 55, 64, 72] is proposed. Debloating is also used to reduce the attack surface of Node.js applications [38] and PHP applications [8, 11]. Although these techniques are effective in mitigating web vulnerabilities, their scopes are limited.

Security Patches. Security patches are quite important in fighting against vulnerabilities. However, patch development is quite difficult and usually requires a lot of manual efforts. Existing works propose several techniques to automatically generating patches for vulnerabilities. First, security patches are generated by learning from human-written patches [47,49]. However, such method fails to fix the vulnerability sometimes because it can only approximate the properties that are necessary to prevent the vulnerability. Second, pre-defined safe properties and exploit inputs are used to generate security patches by understanding the root cause of a vulnerability [31]. Besides, search-based patch generation could infer a correct patch from many normal test cases [36, 48, 73]. Due to the requirement on qualified inputs, the above method is hard to apply to patch backporting. In addition to the difficulties in developing patches, deploying patches is also quite difficult. To reduce the testing efforts when deploying patches, Machiry *et al.* [50] proposed a safe patch identification approach which could ease the propagation of some patches. Besides, hot-patches and hot-patching frameworks [18,56,76] are proposed to directly fix the vulnerability without requiring users to explicitly updating their software. However, these works usually assume they have the correct corresponding patches for the target software, which does not fit the assumption of patch backporting.

10 Conclusion

Vulnerable web applications are often left unpatched because the version targeted by the patch may be different from the one running in real-world websites. The problem of applying a security patch to an older version of web application is called security patch backporting. The general problem of backporting is hard because it is challenging to model vulnerable and safe logic across various application versions. In this paper, we focus on injection vulnerabilities with explicit sinks and model vulnerable logic as a new concept called sink capability. A sink capability contains all the control-flow constraints (called reaching conditions) and the symbolic expression of the critical parameter of the sink function (called data-flow expressions) of all control-flows leading to the sink. We implemented a prototype of security patch backporting on injection vulnerabilities, called SkyPort, and evaluated it on a dataset of 155 security patches and 1,526 target old versions. Our evaluation shows that SkyPort outperforms the state-of-the-art, i.e., direct application of the security patch and web application upgrade to the safe version.

Acknowledgement

We would like to thank our shepherd, Giancarlo Pellegrino, and the anonymous reviewers for their helpful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (U1836210, U1836213, 62172105, 61972099, 62172104, 62102091, 62102093), Natural Science Foundation of Shanghai (19ZR1404800). Yuan Zhang was supported in part by the Shanghai Rising-Star Program 21QA1400700 and the Shanghai Pilot Program for Basic Research-Fudan University 21TQ1400100 (21TQ012). Yinzhi Cao was supported in part by National Science Foundation (NSF) under grants CNS-20-46361 and CNS-18-54001 and Defense Advanced Research Projects Agency (DARPA) under AFRL Definitive Contract FA875019C0006. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF or DARPA. Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of CyberSecurity Auditing and Monitoring, Ministry of Education, China.

References

[1] CVE Details Website. https://cve.mitre.org.

[2] Drupalgeddon 2.0 Still Haunting 115K+ Sites. https://threatpost.com/drupalgeddon-2-0-still-haunting-115k-sites/132518/.

[3] National Vulnerability Database. https://nvd.nist.gov/.

[4] Neo4j Graph Platform. https://neo4j.com/.

[5] Open EMR. https://www.open-emr.org/.

[6] Salsi, U. PHPLint Reference Manual. http://www.icosaedro.it/phplint/manual.html.

[7] WolframScript Website. https://www.wolfram.com/wolframscript/.

[8] R. J. Alexander Bulekov and M. Egele. Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021.

[9] A. Alhuzali, B. Eshete, R. Gjomemo, and V. Venkatakrishnan. Chainsaw: Chained Automated Workflow-based Exploit Generation. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 641–652, 2016.

[10] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pages 377–392, 2018.

[11] B. A. Azad, P. Laperdrix, and N. Nikiforakis. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019.

[12] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 334–349. IEEE, 2017.

[13] S. Calzavara, A. Rabitti, and M. Bugliesi. Content Security Problems? Evaluating the Effectiveness of Content Security Policy in the Wild. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[14] Y. Cao, Z. Li, V. Rastogi, Y. Chen, and X. Wen. Virtual browser: a virtualized browser to sandbox third-party javascripts with enhanced security. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 8–9, 2012.

[15] Y. Cao, V. Rastogi, Z. Li, Y. Chen, and A. Moshchuk. Redefining web browser principals with a configurable origin policy. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[16] Y. Cao, Y. Shoshitaishvili, K. Borgolte, C. Kruegel, G. Vigna, and Y. Chen. Protecting web-based single sign-on protocols against relying party impersonation attacks through a dedicated bi-directional authenticated secure channel. In *International Workshop on Recent Advances in Intrusion Detection*, pages 276–298. Springer, 2014.

[17] Y. Cao, V. Yegneswaran, P. A. Porras, and Y. Chen. Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks. In *NDSS*, 2012.

[18] Y. Chen, Y. Li, L. Lu, Y.-H. Lin, H. Vijayakumar, Z. Wang, and X. Ou. InstaGuard: Instantly Deployable Hot-patches for Vulnerable System Programs on Android. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium (NDSS)*.

[19] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei. Adaptive Android Kernel Live Patching. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017.

[20] B. Chinthanet, S. E. Ponta, H. Plate, A. Sabetta, R. G. Kula, T. Ishio, and K. Matsumoto. Code-based Vulnerability Detection in Node.js Applications: How Far Are We? In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1199–1203, 2020.

[21] J. Dahse and T. Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Proceedings of the 21tth ISOC Network and Distributed System Security Symposium (NDSS)*, pages 23–26, 2014.

[22] J. Dahse and T. Holz. Static Detection of Second-order Vulnerabilities in Web Applications. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, pages 989–1003, 2014.

[23] J. Dahse, N. Krein, and T. Holz. Code Reuse Attacks in PHP: Automated POP Chain Generation. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[24] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[25] A. Doupé, B. Boe, C. Kruegel, and G. Vigna. Fear the EAR: Discovering and Mitigating Execution after Redirect Vulnerabilities. In *Proceedings of the 18th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 251–262, 2011.

[26] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee. Automating Patching of Vulnerable Open-source Software Versions in Application Binaries. In *Proceedings of the 26th ISOC Network and Distributed System Security Symposium (NDSS)*, 2019.

[27] T. Durieux, Y. Hamadi, and M. Monperrus. Production-driven Patch Generation. In *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*, pages 23–26. IEEE, 2017.

[28] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the 19th USENIX Security Symposium (USENIX Security)*, 2010.

[29] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos. ShrinkWrap: VTable Protection without Loose Ends. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*.

[30] J. Huang, Y. Li, J. Zhang, and R. Dai. UChecker: Automatically Detecting PHP-based Unrestricted File Upload Vulnerabilities. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019.

[31] Z. Huang, D. Lie, G. Tan, and T. Jaeger. Using Safety Properties to Generate Vulnerability Patches. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, pages 539–554. IEEE, 2019.

[32] R. Jahanshahi, A. Doupé, and M. Egele. You Shall Not Pass: Mitigating SQL Injection Attacks on Legacy Web Applications. In *Proceedings of the 15th ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*, 2020.

[33] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proceedings of the 21st ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[34] J. Jang, A. Agrawal, and D. Brumley. ReDeBug: Finding Unpatched Uode Clones in Entire OS Distributions. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, pages 48–62. IEEE, 2012.

[35] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2006.

[36] D. Kim, J. Nam, J. Song, and S. Kim. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013.

[37] S. Kim, S. Woo, H. Lee, and H. Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2017.

[38] I. Koishybayev and A. Kapravelos. Mininode: Reducing the Attack Surface of Node.js Applications. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[39] D.-g. Le, X. Li, S.-r. Gong, and L.-x. ZHENG. Research on Second-order SQL Injection Techniques. *Journal on Communications*, 36(Z1):85, 2015.

[40] T. Lee, S. Wi, S. Lee, and S. Son. FUSE: Finding File Upload Bugs via Penetration Testing. In *Proceedings of the 27th ISOC Network and Distributed System Security Symposium (NDSS)*. Network & Distributed System Security Symposium, 2020.

[41] S. Li, M. Kang, J. Hou, and Y. Cao. Detecting node.js prototype pollution vulnerabilities via object lookup analysis. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 268–279, 2021.

[42] S. Li, M. Kang, J. Hou, and Y. Cao. Mining node.js vulnerabilities via object dependence graph and query. In *Proceedings of the 31th USENIX Security Symposium (USENIX Security)*, 2022.

[43] Y. Li, S. Wang, and T. N. Nguyen. DLfix: Context-based Code Transformation Learning for Automated Program Repair. In *Proceedings of the 42th International Conference on Software Engineering (ICSE)*, 2020.

[44] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu. VulPecker: An Automated Vulnerability Detection System based on Code Similarity Analysis. In *Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC)*, pages 201–213, 2016.

[45] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. VulDeePecker: A Deep Learning-based System For Vulnerability Detection. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium (NDSS)*, 2018.

[46] B. Livshits and M. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium (USENIX Security)*, 2005.

[47] F. Long, P. Amidon, and M. Rinard. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 727–739, 2017.

[48] F. Long and M. Rinard. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. IEEE.

[49] F. Long and M. Rinard. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016.

[50] A. Machiry, N. Redini, E. Camellini, C. Kruegel, and G. Vigna. SPIDER: Enabling Fast Patch Propagation in Related Software Repositories. In *Proceedings of the 41th IEEE Symposium on Security and Privacy (S&P)*.

[51] N. Magnezi and M. Kolesnik. Backporting of Bug Patches, Mar. 28 2017. US Patent 9,606,793.

[52] H. Man, J. An, W. Huang, and W. Fan. JSEFuzz: Vulnerability Detection Method for Java Web Application. In *Proceedings of the 3rd International Conference on System Reliability and Safety (ICSRS)*, page 6, 2018.

[53] M. Monshizadeh, P. Naldurg, and V. Venkatakrishnan. MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 690–701, 2014.

[54] B. Nielsen, B. Hassanshahi, and F. Gauthier. Nodest: Feedback-driven Static Analysis of Node.js Applications. In *Proceedings of the 27th Joint Meeting on Foundations of Software Engineering (FSE)*, 2019.

[55] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou. CSPAutoGen: Black-box Enforcement of Content Security Policy upon Real-World Websites. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[56] M. Payer and T. R. Gross. Hot-patching A Web Server: A Case Study of ASAP Code Repair. In *Proceedings of the 11th Eleventh Annual Conference on Privacy, Security and Trust (PST)*, pages 143–150. IEEE, 2013.

[57] P. Payet, A. Doupé, C. Kruegel, and G. Vigna. EARs in the Wild: Large-scale Analysis of Execution after Redirect Vulnerabilities. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*.

[58] G. Pellegrino and D. Balzarotti. Toward Black-Box Detection of Logic Flaws in Web Applications. In *Proceedings of the 21st ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[59] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow. Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[60] B. Qu, B. Liang, S. Jiang, and C. Ye. Design of Automatic Vulnerability Detection System for Web Application Program. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 89–92, 2013.

[61] S. Son, K. McKinley, and V. Shmatikov. Diglossia: Detecting Code Injection Attacks with Precision and Efficiency. In *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1181–1192, 2013.

[62] S. Son and V. Shmatikov. SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 1–13, 2011.

[63] C.-A. Staicu and M. Pradel. Freezing the Web: A Study of ReDoS Vulnerabilities in Javascript-based Web Servers. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pages 361–376, 2018.

[64] S. Stamm, B. Sterne, and G. Markham. Reining in the Web with Content Security Policy. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pages 921–930, 2010.

[65] B. Stock, G. Pellegrino, F. Li, M. Backes, and C. Rossow. Didn't You Hear Me?—Towards More Successful Web Vulnerability Notifications. In *Proceedings of the 25th ISOC Network and Distributed System Security Symposium (NDSS)*, 2018.

[66] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes. Hey, You Have a Problem: On the Feasibility of Large-scale Web Vulnerability Notification. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, pages 1015–1032, 2016.

[67] F. Sun, L. Xu, and Z. Su. Static Detection of Access Control Vulnerabilities in Web Applications. In *Proceedings of the 18th ISOC Network and Distributed System Security Symposium (NDSS)*, 2011.

[68] F. Sun, L. Xu, and Z. Su. Detecting Logic Vulnerabilities in E-commerce Applications. In *Proceedings of the 21st ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[69] Y. TIAN. *Mining Software Repositories for Automatic Software Bug Management from Bug Triaging to Patch Backporting*. Dissertations and theses collection, Singapore Management University, 2017.

[70] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java Bytecode Optimization Framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, page 13. 1999.

[71] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 32–41, 2007.

[72] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. CSP is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[73] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung. Context-aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018.

[74] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, and W. Zou. MVP: Detecting Vulnerabilities using Patch-enhanced Vulnerability Signatures. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, pages 1165–1182, 2020.

[75] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th USENIX Security Symposium (USENIX Security)*, pages 179–192, 2006.

[76] Z. Xu, Y. Zhang, L. Zheng, L. Xia, C. Bao, Z. Wang, and Y. Liu. Automatic Hot Patch Generation for Android Kernels. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, pages 2397–2414, 2020.

[77] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2014.

[78] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song. VTrust: Regaining Trust on Virtual Calls. In *Proceedings of the 23rd ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.

[79] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 114–124, 2013.