Automated Patch Transplantation

RIDWAN SALIHIN SHARIFFDEEN, National University of Singapore, Singapore
SHIN HWEI TAN, Southern University of Science and Technology, China
MINGYUAN GAO and ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

Automated program repair is an emerging area that attempts to patch software errors and vulnerabilities. In this article, we formulate and study a problem related to automated repair, namely automated patch transplantation. A patch for an error in a donor program is automatically adapted and inserted into a "similar" target program. We observe that despite standard procedures for vulnerability disclosures and publishing of patches, many un-patched occurrences remain in the wild. One of the main reasons is the fact that various implementations of the same functionality may exist and, hence, published patches need to be modified and adapted. In this article, we therefore propose and implement a workflow for transplanting patches. Our approach centers on identifying patch insertion points, as well as namespaces translation across programs via symbolic execution. Experimental results to eliminate five classes of errors highlight our ability to fix recurring vulnerabilities across various programs through transplantation. We report that in 20 of 24 fixing tasks involving eight application subjects mostly involving file processing programs, we successfully transplanted the patch and validated the transplantation through differential testing. Since the publication of patches make an un-patched implementation more vulnerable, our proposed techniques should serve a long-standing need in practice.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; **Software testing and debugging**; *Software reliability*;

Additional Key Words and Phrases: Program repair, code transplantation, patch transplantation, dynamic program analysis

ACM Reference format:
Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. 2020. Automated Patch Transplantation. *ACM Trans. Softw. Eng. Methodol.* 30, 1, Article 6 (December 2020), 36 pages.
https://doi.org/10.1145/3412376

6

This work was partially supported by the National Natural Science Foundation of China (Grant No. 61902170) and Natural Science Foundation of Guangdong Province (Grant No. 2020A1515011494). This work was partially supported by the National Satellite of Excellence in Trustworthy Software Systems, funded by NRF Singapore under the National Cybersecurity R & D (NCR) program.
Authors' addresses: R. S. Shariffdeen, M. Gao, and A. Roychoudhury, National University of Singapore, Singapore; emails: {ridwan, gaomy, abhik}@comp.nus.edu.sg; S. H. Tan (corresponding author), Southern University of Science and Technology, China; email: tansh3@sustech.edu.cn.

1049-331X/2020/12-ART6 $15.00
https://doi.org/10.1145/3412376

1 INTRODUCTION

Automated program repair [15] is an emerging research area that involves automatically fixing observable software errors or vulnerabilities. The goal of automated program repair is to (minimally) modify a program to meet a correctness criterion such as passing of given test(s) and/or satisfying a formal specification such as an assertion. The last decade has witnessed a flurry of research activities in building automated repair techniques that use search [61], program analysis [37], and machine learning [26], as well as judicious combinations of such machinery. In this article, we propose and study a related but different problem. If the patch for an error or a vulnerability in a buggy program P is available (e.g., the vulnerability has been patched manually and the fix is available), can the patch be automatically transplanted or adapted into another "similar" buggy program P'? We call this the *automated patch transplantation* problem.

Many use-cases that can benefit from a solution to automate patch transplantation exist. First, security fixes in the latest software version may be "backported" to older program versions. Such backporting is not restricted to security fixes but also can be used to enhance compatibility issues in software versions, such as managing the collateral evolution of device drivers to enable their functioning despite evolution of the operating system (e.g., prior work on evolution of Linux backporting [38]).

Second, patch transplantation can be useful for propagating fixes to different implementations of the same protocol or functionality, as opposed to different versions of the same program. Implementations of the same protocol or functionality can differ due to the difference in the programming language or difference in implementation while using the same programming language. Porting a patch across languages is more challenging and beyond the scope of the patch transplantation problem. To elaborate on the transplantation across different implementations, let us consider the *Heartbleed* vulnerability (CVE-2014-0160), which could lead to disclosure of private information by applications using OpenSSL [10]. Although a patch for the Heartbleed vulnerability is available, it cannot be immediately inserted into any OpenSSL implementation; instead, the patch needs to be *adapted*. As different web servers rely on different implementations of OpenSSL, Heartbleed continues to persist in the wild [49], despite the patch being widely available. Thus, by automatically adapting patches of Heartbleed to other vulnerable OpenSSL implementations, we can reduce the exposure to vulnerabilities.

In general, one of the crucial steps towards defense against published exploits is to integrate available patches into one's system as quickly as possible. The challenge in incorporating patches from different sources is to be able to adapt the code modifications involved. Often, shared libraries are customized with new features, different data structures, or rewriting of previous implementation to match the integrated environment. Hence, directly applying a general patch is not trivial and sometimes difficult. This is the problem addressed by our work.

Problem Statement. Given buggy and fixed donor programs P_a, P_b, and a buggy program similar to P_a, also called a host or target program P_c, the goal is to fix P_c to produce a fixed version of P_c, namely P_d. We assume that P_a and P_c fail on the same failing input t_F.

For security patches, t_F is an exploit that takes advantage of an existing software vulnerability. A formulation of the automated patch transplantation problem explaining the inputs and outputs of the problem appears in Figure 1. Note that P_a, P_b, P_c, t_F are inputs to the patch transplantation problem, and the output is P_d, the program with the transplanted patch.

Differences with Other Problems Studied. We note that the patch transplantation problem formulated by our work is *different*, though related, to the program repair problem and the program transplantation problem. The program repair problem seeks to (minimally) modify a program so

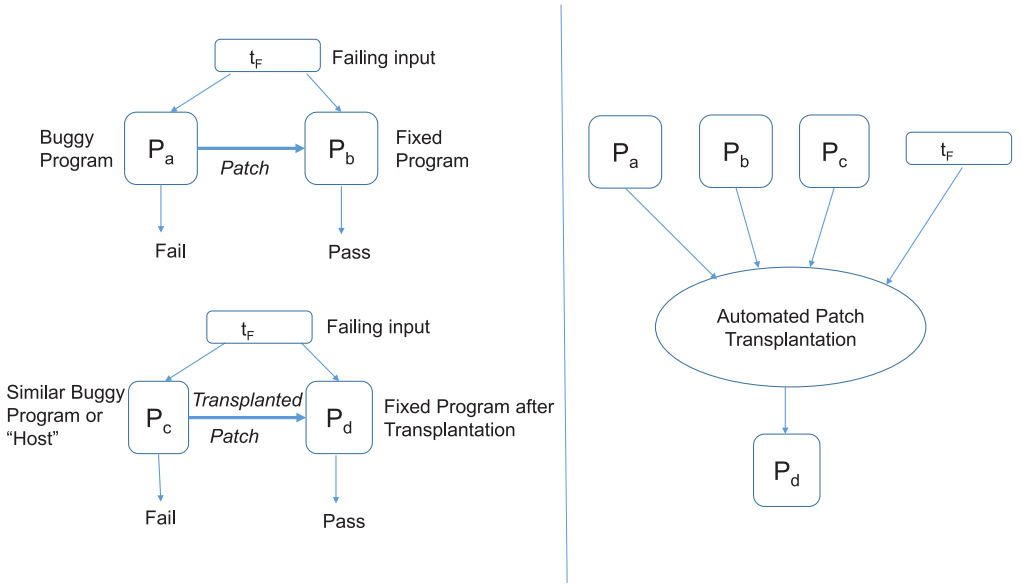


Fig. 1. The automated patch transplantation problem.

as to meet a correctness criterion such as passing a given test suite. To relieve the burden of fixing bugs, many techniques have been previously proposed for automated program repair such as genetic programming, semantic-analysis-based repair techniques, and machine-learning-guided techniques. Unlike the automated repair problem, we do not try to synthesize patches or fit into patch patterns; instead, the patch transplantation problem is more "goal directed"—it automatically identifies a patch from the donor, extracts the identified patch, computes an insertion location for the patch in the target program, and inserts that patch by adapting to the context of the insertion point. The patch transplantation problem is also different from the program transplantation problem. The program transplantation problem deals with transplanting a feature from program P into another program P' such that the transplanted feature must not disrupt the existing functionality of its target P' and must actually execute and add the functionality of the desired feature to its target. These techniques are limited to transferring a logical block of code (such as a function or a check) instead of patches that may involve several disjoint blocks of code. Moreover, program transplantation techniques typically require manual identification of the insertion point, unlike our patch transplantation problem, where the insertion point is automatically identified.

Contributions. The contributions of this article can be summarized as follows:

(1) We propose the patch transplantation problem, which demonstrates a real practical issue in security of open source software. When vulnerabilities are detected and manually fixed, it provides an opportunity to the attackers to easily exploit the vulnerability on similar un-fixed implementations. The patch transplantation task takes care of this concern by automatically adapting available manual fixes for "similar" un-fixed open source programs.
(2) We present PatchWeave, a technique that automatically extracts patches and transplants them to another target program that fails on the same test case. We also propose a classification for patch transplantation based on the difficulty and use this classification to adapt the transplantation process.

```
405  static void j2k_read_siz (opj_j2k_t *j2k) {
415    image->y1 = cio_read(cio, 4);    /* Ysiz */
416    image->x0 = cio_read(cio, 4);    /* X0siz */
417    image->y0 = cio_read(cio, 4);    /* Y0siz */
418    cp->tdx = cio_read(cio, 4);      /* XTsiz */
419    cp->tdy = cio_read(cio, 4);      /* YTsiz */
420    cp->tx0 = cio_read(cio, 4);      /* XT0siz */
421    cp->ty0 = cio_read(cio, 4);      /* YT0siz */

423    if ((image->x0 < 0)||(image->x1 < 0)||(image->y0 < 0)||(image->y1 < 0)) {
424      opj_event_msg(j2k->cinfo, ...);
427      return;
428    } ...
517    cp->tw = int_ceildiv(image->x1 - cp->tx0, cp->tdx);
518    cp->th = int_ceildiv(image->y1 - cp->ty0, cp->tdy);
519    ...
       /* the overflow error */
560    cp->tcps = (opj_tcp_t*) opj_calloc(cp->tw * cp->th, sizeof(opj_tcp_t)); ...
622  }
```

Fig. 2. Overflow error in OpenJPEG 1.5.1.

```
405  static void j2k_read_siz (...) { ....
423  - if ((image->x0<0)||(image->x1<0)||
         (image->y0<0)||(image->y1<0)) {
423  + if ((image->x0==0)||(image->x1<0)||
         (image->y0<0)||(image->y1<0)) {
       ....
622  }
```

(a) Patch generated by F1X

```
405  static void j2k_read_siz (...) { ....
518  - cp->th = int_ceildiv (image->y1 - cp->ty0, cp->tdy);
518  + if (!(1) )
519  +   cp->th = int_ceildiv (image->y1 - cp->ty0, cp->tdy);
520
       ....
623  }
```

(b) Patch generated by Prophet

Fig. 3. Patches generated using Automated Program Repair.

(3) We conduct an evaluation of our approach on real-world programs, specifically in transplanting fixes of reported vulnerabilities. Our evaluation also compares PatchWeave against several approaches (including F1X, Prophet, μSCALPEL, and our version of LASE for C programs). These experiments demonstrate the efficacy of our method in adapting patches for software vulnerabilities.

2 MOTIVATIONAL EXAMPLE

We next present an example of an integer overflow error in OpenJPEG (a C library for the open source JPEG2000 codec) to have a better understanding of the challenges in the patch transplantation problem. Figure 2 shows the integer overflow error in OpenJPEG; this code snippet is simplified for brevity. There is a potential overflow at line 560, where OpenJPEG allocates memory to `cp->tcps` by computing the value as `cp->tw * cp->th`. Input image files with large width and height fields may cause the calculation at line 560 to overflow, eventually writing beyond the end of the allocated buffer. In the error-triggering input, the JPG file height field is 210 and the width field is 2147483646.

Automated Program Repair (APR). Consider the scenario where we attempt to fix this bug using two state-of-the-art program repair tools (F1X [31] and Prophet [26]). Since these two APR techniques require a test suite, we created a test suite inclusive of the failing test case and a passing test case, and also provided the correct location for the fix to generate the patch, which enables us to compare the two patches generated at the same location. Figures 3(a) and 3(b) show the

two patches generated by F1X and Prophet, respectively. F1X was able to generate a patch that modifies an existing if statement to avoid the failing test case. However, the fix in Figure 3(a) does not generalize for test cases beyond the given test suite, since it only fixes the two given test cases. Similarly, Prophet generated a patch that omits the execution of a statement (i.e., Line 518 in Figure 2) by inserting a condition that is always false (i.e., semantically equivalent to deleting a statement). The correct patch for this bug would be to evaluate if the computation of `cp->tw * cp->th` would result in an overflow and avoid the overflow by following an error handling procedure. Both APR-generated patches failed to generalize the patch to the extent of avoiding the overflow but rather generated a patch that could simply pass the failing test suite. Prophet is a technique based on machine learning that generates arbitrary code changes and relies on the test suite for correctness, whereas F1X is semantic based, which uses code analysis to generate the patch that attempts to address the underlying bug rather than changing the code just enough to satisfy the test suite. However, it relies on the test suite to generate the correct constraints to obtain the correct patch. If sufficient test cases are provided, the generated patch would be more generalized. This example highlights one of the limitations of current automated program repair techniques, which is generally known as the overfitting problem [29, 43, 52]. As illustrated in our motivational example, APR-generated patches cannot repair bugs without a sufficient number of test cases to generalize the patch, and the sufficient number differs from one bug to another. When used to fix security vulnerabilities, such inaccurate patches could lead to undesirable effects by leading one to believe that the vulnerability has been fixed when in fact it has not.

Patch Transplantation. A different implementation of the JPEG2000 codec can be found in JasPer (a utility for image manipulation), which could also serve as a fix for this vulnerability in OpenJPEG. We discovered that the input file that exploits the vulnerability in OpenJPEG 1.5.1 is able to exploit the same vulnerability in JasPer 1.900.12, and also fixes the bug in JasPer 1.900.13. Our approach will extract the code that fixes the bug in JasPer 1.900.13 and insert it into OpenJPEG 1.5.1 in the following way. First, we build the binaries with an integer sanitizer to identify potential overflow locations. Next, we check if the two programs share the same vulnerability. In our example, both JasPer 1.900.12 and OpenJPEG 1.5.1 throw integer overflow errors due to multiplication operation of 210 * -2147483646. We identify line number jpc_dec.c@1234 in JasPer 1.900.12 as the donor buggy location and line number j2k.c@560 in OpenJPEG 1.5.1 as the target buggy location.

Donor Selection. Figure 5(a) shows the patch for JasPer 1.900.13, which includes a check. Specifically, JasPer at commit b9be3d9 contains the vulnerability and JasPer at commit d91198a contains the fix (i.e., the pair selection is JasPer-b9be3d9 (P_a), JasPer-d91198a (P_b), OpenJPEG-1.5.1 (P_c)).

Patch Extraction. Next, PatchWeave analyzes the source code diff and aligns the execution trace of P_a and P_b to narrow down the changes to the patch that fix the bug. Lines 1198 and 1235–1238 in Figure 5(a) are the changes in JasPer 1.900.13 that check if the multiplication results in an overflow through a function `jas_safe_size_mul`. This function determines if the two parameters (`dec->numhtiles` and `dec->numvtiles`) can be multiplied without an overflow. If an overflow occurs, it will return false; otherwise, it will return true and assign the multiplication result to the `size` variable.

Concolic Execution. Once we identify the patch for transplantation, we need to translate the statements in the patch and find the insertion point. For this purpose, we perform concolic execution [48] on all three programs in our pair selection with the same input file to capture the symbolic paths for the execution of each program. When identifying the insertion point for the transplantation, the patch can be inserted at any location from the starting point of the execution trace to the crashing point (or a suspicious buggy point). As there could be potentially many candidate locations, we identify a divergent point (see Definition 4.1) in P_c, similar to the divergence caused by the patch in P_b with respect to P_a. A divergence in the trace of P_b with respect

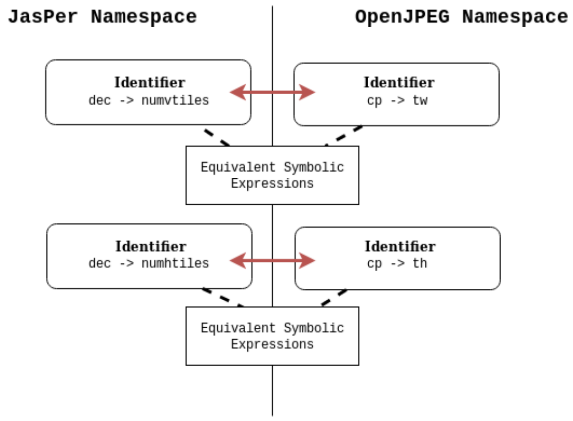


Fig. 4. Variable mapping between OpenJPEG and JasPer.

to P_a is caused by a code change (i.e., patch) that resulted in the difference and is most likely to be a potential divergent point at which the patch has been applied. One of the potential divergent points is the condition at line 1235 in Figure 5(a). We find a similar location in P_c using partial path condition dominance (see Definition 4.3). We calculate the partial path condition in P_b at line 1235 in Figure 5(a), and we traverse through the execution trace of P_c to find a similar location where the partial path condition dominates, i.e., d_c.

Candidate Function. Once we have identified a divergent point in P_c, the next step is to traverse through the estimated divergent point d_c and the crashing point l_c in P_c, in order to identify the candidate functions to transplant the patch. For each function f executed in P_c from and inclusive of d_c up to l_c, we consider the variables used within the function f. For each variable v in f, we capture the symbolic expressions and generate a mapping with the variables used in our patch. More precisely, we check for a function that has a mapping for the variables `dec->numvtiles` and `dec->numhtiles`, which is likely to be a candidate function. Among all candidate functions, we choose the first candidate in the trace (i.e., the function that executes first) for two reasons: (1) the patch can impact more paths and (2) the vulnerability is fixed earlier in the execution of the failing input. In our example, the divergent point and the crashing point lie in the same function `j2k_read_siz`. However, this may not hold for the general cases as the divergent point and the crashing point can be in two different functions.

Candidate Location. We consider the availability of the variables we mapped at the previous stage for each location in our candidate function to find candidate locations for our patch. For each statement inside the function, we compute the list of available variables and find the candidate points where the variables in our generated mapping are usable. Among the candidate locations, we choose the first candidate for the same reasons mentioned above. In our example, we choose line number 519 in Figure 2.

Code Transplantation. Once we have identified the insertion location as line number 519 in Figure 2, our next step is to translate the patch to the namespace of P_c and insert the code at the identified insertion location. We make use of Abstract Syntax Tree (AST) node context information from both P_a and P_c programs to adapt to the insertion point context. Then, we translate the variables to the namespace of OpenJPEG using symbolic analysis (explained in Section 4.3), where we obtain the mappings of `dec->numvtiles` into `cp->tw` and `dec->numhtiles` into `cp->th` as illustrated in Figure 4. We use this mapping to translate the variable names in the patch while weaving the patch into the insertion point in P_c. Using dependency analysis, we identify that the

```
1188   static int jpc_dec_process_siz(...)
       {
           ....
1198   + size_t size;
           ...
       /* overflow check */
1235   +   if (!jas_safe_size_mul(dec->numhtiles, dec->
             numvtiles, &size)){
1236   +       return -1;
1237   +   }
       -   dec->numtiles = dec->numhtiles * dec->numvtiles;
1238   +   dec->numtiles = size;
1241       if (!(dec->tiles=jas_alloc2(dec->numtiles,sizeof(
                 jpc_dec_tile_t)))){

1242           return -1;

1243       }
           ....
1290   }
```

(a) Developer patch in JasPer 1.900.13

```
       /* Adapted patch for OPENJPEG 1.5.1 */
48     + #define SIZE_MAX    (18446744073709551615UL)
56     + inline static bool jas_safe_size_mul(size_t x,
             size_t y, size_t *result){
57     + if (x && y > SIZE_MAX / x) {
58     +   return false;
59     +   }
60     + *result = x * y;
61     +   return true;
62     + }

440      static void j2k_read_siz (opj_j2k_t *j2k) {
554    + size_t size;
556    + if (!jas_safe_size_mul(cp->tw, cp->th, &size)) {
557    +     return -1;
558    + }
         /* the overflow error */
601      cp->tcps = (opj_tcp_t*) opj_calloc(cp->tw * cp
               ->th, sizeof(opj_tcp_t));
663    }
```

(b) Transplanted Patch in OpenJPEG 1.5.1

Fig. 5. Patch generated using transplantation.

`jas_safe_size_mul` function is missing in our target program P_c. We perform transplantation of this function by following the same steps (i.e., extract the missing source, translate the variables, find a suitable insertion location, and transplant into the target program) such that the function can be called within the inserted patch.

Final Result. Patch transplantation is able to successfully repair the bug in OpenJPEG 1.5.1, preventing any potential buffer overflows due to the integer overflow caused during the calculation of the buffer size as shown in Figure 5(b). Although F1X [31] and Prophet [26] were able to generate patches that pass the failing test case, the generated patches are of poor quality due to the quality of the test suite. This is where patch transplantation differs from test-driven patch generation: (1) The patch is a human-written patch that is extracted and then adapted to the context of the target program (i.e., P_c). Since human-written patches are more reliable and general than the generated patches via APR, we eliminate the problem of overfitting as further proved in our experimental results in Table 6 (Section 6). The result of differential fuzzing highlights that the patches generated using APR do not generalize for unseen test cases. (2) Patch transplantation can fix more bugs compared to program repair techniques because APR relies on a good test suite for fault localization to identify patch locations. Since patch transplantation does not depend on a test suite, with the use of the partial path condition dominance relationship (see Definition 4.3), we can find the correct patch location.

3 PROBLEM FORMULATION

Although programs with vulnerabilities may not share common code, they can share different implementations of the same protocol (e.g., OpenSSL) or same standard (e.g., JPEG 2000). Hence, finding a vulnerability in one program can lead to malicious users adapting attacks to other similar programs. This is the scenario we seek to prevent via automated patch transplantation. In this section, we first introduce the notations that we will use throughout the article and then formulate the problem of patch transplantation. P_a represents the buggy version of a program, whereas P_b (also known as the *donor* in the terminology used in software transplantation [8]) denotes the subsequent version in which the fault in P_a is fixed. t_F represents the test that failed in P_a but passes in P_b, while P_c denotes the target program (also known as the *host* in the terminology used in software transplantation [8]) that fails in t_F.

Definition 3.1 (Similar Vulnerability). We consider two vulnerabilities as similar if there exists a failing test t_F exploiting both vulnerabilities and the two vulnerabilities exhibit the same output in

Table 1. Types of Patches

Type	Missing Dependency?	Namespace Translation?	Example
Class I	No	No	Porting across forks
Class II	Yes	No	Backporting
Class III	No	Yes	Collateral evolution
Class IV	Yes	Yes	Collateral evolution

terms of the return code and crashing/buggy instruction, for instance, the motivational example in Section 2 where both JasPer 1.900.12 and OpenJPEG 1.5.1 exhibited similar integer overflow vulnerability for the same test case.

Definition 3.2 (Similar Programs). We consider two programs P_a and P_c as similar if there exists a failing test t_F exploiting a similar vulnerability in both programs. Our goal is to transplant a fix of P_a into the other similar program P_c.

Definition 3.3 (Patch Transplantation). Given a pair of buggy and fixed programs (P_a, P_b) and a program P_c similar to P_a, we try to extract the patch between P_a and P_b. The patch is then inserted into P_c, which involves finding an insertion point and adapting the patch with new variable mappings and context information.

3.1 Classes of Patch Transplantation

There are three major challenges in transplanting a patch from P_b to P_c due to the differences in the two programs. The first challenge is the difference between the namespace and data structures used in the two implementations where the identifiers are not identical, and hence an adaptation for the variables is required. The second challenge is to identify and transplant missing dependencies for the patch to work. For example, the patch would require a supplementary function, a subroutine or a definition that is used in the patch, which is missing in the target program P_c. The third challenge is to correctly identify the insertion location of the patch in P_c.

To perform a thorough analysis of the patches, we identify four classes of patches based on the origin of the patch and the adaptation required for the target system to apply the patch (Table 1). Further, we define an equivalence relation between the original patch and transplanted patch based on the adaptation required as given below.

Syntactically Equivalent. $Patch_{fix}$ is "Syntactically Equivalent" if $Patch_{orig}$ and $Patch_{fix}$ are exactly the same code. If the namspace and data structure of the two programs P_b and P_c are identical and the code is identical, the transplanted patch would be syntactically equivalent to the original patch.

Semantically Equivalent. $Patch_{fix}$ is "Semantically Equivalent" if $Patch_{orig}$ and $Patch_{fix}$ are not syntactically the same but produce the same semantic behavior. This requires a namespace and/or data structure translation.

3.2 Class I: Syntactically Equivalent Transplantation

No adaptation is required for the transplantation. This is the trivial case where the original patch can be applied directly. An example is the backporting of patches to past versions of the same program or porting patches across forked projects.

```
int wpa_sm_rx_eapol(...){
  ....
  if ((sm->proto == WPA_PROTO_RSN ||
       sm->proto == WPA_PROTO_OSEN) &&
       (key_info & WPA_KEY_INFO_ENCR_KEY_DATA) &&
       mic_len) {
+  if (!(key_info & WPA_KEY_INFO_MIC)) {
+     wpa_msg(sm->ctx->msg_ctx, MSG_WARNING,
     "WPA: Ignore EAPOL-Key with encry..");
+   goto out;
+  }

   if (wpa_supplicant_decrypt_key_data(sm, key,
        mic_len, ver, key_data,&key_data_len))
   ....
}
```

(a) Developer's patch for wpa_supplicant 2018-1

```
int wpa_sm_rx_eapol(...){
   ....

   if (sm->proto == WPA_PROTO_RSN &&
       (key_info & WPA_KEY_INFO_ENCR_KEY_DATA)) {

+  if (!(key_info & WPA_KEY_INFO_MIC)) {
+   wpa_msg(sm->ctx->msg_ctx, MSG_WARNING,
     "WPA: Ignore EAPOL-Key with encry..");
+  goto out;
+  }

   if (wpa_supplicant_decrypt_key_data(sm, key, ver))

   ....
}
```

(b) Developer's patch for FreeBSD SA-18:11

Fig. 6. Example for Class II: CVE-2006-4806.

The patch for CVE-2018-14526 is an example for Class I. It is a vulnerability in the processing of EAPOL-Keyframes for wpa_supplicant.[1] An attacker could modify the frame to bypass authentication. To fix this vulnerability, an official patch[2] in Figure 6(a) was released and adapted by every operating system that provides the wpa_supplicant driver. The FreeBSD driver had to integrate this patch to two different versions of its forks, and Figure 6(b) shows the patch[3] for FreeBSD 10.4. The FreeBSD developer had to identify the insertion point in the FreeBSD driver, which is different from the original patch, but no adaptation was required for the patch code itself. This highlights the fact that even for identical patches, finding the insertion point is nontrivial as the predicates in which the patch is inserted in Figure 6(a) and Figure 6(b) are different.

There is a potential issue in the case where the WPA2/RSN style of EAPOL-Key construction is used with TKIP negotiated as the pairwise cipher. Hence, a patch was released by the standard organization and adapted by every operating system that provides the wpa_supplicant driver. The developers at FreeBSD had to identify the correct insertion point, but no adaptation was required for the patch code itself as depicted above. The context is different with respect to the variable names and additional code in place in FreeBSD as shown in Figure 6(b), compared to the original patch shown in Figure 6(a).

3.3 Class II: Syntactically Equivalent Transplantation with Dependency

A dependent function is required for the patch to apply the solution. The dependent component could be from the original patch or could be a missing supplementary code in the recipient program, for example, adding functions for the patch from latest version, which is missing in the old version.

CVE-2006-4806 is an example for Class II, where the patch requires a dependent function to transplant the patch in the recipient program. It occurs due to a buffer overflow in imlib2 (an image file processing library), which could allow remote attackers to cause a denial-of-service attack. To fix the vulnerability, developers of imlib2 applied a patch (Figure 7(a)) that includes a supplementary function named `IMAGE_DIMENSIONS_OK`, which checks if the provided width and height (`im->w` and `im->h`) are within standard limits of the application to prevent memory overflow. The same vulnerability exists in older versions of imlib2, specifically in imlib2 1.4.0, which does not include the definition of the function `IMAGE_DIMENSIONS_OK`; hence, the transplantation of the

[1] wpa_supplicant is a WPA Supplicant for Linux, BSD, Mac OS X, and Windows with support for WPA and WPA2 (IEEE 802.11i /RSN).
[2] https://w1.fi/security/2018-1/0001-WPA-Ignore-unauthenticated-encrypted-EAPOL-Key-data.patch.
[3] https://www.freebsd.org/security/patches/SA-18:11/hostapd-10.patch.

```
char load(ImlibImage * im ..) {

 ....
 im->w = w = cinfo.image_width;
 im->h = h = cinfo.image_height;

 + if (!IMAGE_DIMENSIONS_OK(w, h)){
 +  im->w = im->h = 0;
 +  jpeg_destroy_decompress(&cinfo);
 +  fclose(f);
 +  return 0;

 + }
 ....
}
```

(a) Developer's patch in imlib2 1.4.3

```
+ # define IMAGE_DIMENSIONS_OK(w, h) \
+   ( ((w) > 0) && ((h) > 0) && \
+     ((unsigned long long)(w) * \
+     (unsigned long long)(w) <= \
+     (1ULL << 29) - 1) )

char load(ImlibImage * im ..) {
 ....
 im->w = w = cinfo.image_width;
 im->h = h = cinfo.image_height;
 + if (!IMAGE_DIMENSIONS_OK(w, h)){
 +  return 0;
 + }
 ....
}
```

(b) Adapted patch for imlib2 1.4.0

Fig. 7. Example for Class II: CVE-2006-4806.

```
static void ReadImage(....) {
   ....
   if(!ReadOK(fd, &c, 1)) {
        return;
   }

+  if (c > 12)
+     return;
   ....
}
```

(a) developer patch for LibGD 2.0.34 RC1

```
int readraster(void) {
   ....
   datasize = getc(infile);
+  if (datasize > 12)
+     return 0;
   clear = 1 << datasize;
   eoi = clear + 1;

   ....
}
```

(b) developer patch for Libtiff 4.0.4

Fig. 8. Example for Type II: CVE-2013-4231.

patch (Figure 7(b)) involves the dependency for the patch to correctly fix the vulnerability in imlib2 1.4.0.

3.4 Class III: Semantically Equivalent Transplantation

In this class of patches, adaptation is required to apply the transformation into the target system due to syntactic differences. For instance, when two programs are semantically equivalent but syntactically different, the patch needs to be modified before transplanting into the recipient program. This requires a namespace translation between P_b and P_c.

CVE-2013-4231 is an example of Class III class, where the patch requires an adaptation in terms of namespace translation. It occurs due to a buffer overflow in Libtiff, a library for processing Tagged Image File Format files. A bug in one of the library modules that processes GIF images causes an overflow, which can be fixed by inserting a check as shown in Figure 8(b). Since the maximum LZW bits allowed in GIF standard is 12, the patch for the overflow error involves inserting a check condition. This vulnerability also exists in LibGD and ImageMagick libraries, which are also image processing software similar to Libtiff. All three programs are vulnerable to the same exploit because they follow the same standard for GIF image processing. The adaptation required for the patch is the namespace mapping from `datasize` in Libtiff to `c` in LibGD and change of return type to match the function return type (Figure 8(a)).

```
bool jas_image_cmpt_domains_same(jas_image_t *image)
{
 int cmptno;
 jas_image_cmpt_t *cmpt;
 jas_image_cmpt_t *cmpt0;
 cmpt0 = image->cmpts_[0];
 for(cmptno = 1;
     cmptno < image->numcmpts_;
     ++cmptno){

     cmpt = image->cmpts_[cmptno];
     if (cmpt->tlx_ != cmpt0->tlx_ ||
         cmpt->tly_ != cmpt0->tly_ ||
         cmpt->hstep_ != cmpt0->hstep_ ||
         cmpt->vstep_ != cmpt0->vstep_ ||
         cmpt->width_ != cmpt0->width_ ||
         cmpt->height_ != cmpt0->height_) {
         return 0;
     }}
 return 1;
}
```

(a) Original function in Jasper 1.900.14

```
bool jas_image_cmpt_domains_same(opj_tcd_tile_t *t)
{
 int cmptno;
 opj_tcd_tilecomp_t *cmpt;
 opj_tcd_tilecomp_t *cmpt0;
 cmpt0 = &t->comps[0];
 for(cmptno = 1;
     cmptno < t->numcomps;
     ++cmptno){

     cmpt = &t->comps[cmptno];

     if (cmpt->x0 != cmpt0->x0 ||
         cmpt->y0 != cmpt0->y0 ||
         cmpt->x1 != cmpt0->x1 ||
         cmpt->y1 != cmpt0->y1) {
         return 0;
     }}
 return 1;

}
```

(b) Manually adapted function for OpenJPEG 1.5.1

Fig. 9. Example for Class IV: CVE-2016-9389.

3.5 Class IV: Semantically Equivalent Transplantation with Dependency

For a syntactically different yet semantically equal patch, which requires a dependency to be transplanted, Class IV patches are considered. The dependent component itself may require adaptation due to namespace differences between the donor P_b and the recipient P_c. This requires a namespace translation between P_b and P_c and data structure translation for the patch to work.

CVE-2016-9389 is an example for Class IV, where the patch requires a dependency to transplant the patch in the recipient program. CVE-2016-9389 is a buffer overflow vulnerability in JasPer 1.900.13 version and fixed in 1.900.14, which allows remote attackers to cause a denial-of-service attack. The same vulnerability also exists in OpenJPEG 1.5.1: adapting the patch for OpenJPEG 1.5.1 requires mapping variables across different data structures, specifically `jas_image_t -> opj_tcd_tile_t`, and transplanting the missing function `jas_image_cmpt_domains_same`. Figure 9(a) and Figure 9(b) depict the difference between the original function and adapted function used in the patch.

4 DESIGN

The goal of `PatchWeave` is to extract a patch from a given donor program and insert into a target program by computing the patch location and adapting the patch to the context of the target program. First, we will introduce the notations that we will use throughout the rest of the article, and then we present an overview of our approach and discuss in detail how each phase in our approach works. We will make use of our motivational example presented earlier in Section 2 to guide through each phase.

4.1 Symbols and Definitions

Table 2 summarizes the notations used in our article, where P_a is used to identify the donor program before the patch and P_b identifies the donor program with the developer fix. Similarly, P_c is used to identify the target program in which we aim to repair the bug, and P_d denotes the patched target program after the transplantation.

We now define a *Divergent Point,* which identifies a location in the program that will be used to compute the insertion location for the transplantation. Figure 10 shows the divergent points with respect to the source code that differs for P_a, P_b, P_c.

Table 2. Annotations Used and Their Description

Symbol	Description
P_a	the buggy version of the donor program
P_b	the fixed version of the donor program
P_c	the buggy version of the target program
P_d	the fixed version of the target program
t_F	the test case that failed in P_a but passes in P_b
d_a	a divergent point in P_a
d_c	a divergent point in P_c that is mapped to d_a
l_c	a buggy location in P_c where the program produces an observable error
π_a^F	the execution trace of t_F in P_a
π_b^F	the execution trace of t_F in P_b
π_c^F	the execution trace of t_F in P_c

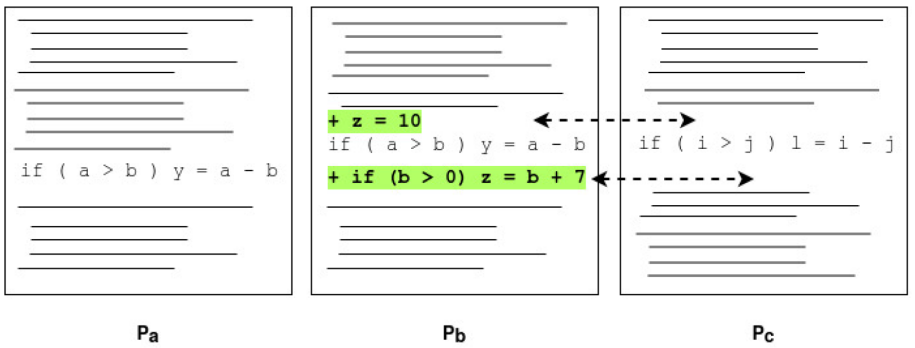


Fig. 10. Illustration of divergent points for P_a, P_b, P_c.

Definition 4.1 (Divergent Point). Given two traces π_a^F and π_b^F in P_a and P_b of a failing input t_F, the set of divergent points between π_a^F and π_b^F are the set of locations where π_b^F starts deviating from π_a^F in terms of instructions executed.

We make use of a relation between two given path conditions π_a and π_b to map a divergent point from one program to another using the following definitions.

Definition 4.2 (Partial Path Condition). Given a trace π_i of an input i in a program P, and given a point l in the trace π_i, the partial path condition of i in program P at l, denoted $ppc(P, i, l)$, is the path condition of π_i up to and including l.

For the patch transplantation problem that we investigate in this research, we extract the patch from one program and transplant to another similar program (Definition 3.2). Due to the similarity of the two programs, an inherent property is following a standard or a protocol in which the data processing order is the same. For instance, the order of reading/processing input bytes from the input is more or less the same. Making use of this inherent property, we define a relation "partial path condition dominance" to identify a mapping of program locations from P_a to P_c.

Definition 4.3 (Partial Path Condition Dominance). Given two partial path conditions α and γ, we define partial path condition dominance for γ, denoted $dom(\gamma, \alpha)$, if γ satisfies the condition where input bytes appearing in α are a subset of the input bytes appearing in γ.

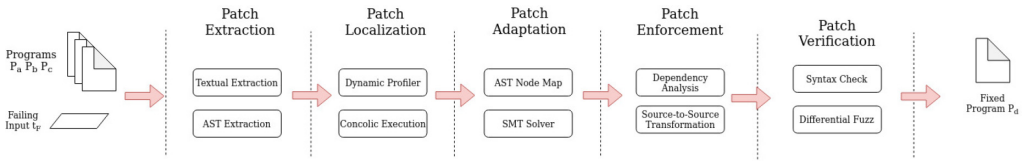


Fig. 11. The overall workflow for PatchWeave.

4.2 Overview

Figure 11 shows the overall workflow of PatchWeave. Given P_a, P_b, P_c, and t_F, we first verify that two programs are similar programs as stated in Definition 3.2. Then, PatchWeave transplants the patch from P_b to P_c in five steps: patch extraction, patch localization, patch adaptation, patch enforcement, and patch verification as described in Algorithm 1.

First, during patch extraction (Lines 1–2 in Algorithm 1), PatchWeave takes as input program P_a, program P_b and outputs the difference of the two programs in two formats: textural difference and AST structural difference. A textual difference ($text\_d$) between the two programs provides a list of diff locations in terms of source file paths and line numbers. We use information from the textual difference to identify potentially changed locations that are relevant for the patch and use this information to generate the AST in a granular level instead of generating AST for the complete program. Since the difference between the two programs may contain modifications that are irrelevant for the bug fix, we use trace-based filtering to identify the correct patch from the difference of the two programs. In the initial steps in Algorithm 1, we preprocess the diff locations using the traces generated by executing P_a, P_b, and P_c for the failing test case t_F. From the textual difference at each diff location, we identify a *code chunk* that represents the textual difference from P_a to P_b at the given diff location. Using GumTree, we obtain the AST structural difference at each diff location from the two programs P_a and P_b, which captures the transformation from P_a to P_b with respect to its abstract syntax tree. The objective of this phase is to correctly identify the patch that fixes the bug, expressed in the form of an AST transformation script. Since the transformation of the AST abstracts concrete identifiers and captures the difference at a fine-grained level, PatchWeave could adapt the patch to different contexts.

Second, during patch localization (Lines 3–5 in Algorithm 1), PatchWeave computes a patch location for the transplantation of the filtered patch. PatchWeave divides the task of patch localization into two sub-tasks: (1) finding the patch function and (2) finding the patch location within the identified function. At Line 3 in Algorithm 1, the EstimateDivergentPoint method uses concolic execution [48] to obtain the partial path conditions of P_a, P_b, and P_c for the input t_F to find a divergent point in P_c (i.e., d_c) similar to the divergent point observed in P_a (i.e., d_a) with respect to the filtered patch. Once we identified a similar divergent point in P_c, PatchWeave iterates over the trace of the target program P_c to find a patch location. We locate the patch function using the FindPatchFunction method, which uses the estimated divergent location and the variables used in the code chunk to search for a candidate patch function. First, it filters the functions invoked by P_c in $trace_c$ starting from the estimated divergent point d_c. Then, it finds the first function in the filtered list, which can be mapped to variables used in the code chunk into variables used in the function. Similarly, the FindPatchLoc method searches for a patch location within the identified patch function using live analysis of the variables mapped by the FindPatchFunction method. Finally, patch localization provides the identified insertion location for the patch in terms of a target function and the position within the target function to insert the patch, which also gives the context information (i.e., variable mapping) required to translate the patch from the namespace of P_a into the namespace of P_c.

ALGORITHM 1: PatchWeave Algorithm

input: Buggy version of Donor P_a
Fixed version of Donor P_b
Buggy version of Target P_c
Failing test case t_F
output: Fixed version of Target P_d or ϕ

$trace_a \leftarrow \text{Trace}(P_a, t_F)$
$trace_b \leftarrow \text{Trace}(P_b, t_F)$
$trace_c \leftarrow \text{Trace}(P_c, t_F)$
$text\_d \leftarrow \text{Diff}(P_a, P_b)$
$text\_d\_filtered \leftarrow \text{FilterDiff}(text\_d, trace_a, trace_b)$
$text\_d\_filtered \leftarrow text\_d\_filtered.\text{reverse}()$
while $text\_d\_filtered$ **do**
  /* Patch Extraction */
1 $d_a, code\_chunk \leftarrow text\_d\_filtered.\text{pop}()$
2 $transformation\_script \leftarrow \text{GumTree}(code\_chunk)$
  /* Patch Localization */
3 $d_c \leftarrow \text{EstimateDivergentPoint}(d_a, trace_c)$
4 $candidate\_function, var\_map \leftarrow \text{FindPatchFunction}(d_c, trace_c)$
5 $candidate\_loc \leftarrow \text{FindPatchLoc}(candidate\_function, var\_map, transformation\_script)$
  /* Patch Adaptation */
6 $translated\_script \leftarrow \text{TranslateScript}\ (transformation\_script, d_a, d_c)$
  /* Patch Enforcement */
7 $P_d \leftarrow \text{Transform}(translated\_script, var\_map, candidate\_loc)$
end
if $SyntaxCheck(P_d)$ **then**
  return P_d
end
return ϕ

In the patch adaptation phase (Line 6 in Algorithm 1), `PatchWeave` obtains a translated patch for P_c. The first step of this phase is to obtain an AST transformation that can convert P_c into P_d. At this point, we have the AST transformation script for P_a, and we have computed the target location for the insertion of the patch. The `TranslateScript` method in Line 6 uses an AST node matching algorithm to obtain a mapping between the target function identified from the patch localization phase and the AST of P_a. Using this mapping we can translate the AST transformation script into the context of the target program P_c. Second, using the variable map computed earlier in the patch localization phase, we translate the concrete identifiers (i.e., variable names and data structures) from P_b to P_c.

In patch enforcement (Line 7 in Algorithm 1), `PatchWeave` uses the mapping of concrete identifiers and the adapted AST transformation to weave the patch into the identified patch location in P_c. In this phase, dependency analysis is used to locate and evolve the patch such that all required dependencies (i.e., header files, macro definitions, etc.) for the patch are transplanted such that the patch is syntactically correct. Finally, after successful transplantation, we validate our transplanted patch. Given the patched version of P_c, we call it P_d, and we validate P_d as follows. First, we use a syntax checker (`SyntaxCheck`(P_d) in Algorithm 1), which performs static analysis on P_d with a set of syntax rules to fix any found plausible errors (i.e., unused variables, implicit conversion). Second, we recompile the patched target application and check that the build is successful without

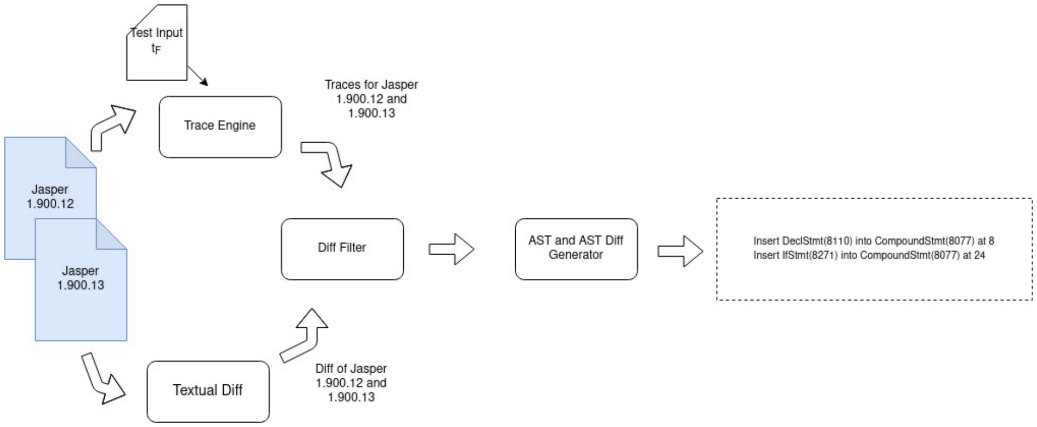


Fig. 12. Patch extraction phase of PatchWeave.

any syntactical errors. Third, we execute the patched application on the bug-triggering input to verify that the patch has successfully eliminated the vulnerability for that input. Finally, to check for the deviation of P_d's behavior from P_c's behavior, we perform differential fuzz testing over 100 generated test cases using the input t_F as the seed.

4.3 Step 1: Patch Extraction

PatchWeave uses trace-based filtering to narrow down the changes from P_a to P_b, which only consist of code modifications relevant for the bug. The dynamic profiler used by PatchWeave during trace collection is a modified version of KLEE [9]. PatchWeave executes the programs P_a and P_b in LLVM IR instructions with the vulnerability triggering input t_F until the buggy location is reached or the program crashes. The modified version of KLEE uses the debugging information in the program to translate each instruction executed to a location in a source file. Using the traces collected, combined with the textual difference obtained from the difference of the two source codes of P_a and P_b, we filter the differences not witnessed in the trace. The underlying assumption is that any modification required for the fix of the bug should be executed in the patched version of the donor P_b. In essence, what we extract as the patch is the necessary and sufficient modification required for the fix.

The filtered patch can be viewed as a code difference composed of multiple code chunks across different locations, each of which is a contiguous sequence of lines corresponding to a sequence of insertions, deletions, or both. Once we identified the necessary code chunks required for the patch, we captured the modification as a transformation of an abstract syntax tree. PatchWeave constructs an AST for the function that contains the identified chunk in both P_a and P_b. Using a tree difference algorithm GumTree [11], we generate a transformation script for the ASTs constructed earlier. This transformation script captures the modifications of line insertion, deletion, or both in the context of the AST. The set of such transformations at each identified chunk is the output of this phase.

In our motivational example in Section 2, for the two versions of our donor program JasPer, we obtained an AST script as depicted in Figure 12. Although the original developer patch includes a statement replacement in line 1238 in Figure 5(a), this statement is never executed for the input t_F in the fixed version of JasPer. Hence, we filter this statement and only include the insertion of the variable declaration and the if-condition, which checks for the overflow.

4.4 Step 2: Patch Localization

For each code chunk collected in the previous phase, the patch localization step aims to search for the location in P_c to insert the respective code chunk. Algorithm 2 explains how PatchWeave estimates a similar location in P_c. The diff location of each chunk is a divergent point in P_a since the execution of the t_F in P_a and P_b start to differ at this location. For each such divergent point d_a, we first calculate the partial path condition of the failing input t_F in P_a at d_a. This is the path condition of the trace π_a^F of input t_F in program P_a up to and including the divergent point. We estimate a similar location in P_c using partial path condition dominance on the trace of input t_F in program P_c. We map a divergent point d_a to the earliest point d_c in the execution trace of t_F in program P_c, which satisfies

$$dom(ppc(P_c, t_F, d_c), ppc(P_a, t_F, d_a)),$$

where *dom* is a dominance relation defined in Definition 4.3. Note that the two programs, although semantically similar, may have different input verification; hence, a superset of the input bytes of d_a in P_c may not exist. For instance, in our motivational example, the divergent point is at "libjasper/jpc/jpc_dec.c:1234" and the input bytes appearing in the partial path condition at this point are shown below in "input_bytes_a." Similarly, OpenJPEG 1.5.1 input bytes appearing at the full path condition are given below in "input_bytes_c."

input_bytes_a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 62, 63, 64, 65, 66, 67, 68, 69, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, **89**, **90**, 93, 94, 95, 96, 97, 98, 99, 100, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138]

input_bytes_c = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 58, 62, 63, 64, 65, 66, 67, 68, 69, 70, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138]

Note that input bytes [89, 90] are missing in "input_bytes_c," because OpenJPEG 1.5.1 does not use these bytes at any control location. Hence, we need to rule out any byte that does not appear in P_c before checking for the dominance relation. Algorithm 2 explains how PatchWeave overcomes this issue by taking the intersection of input bytes of the full path condition of P_c and the input bytes of the partial path condition at divergent point d_a in P_a. Once the filtered input bytes for the dominance relation are obtained, PatchWeave iterates through all control locations in P_c in a reverse order (i.e., starting from the last execution location) and traverses until the dominance relation does not hold, which would give us the estimated divergent location. We iterate in reverse order to be time efficient because the patch location is much closer to the crash/buggy location.

Once we identify a location in P_c, the patch can be inserted at any location between d_c and the crash location in the execution trace of the failing test in P_c. Note that there can be multiple divergent points due to multiple changes made between P_a and P_b. Once a location d_c is found for a given d_a, the adaptation of the code chunk can be applied at any location visited between d_c and the crashing point (in the execution trace of t_F in P_c). The search for this patch location consists of two steps: (1) identifying patch function and (2) finding the correct patch line within the patch function. Given t_F, our approach performs concolic execution on the three programs (P_a, P_b, and P_c) along the paths taken by t_F.

4.4.1 Patch Function. Algorithm 3 presents how PatchWeave finds the candidate patch function. The insertion point is bounded between d_c (divergent point) and l_c (crash location); a candidate function is any function invoked between d_c and l_c in the execution trace of t_F in P_c. In Algorithm 3, `list_functions` denotes all such functions executed between d_c and l_c in

ALGORITHM 2: Estimating a divergent point (`estimateDivLoc(d,p)`)

routine: getPartialPathCondition(p) takes a program location and outputs the partial path condition
extractInputBytes(p) takes a partial path condition and outputs the input bytes in the path condition
extractControlLocations(p) takes a program trace and outputs control locations
input: a divergent point in P_a (d_a), execution trace of P_c for input t_F (π_c^F)
output: a location in P_c or ϕ

$ppc_a \leftarrow getPartialPathCondition(d_a)$
$bytes\_list_a \leftarrow extractInputBytes(ppc_a)$
$list_{control} \leftarrow extractControlLocations(\pi_c^F)$
$location_{end} \leftarrow list_{control}.last()$
$ppc_{end} \leftarrow getPartialPathCondition(location_{end})$
$bytes\_list_{end} \leftarrow extractInputBytes(ppc_{end})$
$bytes\_list_a \leftarrow bytes\_list_a \cap bytes\_list_{end}$
`/* Iterate through the control locations in` P_c `to find a location which satisfies the dominance relation */`
$estimate\_loc \leftarrow \phi$
while $list_{control}$ **do**
  $location_c \leftarrow list_{control}.pop()$
  $ppc_c \leftarrow getPartialPathCondition(location_c)$
  $bytes\_list_c \leftarrow extractInputBytes(ppc_c)$
  if $bytes\_list_a \sqsubseteq bytes\_list_c$ **then**
    $estimate\_loc \leftarrow location_c$
  else
    return $estimate\_loc$
  end
end
return $estimate\_loc$

the execution trace of t_F in P_c. PatchWeave traverses through each such function and generates symbolic expressions for the variables to find a mapping between the variables in the code chunk at the divergent point d_a. Given a function f_c in P_c, our goal is to map each variable in the code chunk to variables in function f_c in P_c. Thus, for each variable in the code chunk, if a variable can be mapped to a variable in f_c of P_c with the same symbolic expression[4] (`getMap(l,f)` in Algorithm 3), we consider f_c as a candidate patch function. In our motivational example, function `j2k_read_siz` is our candidate insertion function since it is within the range of the estimated divergent point and the crashing location, and it includes variables that match the symbolic expressions of the variables in the code chunk that we want to transplant.

4.4.2 Patch Location. Computing the patch location has two variants based on the type of transformation of the patch. If the original patch is modifying an existing statement in P_a, then the objective is to find a similar statement in P_c. However, if the patch is introducing new statements, the patch line will be determined by the liveness property of the variables required for the patch; that is, the patch line should be a line at which all variables required for the patch holds a value. We identify the line where the transformation of the patch needs to be inserted based on the mapping of the variables in the patch in P_b and to variables in the patch function in P_c. Let the variables appearing in patch be Vars_{ab} and let them be mapped to variables Vars_c in P_c in the previous step of identifying the patch function. Recall that the mapping of Vars_{ab} to Vars_c was achieved by (1)

[4]These symbolic expressions are calculated by the concolic execution of t_F in programs P_b and P_c.

ALGORITHM 3: Finding patch function (`FindPatchFunction`($d_c, trace_c$))

routine: extractCode(p) takes a program location and outputs the program statements at given location
estimateDivLoc(d,p) is the routine described in Algorithm 2
extractIdentifiers(c) takes a code chunk and outputs the list of variable names
extractFunctions(d,p) takes a program location d, a trace p, outputs the functions executed up to d
getMap(l,f) takes a list of variable names and a function, outputs if variable mapping is possible
input: a divergent point in P_a (d_a), execution trace of P_c for input t_F (π_c^F)
output: a candidate function f_c in P_c or ϕ

```
code_chunk ←extractCode(d_a)
d_c ←estimateDivLoc(d_a, π_c^F)
list_identifiers ← extractIdentifiers(code_chunk)
list_functions ←extractFunctions(d_c, π_c^F)
for function f_c in list_functions do
    if getMap(list_identifers, f_c) then
        return f_c
    end
end
return null
```

concolic execution of input t_F in P_b and P_c and then (2) mapping variables based on which variables in P_c have the same symbolic expressions as the symbolic expressions of variables Vars_{ab} in program P_b. Given a patch function f_c in P_c, we filter out all control locations in f_c where the variables Vars_c are not live. Moreover, if the patch function is on the stacktrace when the crash occurs in P_c when executing t_F, we can further narrow down the patch locations to all locations in f_c executed. For any selected candidate patch location in function f_c, we check if Vars_{ab} in the patch possess the same symbolic expressions as the variables Vars_c at the patch location.

4.5 Step 3: Patch Adaptation

The patch adaptation phase processes the translation of the AST transformation script obtained from the patch extraction phase and translates the concrete identifiers to the namespace at the insertion location identified from the patch localization phase. `PatchWeave` first translates the AST transformation script into the context at the insertion location. For each code chunk identified for transplantation at the extraction phase, `PatchWeave` obtains the corresponding AST transformation. Translating the transformation script to the context of P_c involves three steps: node translation, position translation, and namespace translation. First, we present the structure of a transformation step in the transformation script as follows:

- **Delete *NodeA*:** Deletes node *NodeA* from AST_a
- **Insert *NewNode* into *NodeB* at *k*:** Inserts the node *NewNode* as the $k-th$ child of node *NodeB* in AST_b
- **Move *NodeA* into *NodeB* at *k*:** Moves the node *NodeA* in AST_a to be the k-th child of node *NodeB* in AST_b
- **Update *NodeA* to *NodeB*:** Replaces the label in node *NodeA* with the label of node *NodeB*
- **Update and Move *NodeA* into *NodeB* at *k*:** First updates the label of node *NodeA* from the matching node and then moves node *NodeA* to the k-th position of node *NodeB*

For instance, in our motivational example in Figure 12, one of the transformation actions is "Insert IfStmt(8271) into CompoundStmt(8077) at 24," which is described as inserting the node of

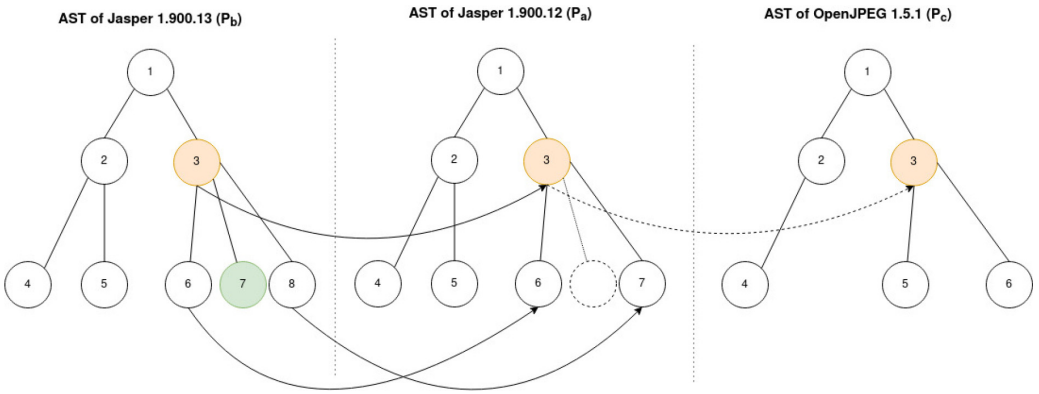


Fig. 13. AST node mapping in patch adaptation phase.

type "IfStmt" identified by the id 8271 in P_b into the node of type "CompoundStmt" identified by the id 8077 in P_b.

4.5.1 Node Translation. Given a transformation of an AST node in P_a into an AST node in P_b, we want to replace the node with a node in P_c and apply the same transformation. For this purpose, we use the tree differencing algorithm GumTree [11] implementation on LLVM AST. GumTree maps nodes in the two input ASTs based on certain heuristics [11]. It outputs a set of mapped nodes denoted as $(X, Y) = (X_1, Y_1), \ldots, (X_i, Y_i)$, where $X = X_1, \ldots, X_i$ and $Y = Y_1, \ldots, Y_i$ are the mapped nodes in the two ASTs and "i" is the number of mapped nodes. We use this technique to generate a mapping between P_a and P_c. Specifically, we generate the AST of the function, which contains the AST node of P_a and the AST of the candidate function in P_c identified in the previous phase. Once we obtain the mapping of nodes in the ASTs of P_a and P_c, `PatchWeave` uses this mapping to translate the transformation script obtained in the extraction phase.

Figure 13 shows the AST node translation using our motivational example. The arrows indicate a mapping identified by GumTree from one AST to another. To translate the transformation operation into P_c, we first translate the operation into the context of P_a and then translate back from P_a into P_c. In Figure 13, nodes 6 and 8 in the AST of JasPer 1.900.13 are mapped to nodes 6 and 7 in the AST of JasPer 1.900.12 respectively. Similarly, nodes 6 and 7 in the AST of JasPer 1.900.12 are mapped to nodes 5 and 6 in the AST of OpenJPEG 1.5.1. This gives us the translation of nodes 6 and 8 from AST of JasPer 1.900.13 into nodes 5 and 6 in OpenJPEG 1.5.1 respectively.

Let us consider in our motivational example where we transplant an "if" statement from P_b into P_c. Figure 13 shows the translation of the AST nodes. The diagram depicts the node translation for the transformation operation of "Insert IfStmt(8271) into CompoundStmt(8077) at 24" discussed earlier. The green node in Figure 13 represents the "IfStmt" node that needs to be inserted, while the yellow nodes represent the "CompoundStmt" nodes in all three programs. First, we map the "CompoundStmt(8077)" node from P_b into the "CompoundStmt(8077)" node in P_a, and then "CompoundStmt(8077)" node from P_a into the "CompoundStmt(3886)" node in P_c as shown in Figure 13. At this point of the adaptation, the transformation operation is "Insert IfStmt(8271) into CompoundStmt(3886) at X." Next, we adjust the position of the transformation related to the translated AST node "CompoundStmt(3886)" in P_c.

4.5.2 Position Translation. Given a translation of nodes for the transformation action, we convert the position relative to the translated AST node. For example, some transformations such as INSERT, MOVE require a position attribute that describes a position relative to the target node in

which the transformation would take place. PatchWeave uses the patch location identified in the previous phase to identify the target AST position by translating the patch location into a relative AST node position.

4.5.3 Namespace Translation. To translate the namespace of patch from P_b into P_c, the first step is to generate a mapping between the variables in the patch and P_c. For each variable in the patch, PatchWeave finds the corresponding variable in P_c despite the difference in data structures. As a single variable could have multiple instances due to different invocations, PatchWeave keeps track of all such instances to identify any instance of a variable in P_c that matches with a variable in the patch. For each such instance, PatchWeave records the bit value of KLEE [9], the symbolic expression, and the variable name. PatchWeave is relying on the data-type agnostic representation of KLEE to translate values across different data structures. (i.e., same numeric input represented in a signed integer value and unsigned integer differ). For each variable v in the patch, PatchWeave lists variables u from P_c where any instance of the variable u is equal to the bit value of the given variable v. PatchWeave then filters the variables using equivalence of symbolic expressions of v and u, which guarantees that the two variables are equal not only in terms of the value computed but also in terms of symbolic expressions. For any given variable v in the patch, if there are several variables in P_c having the same bit value and symbolic expression as v, PatchWeave selects the variable whose name has the minimum Levenshtein distance with the name of variable v (e.g., if $array[i]$ is matched against $temp$ and $arr[j]$, PatchWeave selects $arr[j]$ due to the minimum Levenshtein distance).

4.6 Step 4: Patch Enforcement

Once the translation of the variables in the patch into the namespace of P_c is complete, we obtain the translated patch. Although the patch has been translated successfully, we still need to identify and transplant any missing definitions used in the patch, such as a function specific for P_a or any macro definition only defined in P_a. The final step of the transplantation is to analyze each AST node in P_c to identify such missing definitions and ensure that the relevant missing functions can be called from within P_c at the insertion point. For example, if the patch is using a function defined in an external library that is not used in P_c, PatchWeave would include the relevant header files automatically so that the library can be called from P_c. In the case where a custom function defined in P_a is used in the patch, PatchWeave translates the function to match the namespace of P_c as shown in our motivational example.

5 IMPLEMENTATION

We have implemented PatchWeave in Python 2.7, in combination with Clang 7.1 [2] and KLEE 1.4 [9]. Clang is used for compiling the ASTs and obtaining the LLVM IR for symbolic execution in KLEE. We extend KLEE to support concolic execution for C/C++ programs. We leverage the Z3 [3] SMT solver for equivalence checking and LibTooling to enforce the textual edits and for source code instrumentation required in symbolic analysis.

We use the Clang AST because it offers source-to-source transformation for C/C++ code and the C++ standard. For example, parenthesis expressions and compile-time constants are available in an unreduced form in the AST. This makes Clang's AST a good fit for refactoring tools such as ours. Since we use a compiled AST contrast to a static AST built from the source code, the AST is lightweight but also does not include pragmas, which are commonly used in software to produce different code for different environments. For our analysis, we do not require such level of details since we only repair the bug for the vulnerability observed in one environment.

Table 3. Experiment Subjects and Their Details

Name	Description	LOC	Version Range	Count
JasPer	Image manipulations	26k	1.900.2 – 2.0.14	44
OpenJPEG	JPEG 2000 image manipulations	200k	1.4 – 2.1.1	8
LibWebP	WebP image manipulations	67k	0.1.2 – 1.0.2	14
LibTIFF	TIFF images processing	70k	4.0.0 – 4.0.9	10
LibMing	SWF processing	66k	0.4.3 – 0.4.8	6
Libsndfile	Audio manipulation	52k	1.0.25 – 1.0.28	4
Libzip	ZIP archive processor	13k	1.0.0 – 1.5.2	14
WavPack	Lossless Wave file compressor	33k	4.40.0 – 5.1.0	12

`PatchWeave` is implemented with 6,394 lines of Python code and 4,164 lines of C/C++ code not including the modified KLEE used for symbolic analysis.

6 EXPERIMENT AND EVALUATION

First, we evaluate the efficacy of PatchWeave on eight real-world applications to transplant fixes for vulnerabilities reported in the CVE database and recurring bugs [42] discovered in our experiments, where each subject consists of (*donor program*, *target program*, *detected error*). In terms of the problem formulation shown in Figure 1, donor program corresponds to P_b, target program corresponds to P_c, and detected error corresponds to the failing input t_F. Thus, a patch from P_b is transplanted into P_c. Second, we evaluate the quality of the transplanted patch using differential fuzzing and manually examine the patches for correctness and compare them to the developer fix (if any exist). Third, we compare `PatchWeave` against state-of-the-art APR tools including F1X [31] and Prophet [26]. Next, we provide our comparison effort with feature transplantation tools and the results. Last, we compare against a syntactic approach for transplantation, namely LASE [36].

6.1 Data Set

We evaluate PatchWeave on five classes of errors, including integer overflow, division by zero, null pointer dereference, buffer overflow, and memory errors. We obtain our subjects from a public repository [1] that contains exploits and steps to reproduce vulnerabilities published in the CVE database. To evaluate the patch transplantation ability of PatchWeave, we studied the transplantation of patches on one application program processing an input format to another application program exercising similar functionality on the same input format. We select subjects based on two criteria: (1) they have exploits reported in the referenced public repository [1], and (2) they should be popular C and C++ open source programs that have been cited in literature [5, 23]. Table 3 shows the summary of each selected subject, together with a short description about the functionality of the subject, lines of code (LOC), the range of the program versions, and the number of versions we evaluated.

For our experiments, we select vulnerabilities and their corresponding exploits from reported issues in the projects listed in Table 3 using the following criteria: (1) the vulnerability should be exploitable and its report includes a proof of exploit, (2) the vulnerability should be fixed and verified by the developer, and (3) the vulnerability should be reported in the CVE database between 2016 and 2018. For each vulnerability collected, we run against each of the subjects (each version of each program listed in Table 3) to find any similar vulnerability exhibited in a different program other than the one it was reported on. Any two pairs that exhibit similar vulnerability (Definition 3.1) are considered for evaluation under two criteria: (1) if the two programs are not from the

same project, we select the latest version of the target project in the range we presented in Table 3, or (2) if both programs are from the same project, it implies a backporting pair. For backporting, we select the oldest version in the range we presented in Table 3 that exhibits the same vulnerability. We further filtered commits that did not compile as we cannot verify if the vulnerability has been fixed for the exploit. Note that the exploits we collected can trigger vulnerabilities that are common to two different programs that are not relevant for the original program it was reported with. For example, an exploit that was reported with OpenJPEG could trigger a vulnerability that is common to Jasper and LibTiff.

6.2 Experimental Setup

All experiments are conducted on a Dell PowerEdgeR530 with Intel(R) Xeon(R) CPU E5-2660 processor and 64GB RAM. We use Docker [4] containers to exploit and repair the vulnerable applications.

6.3 Evaluation

Our evaluation aims to address the following research questions:

RQ1 How effective is PatchWeave on real-world programs?
RQ2 Can we localize the correct function for transplantation using our patch localization algorithm?
RQ3 What is the quality of the transplanted code?
RQ4 How effective is our approach compared to APR tools?
RQ5 How effective is our approach compared to existing transplantation techniques?
RQ6 How effective is a semantic transplantation technique compared with syntactic transplantation techniques?

6.4 [RQ1] Efficacy of PatchWeave

We select a set of real-world applications and CVE bug reports as described in Section 6.1 and run `PatchWeave` to transplant patches from the donor program to the target program in the identified pair list. We validate the efficacy of `PatchWeave` by comparing our results to developer patches (if any exist) for the bugs as the ground truth. We extract the developer patches from the bug reports or the commits provided by the developers. Table 4 presents the results of our experiments. The "Bug ID" column specifies an identifier of the bug if the bug has been reported. If the bug has been reported in CVE, we indicate the CVE identifier; if the bug has been reported in the project bug tracker, we indicate the bug identifier. Some bugs have been fixed without being reported and hence may not have a bug ID, which is indicated by "N/A." The "Exploit ID" column indicates the CVE where we obtained the exploits/test cases (t_F). The "Donor" column specifies the program name and version of the donor, while the "Target" column shows the same information for the target program. The "Target Location" column indicates the source code location in the target program that contains the vulnerability. The "Error" column specifies the type of vulnerability. The "Patch Commit" column represents the commit id for the patch in the donor program, and "Patch Class" shows the classes of patches defined in Section 3. The "Time (min)" column shows the total time taken in minutes for PatchWeave to fix the error/vulnerability by transferring the patch, starting from patch extraction to patch verification, or ✗ to indicate the patch transplantation was unsuccessful.

The "Diff. Fuzz." column denotes the results of differential fuzz testing in the form x/y, where x is the number of test cases where P_c results in a crash and P_d gracefully exits; similarly, y is the number of test cases where P_d crashes or produces a different output than P_c. The "Patch Size"

Table 4. Summary of PatchWeave Experiment Results

ID	Bug ID	Exploit ID	Donor	Target	Target Location	Error	Patch Commit	Patch Class	Time (min)	Diff. Fuzz.	Patch Size	Function Hops	Patch Similarity
1	Bug-169	CVE-2016-8691	JasPer-1.900.3	OpenJPEG-1.5.1	int.h@87	DZ	d8c2604	III	5.0	37/0	3	1	C2
2	CVE-2016-8691	CVE-2016-8691	OpenJPEG-1.5.2	JasPer-1.900.2	*jpc_dec.c*@1194	DZ	e55d5e2	III	8.0	22/0	3	4	C2
3	N/A	CVE-2016-9387	JasPer-1.900.13	OpenJPEG-1.5.1	j2k.c@560	IO	d91198a	IV	7.5	5/0	27	1	C2
4	CVE-2016-9387	CVE-2016-9387	OpenJPEG-1.5.2	JasPer-1.900.12	*jpc_dec.c*@1234	IO	6e0162a	III	8.5	42/0	3	1	C2
5	Bug-155	CVE-2017-6850	JasPer-2.0.12	OpenJPEG-1.5.1	cio.c@146	NPD	7692d6d	III	✗	-/-	-	-	-
6	CVE-2016-6850	CVE-2017-6850	OpenJPEG-1.5.2	JasPer-1.900.30	*jas_malloc.c*@111	NPD	7720188	III	15.0	9/0	4	1	C2
7	N/A	CVE-2016-8692	JasPer-1.900.3	OpenJPEG-1.3	int.h@87	DZ	3c55b39	III	3.5	52/0	3	1	C2
8	CVE-2016-8692	CVE-2016-8692	OpenJPEG-1.4	JasPer-1.900.2	*jpc_dec.c*@1196	DZ	f4d394d	III	5.5	20/0	3	4	C2
9	N/A	CVE-2016-9387	JasPer-1.900.14	OpenJPEG-2.1.0	j2k.c@2099	UIO	ba2b9d00	III	6.0	42/0	4	1	C2
10	N/A	CVE-2016-9387	OpenJPEG-2.1.1	JasPer-1.900.13	*jpc_dec.c*@1244	UIO	58fc8645	III	18.0	46/0	13	2	C2
11	Bug-312	CVE-2016-9262	OpenJPEG-1.5.1	LibWebP@0.3.0	jpegdec.c@251	SIO	6280b5ad	III	2.5	90/0	4	1	C2
12	N/A	CVE-2016-9830	JasPer-1.900.4	LibWebP@0.2.0	cwebp.c@120	ShO	6109f6a	II	5.0	99/0	2	1	C1
13	N/A	CVE-2016-9830	LibWebP@0.3.0	JasPer-1.900.3	*mif_cod.c*@394	ShO	7a650c6a	I	11.0	98/0	1	13	C1
14	CVE-2016-9390	CVE-2016-9390	OpenJPEG-1.5.2	JasPer-1.900.13	*jpc$_m$ct.c*@151	HBO	69cd4f9	III	✗	-/-	-	-	-
15	Bug-297	CVE-2016-9390	JasPer-1.900.14	OpenJPEG-2.1.0	libopenjpeg/mct.c@84	HBO	dee11ec	IV	✗	-/-	-	-	-
16	CVE-2016-9393	CVE-2016-9393	JasPer-1.900.17	OpenJPEG-1.5.2	j2k.c@447	UIO	f7038068	III	12.0	42/0	3	1	N.A.
17	CVE-2016-8884	CVE-2016-8884	LibTiff-3.8.0	Jasper-1.900.8	*bmp_dec.c*@394	MWE	50373d7d	III	26.0	18/0	8	23	C2
18	N/A	CVE-2017-6828	Libsndfile-1.0.26	WavPack-5.1.0	common.c@992	ShO	3e91aaf7	III	✗	-/-	-	-	-
19	CVE-2016-9387	CVE-2016-9387	Jasper-1.900.13	Jasper-1.900.2	*jpc_dec.c*@1206	IO	d91198a	II	8.0	34/0	29	1	C1
20	CVE-2016-9265	CVE-2016-9265	LibMing-0.4.8	LibMing-0.4.6	listmp3.c@187	DZ	b0704f80	I	5.5	98/0	2	1	C1
21	CVE-2016-9266	CVE-2016-9266	LibMing-0.4.8	LibMing-0.4.6	listmp3.c@94	NS	2e5a98a0	I	5.5	29/0	17	1	C1
22	Bugzilla-2634	CVE-2017-14039	LibTiff-4.0.8	LibTiff-4.0.0	tiff2ps.c@2443	HBO	5ed9fea5	I	26.0	75/0	5	18	C1
23	CVE-2017-8365	CVE-2017-8365	Libsndfile-master	Libsndfile-1.0.26	src/pcm.c@670	GBO	fd0484ab	I	16.0	73/0	9	3	C1
24	CVE-2017-14107	CVE-2017-14107	Libzip-1.3.0	Libzip-1.1.2	*zip_dirent.c*@108	MAF	9b46957e	I	42.0	90/0	4	13	C1

Division by Zero Error {DZ - Divide by Zero}, *Integer Overflow* {IO - Integer Overflow, UIO - Unsigned Integer Overflow, SIO - Signed Integer Overflow}, *Memory Error* {NPD - Null Pointer Dereference, MAF - Memory Allocation Failure, MWE - Memory Write Error}, *Shift Overflow* {ShO - Shift Overflow, NS - Negative Shift}, *Buffer Overflow* {HBO - Heap Buffer Overflow, GBO - Global Buffer Overflow}.

column denotes the modified lines of code in the transferred patch (e.g., if an expression within an if-statement is modified, we consider "Patch Size" = 1). The "Function Hops" column is a measure of efficiency in finding the patch function (e.g., "Function Hops" = 1 means that we find the patch function at the first iteration of the loop in Algorithm 3). The "Patch Similarity" column denotes the patch quality of PatchWeave relative to human-written patches (defined in Section 6.6).

Overall, PatchWeave has successfully fixed the errors for all evaluated pairs via patch transplantation, except for four errors. PatchWeave fails to fix ID 5 in Table 4 because the patch for JasPer includes program-specific changes that cannot be translated to OpenJPEG as intended. Specifically, the changes involve refactoring of existing functions and modifications to internal function calls, which are irrelevant for the bug fix. Similarly, the developer patch in ID 18 contains changes for multiple bug fixes, so our approach fails to extract the specific patch for the bug. In ID 14 and 15 of Table 4, the transplantation was unsuccessful due to the higher number of iterations executed within the execution of t_F, which produced a large symbolic path condition in which checking of partial path conditions became computationally infeasible.

Artifact and Tool Release. Experiment results in the form of generated patches and developer patches are publicly available at our website.[5] Other experimental data is publicly available at Docker Hub via docker image rshariffdeen/patchweave:experiments. Meanwhile, our tool is publicly available at https://github.com/rshariffdeen/PatchWeave.

[5] https://patchweave.github.io/.

Table 5. Effectiveness of Patch Localization in PatchWeave

ID	Donor	Target	Error	Patch Class	Time (min)	In Top-3?	In Top-5?	Localized Function Hops	Non-Localized Function Hops	Filter Count	Reduction Ratio
1	JasPer-1.900.3	OpenJPEG-1.5.1	DZ	III	5.0	✓	✓	1	18	17	17/17 = 100%
2	OpenJPEG-1.5.2	JasPer-1.900.2	DZ	III	8.0	✗	✓	4	65	61	61/64 = 95%
3	JasPer-1.900.13	OpenJPEG-1.5.1	IO	IV	7.5	✓	✓	1	34	33	33/33 = 100%
4	OpenJPEG-1.5.2	JasPer-1.900.12	IO	III	8.5	✓	✓	1	82	81	81/81 = 100%
5	JasPer-2.0.12	OpenJPEG-1.5.1	NPD	III	✗	-	-	-	-	-	-
6	OpenJPEG-1.5.2	JasPer-1.900.30	NPD	III	15.0	✓	✓	1	33	32	32/32 = 100%
7	JasPer-1.900.3	OpenJPEG-1.3	DZ	III	3.5	✓	✓	1	18	17	17/17 = 100%
8	OpenJPEG-1.4	JasPer-1.900.2	DZ	III	5.5	✗	✓	4	65	61	61/64 = 95%
9	JasPer-1.900.14	OpenJPEG-2.1.0	UIO	III	6.0	✓	✓	1	57	56	56/56 = 100%
10	OpenJPEG-2.1.1	JasPer-1.900.13	UIO	III	18.0	✓	✓	2	70	68	68/69 = 98%
11	OpenJPEG-1.5.1	LibWebP@0.3.0	SIO	III	2.5	✓	✓	1	82	81	81/81 = 100%
12	JasPer-1.900.4	LibWebP@0.2.0	ShO	II	5.0	✓	✓	1	7	6	6/6 = 100%
13	LibWebP@0.3.0	JasPer-1.900.3	ShO	I	11.0	✗	✗	13	13	0	0/12 = 0%
14	OpenJPEG-1.5.2	JasPer-1.900.13	HBO	III	✗	-	-	-	-	-	-
15	JasPer-1.900.14	OpenJPEG-2.1.0	HBO	IV	✗	-	-	-	-	-	-
16	JasPer-1.900.17	OpenJPEG-1.5.2	UIO	III	12.0	✗	✗	1	34	33	33/33 = 100%
17	LibTiff-3.8.0	Jasper-1.900.8	MWE	III	26.0	✗	✗	23	46	23	23/45 = 51%
18	Libsndfile-1.0.26	WavPack-5.1.0	ShO	III	✗	-	-	-	-	-	-
19	Jasper-1.900.13	Jasper-1.900.2	IO	II	8.0	✓	✓	1	79	78	78/78 = 100%
20	LibMing-0.4.8	LibMing-0.4.6	DZ	I	5.5	✓	✓	1	2	1	1/1 = 100%
21	LibMing-0.4.8	LibMing-0.4.6	NS	I	5.5	✓	✓	1	2	1	1/1 = 100%
22	LibTiff-4.0.8	LibTiff-4.0.0	HBO	I	26.0	✗	✗	18	76	58	58/75 = 77%
23	Libsndfile-master	Libsndfile-1.0.26	GBO	I	16.0	✓	✓	3	17	14	14/16=87.5%
24	Libzip-1.3.0	Libzip-1.1.2	MAF	I	42.0	✗	✗	13	25	12	12/24 = 50%

Division by Zero Error {DZ - Divide by Zero}, *Integer Overflow* {IO - Integer Overflow, UIO - Unsigned Integer Overflow, SIO - Signed Integer Overflow}, *Memory Error* {NPD - Null Pointer Dereference, MAF - Memory Allocation Failure, MWE - Memory Write Error}, *Shift Overflow* {ShO - Shift Overflow, NS - Negative Shift}, *Buffer Overflow* {HBO - Heap Buffer Overflow, GBO - Global Buffer Overflow}.

6.5 [RQ2] Effectiveness of Patch Localization

We evaluate the effectiveness of the patch localization technique in `PatchWeave` (i.e., partial path condition dominance relations defined in Definition 4.3). Table 5 summarizes the efficacy of `PatchWeave` for patch localization. The "Donor" column specifies the program name and version of the donor, while the "Target" column shows the same information for the target program. The "Error" column specifies the type of vulnerability, and "Patch Class" shows the classes of patches defined in Section 3. The "Time (min)" column shows the total time taken in minutes for `PatchWeave` to fix the error/vulnerability by transferring the patch, starting from patch extraction to patch verification, or ✗ to indicate the patch transplantation was unsuccessful.

The "In Top-3?" column and "In Top-5?" column indicate if the patch function has been located after localization within three hops and three hops, respectively. The "Localized Function Hops" column indicates the absolute measure of efficiency in finding the patch function (same as "Function Hops" column in Table 4). Similarly, the "Non-Localized Function Hops" column indicates the number of hops required to iterate in order to find the identified patch function without patch localization. "Filter Count" represents the number of functions filtered from the search space using patch localization, and "Reduction Ratio" represents the efficiency of the patch localization in terms of the number of hops saved as a percentage of the total number of hops without patch localization. The percentage is calculated in the form x/y, where "x" is the number of functions filtered and "y" is the total number of function hops required when not using patch localization.

PatchWeave successfully locates the correct patch function for 20 test cases presented in Table 5. Among these test cases 11 are the first candidate function, while 13 hits the Top-3 and 15 hits the Top-5.

PatchWeave can correctly identify the insertion points for all patches transplanted for all evaluated (*donor, target*) pairs. In general, there could be several potential insertion points for a given patch; our algorithm has successfully identified one of these insertion points. On average, our approach requires six iterations to find the insertion points for all evaluated patches ("Function Hops" column in Table 4 and "Localized Function Hops" column in Table 5). As our approach can automatically identify all insertion points within a reasonable number of iterations, this serves as evidence for the effectiveness of our localization algorithm using the partial path condition dominance relationship.

6.6 [RQ3] Quality and Diversity of Patches

Given a developer-written patch $Patch_{dev}$ and an automatically transplanted patch $Patch_{auto}$, we measure patch quality using the following criteria:

- **(C1) Syntactically Equivalent.** $Patch_{auto}$ is "Syntactically Equivalent" if $Patch_{auto}$ and $Patch_{dev}$ are syntactically the same.
- **(C2) Semantically Equivalent.** $Patch_{auto}$ is "Semantically Equivalent" if $Patch_{auto}$ and $Patch_{dev}$ are not syntactically the same but could be refactored to produce the same semantic behavior.

Table 4 shows that our approach could successfully generate patches of comparable quality to the developer-written patches (i.e., eight generated patches are "Syntactically Equivalent" and 11 are "Semantically Equivalent"). We attribute the success of PatchWeave in generating patches that are of comparable quality to developer-written patches to the code reuse advocated by our approach in the form of patches. The "Patch Size" column in Table 4 indicates that our approach is effective in transplanting compact patches (i.e., they are all expressible within 1–29 lines of code). Meanwhile, the "Diff. Fuzz." column shows that all of the transplanted patches have been validated through differential fuzzing.

In terms of the patch types covered, the "Patch Type" column shows that PatchWeave could successfully transplant patches for all types of patches defined in Table 1. In terms of the class of errors covered, the "Error" column shows that PatchWeave could successfully transplant patches for all evaluated five classes of vulnerabilities, namely: buffer overflow, integer overflow, divide-by-zero, memory errors (including null pointer dereferences), and shift overflows.

6.7 [RQ4] Comparison with APR

Although not directly comparable, automated program repair can be used to directly repair program bugs instead of transplanting from a different program. Hence, we compare our technique with two state-of-the-art program repair techniques to show how transplantation addresses the limitations of program repair (i.e., bounded search space, overfitting problem). Comparison results are shown in Table 6. The "Time" column shows the total time taken in minutes for each tool to fix the error/vulnerability, or ✗ to indicate the repair was unsuccessful. The "Fuzz" column denotes the results of differential fuzz testing in the form x/y, where x is the number of test cases where P_c results in a crash and P_d gracefully exits; similarly, y is the number of test cases where P_d crashes or produces a different output than P_c.

F1X [31]: We evaluate our benchmark with $F1X$, one of the latest semantic-based automated program repair tool for C programs; the authors of [31] provided us access to the tool at our email request. We choose to evaluate on $F1X$ because it represents a state-of-the-art program repair tool

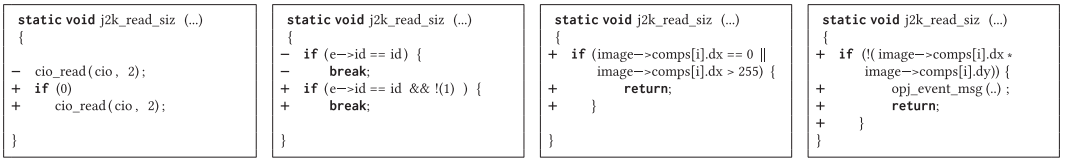
Table 6. Comparison with Program Repair Techniques

					PatchWeave		F1X		Prophet	
ID	Donor	Target	Error	Patch Class	Time	Fuzz	Time	Fuzz	Time	Fuzz
1	JasPer-1.900.3	OpenJPEG-1.5.1	DZ	Class III	5.0	37/0	0.16	20/13	10	69/2
2	OpenJPEG-1.5.2	JasPer-1.900.2	DZ	Class III	8.0	22/0	1.5	24/0	3.5	23/0
3	JasPer-1.900.13	OpenJPEG-1.5.1	IO	Class IV	7.5	5/0	0.33	52/0	2	38/7
4	OpenJPEG-1.5.2	JasPer-1.900.12	IO	Class III	8.5	42/0	6	14/27	2	0/41
5	JasPer-2.0.12	OpenJPEG-1.5.1	NPD	Class III	✗	-/-	0.16	11/3	✗	-/-
6	OpenJPEG-1.5.2	JasPer-1.900.30	NPD	Class III	15.0	9/0	0.5	8/0	3	0/9
7	JasPer-1.900.3	OpenJPEG-1.3	DZ	Class III	3.5	52/0	0.25	30/29	1	1/69
8	OpenJPEG-1.4	JasPer-1.900.2	DZ	Class III	5.5	20/0	1.5	33/0	3.5	20/0
9	JasPer-1.900.14	OpenJPEG-2.1.0	UIO	Class III	6.0	42/0	0.67	37/49	✗	-/-
10	OpenJPEG-2.1.1	JasPer-1.900.13	UIO	Class III	18.0	46/0	6	35/7	✗	-/-
11	OpenJPEG-1.5.1	LibWebP@0.3.0	SIO	Class III	2.5	90/0	✗	-/-	3	41/0
12	JasPer-1.900.4	LibWebP@0.2.0	ShO	Class II	5.0	99/0	0.25	38/0	✗	-/-
13	LibWebP@0.3.0	JasPer-1.900.3	ShO	Class I	11.0	98/0	0.5	93/4	✗	-/-
14	OpenJPEG-1.5.2	JasPer-1.900.13	HBO	Class III	✗	-/-	✗	-/-	✗	-/-
15	JasPer-1.900.14	OpenJPEG-2.1.0	HBO	Class IV	✗	-/-	✗	-/-	✗	-/-
16	JasPer-1.900.17	OpenJPEG-1.5.2	UIO	Class III	12.0	42/0	1	44/9	1	0/69
17	LibTiff-3.8.0	Jasper-1.900.8	MWE	Class III	26.0	18/0	0.33	15/0	✗	-/-
18	Libsndfile-1.0.26	WavPack-5.1.0	ShO	Class III	✗	-/-	✗	-/-	✗	-/-
19	Jasper-1.900.13	Jasper-1.900.2	IO	Class II	8.0	34/0	7.5	13/26	2	0/42
20	LibMing-0.4.8	LibMing-0.4.6	DZ	Class I	5.5	98/0	0.33	100/0	2	0 /42
21	LibMing-0.4.8	LibMing-0.4.6	NS	Class I	5.5	29/0	0.33	100/0	6	94/6
22	LibTiff-4.0.8	LibTiff-4.0.0	HBO	Class I	26.0	75/0	0.33	0/100	✗	-/-
23	Libsndfile-master	Libsndfile-1.0.26	GBO	Class I	16.0	73/0	1	66/2	✗	-/-
24	Libzip-1.3.0	Libzip-1.1.2	MAF	Class I	42.0	90/0	✗	-/-	✗	-/-
	Total			24	20	20	19	8	12	3

Division by Zero Error {DZ - Divide by Zero}, *Integer Overflow* {IO - Integer Overflow, UIO - Unsigned Integer Overflow, SIO - Signed Integer Overflow}, *Memory Error* {NPD - Null Pointer Dereference, MAF - Memory Allocation Failure, MWE - Memory Write Error}, *Shift Overflow* {ShO - Shift Overflow, NS - Negative Shift}, *Buffer Overflow* {HBO - Heap Buffer Overflow, GBO - Global Buffer Overflow}.

that has been shown in prior work to be more efficient and effective than several search-based and semantic program repair tools. *F1X* uses test equivalence relations to partition patch candidates, which leads to significant improvement of the patch generation time without sacrificing patch quality. However, *F1X*, like most existing repair tools, may suffer from the overfitting problem, where a generated patch may be plausible (passing all given tests) but overfitting (fails for tests outside the given tests). For each vulnerability (each row of Table 4), we created a test suite T_m that includes the failing test case and a passing (noncrashing) test case. Then, we give T_m and the fix location to *F1X* for patch generation. In summary, *F1X* was able to fix 19 bugs out of 24, but only 8 out of the 19 fixes are not overfitting (as shown in rows with x/0 in the "Fuzz" column in Table 6). Effectively, only eight bugs have been successfully repaired by *F1X*.

Prophet [26]: We also evaluate our benchmark with Prophet, one of the popular search-based automated repair tools. We choose to evaluate on Prophet because it represents a state-of-the-art program repair tool that has been shown in [26] to be more efficient and effective than other search-based tools. Prophet uses a probabilistic, application-independent model generated from a collection of human-written patches to find the correct code. We run Prophet using the de-



(a) F1X Patch (b) Prophet Patch (c) Transplanted Patch (d) Developer Patch

Fig. 14. Comparison of transplantation versus APR patch for bug 1.

fault configuration (with the pretrained model and enabling the `condition-ext` option and the `replace-ext` option). For each vulnerability (each row of Table 4), we created a test suite T_m, which includes the failing test case and a passing (noncrashing) test case. Then, we give T_m and fix location to Prophet to repair the bug. In summary, Prophet was able to repair 12 bugs out of 24; however, only 3 of the 12 fixes are not overfitting. Effectively, only three bugs have been successfully repaired by Prophet.

Compared to Prophet, F1X was able to generate a higher number of correct patches for our benchmark. Prophet is a search-based technique that modifies the code and relies on the test suite for correctness of the code, whereas F1X generates a patch using the constraints generated by the test suite. Hence, the semantic-based repair approach is more effective than the search-based repair approach in our experiments.

Although *F1X* and Prophet were able to fix most of the bugs listed in our benchmark, the quality of the patches is low. This is indicated in Table 6 columns "*F1X*" and "Prophet" with the patches produced by both failing on a large number of differential fuzzing test cases, where the output generated by the program before the fix and after the fix differs for noncrashing instances. Figure 14 shows the patch comparison for bug ID 1 in Table 4. F1X generates an overfitting patch for the divide-by-zero bug (i.e., bug ID 1 in Table 4), which avoids the execution of an API call by inserting a condition that is always false. The quality of the patch is determined by the differential fuzz testing of the patch, which shows failure for 13 out of 100 test cases generated using fuzzing, as indicated in Table 6. Similarly, Prophet generates a patch that excludes the execution of a statement by inserting a false logical connective. The quality of the patch is revealed to be overfitting by the differential fuzz testing of the patch, which shows failure for two test cases on average for 100 generated test cases. However, the transplanted patch that is adapted from a developer patch from the JasPer program does not fail on any of the fuzz input generated for differential analysis. Moreover, manual inspection of the patch, as shown in Figure 14(d), is correct (semantically equivalent to human patches) for the divide-by-zero bug, which checks if the denominator is zero. Furthermore, our evaluation also emphasizes the following: (1) quality of the patch in automated repair tools depends on the quality of the test suite, and (2) it is difficult for automated repair tools to produce multiline fixes. Compared to human-written patches, current methods produce lower-quality patches [13]; hence, augmenting transplantation techniques can improve the quality of the patches.

6.8 [RQ5] Comparison with Transplantation Tools

We compare our tool with existing transplantation tools to investigate the effectiveness of our approach. There are several transplantation tools proposed by prior work. CodePhage [51] is the most relevant tool for comparison as it transfers security fixes, compared to μSCALPEL [8] and CodeCarbonCopy [50], which transfer functionality. However, both CodePhage and CodeCarbonCopy are not publicly available for evaluation. We requested the authors of CodePhage [51] to provide access to the tool for comparison purposes; however, due to unavoidable circumstances, the primary developer was not available, and hence we were not able to obtain the tool. Since μSCALPEL

Table 7. Comparison with Transplantation Techniques

ID	Donor	Target	Error	Patch Class	PatchWeave	μSCALPEL
1	JasPer-1.900.3	OpenJPEG-1.5.1	DZ	Class III	✓	N/A
2	OpenJPEG-1.5.2	JasPer-1.900.2	DZ	Class III	✓	N/A
3	JasPer-1.900.13	OpenJPEG-1.5.1	IO	Class IV	✓	Seg Fault
4	OpenJPEG-1.5.2	JasPer-1.900.12	IO	Class III	✓	N/A
5	JasPer-2.0.12	OpenJPEG-1.5.1	NPD	Class III	✗	N/A
6	OpenJPEG-1.5.2	JasPer-1.900.30	NPD	Class III	✓	N/A
7	JasPer-1.900.3	OpenJPEG-1.3	DZ	Class III	✓	N/A
8	OpenJPEG-1.4	JasPer-1.900.2	DZ	Class III	✓	N/A
9	JasPer-1.900.14	OpenJPEG-2.1.0	UIO	Class III	✓	N/A
10	OpenJPEG-2.1.1	JasPer-1.900.13	UIO	Class III	✓	N/A
11	OpenJPEG-1.5.1	LibWebP@0.3.0	SIO	Class III	✓	N/A
12	JasPer-1.900.4	LibWebP@0.2.0	ShO	Class II	✓	Seg Fault
13	LibWebP@0.3.0	JasPer-1.900.3	ShO	Class I	✓	N/A
14	OpenJPEG-1.5.2	JasPer-1.900.13	HBO	Class III	✗	N/A
15	JasPer-1.900.14	OpenJPEG-2.1.0	HBO	Class IV	✗	Seg Fault
16	JasPer-1.900.17	OpenJPEG-1.5.2	UIO	Class III	✓	N/A
17	LibTiff-3.8.0	Jasper-1.900.8	MWE	Class III	✓	N/A
18	Libsndfile-1.0.26	WavPack-5.1.0	ShO	Class III	✗	N/A
19	Jasper-1.900.13	Jasper-1.900.2	IO	Class II	✓	Seg Fault
20	LibMing-0.4.8	LibMing-0.4.6	DZ	Class I	✓	N/A
21	LibMing-0.4.8	LibMing-0.4.6	NS	Class I	✓	N/A
22	LibTiff-4.0.8	LibTiff-4.0.0	HBO	Class I	✓	N/A
23	Libsndfile-master	Libsndfile-1.0.26	GBO	Class I	✓	N/A
24	Libzip-1.3.0	Libzip-1.1.2	MAF	Class I	✓	N/A
	Total			24	20	0

Division by Zero Error {DZ - Divide by Zero}, *Integer Overflow* {IO - Integer Overflow, UIO - Unsigned Integer Overflow, SIO - Signed Integer Overflow}, *Memory Error* {NPD - Null Pointer Dereference, MAF - Memory Allocation Failure, MWE - Memory Write Error}, *Shift Overflow* {ShO - Shift Overflow, NS - Negative Shift}, *Buffer Overflow* {HBO - Heap Buffer Overflow, GBO - Global Buffer Overflow}.

is the only publicly available tool for evaluation in C programs, we evaluate our approach against μSCALPEL [8]. Table 7 gives an overview of the comparison.

We compare PatchWeave with μSCALPEL [8] for four errors. Since μSCALPEL is designed for transplantation of a feature, we only consider PatchWeave with μSCALPEL for patches that involve transplantation of a new function. Specifically, we manually specify the insertion point for the patch and evaluate the effectiveness of μSCALPEL for four errors. Note that μSCALPEL has an advantage over PatchWeave under this setup because it does not need to search for the entry points and the insertion points. For each of these errors, as μSCALPEL is based on genetic programming (GP), we rerun μSCALPEL 10 times with different seeds for 30 minutes for each run due to the stochastic nature of GP. Table 7 shows that all of the runs for μSCALPEL could not complete as they resulted in segmentation fault. The reason is because the implementation of μSCALPEL was unable to parse the function in the donor program, specifically "JasPer," which is the donor program for the four errors we tried for transplantation. μSCALPEL fails to extract the function from the donor program and hence is unable to successfully repair the bug. We have reported this issue to the developers of μSCALPEL [8] and the developers have acknowledged the issue.

Similar inefficiency of μSCALPEL was confirmed with the results of another prior experiment of μSCALPEL independently conducted by other researchers [27].

6.9 [RQ6] Syntactic versus Semantic Patch Transplantation

While PatchWeave performs semantic patch transplantation based on concolic execution, the existing approach (i.e., LASE [36]) infers syntactic edit scripts from examples (in our problem formulation in Figure 1, P_b serves as one such example) and uses the inferred scripts to find edit locations, customizes the script to each location, and applies the customized script. Since LASE targets Java programs and there are no other syntactic patch transplantation tools available for C programs, we implemented a prototype inspired by the LASE [36] technique for comparison purposes. Our comparison tool uses the same technique of clone detection and AST transformation to locate the insertion point and transplant the patch. This allows us to perform a comparison of syntactic versus semantic approaches for patch transplantation. Our prototype implementation uses clone detection to identify a similar function in P_c with the help of Deckard [18] syntactic distance calculation. It uses the GumTree [11] algorithm to formulate an AST transformation script that can be used to transplant the patch from P_b to P_c. Table 8 shows the overall results of the comparison. The "Time" column shows the total time taken in minutes for each tool to fix the error/vulnerability, or ✗ to indicate the repair was unsuccessful. The "Fuzz" column denotes the results of differential fuzz testing in the form of x/y, where x is the number of test cases where P_c results in a crash and P_d gracefully exits, whereas y is the number of test case where P_d crashes or produces a different output than P_c. The "Function" column indicates if the syntactic approach was able to correctly identify the insertion function for the transplantation, and the "Var Map" column represents if the AST node matching for variable mapping based on the GumTree [11] algorithm was able to correctly map variables used for the transplantation.

The syntactic approach was able to successfully repair almost all bugs in Class I and Class II (six out of eight bugs) because these bugs do not require variable translations. The donor fix included changes that are not relevant for the bug in ID 13; hence, the syntactic approach failed to filter the relevant fix. These two classes represent syntactically very similar programs such as backporting versions or forked projects, and hence AST node matching could find the correct insertion point once the inserting function is located using clone detection. One interesting observation in our experiment is that for Class I and Class II, the syntactic method performs better compared to the semantic method. This is because the semantic method requires expensive symbolic execution to calculate the insertion point, while the syntactic method only requires clone detection, which is relatively lightweight. The syntactic method was not able to transplant any of the fixes that require a translation of variables, although in few cases it was able to identify the correct insertion function. This is due to the failure of the syntactic approach to map variables across different data structures. In these cases, our approach PatchWeave is more effective.

7 DISCUSSION

Variable Mapping Accuracy. For all our experimental evaluations, the variable mapping is successful and we did not evaluate the mapping separately, because our technique relies on the symbolic relationship, which defines the equivalence in terms of the symbolic expressions of the variables. The symbolic expressions are generated from the same input provided to both programs. There are few limitations in mapping variables using symbolic expressions for memory pointers and function return values that are not stored as a variable.

Requirement on the Number of Test Cases. Our approach requires only one test case for patch transplantation. Many vulnerability reports include a test case that is mostly

Table 8. Comparison with Syntactic Patch Transplantation

ID	Donor	Target	Error	Patch Class	Semantic Method Time	Semantic Method Fuzz	Syntactic Method Function	Syntactic Method Var Map	Syntactic Method Time	Syntactic Method Fuzz
1	JasPer-1.900.3	OpenJPEG-1.5.1	DZ	Class III	5.0	37/0	✓	✗	✗	-/-
2	OpenJPEG-1.5.2	JasPer-1.900.2	DZ	Class III	8.0	22/0	✗	✗	✗	-/-
3	JasPer-1.900.13	OpenJPEG-1.5.1	IO	Class IV	7.5	5/0	✓	✗	✗	-/-
4	OpenJPEG-1.5.2	JasPer-1.900.12	IO	Class III	8.5	42/0	✗	✗	✗	-/-
5	JasPer-2.0.12	OpenJPEG-1.5.1	NPD	Class III	✗	-/-	✗	✗	✗	-/-
6	OpenJPEG-1.5.2	JasPer-1.900.30	NPD	Class III	15.0	9/0	✗	✗	✗	-/-
7	JasPer-1.900.3	OpenJPEG-1.3	DZ	Class III	3.5	52/0	✗	✗	✗	-/-
8	OpenJPEG-1.4	JasPer-1.900.2	DZ	Class III	5.5	20/0	✓	✗	✗	-/-
9	JasPer-1.900.14	OpenJPEG-2.1.0	UIO	Class III	6.0	42/0	✗	✗	✗	-/-
10	OpenJPEG-2.1.1	JasPer-1.900.13	UIO	Class III	18.0	46/0	✗	✗	✗	-/-
11	OpenJPEG-1.5.1	LibWebP@0.3.0	SIO	Class III	2.5	90/0	✗	✗	✗	-/-
12	JasPer-1.900.4	LibWebP@0.2.0	ShO	Class II	5.0	99/0	✓	✓	4.0	99/0
13	LibWebP@0.3.0	JasPer-1.900.3	ShO	Class I	11.0	98/0	✓	✓	✗	-/-
14	OpenJPEG-1.5.2	JasPer-1.900.13	HBO	Class III	✗	-/-	✗	✗	✗	-/-
15	JasPer-1.900.14	OpenJPEG-2.1.0	HBO	Class IV	✗	-/-	✗	✗	✗	-/-
16	JasPer-1.900.17	OpenJPEG-1.5.2	UIO	Class III	12.0	42/0	✓	✗	✗	-/-
17	LibTiff-3.8.0	Jasper-1.900.8	MWE	Class III	26.0	18/0	✗	✗	✗	-/-
18	Libsndfile-1.0.26	WavPack-5.1.0	ShO	Class III	✗	-/-	✗	✗	✗	-/-
19	Jasper-1.900.13	Jasper-1.900.2	IO	Class II	8.0	34/0	✓	✓	5.5	40/0
20	LibMing-0.4.8	LibMing-0.4.6	DZ	Class I	5.5	98/0	✓	✓	10.5	100/0
21	LibMing-0.4.8	LibMing-0.4.6	NS	Class I	5.5	29/0	✓	✓	6.0	32/1
22	LibTiff-4.0.8	LibTiff-4.0.0	HBO	Class I	26.0	75/0	✓	✓	8.0	76/1
23	Libsndfile-master	Libsndfile-1.0.26	GBO	Class I	16.0	73/0	✗	✗	✗	-/-
24	Libzip-1.3.0	Libzip-1.1.2	MAF	Class I	42.0	90/0	✓	✓	6.0	95/0
	Total			24	20	20	11	7	6	4

Division by Zero Error {DZ - Divide by Zero}, *Integer Overflow* {IO - Integer Overflow, UIO - Unsigned Integer Overflow, SIO - Signed Integer Overflow}, *Memory Error* {NPD - Null Pointer Dereference, MAF - Memory Allocation Failure, MWE - Memory Write Error}, *Shift Overflow* {ShO - Shift Overflow, NS - Negative Shift}, *Buffer Overflow* {HBO - Heap Buffer Overflow, GBO - Global Buffer Overflow}.

available for the developers. Furthermore, recurring vulnerabilities can find test cases from other similar projects (i.e., donor program or upstream repo of the fork).

Patch Size. As shown in the "Patch Size" column in Table 4, patch size does not have an effect on the transplantation since the major task is generating the translation between the variables. In fact, the difficulty lies when the patch is inside a loop, which is difficult for symbolic execution to reach (which is one of the difficulties in symbolic execution); an example is Test Case 15 in Table 4.

Syntactic Prototype. One significant difference in our prototype and the LASE [36] technique is that our prototype does not infer a bug fix pattern based on generalizations of multiple examples. In the patch transplantation problem, only one patch is provided to transplant from the donor to the target program. Hence, the implemented prototype does not infer a bug fix pattern from multiple examples as defined in LASE [36]. Nevertheless, our prototype can still be used to compare the effectiveness of clone detection versus symbolic path condition for identifying the insertion point, as well as the AST-based

node matching GumTree algorithm versus symbolic equivalence for variable mapping as discussed in Section 6.9.

8 RELATED WORK

Software Transplantation. Automated software transplantation has been applied for solving several software maintenance tasks, including feature transplantation [8, 28], transplanting validation checks [51], and transplanting shellcode for remote exploits [7]. μSCALPEL [8] and CodeCarbonCopy [50] transplant new functionality from a donor application P_b into a recipient application P_c. Both of these approaches require the developer to specify the insertion point and identify the functionality to be extracted for transplantation. μSCALPEL [8] uses genetic programming with program slicing to transplant functionality from a donor system to a target system but requires manually specifying the entry point of code and the insertion point in the host program. Meanwhile, CodeCarbonCopy [50] requires manual specification of (1) the donor function that captures the functionality to be transplanted, (2) the insertion point in the host program, and (3) extra parameters for removing irrelevant functionality in the transferred code. Code Phage [51] eliminates errors such as integer overflow by transferring checks from P_b to P_c. All three approaches are limited in transplanting new code (e.g., function and check condition) and could not handle the case where the donor and the recipient are code edits in the form of a patch. Meanwhile, PatchWeave provides a fully automated approach for patch transplantation without the manual effort required to specify the donor function and the insertion point.

Several techniques transplant Java code edits [27, 66]. Although the donor P_a and the recipient P_c in the patch transplantation problem could be regarded as code clones, GRAFTER [66] uses code clones for test reuse and differential testing, whereas PatchWeave uses the similarity between P_a and P_c to automatically identify insertion points. Similar to our work that encourages code reuse, program splicing [27] reuses existing code from the web, whereas we reuse security patches. Overall, [66] and [27] support Java programs instead of C programs, but prior study shows that there are less exact clones in C functions than Java methods [47]. The lack of exact clones in C functions implies that the clones' adaptation in C programs could be more challenging than in Java programs.

Patch Transformation Inference. To improve the quality of generated patches, several approaches infer code edits from Java methods [17, 22, 25, 35, 36] and student-written examples from C# programs [46]. One of the main differences in transplanting code edits for Java and C, apart from the inherent structure enforced in Java (classes, methods, etc.), is locating the insertion point. Java-based transformation tools rely on the existence of code clones (a syntactically similar method) to locate a suitable insertion point. Since such clones are not common for C functions [47], we cannot rely on clone detection methods to locate an insertion point. For other work on inferring code edits, prior approaches either infer specifications from reference implementation [32] (with code edits being synthesized from specifications), or employ predefined repair operators derived from previous program versions and/or patch patterns [19, 55]. Our approach differs from all of these approaches in that (1) it infers C code edits from existing patches of vulnerabilities, (2) the insertion point of the code edit is found automatically, and (3) the approach is not restricted to any specific pattern of code edits. Long et al. use multiple examples to infer a transformation to fix a specific type of bug [25]. However, in reality, many examples to fix a particular vulnerability may not exist, thus preventing such inference techniques.

Automated Program Repair. Several approaches have been proposed to automatically generate patches [24, 26, 30, 33, 34, 37, 41, 54, 56, 59, 60, 62–64]. A review of the area appears in [15]. Though some of the vulnerabilities evaluated could be fixed automatically by existing repair approaches,

our approach does not generate patches from a set of predefined templates or mutations. Instead, our approach extracts patches from the donor program automatically and transplants the extracted patches into the host program. Prior research on automated repair [53] have identified the overfitting problem where the generated patches can overfit a test suite. We alleviate overfitting by transplanting a fix obtained from a similar program.

Backporting. Backporting new features to older software versions is important in systems such as Linux kernel [58]. Several approaches were proposed to support backporting [14, 39, 45, 57]. Although PatchWeave could be applied for backporting patches to older versions, our approach is not limited to the case where the donor and the host are different versions of the same program.

Patch Differencing and Patch Analysis. Various program differencing algorithms have been introduced for computing source code modifications between two program versions [6, 11, 12, 16, 20, 21, 44, 65]. Our approach relies on GumTree [11] for (1) extracting edit actions between the donor and the host program and (2) matching the AST between P_b and P_c. While other program differencing algorithms may be used to enhance the effectiveness of the patch adaptation step, we studied a different problem than existing works.

9 THREATS TO VALIDITY

There are several threats to validity of our approach related to the external tools we use or the datasets we use in our experiments. We seek to mitigate such threats by using actively maintained external tools such as KLEE and conducting our experiments on a wide variety of subjects, including those studied by previous work on software transplantation [51].

We would like to highlight two issues related to the availability of tests. First, we assume the presence of at least one test (i.e., the exploit). If no test is available, as may be the case in certain domains (e.g., backporting of Linux patches [40]), our technique cannot be applied as it is. Furthermore, relying only on one test case could result in a patch that is overfitted, which is likely to disrupt correct behavior of the application. We address this concern with the use of differential fuzz testing where we generate mutated inputs from the single failing test case. Then we employ differential behavior analysis to detect if the transplanted patch introduces disruption to correct behavior by comparing the output of P_c and P_d for each generated test case. Second, due to practical considerations, we have not assumed the presence of a large number of tests. Moreover, requiring more than one test case is often an impractical requirement since more often vulnerability reports contain a single exploit; hence, our focus is to provide an automation to help developers fix the issue given a single test case is provided. If more tests are available, we could extend our technique to efficiently navigate all candidate patches and select one (similar to the selection of patches in prior work in program repair, e.g., *F1X*[31]). However, such a patch space exploration is *not* part of our current PatchWeave implementation.

10 CONCLUSION

We formulate the *patch transplantation* problem in this article. We also propose a fully automated solution to patch transplantation based on concolic execution. The patch transplantation problem caters to a *real-life need* in the practice of software security: even when an important vulnerability is detected and a patch is constructed, it is nontrivial to adapt the patch for other similar implementations exhibiting the same vulnerability. Indeed, associating a CVE to a newly found vulnerability and publishing the patch may make these other similar un-patched implementations more vulnerable since attackers will have more knowledge on how to exploit the vulnerability. The patch transplantation problem studied in this article provides a solution to reduce or remove such exposure to vulnerabilities.

REFERENCES

[1] Agostino Sarubbo. 2019. Agostino's blog. Retrieved February 24, 2019, from https://blogs.gentoo.org/ago.

[2] LLVM Developer Group. 2019. The Clang Project. Retrieved March 11, 2019, from https://clang.llvm.org.

[3] Microsoft Research. 2019. The Z3 Theorem Prover. Retrieved March 11, 2019, from https://github.com/Z3Prover/z3.

[4] Docker Inc. 2020. The Docker Project. Retrieved March 5, 2019, from https://www.docker.com/.

[5] Michael D. Adams and Faouzi Kossentini. 2000. JasPer: A software-based JPEG-2000 codec implementation. In *Proceedings 2000 International Conference on Image Processing (Cat. No. 00CH37101)*, Vol. 2. IEEE, 53–56.

[6] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2007. JDiff: A differencing technique and tool for object-oriented programs. In *29th IEEE/ACM International Conference on Automated Software Engineering (ASE'14)*. 3–36.

[7] Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. 2017. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *2017 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 824–839.

[8] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'18)*. ACM, 257–269.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756.

[10] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. 2014. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC'14)*. ACM, 475–488.

[11] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering (ASE'14), Vasteras, Sweden - September 15-19, 2014*. 313–324.

[12] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. 2007. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743.

[13] Zachary P. Fry, Bryan Landau, and Westley Weimer. 2012. A human study of patch maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA'12)*. ACM, New York, NY, 177–187. DOI: https://doi.org/10.1145/2338965.2336775

[14] V. U. Gomez, S. Ducasse, and T. D'Hondt. 2010. Visually supporting source code changes integration: The torch dashboard. In *2010 17th Working Conference on Reverse Engineering*. 55–64. DOI: https://doi.org/10.1109/WCRE.2010.15

[15] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Communications of the ACM* 62, 12 (2019), 56–65.

[16] Kaifeng Huang, Bihuan Chen, Xin Peng, Daihong Zhou, Ying Wang, Yang Liu, and Wenyun Zhao. 2018. ClDiff: Generating concise linked code differences. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. ACM, 679–690.

[17] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18)*. ACM, 298–309.

[18] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. IEEE Computer Society, 96–105. DOI: https://doi.org/10.1109/ICSE.2007.30

[19] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. IEEE Press, 802–811.

[20] Miryung Kim and David Notkin. 2009. Discovering and representing systematic code changes. In *2009 IEEE 31st International Conference on Software Engineering (ICSE'09)*. IEEE, 309–319.

[21] Janusz Laski and Wojciech Szermer. 1992. Identification of program modifications and its applications in software maintenance. In *Proceedings Conference on Software Maintenance 1992*. IEEE, 282–290.

[22] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, Vol. 1. IEEE, 213–224.

[23] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE Press, Piscataway, NJ, 3–13. http://dl.acm.org/citation.cfm?id=2337223.2337225.

[24] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan. 2012), 54–72. DOI: https://doi.org/10.1109/TSE.2011.104

[25] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. Association for Computing Machinery, New York, NY, 727–739. DOI: https://doi.org/10.1145/3106237.3106253

[26] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*. ACM, New York, NY, 298–312.

[27] Yanxin Lu, Swarat Chaudhuri, Chris Jermaine, and David Melski. 2018. Program splicing. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. ACM, 338–349.

[28] Alexandru Marginean, Earl T. Barr, Mark Harman, and Yue Jia. 2015. Automated transplantation of call graph and layout features into Kate. In *International Symposium on Search Based Software Engineering (ICSE'13)*. Springer, 262–268.

[29] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering (Empirical Softw. Engg)* 22, 4 (Aug. 2017), 1936–1964. DOI: https://doi.org/10.1007/s10664-016-9470-4

[30] Matias Martinez and Martin Monperrus. 2016. ASTOR: A program repair library for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. ACM, New York, NY, 441–444. DOI: https://doi.org/10.1145/2931037.2948705

[31] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-equivalence analysis for automatic patch generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 4 (2018).

[32] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. ACM, 129–139.

[33] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15), Volume 1*. IEEE Press, 448–458.

[34] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *IEEE/ACM 38th International Conference on Software Engineering (ICSE'16)*. IEEE, 691–701.

[35] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic editing: Generating program transformations from an example. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 329–342.

[36] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE'13)*. IEEE Press, 502–511.

[37] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE'13)*. IEEE Press, Piscataway, NJ, 772–781. http://dl.acm.org/citation.cfm?id=2486788.2486890.

[38] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. 2008. Documenting and automating collateral evolutions in Linux device drivers. In *ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys'08)*.

[39] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys'08)*. ACM, New York, NY, 247–260. DOI: https://doi.org/10.1145/1352592.1352618

[40] Nicolas Palix, Gael Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. Faults in Linux: Ten years later. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*.

[41] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. 2009. Automatically patching errors in deployed software. In *22nd ACM Symposium on Operating Systems Principles (SOSP'09)*. 87–102.

[42] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2010. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*. ACM, New York, NY, 447–456. DOI: https://doi.org/10.1145/1858996.1859089

[43] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)*. Association for Computing Machinery, New York, NY, 24–36. DOI: https://doi.org/10.1145/2771783.2771791

[44] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. 2004. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *20th IEEE International Conference on Software Maintenance (ICSM'04). Proceedings.* IEEE, 188–197.

[45] Baishakhi Ray, Miryung Kim, Suzette Person, and Neha Rungta. 2013. Detecting and characterizing semantic inconsistencies in ported code. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE, 367–377.

[46] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. IEEE Press, 404–415.

[47] C. K. Roy and J. R. Cordy. 2008. An empirical study of function clones in open source software. In *2008 15th Working Conference on Reverse Engineering*. 81–90. DOI: https://doi.org/10.1109/WCRE.2008.54

[48] Koushik Sen. 2007. Concolic testing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. Association for Computing Machinery, New York, NY, 571–572. DOI: https://doi.org/10.1145/1321631.1321746

[49] Shodan. 2016. Devices Vulnerable to Heartbleed. Retrieved September 14, 2018, from https://www.shodan.io/report/89bnfUyJ

[50] Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. 2017. CodeCarbonCopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. ACM, 95–105.

[51] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, New York, NY, 43–54. DOI: https://doi.org/10.1145/2737924.2737988

[52] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. 532–543.

[53] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. ACM, New York, NY, 532–543. DOI: https://doi.org/10.1145/2786805.2786825

[54] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing crashes in Android apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. ACM, New York, NY, 187–198. DOI: https://doi.org/10.1145/3180155.3180243

[55] Shin Hwei Tan and Abhik Roychoudhury. 2015. Relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE'15)*. IEEE Press, Piscataway, NJ, 471–482. http://dl.acm.org/citation.cfm?id=2818754.2818813

[56] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'16)*. ACM, 727–738.

[57] Ferdian Thung, Xuan-Bach D. Le, David Lo, and Julia Lawall. 2016. Recommending code changes for automatic backporting of Linux device drivers. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME'16)*. IEEE, 222–232.

[58] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying Linux bug fixing patches. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE Press, Piscataway, NJ, 386–396. http://dl.acm.org/citation.cfm?id=2337223.2337269

[59] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE'18)*. IEEE, 151–162.

[60] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, Piscataway, NJ, 356–366. DOI: https://doi.org/10.1109/ASE.2013.6693094

[61] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *IEEE/ACM International Conference on Software Engineering (ICSE'09)*.

[62] Qi Xin and Steven P. Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'17)*. ACM, New York, NY, 226–236. DOI: https://doi.org/10.1145/3092703.3092718

[63] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. IEEE Press, Piscataway, NJ, 416–426. DOI: https://doi.org/10.1109/ICSE.2017.45

[64] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. Lamelas Marcote, T. Durieux, D. Le Berre, and M. Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Transactions on Software Engineering* PP, 99 (2016), 1–1.

[65] Wuu Yang. 1991. Identifying syntactic differences between two programs. *Software: Practice and Experience* 21, 7 (1991), 739–755.

[66] Tianyi Zhang and Miryung Kim. 2017. Automated transplantation and differential testing for clones. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. IEEE Press, Piscataway, NJ, 665–676. DOI: https://doi.org/10.1109/ICSE.2017.67

Received November 2019; revised June 2020; accepted July 2020