Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications

Stelios Sidiroglou-Douskos Eric Lahtinen Fan Long Martin Rinard
MIT CSAIL
{stelios,elahtinen,fanl,rinard}@csail.mit.edu

Abstract

We present Code Phage (CP), a system for automatically transferring correct code from donor applications into recipient applications that process the same inputs to successfully eliminate errors in the recipient. Experimental results using seven donor applications to eliminate ten errors in seven recipient applications highlight the ability of CP to transfer code across applications to eliminate out of bounds access, integer overflow, and divide by zero errors. Because CP works with binary donors with no need for source code or symbolic information, it supports a wide range of use cases. To the best of our knowledge, CP is the first system to automatically transfer code across multiple applications.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Error handling and recovery; D.2.7 [*Distribution, Maintenance, and Enhancement*]: Corrections

Keywords automatic code transfer, program repair, data structure translation

1. Introduction

Horizontal gene transfer is the transfer of genetic material between cells in different organisms. Examples include plasmid transfer (which plays a major role in acquired antibiotic resistance [17]), virally-mediated gene therapy [28], and the transfer of insect toxin genes from bacteria to fungal symbionts [16]. Because of its ability to directly transfer functionality evolved and refined in one organism into another, horizontal gene transfer is recognized as a significant factor in the development of many forms of life [29].

Like biological organisms, software applications often face challenges and threats from the environment in which they operate. Despite significant software development effort, errors and security vulnerabilities still remain a important concern. Many of these errors are caused by an uncommon case that the developers of one (or more) of the applications did not anticipate. But in many cases, the developers of another application did anticipate the uncommon case and wrote correct code to handle it.

1.1 The Code Phage (CP) Code Transfer System

We present Code Phage (CP), a novel *horizontal code transfer system* that automatically eliminates errors in recipient software applications by finding correct code in donor applications, then transferring that code from the donor into the recipient. The result is a software hybrid that productively combines beneficial code from multiple applications:

- **Donor Selection:** CP starts with an application and two inputs: an input that triggers an error and a seed input that does not trigger the error. Working with a database of applications that can read these inputs, it locates a donor that processes both inputs successfully. The hypothesis is that the donor contains a check, missing in the recipient, that enables it to process the error-triggering input correctly. The goal is to transfer that check from the donor into the recipient (and eliminate the error in the recipient).
- **Candidate Check Discovery:** To identify the check that enables the donor to survive the error-triggering input, CP analyzes the executed conditional branches in the donor to find branches that take different directions for the seed and error-triggering inputs. The hypothesis is that if the check eliminates the error, the seed input will pass the check but the error-triggering input will fail the check (and therefore change the branch direction).
- **Patch Excision:** CP performs an instrumented execution of the donor on the error-triggering input to obtain a symbolic expression tree that expresses the check as a function of the input fields that determine its value. This execution translates the check from the data structures and name space of the donor into an application-independent representation suitable for insertion into another application.
- **Patch Insertion:** CP next uses an instrumented execution of the recipient on the seed input to find *candidate insertion points* at which all of the input fields in the excised check are available as recipient program expressions. At each such point, it is possible to translate the check from the application-independent representation into the data structures and name space of the recipient. This translation, in effect, inserts the excised check into the recipient.
- **Patch Validation:** CP inserts the translated check into the recipient at each candidate insertion point in turn, then attempts to validate the patch. It recompiles the application, uses regression testing to verify that the patch preserves correct behavior on the regression suite, and checks that the patch enables the patched recipient to correctly process the error-triggering input. As available, CP also reruns error detecting tools to generate additional error-triggering inputs, which it then uses to recursively eliminate any residual or newly discovered errors.

  As appropriate, CP can also exploit the semantics of specific classes of errors (such as divide by zero or integer overflow) to perform additional validation steps. For integer overflow errors, for example, CP analyzes the check, the expression that overflows,

PLDI'15, , June 13–17, 2015, Portland, OR, USA.
Copyright © 2015 ACM 978-1-4503-3468-6/15/06. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

and other existing checks in the recipient that are relevant to the error to verify that there is no input that 1) satisfies the checks to traverse the exercised path through the program to the overflow and also 2) triggers the overflow.

- **Retry:** If the validation fails, CP tries other candidate insertion points, other candidate checks, and other donors.

If the transferred check detects an input that may trigger the error, it exits the application before the error occurs. The check therefore introduces no new and potentially unpredictable behaviors — it simply narrows the set of inputs that the application decides to process. This narrowing is conceptually similar to transformations that eliminate concurrency errors by narrowing the set of possible interleavings [27, 43].

1.2 Usage Scenarios

Proprietary Donors: The CP donor analysis operates directly on stripped binaries with no need for source code or symbolic information of any kind. CP can therefore use arbitrary binaries, including closed-source proprietary binaries, as donors for other applications. A developer could, for example, reduce development and testing effort by simply omitting checks for illegal inputs, then using CP to automatically harden the application by automatically transferring in checks from more intensively engineered (including closed-source proprietary) applications.

Multilingual Code Transfer: CP supports multilingual code transfer between applications written in different programming languages. Because CP works with binary donors, the current implementation supports arbitrary (compiled) donors. The current CP implementation generates source-level patches in C. It would be straightforward to extend CP to generate patches in other languages. Given appropriate binary patching capability, it would also be straightforward to generate binary patches, including hot patches for running applications.

Multiversion Code Transfer: In addition to transferring checks between independently developed applications, we have also used CP to transfer checks between different versions of the same application. The motivation is to automatically obtain a targeted update that eliminates an error in an older version without the disruption often associated with full upgrade [22].

Divergent Functionality: Even though CP works with applications that process the same inputs, the recipient and donor do not need to implement the same functionality. Many errors occur in code that parses the input, constructs the internal data structures that hold the input, and/or reads the input into those data structures. Even when the applications have different goals and functionality, the fact that they both read the same input is often enough to enable a successful transfer.

Continuous Multiple Application Improvement: CP can work with any source of seed and error-triggering inputs. Its current integration with the DIODE automatic error-discovery system [55] points the way to future systems that combine 1) large libraries of applications, 2) a variety of automatic error discovery tools (for example, DIODE and BuzzFuzz [25]), and 3) CP along with other automatic error repair tools such as ClearView [48], staged program repair [37], and automatic code fracture and recombination [57]. Continuously running the error-discovery tools across the library of applications, then using horizontal code transfer and other program repair mechanisms to generate repairs delivers an automatic application improvement system that productively leverages the entire global software development enterprise.

Such a system holds out the promise of automatically producing robust software hybrids that incorporate the best code produced anywhere by any mechanism. Given the ability of DIODE and CP to work with stripped binary donors, it is possible to include closed-source software produced by proprietary software development efforts into this continuous application improvement system.

1.3 Scope

CP is currently designed to locate and transfer checks, including all computation required to compute the checks, between applications that process the same inputs. The goal is to change the (incorrect) semantics of the recipient so that it rejects inputs that would otherwise trigger the error. The patch validation phase, along with the rejection of unstable insertion points (Section 3.3), is designed to reduce, but not necessarily eliminate, the possibility of rejecting inputs that the recipient could have processed correctly. The excised computation can be, and in practice always is, distributed across multiple system layers and abstraction boundaries within the donor — the excised computation always includes code from multiple system libraries and procedures within the application.

In the current implementation of CP, a set of values sufficient to compute the check must be available in the recipient at one of the granularities at which they are accessed in the excised computation and in one of the same byte orders. It is straightforward to extend the implementation to reassemble values sufficient to compute the check from bits arbitrarily distributed across the address space of the recipient as long such a set of bits is accessible via the name space of the recipient.

CP is currently designed to transfer code that computes a check. But the basic CP transfer techniques are designed to dynamically track, extract, and insert any computation (or computations) that generate any value (or values) in the donor as long as CP can identify the value(s). The two critical questions are identifying the value(s) in the donor and the insertion point(s) in the recipient. CP automates this identification for checks in the donor that eliminate errors in the recipient.

1.4 Experimental Results

We evaluate CP on 10 errors in 7 recipient applications (JasPer 1.9 [11], gif2tiff 4.0.3 [9], CWebP 0.31 [1], Dillo 2.1 [2], swfplay 0.55 [7], Display 6.5.2-8 [5], and Wireshark-1.4.14 [14]). The donor applications are FEH-2.9.3 [3], mtpaint 3.4 [12], ViewNoir 1.4 [8], gnash 0.8.11 [10], OpenJpeg 1.5.2 [13], Display 6.5.2-9 [5], and Wireshark-1.8.6 [14]. CP was able to successfully generate patches that eliminated the errors, in five cases demonstrating the ability to transfer patches from multiple donors (see Section 4).

For all of the applications except Wireshark-1.4.14 (which uses Wireshark 1.8), CP successfully excises code from an independently developed alien donor and successfully implants the code into the recipient. The ability of CP to translate the check from the donor name space and data structures into the name space and data structures of the recipient is critical to the success of many transfers. Wireshark-1.4.14 demonstrates the ability of CP to deliver targeted updates that eliminate specific errors while leaving the behavior and functionality of the recipient otherwise intact.

1.5 Contributions

This paper makes the following contributions:

- **Basic Concept:** CP automatically eliminates software errors by identifying and transferring correct code from donor applications into incorrect recipient applications. In this way CP can automatically harness the combined knowledge and labor invested across multiple software systems to improve each application.
  To the best of our knowledge, CP is the first system to automatically transfer code across multiple applications.
- **Name Translation:** One of the major challenges in code transfer is translating the names of values from the name space of the donor into the name space of the recipient. CP shows how to use instrumented executions of the donor and recipient to meet this name translation challenge.

- **Data Structure Translation:** Another major code transfer challenge is translating between different data representations. CP shows how to use instrumented executions and data structure traversals to meet this challenge — it takes code that accesses values stored in the data structures of the donor and produces code that accesses values stored in the data structures of the recipient.
- **Donor Code Identification:** It presents a mechanism to identify correct code in donor applications for transfer into recipient applications. CP uses two instrumented executions of the donor to automatically identify the correct code to transfer into the recipient: one on the seed input and one on the error-triggering input (which the donor, but not the recipient, can successfully process). A comparison of the paths that these two inputs take through the donor enables CP to isolate a single check (present in the donor but missing in the recipient) that eliminates the error.
- **Insertion Point Identification:** CP automatically identifies appropriate check insertion points within the recipient at which 1) the values needed to express the transferred check computation are available as valid program expressions in the name space of the recipient and 2) the transferred check will not affect observed computations unrelated to the error.
- **Experimental Results:** We present experimental results that characterize the ability of CP to eliminate ten otherwise fatal errors in seven recipient applications by transferring correct code from seven donor applications. For all of the 10 possible donor/recipient pairs, CP was able to obtain a successful validated transfer that eliminated the error.

The remainder of the paper is structured as follows. Section 2 presents an example that illustrates how CP eliminates an error in CWebp (with FEH as the donor). Section 3 discusses the CP design and implementation. We present experimental results in Section 4, related work in Section 5, and conclude in Section 6.

2. Example

We next present an example that illustrates how CP automatically patches an integer overflow error in CWebP, the Google conversion program for the WepP image format.

Figure 1 presents (simplified) CWebP source code that contains an integer overflow error. CWebP uses the libjpeg library to read JPG images before converting them to the CWebP format. It uses the `ReadJPEG` function to parse the JPG files. There is a potential overflow at line 9, where CWebP calculates the size of the allocated image as `stride * height`, where stride is: `width * output_components * sizeof(rgb)`.

On a 32-bit machine, inputs with large width and height fields can cause the image buffer size calculation at line 9 to overflow. In this case CWebP allocates an image buffer that is smaller than required and eventually writes beyond the end of the allocated buffer.

Error Discovery: In our example, CP works with seed and error-triggering inputs identified by the DIODE integer-overflow discovery tool, which performs a directed search on the input space to discover inputs that trigger integer overflow errors at memory allocation sites [55]. In the error-triggering input in our example, the JPG `height` field is 62848 and the `width` field is 23200.

Donor Selection: CP next searches a database of applications that process JPG files to find candidate donor applications that successfully process both the seed and the error-triggering inputs. In our example, CP determines that the FEH image viewer application processes both inputs successfully.

```
1   int ReadJPEG(...) {
2      ...
3      width   = dinfo.output_width;
4      height  = dinfo.output_height;
5      stride  = dinfo.output_width *
6                  dinfo.output_components *
7                  sizeof(*rgb);
8      /* the overflow error */
9      rgb = (uint8_t*)malloc(stride * height);
10     if (rgb == NULL) {
11        goto End;
12     }
13     ...
14  }
```

Figure 1: (Simplified) CWebP Overflow Error

```
1   # define IMAGE_DIMENSIONS_OK(w, h) \
2      ( ((w) > 0) && ((h) > 0) && \
3        ((unsigned long long)(w) * \
4         (unsigned long long)(h) <= (1ULL << 29) - 1) )
5
6   char load(...) {
7      int    w, h;
8      struct jpeg_decompress_struct cinfo;
9      struct ImLib_JPEG_error_mgr jerr;
10     FILE   *f;
11     ...
12     if (...) {
13        ...
14        im->w = w = cinfo.output_width;
15        im->h = h = cinfo.output_height;
16        /* Candidate check condition */
17        if ((cinfo.rec_outbuf_height > 16) ||
18              (cinfo.output_components <= 0) ||
19              !IMAGE_DIMENSIONS_OK(w, h))
20        {
21           // Clean up and quit
22           ...
23           return 0;
24        }
25     }
26  }
```

Figure 2: (Simplified) FEH Overflow Check

Candidate Check Discovery: CP next runs an instrumented version of the FEH donor application on the two inputs. At each conditional branch that is influenced by the relevant input field values (in this case the JPG `height` and `width` fields), it records the direction taken at the branch and a symbolic expression for the value of the branch condition. The free variables in these expressions represent the values of input fields.

CP operates under the hypothesis that one of the FEH branch conditions implements a check designed to detect inputs that trigger the overflow. Under this hypothesis, the seed input and error-triggering inputs take different directions at this branch (because the error-triggering input would satisfy the check and the seed input would not). CP therefore considers the check for each branch at which the seed and error-triggering inputs take different directions to be a *candidate check*.

In our example, CP discovers a candidate check in the `imlib` library that FEH uses to load and process JPG files. Figure 2 presents (simplified) source code for this check.[1] The macro `IMAGE_DIMENSIONS_OK` (defined on lines 1-4, invoked on line 19), performs an overflow check on the computation of `output_width * output_height`. This check enables FEH to detect and correctly process the error-triggering input without overflow.

Candidate Check Excision: The FEH check is expressed in terms of the FEH data structures. The next step is to translate the check from this form into an application-independent form that expresses the check as a function of the input bytes that determine its value.

[1] Because CP operates on binaries, information about the source code for the donor patch is, in general, not available. So that we can present the FEH source code for the check in our example, we used the symbolic debugging information in FEH to manually locate the source code for the check.

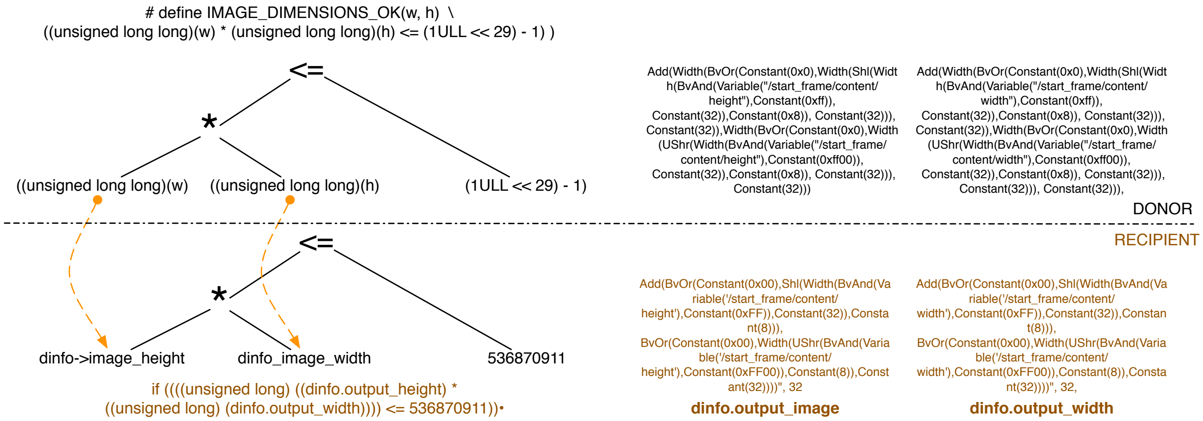
# define IMAGE_DIMENSIONS_OK(w, h) \
((unsigned long long)(w) * (unsigned long long)(h) <= (1ULL << 29) - 1))
<=
*
((unsigned long long)(w)
((unsigned long long)(h)
(1ULL << 29) - 1)
Add(Width(BvOr(Constant(0x0),Width(Shl(Widt
h(BvAnd(Variable("/start_frame/content/
height"),Constant(0xff)),
Constant(32)),Constant(0x8)), Constant(32))),
Constant(32)),Width(BvOr(Constant(0x0),Width
(UShr(Width(BvAnd(Variable("/start_frame/
content/height"),Constant(0xff00)),
Constant(32)),Constant(0x8)), Constant(32))),
Constant(32)))
Add(Width(BvOr(Constant(0x0),Width(Shl(Widt
h(BvAnd(Variable("/start_frame/content/
width"),Constant(0xff)),
Constant(32)),Constant(0x8)), Constant(32))),
Constant(32)),Width(BvOr(Constant(0x0),Width
(UShr(Width(BvAnd(Variable("/start_frame/
content/width"),Constant(0xff00)),
Constant(32)),Constant(0x8)), Constant(32))),
Constant(32))), Constant(32))),
DONOR
RECIPIENT
<=
*
dinfo->image_height
dinfo_image_width
536870911
if ((((unsigned long) ((dinfo.output_height) *
((unsigned long) (dinfo.output_width)))) <= 536870911))•
Add(BvOr(Constant(0x00),Shl(Width(BvAnd(Va
riable('/start_frame/content/
height'),Constant(0xFF)),Constant(32)),Consta
nt(8))),
BvOr(Constant(0x00),Width(UShr(BvAnd(Varia
ble('/start_frame/content/
height'),Constant(0xFF00)),Constant(8)),Const
ant(32))))", 32
dinfo.output_image
Add(BvOr(Constant(0x00),Shl(Width(BvAnd(Va
riable('/start_frame/content/
width'),Constant(0xFF)),Constant(32)),Constan
t(8))),
BvOr(Constant(0x00),Width(UShr(BvAnd(Varia
ble('/start_frame/content/
width'),Constant(0xFF00)),Constant(8)),Consta
nt(32))))", 32,
dinfo.output_width

Figure 3: Patch Transfer

This translation uses an instrumented execution of the donor to dynamically track the flow of input bytes through program. CP uses this instrumentation to obtain symbolic expressions, in terms of the input bytes, for relevant expressions that the application computes. In our example the translated application-independent symbolic expression for the check is:

```
ULessEqual(32,Shrink(32,Mul(64,Shrink(32,Div(32,BvOr(64,Shl(64,
ToSize(64,SShr(32,Sub(32,Add(32,Constant(8),Shl(32,Add(32,Shl
(32,ToSize(32,BvAnd(16,HachField(16,'/start_frame/content/height'),
Constant(0xFF))),Constant(8)),ToSize(32,UShr(32,BvAnd(16,HachField(16,
'/start_frame/content/height'),Constant(0xFF00)),Constant(8)))),
Constant(3))),Constant(1)),Constant(31))),Constant(32)),ToSize(64,
Sub(32,Add(32,Constant(8),Shl(32,Add(32,Shl(32,ToSize(32,BvAnd(16,
HachField(16,'/start_frame/content/height'),Constant(0xFF))),Constant(8)),
ToSize(32,UShr(32,BvAnd(16,HachField(16,'/start_frame/content/height'),
Constant(0xFF00)),Constant(8)))),Constant(3))),Constant(1)))),Constant(8))),
Shrink(32,Div(32,BvOr(64,Shl(64,ToSize(64,SShr(32,Sub(32,Add(32,
Constant(8),Shl(32,Add(32,Shl(32,ToSize(32,BvAnd(16,HachField(16,
'/start_frame/content/width'),Constant(0xFF))),Constant(8)),
ToSize(32,UShr(32,BvAnd(16,HachField(16,'/start_frame/content/width'),
Constant(0xFF00)),Constant(8)))),Constant(3))),Constant(1)),
Constant(31))),Constant(32)),ToSize(64,Sub(32,Add(32,Constant(8),
Shl(32,Add(32,Shl(32,ToSize(32,BvAnd(16,HachField(16,
'/start_frame/content/width'),Constant(0xFF))),Constant(8)),
ToSize(32,UShr(32,BvAnd(16,HachField(16,'/start_frame/content/width'),
Constant(0xFF00)),Constant(8)))),Constant(3))),Constant(1)))),
Constant(8))))),Constant(536870911))
```

There are two primary reasons for the complexity of this excised check. First, it correctly captures how FEH manipulates the input fields to convert from big-endian (in the input file) to little-endian (in the FEH application) representation. The excised check correctly captures the shifts and masks that are performed as part of this conversion. Second, FEH casts the 16-bit input fields to unsigned long long integers before it performs the overflow check. The excised check properly reflects these operand length manipulations.

Patch Transfer: The next step is to insert the check into the recipient CWebP application. There are two related challenges: 1) finding a successful insertion point for the check and 2) translating the check from the application-independent representation into the data representation of the recipient CWebP application. Note that this translation must find CWebP data structures that contain the relevant input field values and express the check in terms of these data structures.

Candidate Patch Insertion Point Identification: CP runs CWebP (the recipient) on the seed input. At each function the CP instrumentation records the input fields that the function reads. CP identifies program points at which the function has read all of the input fields as potential patch insertion points. In our example, CP recognizes that the `ReadJPEG` function has read both the input JPG `width` and `height` fields after line 4 in Figure 1. It therefore identifies the point after this statement as a candidate insertion point. The next step is to use the variables and data structures available at this point to express the check.

Patch Translation: To translate the patch into the recipient, CP first finds the relevant input fields as stored in the variables and data structures of the recipient. It then determines how to use these fields to express the check.

To find the values, CP uses the debugging information from the recipient binary to identify the local and global variables available at that candidate insertion point. Using these variables as roots, it traverses the data structures to find memory locations that store relevant input fields or values computed from relevant inputs fields and constants. As part of this traversal it also records expressions (in the name space of the recipient) that evaluate to each of the input fields or input field expressions. In our example CP determines that `dinfo.height` contains the JPG `height` input field and `dinfo.width` contains the JPG `width` input field.

The next step is to use the extracted recipient expressions to express the extracted check in the name space of the recipient. CP recursively processes the application-independent expression tree to find subtrees that always evaluate to the same value as one of the extracted recipient expressions. CP uses an SMT solver to determine this equivalence (see Section 3.3). In our example, CP produces the following translated check, which it inserts after line 4 in Figure 1:

```
if (!((unsigned long long)dinfo.output_height *
    (unsigned long long)dinfo.output_width)<=536870911)) {
   exit(-1);
}
```

Note that CP was able to successfully convert the complex application-independent excised check into this simple form — the SMT solver detects that CWebP and FEH, even though developed independently, perform semantically equivalent endianess conversions, shifts, and masks on the input fields. CP therefore realizes that the input fields are available in the same format in both the CWebP and FEH internal data structures, enabling CP to generate a simple patch that accesses the CWebP data structures directly with no complex format conversion. The generated patch evaluates the check and, if the input fails the check, exits the application. The rationale is to exit the application before the integer overflow (and any ensuing errors or vulnerabilities) can occur.

Multiple Patch Insertion Points: For CWebP, CP identifies 38 candidate insertion points. 2 of these points are *unstable* — in some executions of the point, the generated expressions reference values other than the desired JPEG `width` and `height` input fields. To avoid perturbing computations not related to the error, CP filters out these unstable points. CP then sorts the remaining generated patches by size and attempts to validate the patches in that order. In our example the above patch is the first patch that CP tries (and this patch validates).

Patch Validation: Finally, CP rebuilds CWebP, which now includes the generated patch, and subjects the patch to a number of tests. First, it ensures the compilation process finished correctly. Second, it executes the patched version of CWebP on the error-triggering input and checks that the input no longer triggers the error (CP runs CWebP under Valgrind memcheck to detect any errors that do not manifest in crashes). Third, it runs a regression test that compares the output of the patched application to the output of the original application, on a regression suite of inputs that the application is known to process correctly. Fourth, CP runs the patched version of the application through the DIODE error discovery tool to determine if DIODE can generate new error-triggering inputs. In our example DIODE finds no new error-trigging inputs — if it had, CP would have rerun the entire patch discovery and generation process, patching the discovered errors, until DIODE discovered no new errors. The end result, in this example, is a version of CWebP that contains a check that completely eliminates the integer overflow error.

3. Design and Implementation

We next discuss how CP deals with the many technical issues it must overcome to successfully transfer code between applications. CP consists of approximately 10,000 lines of C (most of this code implements the taint and symbolic expression tracking) and 4,000 lines of Python (code for rewriting donor expressions into expressions that can be inserted into the recipient, code that generates patches from the bitvector representation, code that interfaces with Z3, and the code that manages the database of relevant experimental results). Figure 4 presents an overview of the CP components.

3.1 Donor Selection

For each input format, CP works with a set of applications that process that format. Given seed and error-triggering inputs, CP considers applications that can successfully process both inputs as potential donors. Open source repositories such as github can be a rich source of independently developed applications that process the same input formats. Different versions, releases, or variants of the same application can also be good sources of patches either for regression errors introduced during maintenance or to obtain targeted updates for specific errors. Our set of benchmark donors includes both sources of applications (Section 4).

3.2 Candidate Check Discovery and Excision

To extract candidate checks from donor applications, CP implements a fine-grained dynamic taint analysis built on top of the Valgrind [46] binary analysis framework. Our analysis takes as input a specified taint source, such as a file or a network connection, and marks all data read from the taint source as tainted. Each input byte is assigned a unique label and is tracked by the execution monitor as it propagates through the application. Our analysis instruments arithmetic instructions (e.g., ADD, SUB), data movement instructions (e.g., MOV, PUSH), and logic instructions (e.g., AND, XOR). It also supports additional instrumentation to reconstruct the full symbolic expression of each computed value, which records how the application computes the value from input bytes and constants.

CP can optionally work with only a specified subset of the input bytes. We call this subset the *relevant bytes*. Working with properly identified relevant bytes can often improve the efficiency of the analysis without hampering its ability to find successful patches (because only a subset of the bytes are relevant to the patch). In our experiments, CP identifies the relevant bytes as those input fields that differ between the seed and error-triggering inputs.

CP uses Hachoir [4] to convert byte ranges into symbolic input fields. If Hachoir does not support a particular input format or is otherwise unable to perform this conversion, CP also supports a raw mode in which all input bytes are represented as offsets. Raw mode is effective, for example, for closely related inputs generated by standard error-finding tools [6, 25, 55, 61].

Identify Candidate Checks: CP runs the dynamic taint analysis on the donor application twice, once with the seed input and once with the error-triggering input. For each execution, CP extracts the executed conditional branch instructions and records which direction each execution of the branch takes. After filtering out branches that are not affected by the relevant bytes, branches that take different directions are the *candidate branches*. CP proceeds under the assumption that the condition associated with one of the candidate branches implements the desired check. Starting with the first (in the program execution order) candidate branch, CP attempts to transfer each check in turn until a transferred check successfully validates.

Check Excision: To obtain the application-independent form of the check, CP reruns the application with additional instrumentation that enables CP to reconstruct the full symbolic expression tree for the candidate check. This expression tree records how the donor application computes the condition of the candidate check from the input byte values and constants. Conceptually, CP generates a symbolic record of all calculations that the application performs. To reduce the volume of recorded information, CP only builds expression trees for calculations that involve the relevant input bytes. This optimization substantially reduces the volume of generated data.

A key challenge in transferring code between applications is translating between the different data representations in the donor and recipient. Translating the check into a symbolic expression over the input bytes performs the first half of this translation — it translates the check out of the naming environment and data structures of the donor into an application-independent representation.

Bit Manipulation Optimizations: As the symbolic expressions are recorded during the instrumented execution of the donor, CP applies several optimizations that reduce the size of the generated expressions. Among the most important of these are optimizations that simplify expressions generated by bit manipulation operations (such as shifts) that extract, align, or combine operands of subsequent computations. Because such bit manipulation operations occur frequently (for example, when the application extracts pieces of data read from the input or because of SSE optimizations) in donor binaries, the rules significantly reduce the size and complexity of the extracted symbolic expressions.

Figure 5 presents several rewrite rules that CP applies to simplify the symbolic expressions that such operations generate. The first two rules simplify symbolic expressions that extract the bottom or top 8-bit byte, respectively, of a 16 bit value. Here Shl(8,E) represents an 8-bit left shift of the 16 bit value E; ShrinkH(8,Shl(8,E)) converts the resulting 16 bit value into an 8 bit value by extracting the top byte. One important consequence of these rules is that, by eliminating discarded bytes from the symbolic representation, they can disentangle bytes from adjacent input fields that were read into the same word as part of the input process.

Note that the rules require the operand of the shift to be represented symbolically as a concatenation of two 8-bit bytes (the operand E must be of the form $[b_1, b_2]$, where b_1 and b_2 are independent bytes). Potential other representations that may appear as an operand include unified 16-bit values produced by addition or subtraction operations. CP does not further optimize the representation of bit manipulation operations involving such unified operands as there is no straightforward way to disentangle the two bytes of the unified operand.

The last two rules simplify symbolic expressions that start with a 16-bit value composed of two 8-bit bytes, shift one of the bytes out of the value, then or another byte into the position vacated by the shift. Here BvOrH(b_1, Shr(8,E)) bitwise ors b_1 into the top byte of the 16 bit value produced by Shr(8,E). The result is a new 16 bit

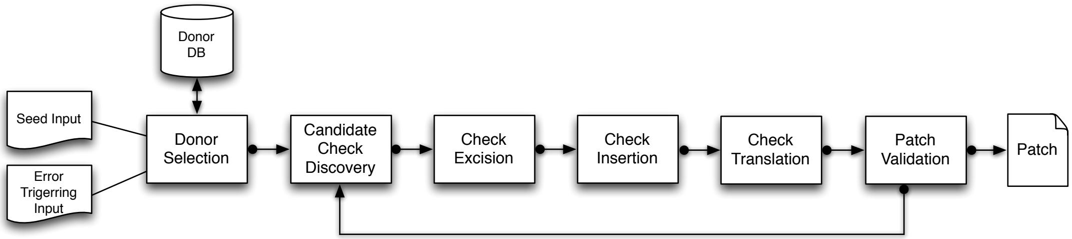

Figure 4: High-level overview of CP's components

$$\frac{E \equiv [b_1, b_2]}{\text{ShrinkH}(8, \text{Shl}(8, E)) \Rightarrow b_2} \qquad \frac{E \equiv [b_1, b_2]}{\text{ShrinkL}(8, \text{Shr}(8, E)) \Rightarrow b_1}$$

$$\frac{E \equiv [b_2, b_3]}{\text{BvOrH}(b_1, \text{Shr}(8, E)) \Rightarrow [b_1, b_2]} \qquad \frac{E \equiv [b_2, b_3]}{\text{BvOrL}(b_1, \text{Shl}(8, E)) \Rightarrow [b_3, b_1]}$$

Figure 5: CP Rewrite Rules for Bit Manipulation Operations

value. Once again, one of the benefits of these rules is that they can eliminate bytes that would otherwise entangle unrelated input fields that appear adjacent in the input. Like the first two rules, the last two rules require the initial 16-bit value to be represented symbolically as a concatenation of two 8-bit values.

CP also implements similar rules for other combinations of operand sizes. Specifically, there are similar rules for expressions that represent results of bit manipulation operations involving combinations of 8, 16, 32, and 64 bit values.

3.3 Check Insertion

To transfer the candidate check to an insertion point in the recipient application, CP rewrites the check to access the input field values as stored in variables and data structures available in the recipient.

Candidate Insertion Points: The first step is to find *candidate insertion points* – program points at which a set of values computed from all of the input bytes in the symbolic check expression are available as program expressions in the recipient. CP runs an instrumented version of the recipient that tracks the flow of the relevant input bytes through the application. Whenever the recipient evaluates an expression that involves the relevant input bytes, CP records the symbolic expression for the computed value. This symbolic expression records how the recipient application computes the value as a function of the input bytes and constants. Using these collected symbolic expressions, CP finds functions that access a set of values computed from all of the input bytes in the check. It then finds points within these functions at which the function has accessed all of these values. These points are the set of candidate insertion points.

Unstable Points: In general, the application may execute a candidate insertion point multiple times, potentially accessing different input bytes or even different values not derived from the input bytes on different executions. Candidate insertion points in multipurpose code such as libraries, for example, may execute with different values when invoked from different parts of the computation. To minimize the risk that the inserted check may affect a computation not related to the error, CP filters out all points that access different values on different executions (we call these points *unstable* points). The goal is choose the insertion point so that the patch performs the check only when it is relevant to the error.

Paths to Relevant Values: CP next attempts to express the extracted symbolic check in terms of the available variables and data structures at the remaining stable candidate insertion points. Given a candidate point, CP uses the debugging information to find the set V of local and global variables available at that point. Starting with these variables as roots, it then uses the debugging information to traverse the data structures to find *relevant values* (values computed from relevant fields and constants) stored in the data structures. As part of the traversal it computes the data structure traversal paths that lead to these relevant values.

Figure 6 presents the traversal algorithm. Starting from a given variable or data structure traversal path, the algorithm computes names that lead to reachable relevant values. Each name has the form $\langle p, E \rangle$. Here p is a path through the reachable data structures. Each path p starts at a variable v, then identifies a sequence of pointer dereferences and data structure field accesses that reaches the relevant value. The symbolic expression E records how the program computed the value from relevant input bytes.

For each variable $v \in V$, CP invokes the traverse algorithm and merges the resulting sets of names. The algorithm recursively traverses the data structures of the recipient program based on type signatures from the debugging information. At line 15, it uses the debugging information to determine the type of the path p. At line 16, it queries the symbolic tracking analysis results to obtain the corresponding symbolic expression for the traversed path p.

Check Translation: The next step is to rewrite the application-independent form of the check to use the variables and data structures of the recipient. Figure 7 presents the CP expression rewrite algorithm. The algorithm takes as input a symbolic expression E and a set of names *Names* produced by the traversal algorithm in Figure 6. It then uses the *Names* to translate E to use the available variables and data structures at the candidate insertion point in the recipient. E may take one of four possible forms, 1) an input field, 2) a constant c, 3) a unary operation expression $\langle unaryop, E \rangle$, or 4) a binary operation expression $\langle binop, E_1, E_2 \rangle$.

The algorithm first uses an SMT solver to try to find a single value in the recipient with the same value as the expression E. In practice, CP is often able to find single recipient values that are equivalent to very complex expressions E — many of these symbolic expressions include complex shift and mask operations that are also performed by the recipient as it reads the input. Otherwise the algorithm decomposes the expression and attempts to rewrite each subexpression recursively (lines 13-15 for expressions with unary operations, lines 16-19 for expressions with binary operations). Constants (line 20) translate directly.

CP implements two optimizations that reduce the number of solver invocations: 1) if two symbolic expressions depend on different sets of input bytes, CP does not invoke the solver and 2) CP caches all queries to the SMT solver so that it can retrieve results from the cache for future duplicate queries. Together, these two optimizations produce an order of magnitude reduction in the translation times.

There are two ways for the Rewrite algorithm to fail. First, it does not attempt to rearrange or reorder input bits as stored in the recipient data structures to match the groups of input bits as accessed by the application-independent representation of the check. So all of the required input bits may be available in the recipient but not stored as a contiguous block in the order accessed by the check. Second, it is possible for the function to access a value required to compute the check, then overwrite the value before it reaches the insertion point.

```
Parameters:
  p: A data structure path.
Subroutines:
  Type(p): The type of the path p.
  Fields(t): If t is a struct type, the set of fields in t.
  Addr(p): The address (at runtime) for the path p.
  Expr(a): The symbolic expression for the value
    stored in the address a.
  Visited(a): A boolean that tracks whether the address
was already processed to avoid infinite recursion.
Returns:
  A set of path, symbolic expression pairs.

Traverse(p) {
  $T \leftarrow Type(p)$
  $E \leftarrow Expr(Addr(p))$
  if (Visited(Addr(p))) return $\emptyset$
  else if (T is Pointer) return Traverse("(*"+p+")")
  else if (T is Struct)
    $Names \leftarrow \emptyset$
    for f in Fields(T)
      $Names \leftarrow Names \cup$ Traverse(p+"."+f)
    return Names
  else if ($E \neq$ NIL) return $\{\langle p,\ E\rangle\}$
  return $\emptyset$
}
```

Figure 6: CP Data Structure Traversal Algorithm

```
Parameters:
  E: A symbolic expression over input values.
  Names: A set of available names.
Subroutines:
  SolverEquiv($E_1$, $E_2$): Query the SMT solver to determine
    whether expressions $E_1$ and $E_2$ are equivalent.
Return:
  Rewritten expression of E or NIL if failed

Rewrite(E, Names) {
  for $\langle p,E'\rangle$ in Names
    if (SolverEquiv(E,E')) return p
  if (E is of the form $\langle unaryop,E_1\rangle$)
    $E_1' \leftarrow$ Rewrite($E_1$,Names)
    if ($E_1' \neq$ NIL) return $\langle unaryop,E_1'\rangle$
  else if (E is of the form $\langle binop,E_1,E_2\rangle$)
    $E_1' \leftarrow$ Rewrite($E_1$,Names)
    $E_2' \leftarrow$ Rewrite($E_2$,Names)
    if ($E_1' \neq$ NIL and $E_2' \neq$ NIL) return $\langle binop,E_1',E_2'\rangle$
  else if (E is Constant c) return c
  return NIL
}
```

Figure 7: CP Rewrite Algorithm

In this case the value may be unavailable at the insertion point even though it was previously accessed by the enclosing function.

If CP successfully constructs the new condition, CP generates a *candidate patch* as an if statement inserted at the insertion point. In the current implementation, CP transforms the constructed bitvector condition into a C expression as the if condition (appropriately generating any casts, shifts, and masks required to preserve the semantics of the transferred check). If the condition is satisfied, the patch exits the application with an `exit(-1)`.

3.4 Patch Validation

CP first recompiles the patched recipient application. It then executes the patched application on the bug-triggering input to verify that the patch successfully eliminates the error for that input. CP also runs the patched build on a set of regression suite inputs to validate that the patch does not break the core functionality of the application. As appropriate, CP may also test other error-triggering inputs or run additional error-finding tools (such as DIODE) to determine if the patch leaves any residual errors behind. If so, CP recursively attempts to find and transfer patches that eliminate the residual errors.

4. Experimental Results

We evaluate CP on three classes of errors — out of bounds access, integer overflow, and divide by zero errors. The two out of bounds access errors occur in JasPer 1.9 [11] and gif2tiff 4.0.3 [9] and are triggered by JPEG2K (JasPer) and gif (gif2tiff) images. OpenJPEG [13] and Display 6.5.2-9 [5] are the donors. We use standard fuzzing techniques to obtain the seed and error-triggering inputs.

The seven integer overflow errors occur in four applications: CWebP 0.31 [1], Dillo 2.1 [2], swfplay 0.55 [7], and Display 6.5.2-8 [5]. Two of these errors were listed in the CVE database; one was first discovered by BuzzFuzz [25]; the other four were, to the best of our knowledge, first discovered by DIODE [55]. The errors are triggered by JPG image files (CWebP), PNG image files (Dillo), SWF video files (swfplay), and TIFF image files (Display). The donor applications include FEH-2.9.3 [3], mtpaint 3.4 [12], ViewNoir 1.4 [8], and 0.8.11 [10]. We use DIODE to obtain the seed and error-triggering inputs.

The two divide by zero errors occur in Wireshark-1.4.14 [14] and are triggered by degenerate network packets with zero size fields. Wireshark-1.8.6 is the donor — in this scenario the goal is to obtain a targeted update that eliminates the error without the potential disruption of a full update to a later version. Starting with an error-triggering input from the corresponding CVE report, we used standard techniques to obtain a corresponding seed input that did not trigger the error.

We obtained integer overflow errors from the DIODE project [55]. The buffer overflow errors are reported as security vulnerabilities in the CVE database (CVE-2012-3352,CVE-2013-4231). We selected donor applications by collecting applications that successfully process the seed and error-triggering inputs. We further filter any applications that use the same underlying library (and version) to process inputs (e.g., we select only one donor application that uses libjpeg to process jpeg images). For every class of errors, we try all combinations of recipient-donor pairs that can process the same inputs.

Results Summary: Figure 8 summarizes the results of these experiments. There is a row in the table for each combination of error and donor. The first column (Recipient) identifies the recipient application that contains the error. The second column (Target) identifies the source code file and line where the vulnerability occurs. The third column (Donor) identifies the donor application. The fourth column (Patch Time) presents the amount of time that CP required to generate the patch.

The fifth column (Relevant Branches) presents the number of branches that depend on relevant values. The sixth column (Flipped Branches) presents the number of branches that take different directions for the seed and error-triggering inputs. Several entries are of the form $[X_1, ..., X_n]$. These entries correspond to errors with multiple error-triggering inputs. The first patch eliminates the error for the first input but there is a residual error. Recursive CP executions transfer patches to eliminate each remaining residual error, with an error eliminated per patch transfer. In all cases the final sequence of patches completely eliminates the exposed errors. For all four cases with multiple patches DIODE, running on the previously patched version, automatically generates the additional error-triggering inputs. The seventh column (Used Checks) presents the number of checks that CP transferred to eliminate the error. In all of our experiments, the transferred checks came from the first (in the execution order) flipped branch.

The eighth column (Candidate Insertion Points) contains entries of the form $X - Y - Z = W$. Here X is the number of candidate insertion points, Y is the number of unstable points (CP filters these points), Z is the number of insertion points at which CP was unable to translate the patch (see Section 3.3), and W is the number of points at which CP is able to insert a successfully translated patch.

Recipient	Target	Donor	Generation Time	# Relevant Branches	# Flipped Branches	# Used Checks	# Candidate Insertion Pts	Check Size
CWebP 0.3.1	jpegdec.c:248	feh-2.9.3	4m	157	5	1	38 - 2 - 31 = 5	$57 \rightarrow 4$
CWebP 0.3.1	jpegdec.c:248	mtpaint-3.40	4m	94	5	1	38 - 2 - 30 = 6	$28 \rightarrow 2$
CWebP 0.3.1	jpegdec.c:248	viewnior-1.4	1m	137	1	1	38 - 2 - 31 = 5	$111 \rightarrow 12$
Dillo 2.1	png.c@203	mtpaint-3.40	3m	29	[1,1]	2	16 - 1 - 8 = 7 16 - 1 - 9 = 6	$[(18 \rightarrow 1),(18 \rightarrow 1)]$
Dillo 2.1	png.c@203	feh-2.9.3	3m	120	[4,1]	2	16 - 1 - 9 = 6 16 - 1 - 9 = 6	$[(76 \rightarrow 8), (37 \rightarrow 3)]$
Dillo 2.1	png.c@203	viewnior-1.4	18m	117	1	1	16 - 1 - 9 = 6	$79 \rightarrow 12$
Dillo 2.1	fltkimagebuf.cc@39	mtpaint-3.40	13m	29	[1,1]	2	22 - 1 - 10 = 11 22 - 1 - 11 = 10	$[(18 \rightarrow 1),(18 \rightarrow 1)]$
Dillo 2.1	fltkimagebuf.cc@39	feh-2.9.3	2m	120	4	1	22 - 1 - 11 = 10	$76 \rightarrow 9$
Dillo 2.1	fltkimagebuf.cc@39	viewnior-1.4	9m	117	1	1	22 - 1 - 11 = 10	$79 \rightarrow 12$
Display 6.5.2	xwindow.c@5619	viewnior-1.4	4m	142	6	1	74 - 5 - 60 = 9	$55 \rightarrow 14$
Display 6.5.2	xwindow.c@5619	feh-2.9.3	4m	147	6	1	74 - 7 - 58 = 9	$17 \rightarrow 4$
Display 6.5.2	display.c@4393	viewnior-1.4	4m	142	6	1	49 - 2 - 45 = 2	$55 \rightarrow 14$
Display 6.5.2	display.c@4393	feh-2.9.3	4m	147	6	1	49 - 2 - 45 = 2	$17 \rightarrow 4$
SwfPlay 0.5.5	jpeg_rgb_decoder.c@253	gnash	12m	264	7	1	43 - 3 - 35 = 5	$53 \rightarrow 12$
SwfPlay 0.5.5	jpeg.c@192	gnash	18m	264	[1,1,3,3]	4	38 - 2 - 34 = 2 38 - 2 - 34 = 2 38 - 0 - 37 = 1 38 - 0 - 37 = 1	$[(5 \rightarrow 1),(5 \rightarrow 1),(4 \rightarrow 1),(3 \rightarrow 1)]$
JasPer 1.9	jpg_dec.c:492	OpenJpeg 1.5.2	1m	63	19	1	18 - 1 - 16 = 1	$188 \rightarrow 3$
gif2tiff 4.0.3	gif2tiff.c:355	Display 6.5.2-9	9m	9	2	1	2 - 1- 0 = 1	$3 \rightarrow 3$
Wireshark 1.4.14	packet-dcp-etsi.c:258	Wireshark 1.8.6	4m	101	2	1	40 - 5 - 15 = 20	$6 \rightarrow 2$

Figure 8: Summary of CP Experimental Results

The ninth column (Check Size) contains entries of the form $X \rightarrow Y$. Here X is the number of operations in the excised application-independent representation of the check. Y is the number of operations in the translated check as it is inserted into the recipient. We attribute the significant size reduction to the ability of the CP Rewrite algorithm (Figure 7) to recognize complex expressions that are semantically equivalent. The typical scenario is that CP recognizes that a complex application-independent expression containing shifts and masks from (for example) the endianess conversion is equivalent to a single variable or data structure field in the recipient.

We next discuss several specific patches in more detail (see Section 2 for a detailed example that illustrates how CP corrects an integer overflow error).

4.1 JasPer 1.9

JasPer 1.9 is an open-source image viewing and image processing utility. It is specifically known for its implementation of the JPEG-2000 standard. JPEG-2000 images may be composed of multiple tiles, with the number of tiles specified by a 16 bit field in the input file. JasPer contains an off-by-one error in the code that processes JPEG-2000 tiles. When JasPer processes the tiles, it includes code that is designed to check that the number of tiles actually present in the image is less than or equal to the number specified in the input file. Unfortunately, the check was miscoded — at jpc_dec.c:492, JasPer checks if the number of the current tile is greater than (>) the specified number of tiles. The correct check is a greater than or equal to (>=) check. The result is that JasPer can write tile data beyond the end of the buffer allocated to hold that data.

The following correct check appears in OpenJPEG 1.5.2 at j2k.c:1394:[2]

```
if ((tileno < 0) || (tileno >= (cp->tw * cp->th))) { ... }
```

CP automatically locates the compiled version of this correct check in the OpenJPEG binary and correctly transfers the check into JasPer at jpc_dec.c:492 as:

```
if (!(!(dec->numtiles <= sot->tileno))) { exit(-1); }
```

To generate this check, CP had to map `tileno` in OpenJPEG 1.5.2 to `dec->numtiles` in JasPer and recognize that `cp->tw * cp->th` in OpenJPEG 1.5.2 has the same value as `sot->tileno` in JasPer. This patch highlights CP's data structure translation capabilities and its ability to recognize different expressions in different applications that produce the same value. We note that the OpenJPEG `tileno < 0` check is redundant — other constraints in both OpenJPEG and JasPer ensure that `tileno` and `dec->numtiles` are always nonnegative.

4.2 gif2tiff

gif2tiff is a utility in the libtiff-4.0.3 library which converts gif images to the tif format. gif2tiff is vulnerable to a buffer overflow attack when processing gif images. gif2tiff iterates over the size of the LZW code size, which under the gif specification should be limited to a size of 12. Without a check to constrain the code size to 12, the loop over the code size in `gif2tif.c:355` can be forced to overwrite over a set of statically allocated buffers.

CP successfully created a patch for this error using ImageMagick-6.5.2-9 as the donor. The transfered check appears in ImageMagick-6.5.2-9 as:

```
#define MaximumLZWBits  12
if (data_size > MaximumLZWBits)
   ThrowBinaryException(CorruptImageError,
         "CorruptImage",image.filename);
```

This check was translated into the following patch for gif2tiff (`gif2tiff.c:357`) as:

```
if (!(datasize <= 12)) {exit(-1);}
```

The check correctly enforces the gif specification that the code size should have a maximum size of 12 and protects gif2tiff from the buffer overflow vulnerability.

4.3 Wireshark

Wireshark is a popular open-source packet analyzer. It is used for a variety of networking tasks such as network analysis, network troubleshooting and protocol development. Wireshark 1.4.14 contains a divide by zero error at packet-dcp-etsi.c:276 in code that processes DCP ETSI packets.

[2] CP does not have access to the OpenJPEG 1.5.2 source code — it instead transfers the check directly from the compiled binary. For presentation purposes, we used the debugging information to manually locate this check in the OpenJPEG source code.

The following check, which appears in a later version of Wireshark (1.8.6) and checks that the length of the packet payload is not zero before attempting to further process the packet, eliminates this error:

```
if (real_len) ...
```

Recognizing that `real_len` and `plen` contain the same input fields (the different names reflect the substantial reengineering between the two versions), CP inserts the check into Wireshark 1.4.14 at packet-dcp-etsi.c:258 as:

```
if (!(!(plen == 0))) { exit(-1); }
```

Empirically, returning zero as the result of divide by zero errors often enables the application to continue to execute productively [40]. We therefore implemented an alternate strategy that returns 0 if the check fires rather than exiting. Our results and manual analysis indicate that this strategy delivers correct continued execution for both of the Wireshark divide by zero errors.

4.4 Discussion

The patches we present above are, in general, representative of the remaining patches (our CP technical report presents these remaining patches [58]). Like the JasPer patch, 10 of the remaining 18 patches access the stored field values via pointers. This fact highlights the critical role that the CP data structure traversal and rewrite algorithms play in enabling the data structure translations required for successful transfers. As the numbers in Figure 8 indicate, the CP rewrite algorithm is effective at generating compact readable patches — like the patches we present above, they are all expressible in at most several lines of code.

Our manual evaluation of the patches indicates that 1) they all completely eliminate the target error and 2) they do not affect computations unrelated to the error. We attribute this success to three factors: 1) the developers of the donor applications were able to write code that correctly handled the case responsible for the error in the recipient, 2) CP was able to locate and transfer the check that handles this case, and 3) eliminating unstable points is an effective way to filter out the many points that appear in multipurpose library code. The result is focused patches that fire only when necessary to eliminate the target error.

The results also highlight several aspects of CP's techniques. Most of the applications contain more than 100 checks that involve relevant input fields. The ability of CP to find the single check (within these more than 100 checks) that eliminates the error highlights the effectiveness of CP's check identification technique (which uses flipped branches to isolate the relevant check). CP's ability to find effective patch insertion points among the many potential source code locations highlights the effectiveness of CP's insertion point location algorithm.

All of the transfers involve naming and/or data structure translations. In some cases the translation could be accomplished via a simple variable renaming (if the source code for the donor was available, which it may not be). In other cases there is a more significant data structure translation that involves finding values stored in different structures or accessed via pointers. Even though the application-independent representation of the checks is typically quite complex, CP's Rewrite algorithm is very effective at finding small recipient representations of the check.

Given that programs often deploy different data representations, any general code transfer system requires some data structure translation technique. CP's technique, which is based on representing values as functions of the input bytes, then traversing the data structures to find desired values, would be equally effective for any approach that can establish a correspondence between executions of the donor and recipient.

CP's current data structure translation technique is effective at translating (potentially quite complex) computations that can be expressed as single expressions. Already this technique enables CP to eliminate significant errors in real-world applications. Generalizing CP to support expressions with simple conditionals would be relatively straightforward — augmenting CP's data structure translation technique with a symbolic execution of the two branches would suffice. An effective loop body identification and generalization technique would enable CP to support loops.

5. Related Work

N-Version Programming: N-version programming [21] aims to improve software reliability by independently developing multiple implementations of the same specification. All implementations execute and the results are compared to detect faulty versions. The expense of N-version programming and a perception that the multiple implementations may suffer from common errors and specification misinterpretations has limited the popularity of this approach [32].

Rather than running multiple versions and comparing the results, CP transfers correct code to obtain a single improved hybrid system. CP has a simpler execution model (run a single hybrid system instead of multiple systems) and can leverage applications with overlapping but not identical functionality. Also unlike traditional N-version programming, CP is also designed to work with applications that are produced by multiple global, spontaneous, and uncoordinated development efforts performed by different organizations. Our results indicate that these development efforts can deliver enough diversity to enable CP to find and transfer correct error checks.

Program Fracture and Recombination: Program fracture and recombination breaks multiple applications into *shards*, analyzes the shards, then transfers shards across applications [57]. Transferred shards may deliver better performance, more correct code, more secure code, more analyzable code, cleaner code, the ability to exploit specialized hardware, or the ability to operate successfully in parallel or distributed computing contexts, to cite a few potential applications of the technique [57]. It may also enable the automatic identification and transfer of functionality from donor to recipient applications to obtain new hybrid applications that incorporate the best or most desirable functionality developed anywhere.

Static Program Repair: Staged Program Repair (SPR) uses *condition synthesis* to instantiate *transformation schemas* to repair errors in large software systems [37]. SPR's novel *staged repair* approach combines a rich space of program repairs with a targeted search algorithm that makes this space viably searchable in practice. The results show that 1) SPR can find significantly more correct repairs than previous automatic patch generation systems [34, 63] and 2) the majority of these correct repairs lie outside the search spaces of these previous systems [34, 63].

GenProg [34, 64] is an automatic program repair tool that uses genetic programming to synthesize program patches. AE is a follow-on tool that uses a set of equivalence tests to reduce the patch search space [63]. Despite what one might reasonably conclude from reading the relevant papers, GenProg and AE are remarkably ineffective at fixing bugs — because of errors in the patch evaluation scripts, 74% (GenProg) and 50% (AE) of the reported patches produce incorrect outputs *even for the inputs in the test suite used to validate the patches* [49]. GenProg and AE produce correct patches for 2 (GenProg) and 3 (AE) of the 105 defects on which they were evaluated [49]. Kali, an automatic patch generation system that aspires only to delete the code that contains the exposed defect, can do as well [49]. SPR finds correct repairs for 11 of the 105 defects [37].

All of these systems work with a single application and require recompilable source code. CP, in contrast, eliminates errors (successfully) by transferring correct code across multiple applications, including binary donor applications.

Khmelevsky et al. [30] present a source-to-source repair tool for missing return value checks after system library calls (e.g., fopen()). The tool scans through the source code for these library calls. For each of these calls, if the source code misses the corresponding check after the call, the tool will automatically add one.

Logozzo and Ball [36] have proposed a program repair technique that provides the guarantee of verified program repair in the form that the repaired program has more good executions and less bad executions than the original program. However, it relies on developer-supplied contracts (i.e., preconditions, postconditions, and object invariants) for scalability, which makes the technique less practical. In contrast, CP is fully automatic — it does not require any human annotations to transfer patches from the donor application to the recipient application.

SJava [23] is a Java type system that exploits common iterative structures in applications. When a developer writes program in SJava, the compiler can prove that the effects of any error will be flushed from the system state after a fixed number of iterations.

Runtime Program Repair: Failure-Oblivious Computing enables applications to survive common memory errors [51]. It recompiles the application to discard out of bounds writes, manufacture values for out of bounds reads, and enable applications to continue along their normal execution paths. RCV [40] enables applications to dynamically recover from divide-by-zero and null-dereference errors. When such an error occurs, RCV attaches the application, applies a fix strategy that typically ignores the offending instruction, forces the application to continue along the normal execution path, contains the error repair effect, and detaches from the application once the repair succeeds. SRS [45] enables server applications to survive memory corruption errors. When such an error occurs, it enters a crash suppression mode to skip any instructions that may access corrupted values. It reverts back to normal mode once the server moves to the next request.

ClearView [48] first learns a set of invariants from training runs. When a learned invariant is violated during the runtime execution, it generates repairs that enforce the violated invariant via binary instrumentation. Jolt [19] and Bolt [31] enable applications to survive infinite loop errors. Bolt attaches to unresponsive applications, detects if the application is in an infinite loop, and if so, either exits the loop or returns out of the enclosing function to enable the application to continue successful execution.

DieHard [18] provides probabilistic memory safety in the presence of memory errors. In stand-alone mode, DieHard replaces the default memory manager with a memory manager that places objects randomly across a heap to reduce the possibility of memory overwrites due to buffer overflows. In replicated mode, DieHard obtains the final output of the application based on the votes of multiple replications. Exterminator [47] automatically generates patches for buffer overflow and dangling pointer errors. Starting with an input that triggers the error, Exterminator patches overflow errors by padding allocated objects and patches dangling pointer errors by deferring object deallocations.

Rx [50] and ARMOR [20] are runtime recovery systems based on periodic checkpoints. When an error occurs, Rx [50] reverts back to a previous checkpoint and makes system-level changes (e.g, thread scheduling, memory allocations, etc.) to search for executions that do not trigger the error. ARMOR [20] reverts back to a previous checkpoint and finds semantically equivalent workarounds for the failed component based on user-provided specifications.

Error Virtualization [53, 54] is a general error recovery technique that retrofits exception-handing capabilities to legacy software. Failures that would otherwise cause a program to crash are turned into transactions that use a program's existing error handling routines to survive unanticipated faults.

Input rectification [38] empirically learns input constraints from benign training inputs and then enforces learned constraints on incoming inputs to nullify potential errors. SIFT [39] can generate sound input filter constraints for integer overflow errors at critical program points (i.e., memory allocation and block copy sites).

All of the above techniques aim to repair the application at runtime to recover from or nullify the error. In contrast, CP is designed to locate and transfer correct code from donors to recipients to, after recompilation, directly eliminate the error.

Deviant Code and Code Clone Errors: Researchers have built tools that analyze programs to discover common resource usage patterns [33] and security checks that guard sensitive program actions [59]. The result is a model that an analysis tool can use find errors in code that deviates from the discovered patterns. Code cloning is a common software development activity. Maintenance of the resulting clones can introduce bugs or leave latent bugs in place when one clone is updated but another is not. One approach uses *linked editing* to ensure that clones are updated consistently [62]. Researchers have also developed approaches detect inconsistent code clones that may contain errors [24, 26, 35]. Because such techniques only provide reports that identify potential errors, and not inputs that demonstrate the actual existence of a suspected error, they are not directly relevant to CP.

Example-Driven Program Edits: SYDIT [41] and LASE [42] are given an original and modified method and synthesize a transformation that, when applied to the original method, produces the modified method. The goal is to obtain a transformation that can be applied to other methods to achieve a similar semantic goal. CP differs in that it works with multiple independent programs to transfer code from alien donors into unrelated recipients without human intervention. To this end, CP contains novel value name and data structure translation capability.

PHP Sanitization and Access Control Repair: PHPQuickFix and PHPRepair use string constraint-solving techniques to automatically repair php programs that generate HTML [52]. By formulating the problem as a string constraint problem, PHPRepair obtains sound, complete, and minimal repairs to ensure the patched php program passes a validation test suite. FixMeUp starts with a high-level specification that indicates the conditional statement of a correct access-control check and automatically computes an interprocedural access-control template (ACT), which includes all program statements involved in this instance of access control logic. The ACT serves as both a low-level policy specification and a program transformation template. FixMeUp uses the ACT to find faulty access-control logic that misses some or all of these statements, inserts only the missing statements, and ensures that unintended dependences do not change the meaning of the access-control policy. FixMeUp then presents the transformed program to the developer, who decides whether to accept the proposed repair [60].[3]

Researchers have developed a technique that is provided with two input validation and sanitization PHP functions (typically from different PHP client and server programs) and uses a static semantic analysis of the string operations to obtain finite state models that (potentially conservatively) characterize relevant input/output relations. It then uses automata operations on the finite state models to obtain patch automata that model different aspects of the desired validation and sanitization operations. From these patch automata it generates new PHP validation and sanitization functions [15]. These new functions return a string only if both of the original input validation and sanitization functions would also return that string. This technique is based on semantic analysis and synthesis of string operations. Its scope is therefore limited to string validation and sanitization func-

[3] These sentences are taken verbatim from the abstract of the FixMeUp paper [60].

tions with input/output relationships that it can accurately analyze and represent with finite state automata.

CP differs in multiple ways. For example, it transfers code between different donor and recipient applications, it works with binary donors, and it implements value naming and data structure transfer algorithms, to cite a few differences. Because CP works directly with code extracted from the donor, it is not limited to any particular semantic model or domain and can transfer arbitrary code relevant to many different problems and domains.

Accuracy-Enhancing Program Transformations: CP, like essentially all research that aspires to eliminate software errors, works with the standard binary correct/incorrect perspective on program behavior. From this perspective, the natural goal is to convert incorrect behavior into correct behavior. But it is also possible to approach program behavior from an accuracy perspective (which characterizes program behaviors not as correct/incorrect or acceptable/unacceptable, but simply as more or less accurate). The QuickStep parallelizing compiler takes this perspective — it automatically generates parallel loops with data races, then applies *accuracy-enhancing transformations* (such as synchronization insertion and privatization) to increase the accuracy of the program and obtain an acceptably (but not necessarily completely) accurate approximate computation [43, 44]. As approximate computing enters the mainstream, we expect accuracy-enhancing transformations to become increasingly visible and important.

6. Conclusion

In recent years the increasing scope and volume of software development efforts has produced a broad range of systems with similar or overlapping goals. Together, these systems capture the knowledge and labor of many developers. But each individual system largely reflects the effort of a single team and, like essentially all software systems, still contains errors.

We present a new and, to the best of our knowledge, the first, technique for automatically transferring code between systems to eliminate errors. The system that implements this technique, CP, makes it possible to automatically harness the combined efforts of multiple potentially independent development efforts to improve them all regardless of the relationships that may or may not exist across development organizations. In the long run we hope this research will inspire other techniques that identify and combine the best aspects of multiple systems. The ideal result will be significantly more reliable and functional software systems that better serve the needs of our society.

Acknowledgements

We thank the anonymous reviewers for their helpful feedback and our shepherd Emery Berger for his help with the camera ready version of the paper. This research was supported by DARPA (Grant FA8650-11-C-7192). We acknowledge an earlier technical report on CP [56].

References

[1] Cwebp. https://developers.google.com/speed/webp/docs/cwebp.

[2] Dillo. http://www.dillo.org/.

[3] Feh - a fast and light Imlib2-based image viewer. http://feh.finalrewind.org/.

[4] Hachoir. http://bitbucket.org/haypo/hachoir/wiki/Home.

[5] Imagemagick. http://www.imagemagick.org/script/index.php.

[6] Peach fuzzing platform. http://peachfuzzer.com/.

[7] Swfdec. http://swfdec.freedesktop.org/wiki/.

[8] Viewnoir - the elegant image viewer. http://xsisqox.github.io/Viewnior/.

[9] Libtiff. http://www.remotesensing.org/libtiff/.

[10] Gnu gnash. https://www.gnu.org/software/gnash/.

[11] The jasper project home page. http://www.ece.uvic.ca/~frodo/jasper/.

[12] mtpaint. http://mtpaint.sourceforge.net/.

[13] Openjpeg. http://www.openjpeg.org.

[14] Wireshark. https://www.wireshark.org/.

[15] M. Alkhalaf, A. Aydin, and T. Bultan. Semantic differential repair for input validation and sanitization. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 225–236, 2014.

[16] K. Ambrose, A. Koppenhofer, and F. Belanger. Horizontal gene transfer of a bacterial insect toxin gene into the epichloe fungal symbionts of grasses. *Scientific Reports*, 4, July 2014.

[17] M. Barlow. What Antimicrobial Resistance Has Taught Us About Horizontal Gene Transfer. *Methods in Molecular Biology*, 532: 397–411, 2009. . URL http://dx.doi.org/10.1007/978-1-60327-853-9_23.

[18] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06', pages 158–168. ACM, 2006. ISBN 1-59593-320-4. . URL http://doi.acm.org/10.1145/1133981.1134000.

[19] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard. Detecting and escaping infinite loops with jolt. ECOOP'11, 2011.

[20] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 782–791.

[21] L. Chen and A. Avizienis. N-version programming: A Fault-tolerance approach to reliability of software operation. In *The Twenty-Fifth International Symposium on Fault-Tolerant Computing Highlights from Twenty-Five Years*. IEEE, 1995.

[22] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel. Staged deployment in mirage, an integrated software upgrade testing and distribution system. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 221–236. ACM, 2007.

[23] Y. h. Eom and B. Demsky. Self-stabilizing java. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12', pages 287–298. ACM, 2012. ISBN 978-1-4503-1205-9. . URL http://doi.acm.org/10.1145/2254064.2254099.

[24] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 175–190, 2010.

[25] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009. ISBN 978-1-4244-3453-4. .

[26] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 55–64, 2007.

[27] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *OSDI*, volume 12, pages 221–236, 2012.

[28] M. A. Kay, J. C. Glorioso, and L. Naldini. Viral vectors for gene therapy: the art of turning infectious agents into vehicles of therapeutics. *Nat Med*, 7(1):33–40, Jan. 2001. . URL http://dx.doi.org/10.1038/83324.

[29] P. J. Keeling and J. D. Palmer. Horizontal gene transfer in eukaryotic evolution. *Nature Reviews Genetics*, 9(8), 8 2008.

[30] Y. Khmelevsky, M. Rinard, and S. Sidiroglou. A source-to-source transformation tool for error fixing. CASCON, 2013.

[31] M. Kling, S. Misailovic, M. Carbin, and M. Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12', pages 431–450. ACM, 2012. ISBN 978-1-4503-1561-6. . URL http://doi.acm.org/10.1145/2384616.2384648.

[32] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, 12:96–109, 1986.

[33] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 161–176. USENIX Association, 2006.

[34] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. ICSE 2012, 2012.

[35] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 289–302, 2004.

[36] F. Logozzo and T. Ball. Modular and verified automatic program repair. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12', pages 133–146, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. . URL http://doi.acm.org/10.1145/2384616.2384626.

[37] F. Long and M. Rinard. Staged Program Repair in SPR. Technical Report MIT-CSAIL-TR-2015-008, 2015. URL http://hdl.handle.net/1721.1/95970.

[38] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard. Automatic input rectification. ICSE '12, 2012.

[39] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard. Sound input filter generation for integer overflow errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14', pages 439–452, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. . URL http://doi.acm.org/10.1145/2535838.2535888.

[40] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via error shepherding. In *Proceedings of the 35th ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '14'. ACM, 2014.

[41] N. Meng, M. Kim, and K. S. McKinley. Sydit: creating and applying a program transformation from an example. In *SIGSOFT/FSE'11*, pages 440–443, 2011.

[42] N. Meng, M. Kim, and K. S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 502–511, 2013.

[43] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. Technical Report MIT-CSAIL-TR-2010-038, 2010. URL http://hdl.handle.net/1721.1/57475.

[44] S. Misailovic, D. Kim, and M. C. Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Trans. Embedded Comput. Syst.*, 12(2s):88, 2013.

[45] V. Nagarajan, D. Jeffrey, and R. Gupta. Self-recovery in server programs. ISMM '09', 2009.

[46] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. PLDI '07, 2007.

[47] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: automatically correcting memory errors with high probability. *ACM SIGPLAN Notices*, 42(6):1–11, 2007.

[48] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. SOSP '09. ACM, 2009.

[49] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. Technical Report MIT-CSAIL-TR-2010-038, 2015. URL http://dspace.mit.edu/handle/1721.1/94337.

[50] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies–a safe method to survive software failures. *ACM Trans. Comput. Syst.*, 2007.

[51] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

[52] H. Samimi, M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, and L. J. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 277–287, 2012.

[53] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *Proceedings of the general track, 2005 USENIX annual technical conference: April 10-15, 2005, Anaheim, CA, USA*, pages 149–161. USENIX, 2005.

[54] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: Automatic software self-healing using rescue points. In *ASPLOS*, pages 37–48, 2009. ISBN 978-1-60558-406-5. . URL http://doi.acm.org/10.1145/1508244.1508250.

[55] S. Sidiroglou, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. Rinard. Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement. In *ASPLOS*, 2015.

[56] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by multi-application code transfer. Technical Report MIT-CSAIL-TR-2014-024, Aug. 2014. URL http://dspace.mit.edu/handle/1721.1/91148.

[57] S. Sidiroglou-Douskos, E. Davis, and M. Rinard. Horizontal code transfer via program fracture and recombination. Technical Report MIT-CSAIL-TR-2015-012, 2015. URL http://dspace.mit.edu/handle/1721.1/96585.

[58] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by multi-application code transfer. Technical Report MIT-CSAIL-TR-2015-013, 2015. URL http://hdl.handle.net/1721.1/96625.

[59] S. Son, K. S. McKinley, and V. Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. *ACM SIGPLAN Notices*, 46(10):1069–1084, 2011.

[60] S. Son, K. S. McKinley, and V. Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *NDSS*, 2013.

[61] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, 2007.

[62] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 173–180. IEEE, 2004.

[63] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013.*

[64] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09', 2009.