Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2014-024 October 22, 2014

Automatic Error Elimination by Multi-Application Code Transfer

Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, Paolo Piselli, and Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Automatic Error Elimination by Multi-Application Code Transfer

Stelios Sidiroglou-Douskos Eric Lahtinen Fan Long Paolo Piselli Martin Rinard

August 11, 2014

ABSTRACT

We present pDNA, a system for automatically transferring correct code from donor applications into recipient applications to successfully eliminate errors in the recipient. Experimental results using three donor applications to eliminate seven errors in four recipient applications highlight the ability of pDNA to transfer code across applications to eliminate otherwise fatal integer overflow errors at critical memory allocation sites. Because pDNA 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, pDNA is the first system to eliminate software errors via the successful transfer of correct code across applications.

1 INTRODUCTION

Over the last decade, the software development community, both open-source and proprietary, has implemented multiple systems with similar functionality (for example, systems that process standard image and video files). In effect, the software development community is now engaged in a spontaneous N-version programming exercise. But despite the effort invested in these projects, errors and security vulnerabilities still remain a significant concern. Many of these errors are caused by an uncommon case that the developers of one (or more) of the systems did not anticipate. A key motivation for our research is the empirical observation that different systems often have different errors — an input that will trigger an error in one system can often be processed successfully by another system.

1.1 The pDNA Code Transfer System

We present pDNA, a system that automatically eliminates errors in recipient software systems by finding correct logic in donor systems, then transferring that logic from the donor into the recipient to enable the recipient to correctly process inputs that would otherwise trigger fatal errors. The result is a software hybrid that productively combines beneficial logic from multiple systems:

- **Error Discovery:** pDNA works with a seed input that does not trigger the error and a related input that does trigger the error. pDNA currently uses the DIODE integer overflow discovery tool, which starts with a seed input, then uses instrumented executions of the recipient program to find related inputs that trigger integer overflow errors at critical memory allocation sites.
- **Donor Selection:** pDNA next uses instrumented executions of other systems that can process the same inputs to find a donor that processes both the seed and error-triggering 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 (thereby eliminating the error in the recipient).
- **Candidate Check Discovery:** To identify the check that enables the donor to survive the error-triggering input, pDNA analyzes the executed conditional branches in the donor program to find branches that 1) are affected by input values involved in the overflow and 2) 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 Transfer:** pDNA next transfers the check from the donor into the recipient. There are two primary (and related) challenges: expressing the check in the name space of the recipient and finding an appropriate location to insert the check.

  pDNA first uses an instrumented execution of the donor on the error-triggering input to express the branch condition as a symbolic expression over the input bytes that determine the value of the branch condition — in effect, excising the check from the donor to obtain a system-independent representation of the check.

  pDNA then uses an instrumented execution of the recipient on the seed input to find *candidate insertion points* at which all of the input bytes in the branch condition are available in recipient program expressions. At these points, pDNA can generate a patch that expresses the condition as a function of these recipient expressions. This translation, in effect, implants the excised check into the recipient. pDNA tries each candidate insertion point in turn until it finds one that validates.

- **Patch Validation:** pDNA first uses regression testing to verify that the patch preserves correct behavior on the regression suite. It then checks that the patch enables the patched recipient to correctly process the error-triggering input.
  pDNA next uses DIODE to verify that the check actually eliminates the error. Specifically, pDNA processes the symbolic check condition, the symbolic expression for the size of the allocated memory block, 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 but also 2) generates an overflow in the computation of the size of the allocated block. If the patch validation fails, pDNA continues on to try other candidate insertion points, other candidate checks, and other donors.

The current pDNA implementation generates source-level recipient patches (given appropriate binary patching capability, it would also be straightforward to generate binary patches). But the donor analysis operates directly on stripped binaries with no need for source code or symbolic information of any kind. pDNA can therefore, for example, use closed-source proprietary binaries to obtain patches for open-source systems. It can also leverage binary donors in any other way that makes sense in a given situation.

1.2 Experimental Results

We evaluate pDNA on seven errors in four recipient applications (CWebP 0.31 [2], Dillo 2.1 [3], swfplay 0.55 [9], and Display 6.5.2-8 [7]). The donor applications are FEH-2.9.3 [4], mtpaint 3.4 [8], ViewNoir 1.4 [10], and gnash 0.8.11 [5]. For all of the 10 possible donor/recipient pairs (the donor and recipient must process inputs in the same format), pDNA was able to successfully generate a patch that eliminated the error.

To fully appreciate the significance of these results, consider that the donor and recipient applications were developed in independent development efforts with no shared source code base relevant to the error. This is not a situation in which pDNA is simply propagating patches from one version of a shared code base to a previous version — the patched code is instead excised from an independently developed alien donor and successfully implanted into the recipient. pDNA's ability to obtain an application-independent representation of the check (by expressing the check as a function of the input bytes) is critical to the success of the transfer.

We also note that the recipient and donor applications do not need to implement the same functionality. Many of the errors occur in the code that parses the input, constructs the internal data structures that hold the input, and 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 files is often enough to enable a successful transfer.

1.3 Contributions

This paper makes the following contributions:

- **Basic Concept:** pDNA automatically eliminates software errors by identifying and transferring correct logic from donor systems into incorrect recipient systems. In this way pDNA can automatically harness the combined knowledge and labor invested across multiple systems to improve each system.
  To the best of our knowledge, pDNA is the first system to demonstrate that it is possible to automatically transfer logic between software systems to eliminate errors.
- **Logic Identification Technique:** pDNA identifies the correct donor logic to transfer into the recipient by analyzing two instrumented executions of the donor: 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 pDNA to isolate a single check (present in the donor but missing in the recipient) that enables systems to correctly process inputs that would otherwise trigger (usually fatal) errors.
- **Transfer Technique:** pDNA excises the check from the donor by expressing the check in a system-independent way as a function of the input bytes that determine the value of the check. It implants the check into the recipient by analyzing an instrumented execution of the recipient to discover program expressions that contain the required input values. Specifically, it uses the availability of these expressions to identify an appropriate check insertion point and translate the check into the name space of the recipient at that point. It then validates the transfer using regression testing and directed input space exploration to verify that there is no input that 1) satisfies the check and relevant enforced DIODE branch conditions but also 2) triggers the error.
- **Experimental Results:** We present experimental results that characterize the ability of pDNA to elim-

inate seven otherwise fatal errors in four recipient applications by transferring correct logic from three donor applications. For all of the 10 possible donor/recipient pairs, pDNA 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 pDNA eliminates an error in CWebp (with FEH as the donor). Section 3 discusses the pDNA 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 pDNA automatically patches an integer overflow error in CWebP, Google's 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: Starting with a seed input that CWebP processes correctly, pDNA uses the DIODE integer overflow discovery tool to obtain a related input that triggers the integer overflow error. DIODE first executes CWebP on the seed input. At each executed memory allocation site, the DIODE instrumentation records a symbolic expression for the size of the allocated memory. The variables in this symbolic expression are the values of the JPG input fields. The symbolic expressions therefore capture the complete computation that CWebP performs on the input fields to obtain the sizes of the allocated memory blocks.

DIODE next leverages branch conditions and the recorded symbolic expressions to efficiently search the input space to find an input that triggers an integer overflow at one (or more) of the memory allocation sites. In the error-triggering input in our example, the JPG `/start_frame/content/height` field is 62848 and the `/start_frame/content/width` field is 23200.

Donor Selection: pDNA next searches a database of applications that process JPG files to find candidate donor applications that successfully process both the seed and

```
1  int ReadJPEG(...) {
2     ...
3     dth     = 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

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

char load(...) {
    int    w, h;
    struct jpeg_decompress_struct cinfo;
    struct ImLib_JPEG_error_mgr jerr;
    FILE   *f;
    ...
    if (...) {
        ...
        im->w = w = cinfo.output_width;
        im->h = h = cinfo.output_height;
        /* Candidate check condition */
        if ((cinfo.rec_outbuf_height > 16) ||
              (cinfo.output_components <= 0) ||
              !IMAGE_DIMENSIONS_OK(w, h))
        {
            // Clean up and quit
            ...
            return 0;
        }
    }
}
```

Figure 2: (Simplified) FEH Overflow Check

the error-triggering inputs. In this example pDNA finds the FEH image viewer application. pDNA will attempt to find a check in FEH that eliminates the integer overflow, then transfer that check from FEH into CWebP to eliminate the overflow in CWebP.

Candidate Check Discovery: pDNA next runs an instrumented version of the FEH donor application on the seed and error-triggering inputs. At each conditional branch that is influenced by the relevant input field values (in this case the `/start_frame/content/height` and `/start_frame/content/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 are the values of input fields).

pDNA 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 seed input would satisfy the branch condition and the error-triggering input

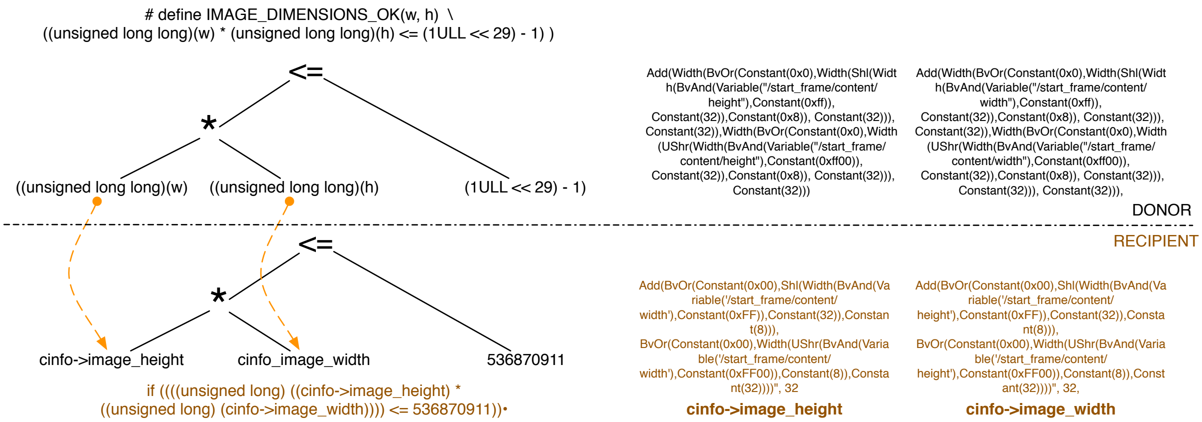

Figure 3: Patch Transfer

would not). pDNA therefore considers the condition at each branch at which the seed and error-triggering inputs take different directions to be a *candidate check condition*.

In our example, pDNA discovers a candidate check condition in the `imlib` library that FEH uses to load and process JPG files. Figure 2 presents the (simplified) source code for this condition.[1] The `IMAGE_DIMENSIONS_OK` macro (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.

pDNA next excises the candidate check condition from the donor by expressing the condition as a function of the input bytes that determine the value of the condition. This excision uses an instrumented execution of the donor that dynamically tracks the flow of input bytes through program to record the bytes that appear in the `output_width` and `output_height` variables. In our example the excised condition is as follows:

```
ULessEqual(Width(Mul(Width(BvOr(Width(Shl(Width(SRemainder
(BvOr(BvOr(Constant(0x00),Width(Sub(Add(Constant(8),
Shl(Add(BvOr(Constant(0x00),Shl(BvAnd(Variable('/start_frame/content/height'),
Constant(0xFF)),Constant(8))),BvOr(Constant(0x00),
UShr(BvAnd(Variable('/start_frame/content/height'),Constant(0xFF00)),
Constant(8)))),Constant(3))),Constant(1)),Constant(64))),
Shl(Width(SShr(Sub(Add(Constant(8),Shl(Add(BvOr(Constant(0x00),
Shl(Width(BvAnd(Variable('/start_frame/content/height'),
Constant(0xFF)),Constant(32)),Constant(8))),BvOr(Constant(0x00),
UShr(BvAnd(Variable('/start_frame/content/height'),Constant(0xFF00)),
Constant(8)))),Constant(3))),Constant(1)),Constant(31)),Constant(64)),
Constant(32))),Constant(8)),Constant(64)),Constant(32)),Constant(64)),
SDiv(BvOr(BvOr(Constant(0x00),Width(Sub(Add(Constant(8),
Shl(Add(BvOr(Constant(0x00),
Shl(BvAnd(Variable('/start_frame/content/height'),
Constant(0xFF)),Constant(8))),BvOr(Constant(0x00),
UShr(BvAnd(Variable('/start_frame/content/height'),Constant(0xFF00)),
Constant(8)))),Constant(3))),Constant(1)),Constant(64))),
Shl(Width(SShr(Sub(Add(Constant(8),
Shl(Add(BvOr(Constant(0x00),
Shl(BvAnd(Variable('/start_frame/content/height'),
Constant(0xFF)),Constant(8))),BvOr(Constant(0x00),
UShr(BvAnd(Variable('/start_frame/content/height'),Constant(0xFF00)),
Constant(8)))),Constant(3))),Constant(1)),Constant(31)),Constant(64)),
Constant(32))),Constant(8))),Constant(64)),
Width(BvOr(Width(Shl(Width(SRemainder(BvOr(BvOr(Constant(0x00),
Width(Sub(Add(Constant(8),Shl(Add(BvOr(Constant(0x00),
Shl(BvAnd(Variable('/start_frame/content/width'),Constant(0xFF)),
Constant(8))),BvOr(Constant(0x00),
UShr(BvAnd(Variable('/start_frame/content/width'),
Constant(0xFF00)),Constant(8)))),Constant(3))),Constant(1)),Constant(64))),
Shl(Width(SShr(Sub(Add(Constant(8),Shl(Add(BvOr(Constant(0x00),
Shl(BvAnd(Variable('/start_frame/content/width'),Constant(0xFF)),
Constant(8))),BvOr(Constant(0x00),
UShr(BvAnd(Variable('/start_frame/content/width'),Constant(0xFF00)),
Constant(8)))),Constant(3))),Constant(1)),Constant(31)),Constant(64)),
Constant(32))), Constant(8)),Constant(64)),Constant(32)),Constant(64)),
SDiv(BvOr(BvOr(Constant(0x00), Width(Sub(Add(Constant(8),Shl(Add(BvOr(Constant(0x00),
Shl(BvAnd(Variable('/start_frame/content/width'),Constant(0xFF)),Constant(8))),
BvOr(Constant(0x00),UShr(BvAnd(Variable('/start_frame/content/width'),
Constant(0xFF00)),Constant(8)))),Constant(3))),Constant(1)),
Constant(64))),Shl(Width(SShr(Sub(Add(Constant(8),
Shl(Add(BvOr(Constant(0x00),Shl(BvAnd(Variable('/start_frame/content/width'),
Constant(0xFF)),Constant(8))),BvOr(Constant(0x00),
UShr(BvAnd(Variable('/start_frame/content/width'),
Constant(0xFF00)),Constant(8)))),Constant(3))),Constant(1)),
Constant(31)),Constant(64)),Constant(32))),Constant(8))),
Constant(64))),Constant(64)),Constant(536870911))
```

There are two primary reasons for the complexity of this excised condition. 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. Second, FEH also casts the 16-bit input fields to long integers before it performs the overflow check. The excised condition correctly captures the shifts and masks that are performed as part of this conversion.

Patch Transfer: pDNA next attempts to transfer the candidate check condition from the donor FEH application to the recipient CWebP application, then use the transferred condition to insert a check into CWebP that eliminates the integer overflow error. Two key challenges are translating the condition into the name space of the CWebP application (i.e., expressing the condition in terms of the variables of the CWebP application) and finding a successful insertion point for the generated check.

pDNA runs CWebP (the recipient) on the seed input. After every assignment that reads a program expression that contains one of the input fields in the candidate check condition, the pDNA instrumentation computes the input field values that are available in CWebP program expressions at that point. If all of the input field values in the

[1] Because pDNA 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.

condition are available at a given point, pDNA can express the candidate check condition in terms of the available CWebP expressions (Figure 3 illustrates the translation). Each such point is a *candidate insertion point*.

pDNA iterates over the candidate insertion points (sorted by the CWebP execution order). At each point pDNA generates *a candidate patch* and attempts to validate the patch to determine if it 1) eliminates the error and 2) does not introduce a new error. The iteration continues until the patch validates.

For CWebP, pDNA identifies 16 candidate insertion points. The first point occurs in *jdmarker.c:267*, which is part of the jpeg-6b library. At this point pDNA (using the `cinfo− >image_height` and `cinfo− >image_width` expressions available in the CWepP source code at that point) generates the following patch:

```
if (!((((unsigned long) ((cinfo->image_height) *
      ((unsigned long) (cinfo->image_width))))
                       <= 536870911))) {
      exit(-1);
}
```

Note that pDNA was able to successfully convert the complex application-independent excised condition into this simple form — pDNA was able to detect that CWebP, even though developed independently, performs the same endianess conversion, shifts, and masks on the input values as FEH. pDNA therefore realizes that the input values are available in the same format in both the CWebP and FEH internal data structures, enabling pDNA to generate a simple patch that accesses the CWebP data structures directly with no complex format conversion. The generated patch checks the candidate check condition and, if the condition is true, exits the application. The rationale is to exit the application before the integer overflow (and any ensuing error or vulnerabilities) can occur.

Figure 4, lines 14-18, shows where pDNA inserts the generated patch into CWebP. A quick inspection of the surrounding code, which also performs a number of input checks, indicates that pDNA selected an appropriate patch insertion point.

Patch Validation: Finally, pDNA 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 (pDNA 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 pre-selected set of inputs that the application is known to process correctly.

```
1   LOCAL(boolean)
2   get_sof (j_decompress_ptr cinfo, ...) {
3     ...
4     // Existing sanity checks
5     if (cinfo->image_height <= 0 ||
6           cinfo->image_width <= 0 ||
7           cinfo->num_components <= 0)
8       ERREXIT(cinfo, JERR_EMPTY_IMAGE);
9
10    if (length != (cinfo->num_components * 3))
11      ERREXIT(cinfo, JERR_BAD_LENGTH);
12    ...
13    /* pDNA transfered patch */
14    if (!((((unsigned long) ((cinfo->image_height) *
15        ((unsigned long) (cinfo->image_width))))
16                 <= 536870911))) {
17        exit(-1);
18    }
19    ...
20    return TRUE;
21  }
```

Figure 4: Transfered Patch In CWebP (from FEH)

Fourth, pDNA runs the patched version of the application through the DIODE error discovery tool to ensure that no more error-triggering inputs can be generated. The end result, in this example, is a version of CWebP that contains a check that eliminates the integer overflow error in the original version.

3 Design and Implementation

We next discuss how pDNA deals with the many technical issues it must overcome to successfully generate source-level patches for discovered errors. pDNA 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 5 presents an overview of the pDNA components. First, we describe our techniques for error discovery. Second, we describe our methodology for selecting donors. Third, we describe our techniques for selecting candidate checks from donor applications. Fourth, we describe our patch transfer algorithms. Finally, we discuss our techniques for patch validation.

3.1 Error Discovery

pDNA uses DIODE [1], a tool that we have previously developed, to automatically generate inputs that trigger integer overflows at memory allocation sites. DIODE is designed to identify relevant checks that inputs must satisfy to trigger overflows at target memory allocation sites, then generate inputs that satisfy these checks to successfully trigger the overflow.

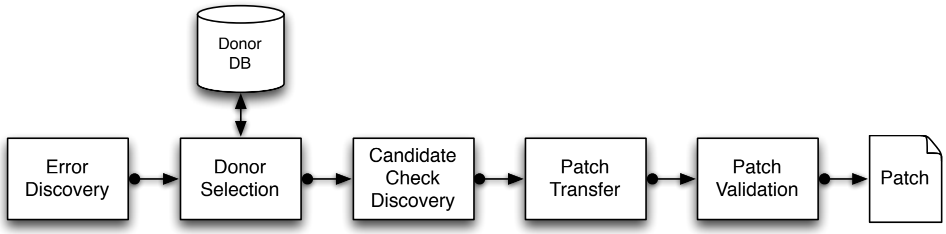

Figure 5: High-level overview of pDNA's components

Starting with a seed input that causes one or more target memory allocation sites to execute, DIODE performs the following steps:

- **Target Allocation Site Identification:** Using a fine-grained dynamic taint analysis on the application running on the seed input, DIODE identifies all memory allocation sites that are influenced by values from the seed input. These sites are the *target sites*.
- **Target Constraint Extraction:** Based on instrumented executions of the application, DIODE extracts a symbolic target expression that characterizes how the application computes the target value (the size of the allocated memory block) at each target memory allocation site from input values. The input bytes that influence this expression are the *relevant input bytes*. Using the target expression, DIODE generates a target constraint that characterizes all inputs that would cause the computation of the target value to overflow (as long as the input also causes the application to compute the target value).
- **Branch Constraint Extraction:** Again based on instrumented executions of the application, DIODE extracts the sequence of conditional branch instructions that the application executes to generate the path to the target memory allocation site.
  To ensure that DIODE productively considers only relevant conditional branches, DIODE discards 1) all conditional branches whose condition is not influenced by relevant input bytes and 2) all conditional branches that implement loop back edges.
  For each remaining conditional branch, DIODE generates a *branch constraint* that characterizes all input values that cause the execution to take the same path at that branch as the seed input. DIODE will use these branch constraints to generate candidate test inputs that force the application to follow the same path as the seed input at selected conditional branches.
- **Target Constraint Solution:** DIODE invokes the Z3 SMT solver [15] to obtain input values that satisfy the target constraint. If the application follows a path that evaluates the target expression at the target memory allocation site, DIODE has successfully generated an input that triggers the overflow. If the application performs no checks on the generated values, this step typically delivers an input that triggers the overflow.
- **Goal-Directed Conditional Branch Enforcement:** If the previous step failed to deliver an input that triggers an overflow, DIODE compares the path that the seed input followed with the path that the generated input followed. These two paths must differ (otherwise the generated input would have triggered an overflow).
  DIODE then finds the first (in the execution order) relevant conditional branch where the two paths diverge (i.e., where the generated input takes a different path than the seed input). We call this conditional branch the *first flipped branch*.
  DIODE adds the branch constraint from the first flipped branch to the constraint that it passes to the solver, forcing the solver to generate a new input that takes the same path as the seed input at the first flipped branch. DIODE then runs the application on this new generated input to see if it triggers the overflow.
  DIODE continues this goal-directed branch enforcement algorithm, incrementally adding the branch constraints from first flipped branches, until either 1) it generates an input that triggers the overflow or 2) it generates an unsatisfiable constraint.

3.2 Donor Selection

For each input file format, pDNA works with a set of applications that process that format. Note that the donor and recipient applications do not have to implement identical functionality — many of the errors that pDNA eliminates occur in the initial input processing phase. Given seed and error-triggering inputs, pDNA considers applications that can successfully process both inputs as potential donors.

3.3 Candidate Check Discovery

To extract candidate checks from donor applications, pDNA contains a fine-grained dynamic taint analysis built on top of the Valgrind [29] binary analysis framework. Our analysis takes as input a specified taint source, such as a filename 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 until it reaches a potential sink in the target application (e.g., branch conditions and memory allocation sites). To track the data-flow dependencies from source to sink, our analysis instruments arithmetic instructions (e.g., ADD, SUB), data movement instructions (e.g., MOV, PUSH) and logic instructions (e.g., AND, XOR). Our analysis also supports additional instrumentation to reconstruct the full symbolic expression of the value at a sink, which represents how the application computes the value from input bytes.

Identify Candidate Check: pDNA runs the dynamic taint analysis on the donor application twice, once with a seed input and once with the bug-triggering input that DIODE generates from the seed input. For each execution, pDNA extracts the conditional branch statements in the execution path that relevant input bytes influence. For each such branch statement, pDNA records which branch direction the execution takes. pDNA then compares the two execution paths to find the flipped conditional branch statements that cause the two executions diverge.

pDNA empirically transfers the condition of the first flipped branch statement into the recipient application. We call the condition of the first flipped branch statement the *candidate check*. If the generated patch does not pass the validation (see Section 3.5), pDNA will transfer the second flipped branch statement to generate a new patch, etc.,.

Generate Target Symbolic Condition: Next, pDNA reruns the application with additional instrumentation that enables pDNA to reconstruct the full target symbolic condition for the candidate check, which characterizes how the donor application computes the condition of the candidate check from the input byte values. Conceptually, pDNA generates a symbolic record of all calculations that the application performs. Obviously, attempting to record all calculations would produce an unmanageable volume of information. pDNA reduces the volume of recorded information with the following optimizations:

- **Relevant Input Bytes:** pDNA only records calculations that involve the relevant input bytes. Specifically, pDNA maintains an expression tree of relevant calculations that only tracks calculations that operate on tainted data (i.e., relevant input bytes). This optimization drastically reduces the amount of recorded information.
- **Simplify Expressions:** pDNA further reduces the amount of recorded information by simplifying recorded expressions at runtime. Specifically, pDNA identifies and simplifies resize, move and arithmetic operations. For example, pDNA can convert the following sequence of VEX IR instructions:

```
t15 = Add32(t10, 0x1:I32)
t16 = Add32(t15,0x1:I32)
t17 = Add32(t16,0x1:I32)
```

that would result in:

```
Add32(Add32(Add32(t10, 0x1), 0x1),0x1)
```

into:

```
Add32(t10, 0x3)
```

To convert relevant input bytes to symbolic representations of the input format, pDNA uses the Hachoir [6] tool to convert byte ranges into input fields (e.g., in the PNG format, bytes 0-3 represent /header/height). If Hachoir does not support a particular input format or is otherwise unable to perform this conversion, pDNA also supports a raw mode in which all input bytes are represented as offsets.

3.4 Patch Transfer

Next, pDNA determines if the symbolic representation of the candidate check can eliminate the error from the recipient. In other words, pDNA verifies that the target constraint solution and relevant branches generated by DIODE, along with the constraints introduced by the candidate check, can no longer be used to generate an input that can cause an integer overflow.

To transfer the candidate check to an insertion point in the recipient application, pDNA rewrites the target symbolic condition with active variables at the insertion point. Therefore, pDNA first needs to track how a recipient application computes the values of program variables that are derived from input bytes.

Specifically, pDNA performs its dynamic taint analysis on the recipient application with the bug-triggering input. For each variable assignment statement that involves relevant input bytes, the analysis records the symbolic expression of the assigned value, which characterizes how the recipient application computes the value from the input bytes.

If all of the required input bytes are available in program expressions after the assignment, pDNA currently considers the program point after each variable assignment statement that involves relevant input bytes to be a candidate check insertion point. For each such insertion point, pDNA identifies active program variables at the insertion point that pDNA can use to construct the patch. pDNA then invokes a rewrite algorithm to synthesize the patch.

Figure 6 presents pDNA's expression rewrite algorithm. The algorithm takes as input a symbolic expression E and a set of variables *Vars* as inputs and rewrites the expression E using variables in *Vars*. The key insight behind the rewrite algorithm is that the synthesized condition in the recipient application should be semantically equivalent to the candidate check in the donor application at least on the error-triggering input. Therefore the symbolic representation of the synthesized patch condition should match the target symbolic condition pDNA obtains using the dynamic analysis on the donor application.

Constant expressions (lines 12-14) are directly used and do not require a rewrite pass. Next, the algorithm attempts to find a single variable to represent the whole expression (lines 15-21). If unsuccessful, the algorithm decomposes the expression and attempts to rewrite each subexpression recursively (lines 22-27 for expressions with unary operations, lines 28-36 for expressions with binary operations).

Note that at line 16, the algorithm queries the SMT solver to determine whether two symbolic expressions are equivalent. The pDNA implementation has two optimizations to reduce the number of invocations to the solver. First, if two symbolic expressions depend on different sets of input bytes, pDNA does not need to invoke the solver because these two expressions cannot be equivalent. Second, pDNA caches all queries to the SMT solver so that it can retrieve results from the cache for future duplicate queries.

For each insertion point in the recipient that the rewrite algorithm successfully constructs the new condition, pDNA generates a *candidate patch* as an if statement inserted at the insertion point. In the current implementation, pDNA transforms the constructed bitvector condition into a C expression as the if condition (appropriately generating the 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)`.

3.5 Patch Validation

pDNA 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. pDNA 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. pDNA finally runs DIODE on the patched recipient application with the seed input. This validates that after the recipient application is patched, DIODE is not able to find another input that triggers the same error. In other words, pDNA validates that there is no input that satisfies the patch condition and the relevant branch conditions that DIODE generates while also triggering an overflow at the target allocation site.

```
1  Parameters:
2    E: A symbolic expression
3    Vars: A set of active variables
4    For each V in Vars, V.var is the variable
5  name; V.exp is the symbolic expression that
6  corresponds to the value of the variable.
7  Return:
8    Rewriten expression of E or
9    false if failed
10
11 Rewrite(E, Vars) {
12     if (E is constant)
13         return E
14     end if
15     for V in Vars
16         if (SolverEquiv(E,V.exp))
17             Ret.opcode ← VAR
18             Ret.op1 ← V.var
19             return Ret;
20         end if
21     end for
22     if (E.opcode is unary operation)
23         Ret.opcode ← E.opcode
24         Ret.op1 ← Rewrite(E.op1, Vars)
25         if (Ret.op1 != false)
26             return Ret
27         end if
28     else if (E.opcode is binary operation)
29         Ret.opcode ← E.opcode
30         Ret.op1 ← Rewrite(E.op1, Vars)
31         Ret.op2 ← Rewrite(E.op2, Vars)
32         if (Ret.op1 ≠ false and
33             Ret.op2 ≠ false)
34             return Ret
35         end if
36     end if
37     return false
38 }
```

Figure 6: pDNA Rewrite algorithm

4 Experimental Results

We evaluate pDNA on seven integer overflow errors that DIODE previously detected in four applications: CWebP 0.31 [2], Dillo 2.1 [3], swfplay 0.55 [9], and Display 6.5.2-8 [7]. Two of these errors were listed in the CVE database; one was first discovered by BuzzFuzz [17]; the other four were, to the best of our knowledge, first discovered by DIODE. The errors are triggered by JPG image files (CWebP), PNG image files (Dillo), SWF video files (swfplay), and TIFF image files (Display). For JPG and PNG files our set of donor applications includes FEH-2.9.3 [4] and mtpaint 3.4 [8]. For TIFF files our donor application is ViewNoir 1.4 [10]. For SWF our donor application is gnash 0.8.11 [5].

For each error we started with seed and corresponding error-triggering inputs previously identified by DIODE. We then deployed pDNA in an attempt to generate validated patches to eliminate each of the errors. Figure 7 summarizes the results of these experiments. There is a row in the table for each combination of error and donor application. The first column (Application) identifies the application. The second column (Target) identifies the source code file and line where the error occurs. The third

Application	Target	Error	Format	Donor App	Patch	Generation Time	# Candidate Checks	# Insertion Points
CWebP 0.3.1	jpegdec.c:248	New	jpeg	feh-2.9.3	✓	13m	7	214
CWebP 0.3.1	jpegdec.c:248	New	jpeg	mtpaint-3.40	✓	7m	16	214
Dillo 2.1	png.c@203	CVE-2009-2294	png	mtpaint-3.40	✓	10m	2	167
Dillo 2.1	png.c@203	CVE-2009-2294	png	feh-2.9.3	✓	13m	5	167
Dillo 2.1	fltkimagebuf.cc@39	New	png	mtpaint-3.40	✓	10m	2	167
Dillo 2.1	fltkimagebuf.cc@39	New	png	feh-2.9.3	✓	13m	5	167
Display 6.5.2	xwindow.c@5619	CVE-2009-1882	tiff	viewnior-1.4	✓	1h	4328	148
Display 6.5.2	cache.c@803	New	tiff	viewnior-1.4	✓	1h	24	148
SwfPlay 0.5.5	jpeg_rgb_decoder.c@253	New	swf	gnash	✓	45m	45	120
SwfPlay 0.5.5	jpeg.c@192	BuzzFuzz [17]	swf	gnash	✓	14m	27	222

Figure 7: pDNA Experimental Results

column (Error) presents either the CVE identifier (if the error was previously known) or new (if the error was first discovered by DIODE). The fourth column identifies the input file format. The fifth column identifies the donor application. The sixth column indicates (with a check mark) if pDNA was able to generate a validated patch for that recipient/donor pair (pDNA succeeded for all pairs). The seventh column presents the amount of time pDNA required to generate and validate the patch.

The eighth column presents the number of candidate checks that pDNA found in the donor. To improve the efficiency of the search, our current pDNA implementation uses the DIODE target overflow constraint from the allocation site, the conditions on the branches the DIODE enforced, and the patch condition to check if any input can simultaneously satisfy all of these conditions. If so, there may be an input that can satisfy the check and still cause an overflow. In this case pDNA immediately filters the candidate check and moves on to the next check. For all of our benchmark errors, the first candidate check that passes this DIODE test eventually validates.

The ninth column presents the number of insertion points that pDNA found in the recipient. For all of our benchmark errors, the first insertion point validates as expected.

4.1 Dillo

Dillo is a lightweight graphical web browser. Dillo 2.1 is vulnerable to an integer overflow when decoding the PNG file format. Dillo computes the size as a 32-bit product of width, height, and pixel depth. An overflow check is present, but the overflow check is itself vulnerable to an overflow. When the buffer size calculation overflows, the allocation at png.c line 203 returns a buffer that is too small to hold the decompressed image (CVE-2009-2294). Both FEH and mtpaint are successful donors for this error. The transferred check appears in FEH as a subexpression generated as part of the following macro invocation:[2]

```
if(!IMAGE\_DIMENSIONS\_OK(w32, h32))
```

After the transfer, the check appears in Dillo (`libpng-1.2.50/pngrutil.c:497`) as:

```
if (!((((unsigned int) (((unsigned int) (((unsigned int)
    ((unsigned int)
    ((width) * 0))) + ((unsigned int) ((unsigned int)
    ((unsigned int) ((height) *0)))))) + ((unsigned int)
    ((unsigned int) ((unsigned int) (((unsigned long
    long) ((height) * ((unsigned long long) (width))))
    >> 32)))))) <= 0)))
{exit(-1);}
```

In this patch the repeated casts to unsigned int and unsigned long long are required to correctly reflect the varying binary representations at which the FEH binary performs the check. The patch eliminates the error by checking that the `width` and `height` values will never generate an overflow. pDNA inserted the patch at `libpng-1.2.50/pngrutil.c:497`.

The mtpaint patch uses the following check:

```
if ((pwidth > MAX\_WIDTH) ||
      (pheight > MAX\_HEIGHT))
```

where `MAX_WIDTH` is equal to 16384. This check generates the following patch:

```
if (!(((i) <= 16384))) {exit(-1);}
```

which pDNA inserts into `libpng-1.2.50/pngrutil.c:65`. Two things are of interest. First, the patch checks only the width field, but this check is enough to eliminate the overflow. Second, the check constrains the width to be small enough (no greater than 16384) so that Dillo may reject some valid input files. But this is consistent with the behavior of the mtpaint donor, which will also reject these same input files.

We note that Dillo 2.1 has an additional overflow vulnerability after the initial allocation. The same function initializes a cache for the image starting at png.c line 212, which leads to an allocation inside the FLTK library at fltkimgbuf.cc line 62 which computes a buffer size as a product of improperly checked variables. If the calculation of the buffer size overflows, the write of the image into the cache will overrun the allocated space. Because the buffer size computation involves the same `width` and `height` values, the previous patches also eliminate this error.

[2] Because pDNA operates on binaries, information about the source code for the donor patch is, in general, not available. To present the source code for the checks in this section, we used the symbolic debugging information in the binary (when available) to locate this source code.

4.2 Display

ImageMagick Display is an image viewing and formatting utility released as part of the ImageMagick suite. Display 6.5.2 is vulnerable to an integer overflow for TIFF files. Display computes the length in bytes needed for a pixel buffer as a product of several values from the input file such as width, height, and bytes per pixel. With no overflow checking at all in this version, this length calculation easily overflows its 32-bit size, resulting in an incorrect size passed to malloc at xwindow.c line 5619 (CVE-2009-1882).

pDNA successfully created a patch for this error using viewnior as the donor application. The transfered check appears in viewnoir as:

```
bytes = height * rowstride;
if (bytes / rowstride != height)
```

This check was translated into the following patch for Display (`cache.c:2056`) as:

```
if (!(((image->rows) == ((unsigned int) (((unsigned
long long) ((unsigned long long) (((unsigned long long)
(((unsigned int) (((unsigned int) ((unsigned int) (0 |
((unsigned long long) ((image->rows) * ((unsigned long
long) ((unsigned int) ((unsigned int) ((image->columns)
<< 2))))))))) | ((unsigned int) ((unsigned int)
((unsigned int) (((unsigned long long) ((unsigned long
long) (((unsigned int) ((image->rows) * ((unsigned int)
((unsigned int) ((unsigned int) ((image->columns) <<
2)))))) >> 31))) << 32))))))
% ((unsigned long long) ((unsigned int) ((unsigned
int) ((image->columns) << 2)))))) << 32))) | ((unsigned
int) ((unsigned int) (((unsigned int) (((unsigned int)
((unsigned int) (0 | ((unsigned long long)
((image->rows) * ((unsigned long long) ((unsigned int)
((unsigned int) ((image->columns) << 2))))))))) |
((unsigned int) ((unsigned int) ((unsigned int)
(((unsigned long long) ((unsigned long long)
(((unsigned int) ((image->rows) * ((unsigned int)
((unsigned int) ((unsigned int) ((image->columns) <<
2)))))) >> 31))) << 32)))))) / ((unsigned int)
((unsigned int) ((unsigned int) ((image->columns) <<
2)))))))))))) {exit(-1);}
```

The multiple casts, shifts, and mask operations are required to correctly reflect the different integer representations at which the viewnoir binary performs the check. This patch eliminates the error by performing an overflow check on `height`, `width`, and the number of `columns` (used to compute `rowstride`)

Display also contains overflow errors when creating a resized version of the image for display within the GUI window (starting at display.c line 4393), and when creating a cache buffer for the image during TIFF decompression (a request for pixel space at tiff.c line 1044 eventually results in an allocation at cache.c line 3717). When the computation of any of these buffer sizes overflows, the allocated memory blocks are too small, causing Display to write beyond the end of the block.

pDNA generated a patch for this error, again using viewnior as the donor. The transfered check appears in viewnoir as:

```
rowstride = width * 4;
if (rowstride / 4 != width)
```

pDNA transfers this check into Display as (`tif_dirread.c:400`):

```
if (!((((unsigned int) ((*(value)) | ((unsigned int)
((unsigned int) ((unsigned int) (((unsigned int)
((unsigned int) (((unsigned int) ((unsigned int) ((m) &
65280))) >> 8))) << 8)))))) == ((unsigned int)
(((unsigned long long) ((unsigned long long)
(((unsigned long long) (((unsigned int) (((unsigned
int) ((unsigned int) (0 | ((unsigned long long)
(((unsigned int) ((*(value)) | ((unsigned int)
((unsigned int) ((unsigned int) (((unsigned int)
((unsigned int) (((unsigned int) ((unsigned int) ((m) &
65280))) >> 8))) << 8)))))) * ((unsigned long long)
((unsigned int) (( unsigned int) (((unsigned int)
((*(value)) | ((unsigned int) ((unsigned int)
((unsigned int) (((unsigned int) ((unsigned int)
(((unsigned int) ((unsigned int) ((m) & 65280))) >>
8))) << 8)))))) << 2))))))))) | ((unsigned int)
((unsigned int) ((unsigned int) (((unsigned long long)
((uns igned long long) (((unsigned int) (((unsigned
int) ((*(value)) | ((unsigned int) ((unsigned int)
((unsigned int) (((unsigned int) ((unsigned int)
(((unsigned int) ((unsigned int) ((m) & 65280))) >>
8))) << 8)))))) * ((unsigned int) ((unsigned int)
((unsigned int) (((unsigned int) ((*(value)) |
((unsigned int) ((unsigned int) ((unsigned int)
(((unsigned int) ((unsigned int) (((unsigned int)
((unsigned int) ((m) & 65280))) >> 8))) << 8)))))) <<
2)))))) >> 31))) << 32)))))) % ((unsigned long long)
((unsigned int) ((unsigned int) (((unsigned int)
((*(value)) | ((unsigned int) ((unsigned int)
((unsigned int) (((unsigned int) ((unsigned int)
(((unsigned int) ((unsigned int) ((m) & 65280))) >>
8))) << 8)))))) << 2)))))) << 32))) | ((unsigned int)
((unsigned int) (((unsigned int) (((unsigned int)
((unsigned int) (0 | ((unsigned long long) (((unsigned
int) ((*(value)) | ((unsigned int) ((unsigned int)
((unsigned int) (((unsigned int) ((unsigned int)
(((unsigned int) ((unsigned int) ((m) & 65280))) >>
8))) << 8)))))) * ((unsigned long long) ((unsigned int)
((unsigned int) (((unsigned int) ((*(value)) |
((unsigned int) ((unsigned int) (( unsigned int)
(((unsigned int) ((unsigned int) (((unsigned int)
((unsigned int) ((m) & 65280))) >> 8))) << 8)))))) <<
2))))))))) | ((unsigned int) ((unsigned int) ((unsigned
int) (((unsigned long long) ((unsigned long long)
(((unsigned int) (((unsigned int) ((*(value)) |
((unsigned int) ((unsigned int) ((unsigned int)
(((unsigned int) ((unsigned int) (((unsigned int)
((unsigned int) ((m) & 65280))) >> 8))) << 8)))))) *
((unsigned int) ((unsigned int) ((unsigned int)
(((unsigned int) ((*(value)) | ((unsigned int)
((unsigned int) ((unsigned int) (((unsigned int) ((uns
igned int) (((unsigned int) ((unsigned int) ((m) &
65280))) >> 8))) << 8)))))) << 2)))))) >> 31))) <<
32)))))) / ((unsigned int) ((unsigned int) ((unsigned
int) (((unsigned int) ((*(value)) | ((unsigned int)
((unsigned int) ((unsigned int) (((unsigned int)
((unsigned int) (((unsigned int) ((unsigned int) ((m)
& 65280))) >> 8))) << 8)))))) << 2))))))))))))
{exit(-1);}
```

This patch successfully protects against the integer overflow error with the added overflow check on `width * 4`. Once again, the patch reflects the conversion of the analyzed viewnoir VEX binary operations into C source code.

4.3 Swfplay

Swfplay is an Adobe Flash player that is released as part of the open source Swfdec library. Swfplay 0.5.5 is vulnerable to an integer overflow for SWF files when decoding embedded JPEG data. When initially allocating buffers for the individual YUVA components of the image, swfplay computes the buffer size for each component buffer as the 32-bit product of width, height, and various sampling factors without sufficient overflow checking (jpeg.c line 192). If the computation overflows, then the decompression procedure will write beyond the allocated space. Even if the computations of individual component buffer sizes do not overflow, there is a potential overflow when merging the individual YUVA components of the image into a single RGBA buffer. Swfplay computes the size of the combined buffer as a 32-bit product of width, height and 4 without performing any overflow checking. This computation is used twice in close succession: once for the allocation of a temporary buffer (jpeg_rgb_decoder.c line 253), and then for the allocation of the image buffer (jpeg_rgb_decoder.c line 257). When this computation overflows, the merge procedure will write beyond the allocated space and ultimately result in a SIGSEGV on an invalid write. pDNA generated a patch for this error, again using Gnash as the donor. Because symbolic information that would allow us to locate the Gnash source code for this patch is not available, we present only the patch in the swfplay recipient:

```
if (!(((image->height) <= 65500))) {exit(-1);}
```

This patch protects the application by limiting height to a 16 bit value, which when used in the product of width, height, and a small constant, cannot generate an overflow on 32 bit machines.

For the error at (jpeg_rgb_decoder.c line 253), pDNA generates the following patch at jpeg_bits.c line 60:

```
if (!((((unsigned int) (((unsigned long long)
((unsigned long long) (((unsigned long long) ((unsigned
long long) (((unsigned int) (((unsigned int) ((unsigned
int) (0 | ((unsigned long long) ((unsigned long long)
(8 * ((unsigned int) ((unsigned int) (((unsigned int)
((unsigned int) (0 | (*(b->ptr))))) & 15)))))))))) |
((unsigned int) ((unsigned int) ((unsigned int)
(((unsigned long long) ((unsigned long long)
(((unsigned int) ((unsigned int) (8 * ((unsigned int)
((unsigned int) (((unsigned int) ((unsigned int) (0 |
(*(b->ptr))))) & 15))))))) >> 31))) << 32)))))) % 8)))
<< 32))) | ((unsigned int) ((unsigned int) ((unsigned
int) (((unsigned int) (((unsigned int) ((unsigned int)
(0 | ((unsigned long long) ((unsigned long long) (8 *
((unsigned int) ((unsigned int) (((unsigned int)
((unsigned int) (0 | (*(b->ptr))))) & 15)))))))))) |
((unsigned int) ((unsigned int) ((unsigned int)
(((unsigned long long) ((unsigned long long)
(((unsigned int) ((unsigned int) (8 * ((unsigned int)
((unsigned int) (((unsigned int) ((unsigned int) (0 |
(*(b->ptr))))) & 15))))))) >> 31))) << 32)))))) /
8)))))) == ((unsigned int) ((unsigned int) (((unsigned
int) ((unsigned int) (0 | (*(b->ptr))))) & 15))))))
{exit(-1);}
```

5 Related Work

We discuss related work in program repair (static and dynamic), N-version programming, and horizontal gene transfer.

Static Program Repair: GenProg [37, 23] is an automatic program repair tool that uses a genetic algorithm to synthesize program patches. GenProg first copies an existing code snippet from another location in the program, then randomly applies a set of mutation rules based on the genetic algorithm in an attempt to find a patch that generates correct results on a set of sample inputs. pDNA, in constrast, eliminates errors by transferring correct code across multiple applications (including stripped binary donor applications).

PAR [20] is a program repair tool that applies a set of ten predefined repair templates that the authors manually summarized from legacy human-written patches. These templates correspond to the structures of common human patches (e.g, inserting null checker, adding a method call, inserting a bound check, etc.). PAR uses a search algorithm to fill in details in the templates (e.g., the variable to be checked, the method to be called.)

In contrast, pDNA transfers correct checks across applications. Instead of random mutations, pDNA uses dynamic analysis techniques to obtain an application-independent representation of the check, then implant the check into the recipient at an appropriate insertion point where the required values are available in program expressions.

Khmelevsky et al. [19] 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 [24] 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, pDNA is fully automatic — it does not require any human annotations to transfer patches from the donor application to the recipient application.

SJava [16] 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 [32] enables an application to survive common memory error. It recompiles the application to discard out of bounds writes, manufacture values for out of bounds reads, and enable the application to continue along their normal execution path. RCV [27] enables an application to recover from divide-by-zero and null-dereference errors on the fly. When such an error occurs, RCV attaches the application, applies 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 [28] 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 backs to normal mode once the server moves to the next request.

Jolt [12] and Bolt [21] enable applications to survive infinite loop errors. When such an error occurs, they control the execution of the application to jump out of the loop or the enclosing function to escape the error.

ClearView [30] 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.

Rx [31] and ARMOR [13] are runtime recovery systems based on periodic checkpoints. When an error occurs, Rx [31] 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 [13] reverts back to a previous checkpoint and finds semantically equivalent workarounds for the failed component based on user-provided specifications.

Error Virtualization [33, 34, 36, 35] 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 from unanticipated faults.

Input rectification [25] empirically learns input constraints from benign training inputs and then enforces learned constraints on incoming inputs to nullify potential errors. SIFT [26] 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, pDNA is designed to transfer correct code from donors to recipients to directly eliminate the error. The final patched application then executes with no dynamic instrumentation overhead.

N-Version Programming: N-version programming [14] 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 [22].

Rather than running multiple versions and comparing the results, pDNA transfers correct logic to obtain a single improved hybrid system. In comparison with traditional N-version programming, pDNA therefore 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, pDNA is 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 pDNA to find and transfer correct error checks.

Horizontal Gene Transfer: Horizontal gene transfer is the transfer of genetic material between individual cells [11]. Examples include plasmid transfer (which plays a major role in acquired antibiotic resistance [11]) and virally-mediated gene therapy [18]. There are strong analogies between pDNA's logic transfer mechanism and horizontal gene transfer — in both cases functionality is transferred from a donor to a recipient, with significant potential benefits to the recipient. The fact that horizontal gene transfer is recognized as significant factor in the evolution of many forms of life hints at the potential that multi-application code transfer may offer for software systems.

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 logic between systems to eliminate errors. The system that implements this technique, pDNA, 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.

References

[1] Anonymized reference.

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

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

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

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

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

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

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

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

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

[11] M. Barlow. What Antimicrobial Resistance Has Taught Us About Horizontal Gene Transfer. *Methods in Molecular Biology*, 532:397–411, 2009.

[12] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard. Detecting and escaping infinite loops with jolt. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 609–633. Springer-Verlag, 2011.

[13] 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.

[14] 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.

[15] L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] 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.

[17] 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.

[18] 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.

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

[20] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13', pages 802–811. IEEE Press, 2013.

[21] 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.

[22] 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.

[23] 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. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 3–13. IEEE Press, 2012.

[24] 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*, OOP-

SLA '12', pages 133–146, New York, NY, USA, 2012. ACM.

[25] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard. Automatic input rectification. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 80–90. IEEE Press, 2012.

[26] 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.

[27] 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.

[28] V. Nagarajan, D. Jeffrey, and R. Gupta. Self-recovery in server programs. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09', pages 49–58. ACM, 2009.

[29] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07. ACM, 2007.

[30] 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. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[31] 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.*, 25(3), Aug. 2007.

[32] 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.

[33] S. Sidiroglou, Y. Giovanidis, and A. Keromytis. A Dynamic Mechanism for Recovery from Buffer Overflow attacks. In *Proceedings of the 8th Information Security Conference (ISC)*, September 2005.

[34] S. Sidiroglou and A. D. Keromytis. A Network Worm Vaccine Architecture. In *Proceedings of the IEEE Workshop on Enterprise Technologies*, June 2003.

[35] 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.

[36] 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.

[37] 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', pages 364–374. IEEE Computer Society, 2009.