MIT Libraries | DSpace@MIT

MIT Open Access Articles

Targeted Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Sidiroglou-Douskos, Stelios et al. "Targeted Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement." Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015) (March 2015): 473–486.

As Published: http://dx.doi.org/10.1145/2694344.2694389

Publisher: Association for Computing Machinery (ACM)

Persistent URL: http://hdl.handle.net/1721.1/96155

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike

Massachusetts Institute of Technology

Targeted Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement

Stelios Sidiroglou-Douskos Eric Lahtinen Nathan Rittenhouse Paolo Piselli Fan Long
Deokhwan Kim Martin Rinard

MIT CSAIL

{stelios,elahtinen,nathan_,ppiselli,fanl,dkim,rinard}@csail.mit.edu

Abstract

We present a new technique and system, DIODE, for automatically generating inputs that trigger overflows at memory allocation sites. DIODE is designed to identify relevant sanity checks that inputs must satisfy to trigger overflows at target memory allocation sites, then generate inputs that satisfy these sanity checks to successfully trigger the overflow.

DIODE works with off-the-shelf, production x86 binaries. Our results show that, for our benchmark set of applications, and for every target memory allocation site exercised by our seed inputs (which the applications process correctly with no overflows), either 1) DIODE is able to generate an input that triggers an overflow at that site or 2) there is no input that would trigger an overflow for the observed target expression at that site.

1. Introduction

Integer overflow errors are an insidious source of software failures and security vulnerabilities [1, 14, 41]. Because programs with latent overflow errors often process typical inputs correctly, such errors can easily escape detection during testing only to appear later in production. Overflow errors that occur at memory allocation sites can be especially problematic as they comprise a prime target for code injection attacks. A typical scenario is that a malicious input exploits the overflow to cause the program to allocate a memory block that is too small to hold the data that the program will write into the allocated block. The resulting out-of-bounds writes can easily enable code injection attacks [14].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '15', March 14–18, 2015, Istanbul, Turkey..
Copyright © 2015 ACM 978-1-4503-2835-7/15/03. . . $15.00.
http://dx.doi.org/10.1145/10.1145/2694344.2694389

1.1 DIODE

We present a new technique and system, DIODE (Directed Integer Overflow Discovery Engine), for automatically generating inputs that trigger integer overflow errors at critical sites. DIODE starts with a *target site* (such as a memory allocation site) and a *target value* (such as the size of the allocated memory block). It then uses symbolic execution to obtain an *target expression* that characterizes how the program computes the target value as a function of the input. It then transforms the target expression to obtain a *target constraint*. If the input 1) satisfies the target constraint while 2) causing the program to execute the target site, then it will trigger the error.

Sanity Checks: A key observation behind the design of DIODE is that programs often perform sanity checks on the input before they use the input to compute target values. If the input does not pass the sanity checks, the program typically emits an error or warning message and does not further process the input. To trigger an overflow, an input must therefore take the *same path* through the sanity checks as typical inputs that the program processes successfully.

One obvious way to obtain an input that satisfies the sanity checks is start with a *seed input* that causes one or more target sites to execute, then use a solver to obtain a new input that 1) satisfies the target constraint as well as 2) additional constraints that force the solver to generate an input that takes the *same path* to the target site as the seed input. This approach ensures that the input passes the sanity checks.

Blocking Checks: Unfortunately, our results indicate that this approach often fails because, in most cases, the path that the seed input takes through the computation contains additional *blocking checks* that prevent any input that satisfies these checks from triggering the error. To trigger an overflow, an input must take a *different path* through these blocking checks. The challenge is therefore to find inputs that 1) satisfy the target constraint, 2) satisfy the sanity checks, and 3) find a path through the blocking checks to execute the target site. DIODE meets this challenge as follows:

- **Target Site Identification:** Using a fine-grained dynamic taint analysis on the program running on the seed input,

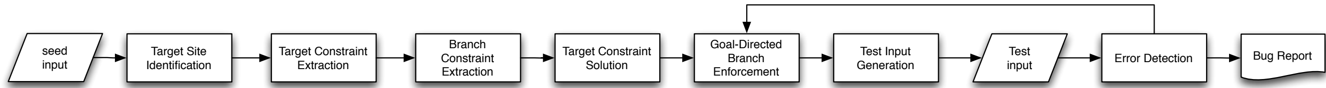

Figure 1: System Overview

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 program, DIODE extracts a symbolic target expression that characterizes how the program computes the target value (the size of the allocated memory block) at each target memory allocation site. The inputs that appear in this expression are the *relevant inputs*. 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 program to compute the target value).

- **Branch Constraint Extraction:** Again based on instrumented executions of the program, DIODE extracts the sequence of conditional branch instructions that the program executes to generate the path to the target memory allocation site. To ensure that DIODE considers only relevant conditional branches, DIODE discards all branches whose condition is not influenced by relevant inputs.

  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 program to follow the same path as the seed input at selected conditional branches.

- **Target Constraint Solution:** DIODE invokes the Z3 SMT solver [13] to obtain input values that satisfy the target constraint. If the program 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 program performs no sanity 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 program 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 that first flipped branch. DIODE then runs the program 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.

If the program does not contain relevant sanity checks, DIODE will typically find an input that triggers the overflow immediately when it solves the target constraint. If the program does contain relevant sanity checks, DIODE enforces flipped sanity checks in the order in which they are executed by the program. Each iteration of the goal-directed conditional branch enforcement algorithm forces the solver to produce an input that satisfies the next relevant unsatisfied sanity check.

As soon as DIODE enforces enough relevant sanity checks, it typically obtains an input that triggers the overflow (if such an input exists). Because the test inputs enforce only relevant branch conditions associated with previously failed relevant sanity checks, this approach gives the input the freedom it needs to navigate the blocking checks that would, if enforced, cause the program to fail to execute the target site (and therefore fail to generate an overflow).

1.2 Experimental Results

We evaluate DIODE on five applications: Dillo 2.1, VLC 08.6h, SwfPlay 0.5.5, CWebP 0.3.1, and ImageMagick 6.5.2. We start by using DIODE to locate the target memory allocation sites (there are 40 of these sites) and extract, for each site, the target constraint. The target constraint for 17 of the 40 target sites is unsatisfiable. For 9 of the remaining 23 target sites, DIODE was unable to generate an overflow-triggering input. Our manual inspection of the source code verified that the applications contain sanity checks that prevent any input from triggering an overflow at these target sites.

DIODE was able to generate inputs that trigger overflows at all of the remaining 14 sites. We were aware of 3 of these overflows prior to starting the study; the remaining 11 were new. We verified that at least 4 of the new overflow errors are still present in the latest versions of these applications as of the submission date of this paper. For 2 of the 14 sites, DIODE was able to generate an overflow-triggering input with a constraint that forced the input to take the same path as the

seed input. For the remaining 12 sites, the presence of relevant blocking checks requires any overflow-triggering input to take a different path to the target site.

For 9 of the 14 sites DIODE was able to generate an overflow-triggering input without enforcing any conditional branches. The remaining 5 sites require the enforcement of a minimum of 2, average of 4, and maximum of 5 conditional branches. Our manual inspection of the source code indicates that all of the enforced conditional branches involve sanity checks on relevant inputs (all but one of which were apparently not specifically designed to check for overflows). Our results also indicate that, if the application does perform relevant sanity checks and the input generation strategy does not take these checks into account, the input generation strategy is unlikely to find inputs that trigger an overflow even when such inputs exist (Section 5).

1.3 Engineering Challenges and Solutions

DIODE works directly on off-the-shelf, production stripped x86 binaries with no need for symbol information or source code. Given a binary and one or more seed inputs, DIODE executes instrumented versions of the binary to extract the symbolic target expressions and branch conditions for each target memory allocation site. For scalability reasons, DIODE stages the symbolic expression extraction as follows.

The first stage runs the application using fine-grained taint tracing to find memory allocation sites in which the input influences the size of the allocated memory block. This size is the target value of the site. This stage also obtains, for each target value, the *relevant input bytes*, i.e., the input bytes that influence the target value. The second stage runs the application again, recording a (compressed for efficiency) symbolic representation of each computation that the relevant input bytes influence. The third stage reads the symbolic representation of the computation to automatically derive the symbolic target expressions at the target memory allocation sites (these expressions capture the computation that the application performs on the relevant input bytes to obtain the target value) and the symbolic branch condition expressions at the relevant conditional branches. This staging is essential in enabling DIODE to scale to real-world applications — attempting to record a symbolic representation of all computations that the application performs is clearly infeasible for real-world applications.

Given a seed input and candidate values from the Z3 SMT solver for relevant input fields within the seed input, DIODE uses Hachoir [3] and Peach [4] to generate a new input file with the candidate values. Together, Hachoir and Peach reconstruct the input file to accommodate the values, applying techniques such as checksum recalculation.

1.4 DIODE and Multi-Application Code Transfer

Once DIODE has identified the error, the next step is to eliminate the error. The standard approach is to report the error to the developers of the application, then wait for them to develop and distribute a patch [12]. Drawbacks of this approach include the patch development and distribution time and the difficulty of obtaining any patch at all if the application is no longer under development or maintained.

In response to this problem, we have developed CodePhage, an automatic code transfer system [37]. CodePhage starts with an input that exposes an error, a related input that the application processes correctly, and a donor application that processes both inputs correctly (such applications are typically readily available for standard input file formats). CodePhage automatically discovers code in the donor that eliminates the error, then transfers this code into the original application to eliminate the error. CodePhage operates directly on stripped x86 binary donors to generate source-level patches. The code transfer includes automatic data structure translation and the automatic location of appropriate code insertion points in the recipient. Combining CodePhage with DIODE produces a system that automatically discovers and eliminates integer overflow errors — DIODE generates inputs that expose errors; CodePhage uses these inputs to locate and transfer code from donor applications to eliminate the errors. To the best of our knowledge, CodePhage is the first system to automatically transfer code between applications.

1.5 Continuous Automatic Improvement

Given a the ability to automatically expose errors via tools such as DIODE and the ability to automatically repair these errors via tools such as CodePhage [37] (as well as the ability to automatically generate repairs using techniques such as ClearView [27], Error Virtualization [35, 36], Failure-Oblivious Computing [29], and RCV [23]), the next step is to build *continuous automatic improvement* systems that automatically search for errors and generate patches that repair the encountered errors. ClearView's automatic patch generation capability provides continuous improvement driven by responses to attacks and errors that users encounter in production use [27]. Augmenting the ClearView continuous improvement approach with continuously executing automatic error detection tools would make it possible to detect and repair errors before users encounter them and before attackers can exploit them. The result would be significantly more secure and robust software systems.

1.6 Contributions

This paper makes the following contributions:

- **Targeted Input Generation:** It introduces the approach of automatically generating error-triggering inputs that target potentially vulnerable program sites.
- **Sanity and Blocking Checks:** It identifies sanity and blocking checks as an important challenge for techniques that aspire to discover error-triggering inputs. Critically, our results indicate that if the program contains relevant sanity checks, one way to identify relevant sanity checks and generate inputs that satisfy these checks is to incrementally find and enforce first flipped conditional branches.

- **DIODE:** It presents DIODE, an implemented system that works with programs that contain relevant sanity checks to automatically generate inputs that trigger overflow errors. Starting with seed inputs that execute a set of target memory allocation sites, DIODE uses (optimized) symbolic execution to obtain symbolic expressions that characterize how input values determine the path through the computation to the target site and control the target value (the number of bytes that the target site allocates).

  Using a targeted approach, DIODE generates a sequence of inputs, each of which enforces the next relevant conditional branch to find and satisfy the sanity checks that would otherwise prevent the input from triggering the overflow at the target site. The goal is to find inputs that satisfy the relevant sanity checks while preserving the ability of the input to successfully traverse relevant blocking checks and reach the target site.
- **Experimental Results:** It presents experimental results that characterize the effectiveness of DIODE in discovering overflow errors. For our benchmark applications, DIODE discovers 14 overflows, 11 of which are new. For 9 of these overflows, DIODE generates overflows without enforcing any conditional branches. We attribute this success to a lack of relevant sanity checks in the program.

  For the remaining 5 overflows, DIODE discovers the overflow after enforcing a modest (2 to 5) number of conditional branches. We attribute this success to the ability of DIODE to 1) successfully identify and satisfy relevant sanity checks that appear in these programs while 2) preserving the ability of the input to traverse relevant blocking checks that would otherwise prevent the execution of the target site.

Fuzzing [4, 6] and concolic execution [9, 10, 18, 25] have been shown to be effective in discovering errors in the initial input parsing stages of computations, but have had little to no success in exposing errors that lie deep within the program. DIODE shows that discovering and targeting specific potentially vulnerable program sites can effectively expose such deep errors. One of the keys to success is new techniques that work appropriately with sanity and blocking checks to obtain inputs that can successfully traverse these obstacles to reach the target site. The success of DIODE in exposing integer overflow vulnerabilities opens up the field to the further development of other targeted techniques that work effectively with sanity and blocking checks to expose deep errors.

The remainder of the paper is structured as follows. Section 2 presents an example that illustrates the operation of DIODE. Section 3 presents the DIODE goal-directed conditional branch enforcement algorithm. Section 4 discusses the engineering challenges that we had to overcome to enable DIODE to operate on stripped x86 binaries. We present experimental results in Section 5, related work in Section 6, and conclude in Section 7.

```
1   // libpng main data process function.
2   void png_process_data(png_structp png_ptr,
3    png_infop info_ptr, ...) {
4     ...
5     while (png_ptr->buffer_size) {
6       // This is a wrapper for png_push_read_chunk
7       png_process_some_data(png_ptr, info_ptr);
8     }
9   }
10  void png_push_read_chunk(png_structp png_ptr,
11   png_infop info_ptr) {
12    if (!png_memcmp(png_ptr->chunk_name,png_IHDR,4)){
13      ...
14      png_handle_IHDR(png_ptr, info_ptr, ...);
15    }
16    else if (!png_memcmp(png_ptr->chunk_name,png_IDAT, 4)) {
17      // Datainfo callback is called
18      png_push_have_info(png_ptr, info_ptr);
19    }
20  }
21  png_check_IHDR(png_structp png_ptr,
22    png_uint_32 width, png_uint_32 height, int bit_depth...) {
23      ...
24      //Check 3: Height < 1000000L
25      if (height > PNG_USER_HEIGHT_MAX) {
26        png_warning(png_ptr,
27              "Image width exceeds user limit in IHDR");
28        error = 1;
29      }
30      //Check 4: Width < 1000000L
31      if (width > PNG_USER_WIDTH_MAX) {
32        png_warning(png_ptr,
33              "Image width exceeds user limit in IHDR");
34        error = 1;
35      }
36  }
37  png_get_uint_31(png_structp png_ptr, png_const_bytep buf) {
38      png_uint_32 uval = png_get_uint_32(buf);
39      // Checks 1 & 2: Checks that width/height < 0x7fffffffL
40      if (uval > PNG_UINT_31_MAX)
41        png_error(png_ptr,
42               "PNG unsigned integer out of range");
43      return (uval);
44  }
45  #define PNG_ROWBYTES(pixel_bits,width) ((pixel_bits)>=8? \
46    ((width)*(((png_uint_32)(pixel_bits))>>3)):\
47    ((((width)*((png_uint_32)(pixel_bits)))+7)>>3))
48  void png_handle_IHDR(png_structp png_ptr,
49   png_infop info_ptr, ...) {
50    ...
51    // read individual png fields from input buffer
52    width    = png_get_uint_31(png_ptr, buf);
53    height   = png_get_uint_31(png_ptr, buf + 4);
54    bit_depth = buf[8];
55    ...
56    png_ptr->width      = width;
57    png_ptr->height     = height;
58    png_ptr->bit_depth   = (png_byte)bit_depth;
59    ...
60    png_ptr->pixel_depth = (png_byte)(
61        png_ptr->bit_depth * png_ptr->channels);
62    png_ptr->rowbytes = PNG_ROWBYTES(
63        png_ptr->pixel_depth, png_ptr->width);
64  }
65  png_memset_check (png_structp png_ptr, png_voidp s1, int value,
66          png_uint_32 length)
67  {
68    png_size_t size;
69    size = (png_size_t)length;
70    if ((png_uint_32)size != length)
71      png_error(png_ptr, "Overflow in png_memset_check.");
72    return (png_memset (s1, value, size));
73  }
74  // Dillo datainfo initialization callback
75  static void
76  Png_datainfo_callback(png_structp png_ptr, png_infop info_ptr)
77  {
78    DilloPng *png;
79    ...
80    // Check 5:  Incorrect check of max image size
81    if (abs(png->width*png->height) > IMAGE_MAX_W * IMAGE_MAX_H) {
82      MSG("suspicious image size request %ldx%ld\n",
83         png->width, png->height);
84      return;
85    }
86    // Where the overflow happens
87    png->image_data = (uchar_t *)dMalloc(png->rowbytes * png->height);
88  }
```

Figure 2: Simplified source code from Dillo 2.1 and libpng

2. Example

We next present an example that illustrates how DIODE automatically generates an input that triggers an integer overflow in Dillo 2.1, a lightweight open source web browser [2]. Figure 2 presents the simplified source code for this example. This code is from the libpng library, which Dillo uses to read PNG images.

Target Site Discovery: DIODE runs Dillo on the seed input, using a fine-grained dynamic taint analysis to track the propagation of input bytes through the program. The libpng runtime calls `png_process_data()` (line 2) to process each PNG image. This function then calls `png_push_read_chunk()` (line 10) to process each chunk in the PNG image. When the libpng runtime reads the first data chunk (the IDAT chunk), it calls the Dillo callback `png_datainfo_callback()` (lines 76-88) in the Dillo PNG processing module. At line 87, Dillo invokes `dMalloc()` to allocate the image buffer. Because the size of the allocated memory block is influenced by the input, DIODE identifies the site as a *target memory allocation site*.

Dillo computes the size of the allocated image buffer as `png→rowbytes * png→height`. This is the *target value*. DIODE's goal is to generate an input that 1) executes the target site at line 87 and 2) causes the computation of the target value `png→rowbytes * png→height` to overflow. The taint information indicates that the target value is influenced by the PNG width, height, and bitdepth fields in the seed input file. These fields are the *relevant input bytes*.

Target Expression Extraction: Next, DIODE runs the application again, this time with additional instrumentation that records all calculations that involve the relevant input bytes. DIODE uses the recorded information to extract the symbolic *target expression*, which characterizes how the application computes the target value (recall that this target value is the size of the allocated image buffer) as a function of the input bytes. Conceptually, this expression is `((width*(4*bitdepth))>>3)*height`, where `width`, `bitdepth`, and `height` are the PNG width, bitdepth, and height fields in the input file. Large values of these fields will cause this expression to overflow. Because of endianness conversions that take place when Dillo reads in the input field values, the actual target expression is:

```
MallocArg(Mul(32,Mul(32,Add(32,ToSize(32,UShr(32,BvAnd(32,
HachField(32, '/header/width'),Constant(0xFF000000)),
Constant(24))),Add(32,Add(32,Shl(32,ToSize(32,
BvAnd(32,HachField(32, '/header/width'), Constant(0xFF))),
Constant(24)),Shl(32,ToSize(32,UShr(32,BvAnd(32,HachField(32,
'/header/width'),Constant(0xFF00)),Constant(8))),
Constant(16))),Shl(32,ToSize(32,UShr(32,BvAnd(32,
HachField(32, '/header/width'),Constant(0xFF0000)),
Constant(16))),Constant(8)))),ToSize(32,Shrink(8,
UShr(32,ToSize(32,Shrink(8,Mul(32,ToSize(32,
HachField(8, '/header/bit_depth')),Constant(4)))),
Constant(3))))),Add(32,ToSize(32,UShr(32,BvAnd(32,
HachField(32, '/header/height'),Constant(0xFF000000)),
Constant(24))),Add(32,Add(32,Shl(32,ToSize(32,
BvAnd(32,HachField(32, '/header/height'),
Constant(0xFF))),Constant(24)),Shl(32,ToSize(32,
UShr(32,BvAnd(32,HachField(32, '/header/height'),
Constant(0xFF00)),Constant(8))),Constant(16))),Shl(32,
ToSize(32,UShr(32,BvAnd(32,HachField(32,
'/header/height'),Constant(0xFF0000)),Constant(16))),
Constant(8))))),Constant(0xFFFFFFFF))
```

Here `TargetSite` indicates that, to overflow, the expression must be greater than the constant `0xFFFFFFFF` (at the end of the last line of the expression). The expression itself references the PNG width, bitdepth, and height fields from the input file as `/header/width`, `/header/bit_depth`, and `/header/height`. The remainder of the expression captures the computation of the target value as described above. It also incorporates constructs (such as `Shl` and `BvAnd`) that capture the conversion of the input values from big-endian to little-endian form. From this target expression, DIODE extracts a target constraint that is satisfied if and only if the computation of the target expression overflows. The variables in this target constraint represent the `/header/width`, `/header/bit_depth`, and `/header/height` PNG input file fields. The target constraint faithfully represents integer arithmetic as implemented in the hardware.

Target Constraint: DIODE next uses the Z3 solver [13] to obtain *candidate values* for the relevant input byte values that would cause the target value to overflow. In this example, the solution sets `/header/width` to 3880563055L, `/header/bit_depth` to 4, and `/header/height` to 689749785L. It then uses Hachoir [3] and Peach [4] to generate a new input file with the candidate values (we call this input file the *initial input file*) and executes Dillo on this new input file. Dillo and libpng contain sanity checks that together prevent the input from triggering the overflow.

Sanity Checks: Dillo and libpng collectively contain five sanity checks. The first two checks occur in `png_get_uint_31` (line 37), which checks that the PNG height and width values are less than `0x7fffffffL`. The third and fourth sanity checks occur in `png_check_IHDR` (lines 21–36), which check that the PNG height and width values are less than one million. The fifth and final sanity check occurs at line 72, immediately before the target memory allocation site at line 87. This final sanity check attempts to ensure that the size of the allocated image does not exceed a specified value (`IMAGE_MAX_W * IMAGE_MAX_H`) (which is 6000 * 6000). This final check contains an overflow error that prevents it from recognizing and correctly rejecting some inputs that cause overflows at the target memory allocation site at line 87.

Symbolic Branch Condition Extraction: DIODE uses the recorded instrumentation information to extract symbolic expressions (the branch conditions) that characterize how the application computes the values of the branch conditions at conditional branch instructions that are directly influenced by the relevant input bytes. For Dillo, the extracted branch conditions characterize how Dillo computes the branch conditions for the sanity checks described above.

Blocking Checks: DIODE is capable of generating a constraint over the relevant input bytes that 1) cause the target value to overflow and 2) cause the application to follow the same path through the conditional branches to the target site as the seed input. If this constraint were satisfiable, DIODE could then use the solution to generate an input file that would trigger the overflow. This constraint is not satisfiable. Dillo and libpng contain *blocking checks* that prevent any input that would trigger an overflow from following the same path through the relevant branches to the target site.

The blocking checks occur in the `png_memset` procedure, which initializes a block of memory whose size is a function of the PNG width and bitdepth input fields. The `png_memset` procedure is hand coded in assembly language using the SSE2 extensions. This procedure contains a loop that iterates over the block of memory initializing the values in the block. The number of iterations of this loop is a function of the size of the block of memory. The conditional branch that controls the number of iterations is therefore a relevant branch — its condition depends on the PNG width and bitdepth fields. Any input that follows the same path as the seed input through the relevant conditions must therefore have PNG width and bitdepth fields that produce the same number of iterations of the loop as the seed inputs. This additional *blocking constraint* makes it impossible to obtain an input that both 1) triggers the overflow and 2) follows the same path through the relevant branches as the seed input.

In our example, the PNG width field is 280. The number of iterations is 8 and the constraint is width $\times$ bitdepth$/8 \leq 1154$. The target expression is (width $\times$ bitdepth$/8$) $\times$ height (which is rowbytes $\times$ height). This value cannot overflow because the maximum value of rowbytes is 1154 and the maximum value of height is 1,000,000 (line 24). These values produce 1154 $\times$ 1,000,000 = 1,154,000,000, which is less than 2^{32}.

Goal-Directed Conditional Branch Enforcement: DIODE next starts goal-directed conditional branch enforcement. It initializes the *current constraint* to the target constraint and the *current input* to the initial input (recall that the initial input was generated to satisfy only the target constraint). It then executes Dillo on the seed input and the current input to find the first (in the program execution order) relevant branch where the seed and current input take different paths. In our example this relevant branch corresponds to the sanity check at function `png_get_uint_31`, line 48 — the seed input satisfies this sanity check, while the current input fails the sanity check (because the generated height is too large).

DIODE therefore adds the branch constraint from the corresponding conditional to the current constraint. Given this new current constraint, Z3 produces a solution that sets `/header/width` to 1632109428L, `/header/bit_depth` to 4, and `/header/height` to 872360950L. The resulting current input fails to generate an overflow because it fails the sanity check at `png_check_IHDR`, line 25.

DIODE adds the branch constraint from the conditional branch that implements the sanity check to the current constraint and obtains a new `/header/width` of 1081489513L and `/header/height` of 732927L. The resulting input file fails to trigger an overflow because it fails the sanity check at `png_check_IHDR`, line 31. After adding the corresponding branch constraint, the solver comes back with `/header/width` 966175L and `/header/height` 484094L. The sanity check at `Png_datainfo_callback`, line 81, which checks for an overly large image size, rejects the resulting current input.

$$
\begin{array}{rcll}
x, y & \in & \text{Var} & = \ \text{PgmVar} \cup \text{InpVar} \\
A, A_1, A_2 & \in & \text{Aexp} & ::= \ n \mid x \mid \text{-}A \mid A_1 \, aop \, A_2 \\
B, B_1, B_2 & \in & \text{Bexp} & ::= \ \texttt{true} \mid \texttt{false} \mid A_1 \, cmp \, A_2 \mid \\
 & & & \quad \texttt{!}B \mid B_1 \, \texttt{\&\&} \, B_2 \mid B_1 \, \texttt{||} \, B_2 \\
C, C_1, \cdots, C_n & \in & \text{Stmt} & ::= \ \texttt{skip} \mid x \texttt{=} A \mid \\
 & & & \quad x \texttt{=} \texttt{alloc(}y\texttt{)} \mid x \texttt{=} y\texttt{[}A\texttt{]} \mid x\texttt{[}A\texttt{]} \texttt{=} y \mid \\
 & & & \quad \texttt{if} \, B \, S_1 \, S_2 \mid \texttt{while} \, B \, S
\end{array}
$$

$$
S, S_1, S_2 \in \text{Seq} ::= C_1 \, ; \, \cdots \, ; \, C_n
$$

Figure 3: Syntax

Successful Generation of Overflow-Triggering Input: This sanity check, designed to detect overflows, is itself vulnerable to an overflow — carefully chosen values can overflow the checked value and cause the sanity check to incorrectly accept an input that overflows the target value at line 87. After adding the branch condition from line 81 to the current constraint, the solver comes back with `/header/width` 689853L and `/header/height` 915210L. With these values, the generated input successfully navigates the sanity checks and the blocking checks to trigger the overflow. The resulting out of bounds writes cause Dillo to crash with a SIGSEGV exception.

3. Goal-Directed Conditional Branch Enforcement Algorithm

We next present the basic DIODE goal-directed conditional branch enforcement algorithm. We first define a core imperative language and a small-step operational semantics for this language. This semantics defines both concrete and symbolic executions for programs written in the core language. We then use this semantics to present the algorithm.

3.1 Core Language

Figure 3 presents the syntax of a core imperative language with variables, arithmetic expressions, boolean expressions, assignments, dynamic memory allocation, memory read/write, conditional statements, while loops, and sequential composition.

Variables: We divide variables into two classes, PgmVar and InpVar. A *program variable* $\in$ PgmVar is a conventional variable and can store integer values or memory addresses as usual. On the other hand, an *input variable* $\in$ InpVar represents an external input value to a program. DIODE uses input variables to symbolically express how the program computes a target value (such as the size of the allocated memory block) from the input values.

Labels: All program statements have a unique label $\ell \in$ Label. before(C) and after(C) denote the labels before and after the statement C, respectively. In a sequence $S = C_1 \, ; \, \cdots ; C_n$, after($C_i$) = before($C_{i+1}$). We define before($C_1 \, ; \, \cdots ; C_n$) and after($C_1 \, ; \, \cdots ; C_n$) as follows:

$$
\begin{array}{rcl}
\text{before}(C_1 \, ; \, \cdots ; C_n) & = & \text{before}(C_1) \\
\text{after}(C_1 \, ; \, \cdots ; C_n) & = & \text{after}(C_n)
\end{array}
$$

$$\rho \vdash n \Rightarrow \langle n, n\rangle \qquad \frac{x \in \text{PgmVar}}{\rho \vdash x \Rightarrow \rho(x)} \qquad \overset{\textsc{InpVar}}{\frac{x \in \text{InpVar}}{\rho \vdash x \Rightarrow \langle \pi_1(\rho(x)), x\rangle}} \qquad \frac{\rho \vdash A \Rightarrow \langle n, n\rangle}{\rho \vdash \texttt{-}A \Rightarrow \langle -n, -n\rangle} \qquad \frac{\rho \vdash A \Rightarrow \langle n, A'\rangle}{\rho \vdash \texttt{-}A \Rightarrow \langle -n, \texttt{-}A'\rangle} \qquad \frac{\rho \vdash A_1 \Rightarrow \langle n_1, n_1\rangle \quad \rho \vdash A_2 \Rightarrow \langle n_2, n_2\rangle}{\rho \vdash A_1 \texttt{+} A_2 \Rightarrow \langle n_1 + n_2, n_1 + n_2\rangle}$$

$$\frac{\rho \vdash A_1 \Rightarrow \langle n_1, A_1'\rangle \quad \rho \vdash A_2 \Rightarrow \langle n_2, n_2\rangle}{\rho \vdash A_1 \texttt{+} A_2 \Rightarrow \langle n_1 + n_2, A_1' \texttt{+} n_2\rangle} \qquad \frac{\rho \vdash A_1 \Rightarrow \langle n_1, n_1\rangle \quad \rho \vdash A_2 \Rightarrow \langle n_2, A_2'\rangle}{\rho \vdash A_1 \texttt{+} A_2 \Rightarrow \langle n_1 + n_2, n_1 \texttt{+} A_2'\rangle} \qquad \frac{\rho \vdash A_1 \Rightarrow \langle n_1, A_1'\rangle \quad \rho \vdash A_2 \Rightarrow \langle n_2, A_2'\rangle}{\rho \vdash A_1 \texttt{+} A_2 \Rightarrow \langle n_1 + n_2, A_1' \texttt{+} A_2'\rangle}$$

Figure 4: Semantics of Arithmetic Expressions

ℓ and ℓ' denote before(C) and after(C) of statement C in question

$$\langle \ell, \rho, m, \phi\rangle =\llbracket\, \texttt{skip} \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \ell', \rho, m, \phi\rangle \qquad \frac{x \in \text{PgmVar} \quad \rho \vdash A \Rightarrow \langle v, w\rangle}{\langle \ell, \rho, m, \phi\rangle =\llbracket\, x \,\texttt{=}\, A \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \ell', \rho[x \mapsto \langle v, w\rangle], m, \phi\rangle}$$

$$\frac{x \in \text{PgmVar} \quad \rho \vdash y \Rightarrow \langle n, _\rangle \quad n > 0 \quad a \notin \text{dom}(m)}{\langle \ell, \rho, m, \phi\rangle =\llbracket\, x \,\texttt{=}\, \texttt{alloc(}y\texttt{)} \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \ell', \rho[x \mapsto \langle a, a\rangle], m[(a,0) \mapsto \langle 0,0\rangle, \cdots, (a, n-1) \mapsto \langle 0,0\rangle], \phi\rangle} \qquad \frac{x \in \text{PgmVar} \quad \rho \vdash y \Rightarrow \langle a, _\rangle \quad \rho \vdash A \Rightarrow \langle n, _\rangle}{\langle \ell, \rho, m, \phi\rangle =\llbracket\, x \,\texttt{=}\, y\texttt{[}A\texttt{]} \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \ell', \rho[x \mapsto m(a,n)], m, \phi\rangle}$$

$$\frac{\rho \vdash y \Rightarrow \langle v, w\rangle \quad \rho \vdash x \Rightarrow \langle a, _\rangle \quad \rho \vdash A \Rightarrow \langle n, _\rangle}{\langle \ell, \rho, m, \phi\rangle =\llbracket\, x\texttt{[}A\texttt{]} \,\texttt{=}\, y \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \ell', \rho, m[(a,n) \mapsto \langle v, w\rangle], \phi\rangle} \qquad \frac{\rho \vdash B \Rightarrow \langle \text{true}, \text{true}\rangle}{\langle \ell, \rho, m, \phi\rangle =\llbracket\, \texttt{if}\, B\, S_1\, S_2 \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \text{before}(S_1), \rho, m, \phi\rangle}$$

$$\frac{\rho \vdash B \Rightarrow \langle \text{true}, B'\rangle}{\langle \ell, \rho, m, \phi\rangle =\llbracket\, \texttt{if}\, B\, S_1\, S_2 \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \text{before}(S_1), \rho, m, \phi \to \langle \ell, B'\rangle\rangle} \qquad \frac{\langle \ell_1, \rho, m, \phi\rangle =\llbracket\, S_1 \,\rrbracket\Rightarrow_{\text{Seq}} \langle \ell_1', \rho', m', \phi'\rangle}{\langle \ell_1, \rho, m, \phi\rangle =\llbracket\, \texttt{if}\, B\, S_1\, S_2 \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \ell_1', \rho', m', \phi'\rangle} \qquad \langle \text{after}(S_1), \rho, m, \phi\rangle =\llbracket\, \texttt{if}\, B\, S_1\, S_2 \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \ell', \rho, m, \phi\rangle$$

$$\frac{\rho \vdash B \Rightarrow \langle \text{false}, \text{false}\rangle}{\langle \ell, \rho, m, \phi\rangle =\llbracket\, \texttt{if}\, B\, S_1\, S_2 \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \text{before}(S_2), \rho, m, \phi\rangle} \qquad \frac{\rho \vdash B \Rightarrow \langle \text{false}, B'\rangle}{\langle \ell, \rho, m, \phi\rangle =\llbracket\, \texttt{if}\, B\, S_1\, S_2 \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \text{before}(S_2), \rho, m, \phi \to \langle \ell, \texttt{!}B'\rangle\rangle}$$

$$\frac{\langle \ell_2, \rho, m, \phi\rangle =\llbracket\, S_2 \,\rrbracket\Rightarrow_{\text{Seq}} \langle \ell_2', \rho', m', \phi'\rangle}{\langle \ell_2, \rho, m, \phi\rangle =\llbracket\, \texttt{if}\, B\, S_1\, S_2 \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \ell_2', \rho', m', \phi'\rangle} \qquad \langle \text{after}(S_2), \rho, m, \phi\rangle =\llbracket\, \texttt{if}\, B\, S_1\, S_2 \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \ell', \rho, m, \phi\rangle \qquad \frac{\rho \vdash B \Rightarrow \langle \text{true}, _\rangle}{\langle \ell, \rho, m, \phi\rangle =\llbracket\, \texttt{while}\, B\, S \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \text{before}(S), \rho, m, \phi\rangle}$$

$$\frac{\langle \ell_1, \rho, m, \phi\rangle =\llbracket\, S \,\rrbracket\Rightarrow_{\text{Seq}} \langle \ell_1', \rho', m', \phi'\rangle}{\langle \ell_1, \rho, m, \phi\rangle =\llbracket\, \texttt{while}\, B\, S \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \ell_1', \rho', m', \phi'\rangle} \qquad \langle \text{after}(S), \rho, m, \phi\rangle =\llbracket\, \texttt{while}\, B\, S \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \ell, \rho, m, \phi\rangle \qquad \frac{\rho \vdash B \Rightarrow \langle \text{false}, _\rangle}{\langle \ell, \rho, m, \phi\rangle =\llbracket\, \texttt{while}\, B\, S \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \ell', \rho, m, \phi\rangle}$$

Figure 5: Small-Step Operational Semantics of Statements

3.2 Operational Semantics

The language has three different kinds of values

$$\begin{array}{rcll} n & \in & \text{Int} & \\ b, b_1, b_2 & \in & \text{Bool} & = \{\text{true}, \text{false}\} \\ a & \in & \text{Addr} & \end{array}$$

where Int is a set of machine integers of finite bit-width, Bool is the standard set of boolean values, and Addr is an address space with an unbounded number of memory addresses.

An *environment* $\rho \in$ Env is a partial mapping from variables to pairs of values and symbolic values. A *value* $v \in$ Val is either an integer or an memory address. A *symbolic value* $w \in$ SymVal can be a symbolic arithmetic expression, integer, or memory address. We use symbolic values to characterize how values were computed as a function of input variables.

$$\begin{array}{rcccl} \rho & \in & \text{Env} & = & \text{Var} \to \text{Val} \times \text{SymVal} \\ v, v_1, v_2 & \in & \text{Val} & = & \text{Int} \cup \text{Addr} \\ w, w_1, w_2 & \in & \text{SymVal} & = & \text{Int} \cup \text{Addr} \cup \text{Aexp} \end{array}$$

Similar to an environment, a *memory* $m \in$ Mem receives a base address and an offset to the base address as its arguments and returns a pair of a value and a symbolic value.

$$m, m_1, m_2 \in \text{Mem} = \text{Addr} \to \text{Offset} \to \text{Val} \times \text{SymVal}$$

$$\frac{\ell \text{ is a label in } C_i \quad \langle \ell, \rho, m, \phi\rangle =\llbracket\, C_i \,\rrbracket\Rightarrow_{\text{Stmt}} \langle \ell', \rho', m', \phi'\rangle}{\langle \ell, \rho, m, \phi\rangle =\llbracket\, C_1 \,\texttt{;}\, \cdots \texttt{;}\, C_n \,\rrbracket\Rightarrow_{\text{Seq}} \langle \ell', \rho', m', \phi'\rangle}$$

Figure 6: Small-Step Operational Semantics of Sequences

A *branch condition* $\phi \in$ BranchCond is a sequence. Each element $\langle \ell, B\rangle$ in this sequence records the symbolic branch condition that determines the path taken at the conditional branch at label ℓ. The elements appear in ϕ in the program execution order.

$$\phi \in \text{BranchCond} := \varepsilon \mid \langle \ell, B\rangle \to \phi$$

A *program state* $\sigma = \langle \ell, \rho, m, \phi\rangle$ is composed of the current program point (represented by a label ℓ), an environment ρ, a memory m, and a branch condition ϕ. At a state $\langle \ell, \rho, m, \phi\rangle$, the program is about to execute a statement C labelled ℓ (i.e. before$(C) = \ell$) in the environment ρ and memory m at the program point ℓ reached by taking the path recorded by the conditional branches in the sequence ϕ.

$$\sigma \in \text{State} = \text{Label} \times \text{Env} \times \text{Mem} \times \text{BranchCond}$$

Expressions: Figure 4 presents the semantics of arithmetic expressions. Each expression evaluates to a pair $\langle v, w\rangle$, where

$v \in$ Val is a concrete value and $w \in$ SymVal is a symbolic expression. The INPVAR rule, for example, defines that the evaluation of an input variable $x \in$ InpVar produces a pair $\langle \pi_1(\rho(x)), x\rangle$, where $\pi_1(\rho(x))$ is the actual input value and x is the variable that symbolically represents that value. The semantics of boolean expressions is defined in a similar way.

Statement: Figures 5 and 6 present the small-step operational semantics of DIODE's core language. Note that the meaning of ℓ and ℓ' is slightly different in Figures 5 and 6. In Figure 5, ℓ is the label for the program point before the relevant statement C; ℓ' is the label for the program point after C. In Figure 6, ℓ and ℓ' are the labels of some program points within (including before for ℓ and after for ℓ') some statement C_i in $C_1; \cdots; C_n$.

3.3 Algorithm

Figure 7 presents the DIODE goal-directed conditional branch enforcement algorithm. Given a program S, an initial program state σ, and a target site ℓ, the algorithm first extracts the symbolic target expression B and the observed path ϕ (from the seed input) for that site (line 1). target$(\langle S,\sigma\rangle,\ell)$ is defined as follows:

$$\text{target}(\langle S,\sigma\rangle,\ell) = \{\langle \pi_2(\rho(y)),\phi\rangle \mid \langle \sigma,\langle \ell,\rho,m,\phi\rangle\rangle \in \tau^*(S)\}$$
$$\text{where } \ell = \text{before}(x = \texttt{alloc}(y))$$

The function target$(\langle S,\sigma\rangle,\ell)$ is defined in terms of the reflexive transitive closure $\tau^*(S)$ of the transition relation of the program S, which contains all possible transitions from a starting state to all reachable states.

The algorithm next uses the overflow(B) function to extract the target constraint β (line 2). The overflow(B) function returns a target constraint β such that any input that satisfies the target constraint β will trigger an overflow during the computation of the target expression B.

The algorithm next compresses the path ϕ to coalesce multiple occurrences of conditional branch constraints of a conditional statement into a single constraint (line 7 and Figure 8). This single constraint is the conjuction of all of the observed branch constraints. The algorithm then extracts the relevant branch constraints (line 8) and performs the goal-directed conditional branch enforcement algorithm (lines 10-16).

The relevant(ϕ,β) function takes a branch condition ϕ and a target constraint β as its arguments, and removes conditions that are not relevant to the target constraint β from the branch condition ϕ. A condition $\langle \ell,B\rangle$ in a branch condition is relevant to a target constraint β if the condition B and the target constraint β share the same input variable.

Input : a program S, an initial state σ, a target label ℓ
Output : an input I that triggers an integer overflow at label ℓ

```
1  for ⟨B, φ⟩ in target(⟨S, σ⟩, ℓ) do
2      β ⟵ overflow(B)
3      if the solver generates an input I that satisfies β then
4          if the input I triggers an overflow at label ℓ then
5              return the input I
6      else continue
7      φ ⟵ compress(φ)
8      φ ⟵ relevant(φ, β)
9      φ′ ⟵ true
10     while true do
11         if the previous input I satisfies φ then break
12         φ′ ⟵
           φ′∧ (the first condition in φ that the previous input I
           does not satisfy)
13         if the solver generates an input I that satisfies φ′ ∧ β
           then
14             if the input I triggers an overflow at label ℓ then
15                 return the input I
16         else break
17 return not found
```

Figure 7: Goal-Directed Conditional Branch Enforcement

Parameters : $\phi \in$ BranchCond
Returns : ϕ's compressed form $\in$ BranchCond

```
1  Function compress(φ) =
2  begin
3      if φ is ε then
4          return ε
5      else if φ is ⟨ℓ, B⟩ → φ then
6          B ⟵ B ∧ (⋀_{⟨ℓ,B′⟩ in φ} B′)
7          φ ⟵ filter out all ⟨ℓ, B′⟩ from φ
8          return ⟨ℓ, B⟩ → compress(φ)
```

Figure 8: Branch Condition Compression

4. System Design and Implementation

We next discuss how DIODE deals with the many complications that it must overcome to effectively operate on stripped x86 binaries. DIODE consists of approximately 9,000 lines of C (most of this code implements the taint and symbolic expression tracking) and 6,000 lines of Python (the target and branch constraint generation algorithms, code that interfaces with Z3, code that manages the database of relevant experimental results, and a distributed work queue system). We first describe our techniques for target site identification. Second, we introduce the dynamic instrumentation used for target and branch constraint extraction. Third, we discuss how DIODE generates and solves target constraints. Fourth, we discuss how DIODE generates new inputs. Fifth, we discuss the implementation of our goal-directed conditional branch enforcement algorithm. Finally, we discuss how DIODE detects any errors caused by the overflow.

4.1 Target Site Identification

To extract the set of symbolic target expressions that characterizes how the application computes the target value at critical

program sites, DIODE uses a fine-grained dynamic taint analysis built on top of the Valgrind [26] 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 program until it reaches a potential target site (e.g., malloc). 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). Using the dynamic taint analysis on the application and a seed input, DIODE generates the set of target sites and relevant input bytes.

4.2 Target and Branch Constraint Extraction

Next, DIODE reruns the program with additional instrumentation that enables DIODE to reconstruct the full symbolic target expression. Conceptually, DIODE generates a symbolic record of all calculations that the application performs (Section 3). Obviously, attempting to record all calculations would produce an unmanageable volume of information. DIODE reduces the volume of recorded information with the following optimizations:

- **Relevant Input Bytes:** DIODE only records calculations that involve the relevant input bytes. Specifically, DIODE 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:** DIODE further reduces the amount of recorded information by simplifying recorded expressions at runtime. Specifically, DIODE identifies and simplifies resize, move and arithmetic operations. For example, DIODE 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, DIODE uses the Hachoir [3] tool to convert byte ranges into input fields (e.g., in the PNG format, bytes 0-3 represent /header/height).

DIODE also uses the recorded information to extract symbolic expressions that characterize how the application computes the values of conditional branch instructions that relevant input bytes directly influence.

4.3 Target Constraint Solution

DIODE uses the Z3 SMT solver [13] to obtain new input values that satisfy the target constraint. Note that the generated target constraint is designed to capture any overflow in the evaluation of the expression, including in the evaluation of subexpressions. For example, if $bbp_8 \in \{8, 16, 32\}$, there are no values that cause the following expression to overflow:

$$((width_{16} \times height_{16}) \times 4)/bbp_8) > 2^{32}$$

But there are values that cause the following subexpression to overflow:

$$((width_{16} \times height_{16}) \times 4)) > 2^{32}$$

4.4 Test Input Generation

DIODE uses a combination of Hachoir [3] and Peach [4] to generate input files with the values obtained from the SMT solver for the target expression. Together, these tools reconstruct the input file such that it satisfies any checksum calculations or any required field orderings. If DIODE needs to operate on an unknown input format, it also supports a raw-byte option, where modifications are made directly on the input bytes. To deal with any required checksum calculations in raw-byte mode, DIODE can use standard checksum reconstruction techniques [40].

4.5 Goal-Directed Branch Enforcement

If a test input that is generated from a target constraint solution fails to trigger an integer overflow error, DIODE turns on instrumentation that records the path taken at all conditional branches that the seed input executes. DIODE uses this instrumentation to find the first conditional branch at which the generated input takes a different path from the seed input. DIODE uses this information to drive the goal-directed branch enforcement algorithm described above (Section 3).

4.6 Error Detection

We use Valgrind's *memcheck* to detect errors (invalid reads and writes; uninitialized reads and writes) that occur as a result of the overflow. Our automated system therefore does not directly detect the overflow; it only detects the overflow indirectly through its effect on the computation (for our benchmark applications, we manually verify that the generated input actually produces an overflow and generates the reported errors as a result of the overflow). Our automated system first filters any errors that occur during t he execution on the seed input.

5. Evaluation

We evaluate DIODE on five applications: Dillo 2.1, VLC 08.6h, SwfPlay 0.5.5, CWebP 0.3.1, and ImageMagick 6.5.2. For each application we obtain a seed input, then use DIODE to automatically generate input files that trigger overflows in the applications. We perform all tests on a quad Intel i7 2.2 GHz machine with 8 GB RAM.

5.1 Benchmark Selection

The benchmark applications were selected as follows. First, we select applications that process input formats supported by Hachoir [3] and Peach [4]. Second, we filter applications that cannot be successfully processed by DIODE's dynamic instrumentation engine. Third, we select applications that contain at least one known integer overflow vulnerability,

Application	Total Target Sites	DIODE Exposes Overflow	Target Constraint Unsatisfiable	Sanity Checks Prevent Overflow
Dillo 2.1	12	3	1	8
VLC 08.6h	4	4	0	0
SwfPlay 0.5.5	8	3	5	0
CWEBP 0.3.1	7	1	6	0
ImageMagick 6.5.2	9	3	5	1

Table 1: Target Site Classification

5.2 Target Site Classification

Table 1 classifies the target sites in our benchmark applications. There is one row for each application. The first column (Application) identifies the application. The second column (Total Target Sites) presents total number of exercised memory allocation sites from the executions on the seed inputs. These sites are the target sites. The third column (DIODE Exposes Overflow) presents the number of sites for which DIODE was able to generate an input that triggered an overflow at the site. The fourth column (Target Constraint Unsatisfiable) presents the number of sites for which the target constraint, by itself, is unsatisfiable. We verified, via a manual inspection, that there is *no* input that will cause an overflow at any of these sites. The fifth column (Sanity Checks Prevent Overflow) presents the number of remaining sites. For all of these remaining sites, we manually verified that the application contains sanity checks that ensure that there is *no* input that triggers an overflow at that site.

Note that, for each target site, either 1) DIODE finds an input that triggers an overflow at that site, or 2) no such input exists. Our analysis indicates that, except for VLC 0.8.6h, whenever DIODE is able to generate an input that triggers an overflow at a given site, the application is missing overflow sanity checks for that site (of course, the applications contain other relevant sanity checks that DIODE must successfully navigate to trigger the overflow). VLC 0.8.6h contains ineffective overflow sanity checks that are designed to protect the application against overflow, but do not, in fact, do so. DIODE is able to generate inputs that successfully evade these checks to trigger overflows at the target sites.

5.3 Overflow Characteristics

Table 2 summarizes the results for each overflow. The table contains one line for each overflow that DIODE discovers. The first column (Application) identifies the application that contains the overflow. The second column (Target) presents the source code file and line that contains the memory allocation statement for which the overflow occurs. The third column (CVE Number) presents either the CVE number of the overflow (if the overflow was known) or "New" if the overflow was new. We note that all but three of the 14 overflows were new. Four of the 11 new overflows persist in the latest versions of the benchmark applications as of the submission date of this paper. Specifically, the latest versions of CWebP and Display, CWebP 0.4.1 and Display 6.8.9-8, are still vulnerable to error triggering inputs discovered by DIODE. We have notified the developers and are awaiting confirmation.

The fourth column (Error Type) characterizes the effect of the overflow on the application for the first input (that DIODE discovers) that triggers the overflow. In most cases the overflow causes the program to generate a SIGSEGV exception and crash, either from an invalid read or from an invalid write as presented in the table. The remaining two overflows cause the application to perform invalid reads and/or writes that do not crash the application. We detect these invalid reads and writes using the Valgrind memcheck tool [26], which monitors the reads and writes and detects invalid reads and writes. All of the invalid reads or writes occur because the overflow makes the memory block allocated at the target allocation site too small to contain the data.

The fifth column (Analysis and Discovery Time) presents the initial analysis time required for each application (performed once) and the subsequent time to generate an error input for each bug. Each entry in this column is of the form (A) B, where A is the analysis time and B is the time required to generate the error input.

The sixth column (Enforced Branches) presents the number of relevant conditional branches that DIODE enforced before generating an input that triggered the overflow. Each entry in this column is of the form X/Y, where X is the number of enforced conditional branches and Y is the total number of relevant conditional branches on the path that the seed input takes to the target memory allocation site. We note that the number of enforced conditional branches is small, especially relative to the total number of relevant conditional branches — to discover the overflow, DIODE enforces only between two to five out of the 35 to 5779 total relevant conditional branches. Our manual inspection of the code indicates that all of the enforced branches are sanity checks, but that (apparently) only one of these checks is designed (obviously incorrectly) to detect an overflow (Section 2).

5.4 Blocking Checks

Recall that DIODE can generate a constraint that requires 1) the computation of the target value to overflow and 2) the input to follow the same path through the relevant conditional branches as the seed input. If this constraint is satisfiable, the solution typically immediately provides an input that will trigger an overflow at the site. Because of blocking checks, this constraint is unsatisfiable for all but two of the sites, specifically SwfPlay 0.5.5 at jpeg.c@192 and CWebP 0.3.1 at jpegdec.c@248.

Application	Target	CVE Number	Error Type	Analysis and Discovery Time	Enforced Branches	Target Success Rate	Target + Enforced Success Rate
Dillo 2.1	png.c@203	CVE-2009-2294	SIGSEGV/InvalidRead	(42m) 8m	4/35	0/200	190/200
Dillo 2.1	fltkimagebuf.cc@39	New	SIGSEGV/InvalidRead	(42m) 7m	5/69	0/200	189/200
Dillo 2.1	Image.cxx@741	New	SIGSEGV/InvalidRead	(42m) 7m	4/5779	0/200	190/200
VLC 0.8.6h	messages.c@355	New	SIGSEGV/InvalidRead	(6m) 1m	2/117	32/200	108/200
VLC 0.8.6h	wav.c@147	CVE-2008-2430	InvalidRead/Write	(6m) 1m	0/62	2/2	N/A
VLC 0.8.6h	dec.c@277	New	SIGSEGV/InvalidRead	(6m) 8m	5/291	57/200	97/200
VLC 0.8.6h	block.c@54	New	InvalidRead	(6m) 4m	0/151	200/200	N/A
SwfPlay 0.5.5	jpeg_rgb_decoder.c@253	New	SIGSEGV/InvalidWrite	(7m) 13m	0/1736	200/200	N/A
SwfPlay 0.5.5	jpeg_rgb_decoder.c@257	New	SIGSEGV/InvalidWrite	(7m) 13m	0/1736	200/200	N/A
SwfPlay 0.5.5	jpeg.c@192	New	SIGABRT/InvalidWrite	(7m) 1m	0/1012	200/200	N/A
CWebP 0.3.1	jpegdec.c@248	New	SIGSEGV/InvalidWrite	(11m) 2s	0/651	155/200	N/A
ImageMagick 6.5.2	xwindow.c@5619	CVE-2009-1882	SIGSEGV/InvalidWrite	(6m) 1m	0/2521	200/200	N/A
ImageMagick 6.5.2	cache.c@803	New	SIGSEGV/InvalidWrite	(6m) 1m	0/306	199/200	N/A
ImageMagick 6.5.2	display.c@4393	New	SIGSEGV/InvalidWrite	(6m) 2m	0/154	200/200	N/A

Table 2: Evaluation Summary

5.5 Inputs That Satisfy Target Constraint Alone

The seventh column (Target Success Rate) presents the results from the experiment in which DIODE generated 200 inputs that satisfied the target constraint by itself (with none of the conditional branch constraints added to the target constraint passed to the solver). Note that all of these inputs will trigger an overflow at the target memory allocation site if they follow a path that evaluates the target expression at that site. Note also that every discovered input that triggers the overflow is in the set of inputs that satisfy the target constraint alone and therefore could potentially be generated as one of the sampled 200 inputs.

Each entry in the column is of the form X/200, where X is the number of generated inputs that actually trigger the overflow. We note that there is a bimodal distribution — in general, either all or the vast majority of the 200 generated inputs trigger the overflow or none or few of the 200 generated inputs trigger the overflow. This bimodal distribution is correlated with the presence or absence of sanity checks on relevant input values — without sanity checks, all or the vast majority of the generated inputs trigger the overflow. If the application contains sanity checks, the generated inputs are unlikely to pass the sanity checks to trigger the overflow. These data indicate that, if the application contains sanity checks and the input generation strategy does not take these checks into account, the input generation strategy is unlikely to find inputs that trigger an overflow (even when such inputs exist).

For CVE-2008-2430, the target expression is of the form $x+2$, where x is an input field. The target constraint for this expression has only two solutions (because there are only two values of x that cause the target expression to overflow).

5.6 Target and Enforced Branch Success Rate

The eighth column (Target + Enforced Success Rate) presents experimental results for those overflows that DIODE discovered only after enforcing some of the conditional branches. DIODE generated 200 inputs that satisfied the corresponding constraint (i.e., the target constraint plus the constraints that enforced the discovered first flipped branches in Algorithm 7). Each entry in the column is of the form X/200, where X is the number of generated inputs that trigger the overflow (note that we do not run this experiment if the majority of the inputs that satisfy the target constraint alone also trigger the overflow).

We note that, for three of the five overflows, the vast majority of the generated inputs trigger the overflow. For the remaining two overflows, approximately half of the generated inputs trigger the overflow. We attribute this success to DIODE's ability to produce inputs that satisfy the sanity checks while preserving their flexibility to satisfy the blocking checks and traverse alternate paths through the computation to reach the target memory allocation site and trigger the overflow.

The success of DIODE in generating these overflows also illustrates the difficulty of writing sanity checks that detect inputs that cause overflows — even though Dillo 2.1 and VLC 0.8.6h contain sanity checks, these checks do not detect all inputs that trigger overflows.

6. Related Work

Random and Directed Fuzzing: Random fuzzing has been shown to be surprisingly effective in uncovering errors [24, 38] and is heavily used by security researchers [4, 6, 30]. But because most randomly generated inputs fail input sanity checks, random fuzzing has been relatively ineffective at generating inputs that trigger errors (such as integer overflows) deep inside applications. Their ability to generate such inputs can be especially limited for programs that process deeply structured formats such as videos.

Motivated by the need to expose errors deep inside applications, researchers have proposed directed fuzzing techniques [15, 16, 40]. BuzzFuzz [16] and TaintScope [40] use taint tracking to identify input bytes that influence values at critical program sites such as memory allocation sites and system calls. In contrast with random fuzzing techniques that modify the entire input, these techniques then fuzz the input bytes that influence critical program points. While successful at reducing the size of the mutation space, our results indicates that these directed techniques are ineffective at finding the carefully crafted inputs required to navigate the sanity checks and

expose integer overflow errors. Because these directed fuzzing systems operate directly on the raw binary input bytes, the modifications can also produce syntactically incorrect inputs that immediately fail the sanity checks.

Symbolic Test Generation: Symbolic test generation (i.e., concolic testing) has been proposed as an alternative to random and directed fuzzing [9, 10, 17–19, 25, 33, 39]. These systems execute programs both concretely and symbolically on a seed input until an interesting program expression is reached (e.g., an assert, a conditional or a specific expression). Although successful in many cases [9, 10, 18, 25], symbolic test generation faces several challenges [8, 34]. Specifically, once past the initial parsing stages, the resulting deeper program paths may produce very large constraints with complex conditions that are currently beyond the capabilities of current state of the art constraint solvers. Our results also show that the path taken by a seed input may contain additional blocking checks that can prevent a constraint solver from generating inputs that satisfy the checks and trigger an overflow.

SmartFuzz [25] is a symbolic test generation tool to discover integer overflows, non-value-preserving width conversions, and potentially dangerous signed/unsigned conversions. SmartFuzz, like other concolic systems, is limited by deep program paths and blocking checks.

Dowser [19] is a fuzzer that combines taint tracking, program analysis, and symbolic execution to find buffer overflows. The key idea is to use program analysis to guide symbolic execution (e.g.,KLEE [9]) along a path that is more likely to discover buffer overflows than running symbolic execution over the entire program. Like most concolic systems, Dowser optimizes for path coverage and is thus unlikely to discover integer overflow errors.

DIODE differs from all of these techniques in that it is *targeted* — instead of exploring paths to find critical sites, it starts with a critical site that is executed by a seed input, then uses a variety of techniques that are designed to produce inputs that successfully navigate sanity and blocking checks to trigger an overflow at the critical site.

Runtime and Library Support: To alleviate the problem of false positives, several research projects have focused on runtime detection tools that dynamically insert runtime checks before integer operations [7, 14, 42]. Another technique is to use safe integer libraries such as SafeInt [5] and CERT's IntegerLib [32] to perform sanity checks at runtime. Using these libraries requires developers to rewrite existing code to use safe versions of integer operations. DIODE, in contrast, pro-actively finds integer errors during testing (i.e., does not rely on observing a malicious input in the wild) and imposes no runtime overhead.

Input Rectification is another technique that can protect applications from integer overflow errors [20, 21, 28] by empirically learning input constraints from benign training inputs and then monitoring inputs for violations of the learned constraints. Instead of discarding inputs that violate the learned constraints, input rectification modifies the input so that it satisfies the constraints. The goal is to nullify potential errors while still enabling the program to successfully process as much input data as possible. Because it learns the constraints from examples, the technique is susceptible to false positives. In contrast, DIODE has no false positives and can proactively discover integer overflow errors.

Static Analysis For Finding Integer Errors: Several static analysis tools have been proposed to find integer overflow and/or sign errors [11, 31, 41]. KINT [41], for example, analyzes individual procedures, with the developer optionally providing procedure specifications that characterize the value ranges of the parameters. Despite substantial effort, KINT reports a large number of false positives [41]. In contrast, DIODE generates inputs that prove the existence of integer errors without any false positives.

SIFT [22] is a system for generating input filters that nullify integer overflow errors associated with critical program sites such as memory allocation and block copy sites. SIFT uses a sound static program analysis to generate filters that discard inputs that may trigger overflow errors. SIFT requires access to source code and is not designed to identify errors. DIODE, in contrast, operates directly on stripped x86 binaries with no need for source code to generate overflow-triggering inputs.

7. Conclusion

We present a system, DIODE, for discovering integer overflow errors in real-world applications. DIODE is designed to work with programs that first perform sanity checks on relevant input fields, then use the input fields to compute target values such as the sizes of allocated memory blocks. Leveraging this pattern, DIODE generates inputs that are designed to satisfy the relevant sanity checks, generate an overflow in the target expression, and impose no other constraints on the specific path that the input takes to trigger the overflow. Our experimental results show that DIODE is effective at generating inputs that trigger integer overflow errors, including previously unknown errors in widely used applications.

Acknowledgements

We thank Jeff Perkins, Michael Gordon and the anonymous reviewers for their insightful comments. This research was supported by DARPA (Grant FA8650-11-C-7192).

References

[1] Common vulnerabilities and exposures. http://cve.mitre.org/.

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

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

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

[5] SafeInt. http://safeint.codeplex.com/.

[6] SPIKE fuzzing platform. http://www.immunitysec.com/resources-freesoftware.shtml.

[7] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song. RICH: Automatically protecting against integer-based vulnerabilities. *Department of Electrical and Computing Engineering*, page 28, 2007.

[8] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 143–157. IEEE, 2008.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[11] E. Ceesay, J. Zhou, M. Gertz, K. Levitt, and M. Bishop. Using type qualifiers to analyze untrusted integers and detecting security flaws in C programs. *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 1–16, 2006.

[12] C. Cowan, H. Hinton, C. Pu, and J. Walpole. The cracker patch choice: An analysis of post hoc security techniques. 2000.

[13] 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.

[14] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 760–770. IEEE Press, 2012.

[15] W. Drewry and T. Ormandy. Flayer: Exposing application internals. In *Proceedings of the first USENIX workshop on Offensive Technologies*, pages 1–9. USENIX Association, 2007.

[16] 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.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[18] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[19] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX conference on Security*, pages 49–64. USENIX Association, 2013.

[20] 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, Piscataway, NJ, USA, 2012. IEEE Press.

[21] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard. Automatic input rectification. MIT-CSAIL-TR-2011-044.

[22] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard. Sound input filter generation for integer overflow errors. 2014.

[23] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 26. ACM, 2014.

[24] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[25] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *Proceedings of the 18th conference on USENIX security symposium*, pages 67–82. USENIX Association, 2009.

[26] 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.

[27] 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.

[28] M. Rinard. Acceptability-oriented computing. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '03) Companion, Onwards! Session*, Anaheim, California, Oct. 2003.

[29] M. C. 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*, volume 4, pages 21–21, 2004.

[30] J. Röning, M. Lasko, A. Takanen, and R. Kaksonen. PROTOS — systematic approach to eliminate software vulnerabilities. *Invited presentation at Microsoft Research*, 2002.

[31] D. Sarkar, M. Jagannathan, J. Thiagarajan, and R. Venkatapathy. Flow-insensitive static analysis for detecting integer anomalies in programs. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, pages 334–340. ACTA Press, 2007.

[32] R. Seacord. *The CERT C Secure Coding Standard*. Addison-Wesley Professional, 2008.

[33] K. Sen, D. Marinov, and G. Agha. *CUTE: A Concolic Unit Testing Engine for C*, volume 30. ACM, 2005.

[34] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[35] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: automatic software self-healing using rescue points. *ACM SIGARCH Computer Architecture News*, 37(1):37–48, 2009.

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

[37] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, P. Piselli, and M. Rinard. Automatic error elimination by multi-application code transfer. Technical Report MIT-CSAIL-TR-2014-024, MIT CSAIL, August 2014.

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

[39] W. Tielei, W. Tao, L. Zhiqiang, and Z. Wei. IntScope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution. In *16th Annual Network & Distributed System Security Symposium*, 2009.

[40] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 31st IEEE Symposium on Security & Privacy (Oakland'10)*, 2010.

[41] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. Kaashoek. Improving integer security for systems with KINT. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 163–177. USENIX Association, 2012.

[42] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou. IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. *Computer Security–ESORICS 2010*, pages 71–86, 2010.