Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement

by

Nathan Rittenhouse

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Masters of Engineering in Computer Science at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2014

© Massachusetts Institute of Technology 2014. All rights reserved.

Signature redacted

Author

Department of Electrical Engineering and Computer Science

January 29, 2014

Signature redacted

Certified by ....

Martin C. Rinard
Professor of Computer Science, MIT Computer Science and Artificial Intelligence Laboratory
Thesis Supervisor

Signature redacted

Accepted by ....

Albert R. Meyer
Undergraduate Officer, Department of Electrical Engineering and Computer Science
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, for every target memory allocation site, 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.

Thesis Supervisor: Martin C. Rinard
Title: Professor of Computer Science, MIT Computer Science and Artificial Intelligence Laboratory
Acknowledgments

I would like to thank my advisor, Professor Martin Rinard for his help in introducing me to program analysis and his valuable feedback on our work. I would especially like to thank my supervisor Stelios Sidiropoulou-Douskous for his excellent leadership of the project. I would also like to thank Deokhwan Kim and Fan Long as well as our research assistants Paolo Piselli and Eric Lahtinen for their collaboration on the project. Further, I would like to extend my thanks to all of the individuals of Professor Rinard’s group for their valuable feedback and creating a stimulating research environment.
Contents

1 Introduction .................................................. 15
   1.1 Prevalence of Integer Overflow Vulnerabilities ............... 15
   1.2 Integer Overflows ........................................... 15
   1.3 Integer Overflow Bugs ......................................... 17
   1.4 Example - Integer Overflow Vulnerability ..................... 17
   1.5 Defenses .................................................... 19
      1.5.1 Run-time Detection ...................................... 19
      1.5.2 Static Analysis .......................................... 19
      1.5.3 Testing .................................................. 19
      1.5.4 Directed Approach ...................................... 21
   1.6 Experimental Results ......................................... 22
   1.7 Outline ..................................................... 22

2 Example Blocker Constraints .................................. 25
   2.1 Loop Induced Constraints ..................................... 25
   2.2 Checksums ................................................... 26
   2.3 Optimized Block Copies ...................................... 27

3 Goal-Directed Conditional Branch Enforcement Algorithm .... 29
   3.1 Core Language ................................................. 29
   3.2 Operational Semantics ........................................ 30
   3.3 Constraint Representation .................................... 33
   3.4 Algorithm .................................................... 33
# System Design and Implementation

4.1 Target Site Identification ........................................... 37
4.2 Target and Branch Constraint Extraction ......................... 38
4.3 Target Constraint Solution ......................................... 39
4.4 Test Input Generation ................................................ 39
4.5 Goal-Directed Branch Enforcement ................................. 40
4.6 Error Detection ....................................................... 40

# Constraint Transformations

5.1 Target Constraint Construction .................................... 41
5.2 Constraint Fixups .................................................... 41
   5.2.1 Implementation ................................................. 42
   5.2.2 Bit-width Promotion .......................................... 42
   5.2.3 Signedness Constraints ....................................... 43
   5.2.4 Boolean Type Error Fixups .................................. 43
   5.2.5 Preventing Divide-By-Zeros ................................ 45

# Evaluation

6.1 Target Site Classification ......................................... 47
6.2 Overflow Characteristics .......................................... 48
6.3 Inputs That Satisfy Target Constraint Alone .................... 49
6.4 Target and Enforced Branch Success Rate ....................... 50
6.5 Case: Dillo - png.c@203 ......................................... 51
6.6 Case: ImageMagick Display - xwindow.c @ 5619 ............... 51

# Prior Work

7.1 BuzzFuzz ............................................................. 53
7.2 SAGE ............................................................... 54
7.3 SmartFuzz .......................................................... 55
7.4 Automatic Patched Based Exploit Generation .................... 55
7.5 KINT ................................................................. 56
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.6 KLEE</td>
<td>56</td>
</tr>
<tr>
<td>8 Conclusion</td>
<td>59</td>
</tr>
<tr>
<td>A Tables</td>
<td>61</td>
</tr>
<tr>
<td>B Figures</td>
<td>63</td>
</tr>
</tbody>
</table>
List of Figures

1-1 Example of legitimate overflow check that needs integer overflow for correctness ........................................ 16
1-2 Example of an integer overflow bug and potential vulnerability ................................................................. 18

2-1 Example of loop that depends on a tainted variable ................................................................................... 26
2-2 Example of file parsing source code ........................................................................................................... 26
2-3 An excerpt from glibc’s optimized memcpy ................................................................................................ 28

3-1 Syntax ......................................................................................................................................................... 30
3-2 Semantics of Arithmetic Expressions ........................................................................................................ 32
3-3 Small-Step Operational Semantics of Statements ..................................................................................... 32
3-4 Small-Step Operational Semantics of Statement Sequence ...................................................................... 32
3-5 Goal-Directed Conditional Branch Enforcement ....................................................................................... 33
3-6 Branch Condition Compression .................................................................................................................. 34

5-1 Constructing bug queries for integer overflows ......................................................................................... 42
5-2 Fixup function for signedness constraints ................................................................................................ 43
5-3 Predicate for signedness fixups .................................................................................................................. 43
5-4 Predicate for boolean inside of bitvector fixup ........................................................................................... 44
5-5 Fixup function for boolean inside of bitvector type error ......................................................................... 44
5-6 Fixup function for bitvector inside of boolean type error ........................................................................... 44
5-7 Predicate for bitvector inside of boolean fixup ........................................................................................... 44
5-8 Predicate for divide-by-zero fixup ............................................................................................................... 45
5-9 Fixup function for avoiding divide-by-zero ................................................................................................ 45
6-1 Target Site Classification ............................................. 47
6-2 Experimental results .................................................. 50

B-1 Code snippet from Dillo PNG parser .............................. 63
B-2 Code for Dillo integer overflow vulnerability .................. 64
B-3 Code for Dillo CRC Checksum ...................................... 65
B-4 Code for ImageMagick integer overflow vulnerability ........ 66
List of Tables

A.1 Bitvector constraints ................................................. 62
A.2 Boolean constraints .................................................. 62
Chapter 1

Introduction

We present DIODE, a directed approach for discovering integer overflow bugs. We establish the motivation for discovering integer overflow bugs by discussing their prevalence, danger, and the difficulties associated with detecting them. We then describe previous methods to detect integer overflows and our directed approach along with its experimental results.

1.1 Prevalence of Integer Overflow Vulnerabilities

Systems level vulnerabilities that result from integer overflow constitute one of the most prominent types of software vulnerability to date. In 2007 study performed on the Common Vulnerabilities and Exposures (CVE) database [12] showed that integer overflow vulnerabilities were the second most prevalent type of software vulnerability aside from integer overflows. In CVEs that pertained to the Linux kernel from 2010 to 2011, integer overflow vulnerabilities were found to constitute one third of the entire corpus [10].

1.2 Integer Overflows

Most lower level programming languages constrain integer types to a size specified by a programmer to improve performance. Integer overflows result when the result of an integer calculation becomes too large for its allotted bit-width and is truncated to fit in the bit-width of the specified type. In the C programming language, overflows in unsigned integer
calculations will wrap modulo $2^n$ where $n$ is the bit-width of the type of the expression. The result is a smaller value than would be found if the expression type had an infinite bit-width.

The C and C++ programming languages do not offer infinite bit-width types for integer calculations. Libraries such as SafeInt [4] aim to replace C integer operations with safe versions. However, they are not widely used, as they negatively impact performance and have been susceptible to overflow bugs in the past[14].

Many cases of software bugs involve improper handling of integer overflows. However, it is not the case that all integer overflows are bugs. As a consequence, false positives are a common problem for many discovery methods.

```c
unsigned int width;
unsigned int height;
unsigned int area;

area = width*height;

if(area/width != height){
    // Raise error
}
```

Figure 1-1: Example of legitimate overflow check that needs integer overflow for correctness

One source of legitimate and intentional uses of integer overflows are integer overflow sanity checks themselves. Figure 1-1 shows a canonical example - determining whether an image has valid dimensions. In one interpretation, valid image dimensions should result in an image area size that the application can hold in the bit-width allotted for the variable holding its area. To determine whether the combination of `width` and `height` are valid, `area` is calculated to be a result of their product. The code reports an error if the calculation overflows- the equality check on `area` / `width` will not hold, as `area` will be truncated and thus less than its expected value.

Other intentional uses are found in hashing algorithms, cryptographic code, and pseudo-random number generators[14, 21].
1.3 Integer Overflow Bugs

Although it is not the case that all integer overflows are bugs, unintended integer overflows can result in disastrous consequences when used as parameters for security critical operations such as data copies, array accesses, and memory allocations. Subsequent code may corrupt data or worse, allow malicious code injection.

Security critical operations can take many forms:

- **Data Copies** Copy operations often depend on a source parameter denoting the source of the data, a dest parameter denoting the destination of the copy, and a size parameter that specifies the size of the copy. An overflowing size parameter could result in less data copied than intended to the dest buffer. Effects such as uninitialized memory reads may result.

- **Array Accesses** An array index calculation susceptible to overflow can result in, at best, the calculation of an index within the bounds of the array that the programmer did not intend. However, certain cases also allow negative array indexes. Such a problem may arise when an index calculation is the result of a subtraction where an overflowable minuend wraps to a value smaller than its subtrahend. In an array write, this can lead to data corruption and possibly remote code execution.

- **Memory Allocations** Memory size calculations susceptible to overflow may allocate a smaller buffer size than the programmer anticipated. This can have disastrous security consequences. The buffer fill operation is often not susceptible to overflow and thus may perform a copy larger in size than the buffer and a buffer overflow is the result. Leveraging stack and heap based buffer overflows into security exploits has been discussed to great length [23, 20, 7, 19].

1.4 Example - Integer Overflow Vulnerability

We present a code sample in figure 1-2 that illustrates a common integer overflow vulnerability pattern - the memory allocation size is susceptible to overflow but the data
// Allocate a buffer to hold a pointer to each row

unsigned int rows;
unsigned short byte_depth;
unsigned int display_byte_depth;
unsigned int height;

// Read rows, bit_depth, and height from file.
// ...

// Set display_bit_depth to bit depth supported by graphics back end
// ...

unsigned long *image_data = malloc(rows*sizeof(unsigned long)*height);

unsigned int i;
unsigned int j;
char current_bytes[4];

for(i = 0; i < height; i++){
    for(j = 0; j < width; j++){
        read_image.next(current_bytes, byte_depth);
        image_data[i][j] = convert_bit_depth(
            current_bytes,
            bit_depth,
            byte_depth);
    }
}

Figure 1-2: Example of an integer overflow bug and potential vulnerability

copy operation is not. In this example, the width, height, and bit depth of an image are read from a file and then multiplied together to calculate the number of bytes needed to store the image. Each byte_depth size block of the image is re-sized to the bit depth supported by the computer's graphics system and subsequently copied into the image_data buffer.

The size of image_data on line 9 is susceptible to overflow but the loop condition is not. For inputs that overflow the size calculation, a buffer less than width * height * sizeof(unsigned long) bytes in size will be allocated for the image, but exactly that number of bytes will be written into the buffer - a potentially large out of bounds memory write.
1.5 Defenses

Due to the danger of integer overflow bugs, a diverse set of approaches has arisen to detect them. Runtime checks alert a programmer if an integer overflow occurred in a calculation. Static analysis tools flag potential integer overflow sites through imprecise analyses. Due to the propensity of these techniques to yield false positives and their inability to provide developers with actionable reports (e.g. they provide no test inputs), automated testing tools have become a very popular method for detecting integer overflow bugs.

1.5.1 Run-time Detection

One approach to detecting integer overflows is to alert their occurrence at run-time. Some approaches insert the checks at compile time [14] and others dynamically [11]. Because of their inability to distinguish intended overflows from unintended overflows, these approaches report many false positives. Dynamic checking incurs an especially high performance penalty.

1.5.2 Static Analysis

An alternative approach elects to detect integer overflows at compile time [25]. Static analysis tools attempt to statically detect integer overflow bugs without running the program or attempting to follow all program paths. Thus there is a trade off of precisely modeling the program for speed.

The path insensitive design of these tools yields two serious problems: 1) false positives and 2) they are unable to generate inputs that can trigger the vulnerable condition.

1.5.3 Testing

The number of false positives reported by static analysis tools and their inability to generate test inputs gave rise to the popularity of automated testing. Automated testing tools aim to generate test inputs that pass program sanity checks, reach the target site (the program point with a security critical operation), and cause its target value (parameter) to overflow. This
approach more precisely models program behavior at the cost of only discovering a subset of all integer overflow bugs within a program.

**Mutation Based Fuzzing**

Mutation based fuzzing starts with a *seed input* and performs random modification of bytes in the input to obtain a *test input* [6]. More evolved forms of this technique use dynamic taint analysis to determine the parts of a seed input that are used in a security critical operation and only modify those ranges of the file [24, 16].

While fuzzing tools have successfully found bugs in the past, they need to run for a large number of iterations. Because they randomly choose modifications, they produce test inputs that satisfy *sanity checks* within a program with low probability. Inputs that do not pass program sanity checks are often denied.

**Symbolic Execution**

Symbolic execution derives inputs that satisfy sanity checks by deriving inputs to exercise every code block in the target application [9]. These tools run the application, beginning from its entry point, inside of an interpreter that treats inputs as symbolic variables. When the interpreter reaches a *tainted branch* (that depends on user input), it will attempt to derive inputs that exercise both branch directions until it discovers a security critical operation. When this occurs, the tool derives a *target constraint* that is satisfied only when the target value overflows. The conjunction of constraints needed to exercise the tainted branches and target constraint are solved to yield a test input.

This approach can find integer overflow bugs in a program's initial input processing step. However, reliance on a breadth first search through tainted program branches from the start of the program causes this approach to miss deeper bugs due to the *path explosion problem* - a program with *n* tainted branches will have $2^n$ possible paths.
Concolic Execution

Concolic execution attempts to reach target sites deep in the program by generating test inputs that diverge from the program path followed when the application is fed a seed input [17, 18, 21]. It discovers new test inputs through generational search - each new test input is formed from the solution to conjunction of the negation of a tainted branch constraint with the preceding tainted branch constraints.

Concolic execution is more equipped to reach deep target sites than symbolic execution. However, it is not optimized for finding integer overflow bugs. It is often the case that blocker constraints (tainted branch constraints that conflict with the target constraint) can stymie efforts to satisfy a target constraint - the generational search will have to find another path to the target site, which still takes exponential time. Blocker constraints come from a myriad of sources including control flow statements that govern loop iterations, size checks in block-size aware code, and checksum validations.

1.5.4 Directed Approach

Although concolic and symbolic execution tools begin their analysis from different points, both types of tools effectively perform an exponential-time search through tainted branches to derive inputs that could exercise any target site. We propose a directed approach - we select target sites obtained from the program path followed when fed the seed input and attempt to select the subset of those constraints necessary to pass the sanity checks and trigger the overflow at the target site.

Filtering Blocker Constraints

One approach to filtering blocker constraints is to simply try every subset of tainted branch constraints present in the path. However, this technique is impractical, as there are $2^n$ combinations of $n$ constraints. Our results show that there are often hundreds of constraints that share input variables with the target expression.
Directing Concolic Execution

As filtering the blocker constraints through subset iteration is impractical, we create a form of concolic execution that is optimized to uncover integer overflow bugs. Whereas concolic execution includes all constraints that depend on user input, we consider only those that include relevant inputs (those used to calculate the target value) and are not characteristic of loop-back edges. We first solve the target constraint and derive a test input. If this input does not trigger the bug, we repeatedly enforce conditional branches from the set collected during the seed run using the goal-directed algorithm described in section 3 to constrain test inputs to exercise the target site.

1.6 Experimental Results

We present experimental results that characterize the effectiveness of DIODE, the tool that implements our directed approach, in finding integer overflow bugs. For our benchmark applications, DIODE discovers 14 overflows, 11 of which are new. For nine 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 five overflows, DIODE discovers the overflow after enforcing a modest (two to five) number of conditional branches. We attribute this success to the ability of DIODE to successfully identify and satisfy the relevant sanity checks that appear in these programs.

Our results also indicate that, for these benchmarks, DIODE generates inputs that trigger every overflow that exists for the observed target expressions at the observed target sites.

1.7 Outline

In section 2, we describe our motivation for developing a directed approach with examples of blocker constraints that prevent same-path approaches from accurately identifying vulnerable target sites. Subsequently, we describe the directed approach to constraint selection in section 3 and the design of the implementation, DIODE, in section 4. We explain the
constraint transformations necessary to process the constraints in section 5. We demonstrate
the success of DIODE in section 5 and compare it with other integer overflow discovery
tools in section 7. Finally, conclude in section 8.
Chapter 2

Example Blocker Constraints

We provide motivation for the creation of a directed approach by describing examples of blocker constraints that would cause concolic execution tools to search for alternate paths to the target site.

2.1 Loop Induced Constraints

Loops that depend on relevant inputs will over-constrain the solutions. In many loops of this kind, loop iteration check will yield a successively higher lower bound on the value of the expression which includes the security parameter. After the last loop iteration, the loop construct will yield a loop termination constraint that asserts that the expression has an upper bound that is only slightly larger than the last lower bound. In effect, this results in the overflow relevant variable assignment being constrained to only a small number of values.

In figure 2-1, each loop iteration will yield a constraint that asserts that width is greater than the current value of the iterator. The last constraint emitted will assert that \((i = width-1) < width\). Before the next run of the loop, a constraint asserting \(\text{not}((i = width) < width))\) or \((i = width) >= width\), thus the value of width can only be assigned the value present in the seed file. If other relevant inputs are sufficiently constrained, a concolic execution tool will be unable to find a bug along this path because the inputs will be artificially over-constrained.
1 // Allocate a buffer to hold a pointer to each row
2 void *columns = malloc(rows*sizeof(void*));
3
4 // Allocate a buffer for each row
5 for(int i =0; i<width;i++){
6    columns[i] = malloc(height*pixel_depth);
7 }
8
9 // Convert it to a canvas that is to be drawn on a screen
10 void * canvas = malloc(width*height*pixel_depth);

Figure 2-1: Example of loop that depends on a tainted variable

2.2 Checksums

In figure 2-2, a file field's checksum is recalculated and checked before processing occurs. Testing tools must give special consideration to checksum validations. Mutation based fuzzers that are not checksum aware will have all of their test inputs discarded as their modifications will lead to the calculation of a different checksum than present in the file. Concolic and symbolic execution tools are greatly slowed down by the characteristic branch constraints of checksums, which are difficult (often intentionally) to solve.

1 if (checksum(file_field) != file_field.checksum)
2     goto error;
3
4 process_field(file_field);
5
6 error:
7     // ...
8    exit(0);

Figure 2-2: Example of file parsing source code

This pattern is common to many file parsing applications. Checksums must be given special consideration because their characteristic constraints are often difficult to solve (often intentionally).
2.3 Optimized Block Copies

It is often the case that parser libraries will perform data copies of inputs using relevant inputs as size parameters before an application uses the data. Optimized data copy functions can yield blocker constraints when used in this context. Recent versions of libc contain memcpy and memset implementations that make heavy use of vectorization. Due to the fact that vectorization instructions take a longer time than simple copy instructions, these implementations only use the vectorized copy implementations when the copy size is large.

The optimized memcpy in figure 2-3 has the potential to prevent symbolic tools from discovering integer overflows. It first checks whether the size is less than 48 bytes. Subsequently, it checks whether the sum of the size and the summand needed to align the destination pointer on a 4-byte boundary is less than half of the shared cache size.

The first constraint has the potential to prevent integer overflow detection on inputs that yield a small size argument as the check will constrain the size to be less than 48. The latter check may also prevent an overflow - a seed input that yields a large size will have the low 32 bits of that size constrained to be large, which may not be possible.

We established that there are numerous sources of blocker constraints. In the next section, we describe goal-directed branch constraint enforcement, our approach to avoiding blocker constraints.
memcpy(char * dst, char * src, size_t len)
// ...
    movl LEN(%esp), %ecx
L(copy_forward):
    #endif
    cmp $48, %ecx
    jae L(48bytesormore)
L(48bytesormore):
    movdqu (%eax), %xmm0
    PUSH (%edi)
    movl %edx, %edi
    and $-16, %edx
    PUSH (%esi)
    cfi_remember_state
    add $16, %edx
    movl %edi, %esi
    sub %edx, %edi
    add %edi, %ecx
    sub %edi, %eax
    #ifdef SHARED_CACHE_SIZE_HALF
    cmp $SHARED_CACHE_SIZE_HALF, %ecx
    #else
    // ...
    #endif
    // ...
    jae L(large_page)

Figure 2-3: An excerpt from glibc's optimized memcpy
Chapter 3

Goal-Directed Conditional Branch Enforcement Algorithm

We next present the basic goal-directed conditional branch enforcement algorithm that is the core of our directed approach. 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.

We note that there are many complex engineering issues that DIODE must successfully navigate to operate successfully on stripped x86 binaries. We discuss these issues in Section 4.

3.1 Core Language

Figure 3-1 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 \( \in \text{PgmVar} \) is a conventional variable and can store integer values or memory addresses as usual. On the other hand, an input variable \( \in \text{InpVar} \) 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 \( \ell \in \text{Label} \). \( \text{before}(C) \) and \( \text{after}(C) \) denote the labels before and after the statement \( C \), respectively. In a sequence \( S = C_1 ; \cdots C_n \), \( \text{after}(C_i) = \text{before}(C_{i+1}) \). We define \( \text{before}(C_1 ; \cdots ; C_n) \) and \( \text{after}(C_1 ; \cdots ; C_n) \) as follows:

\[
\begin{align*}
\text{before}(C_1 ; \cdots ; C_n) &= \text{before}(C_1) \\
\text{after}(C_1 ; \cdots ; C_n) &= \text{after}(C_n)
\end{align*}
\]

### 3.2 Operational Semantics

The language has three different kinds of values

\[
\begin{align*}
n &\in \text{Int} \\
b, b_1, b_2 &\in \text{Bool} = \{ \text{true}, \text{false} \} \\
a &\in \text{Addr}
\end{align*}
\]

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 \( \rho \in \text{Env} \) is a partial mapping from variables to pairs of values and symbolic values. A value \( v \in \text{Val} \) is either an integer or an memory address. A symbolic value \( w \in \text{SymVal} \) 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.

\[ \rho \in \text{Env} = \text{Var} \rightarrow \text{Val} \times \text{SymVal} \]
\[ v, v_1, v_2 \in \text{Val} = \text{Int} \cup \text{Addr} \]
\[ w, w_1, w_2 \in \text{SymVal} = \text{Int} \cup \text{Addr} \cup \text{Aexp} \]

Similar to an environment, a memory \( m \in \text{Mem} \) 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.

\[ m, m_1, m_2 \in \text{Mem} = \text{Addr} \rightarrow \text{Offset} \rightarrow \text{Val} \times \text{SymVal} \]

A branch condition \( \phi \in \text{BranchCond} \) 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 \( \ell \). The elements appear in \( \phi \) in the program execution order.

\[ \phi \in \text{BranchCond} := \epsilon \mid \langle \ell, B \rangle \rightarrow \phi \]

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

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

**Expressions:** Figure 3-2 presents the semantics of arithmetic expressions. Each expression evaluates to a pair \( \langle v, w \rangle \), where \( v \in \text{Val} \) is a concrete value and \( w \in \text{SymVal} \) is a symbolic expression. The \text{INPVAR} rule, for example, defines that the evaluation of an input variable \( x \in \text{InpVar} \) 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:** Figure 3-3 and Figure 3-4 present the small-step operational semantics of DIODE's core language.
Figure 3-2: Semantics of Arithmetic Expressions

Figure 3-3: Small-Step Operational Semantics of Statements

Figure 3-4: Small-Step Operational Semantics of Statement Sequence
Input: a program $S$, an initial state $\sigma$, a target label $\ell$
Output: an input $I$ that triggers an integer overflow at label $\ell$

```
for $\langle B, \phi \rangle$ in target($\langle S, \sigma \rangle, \ell$) do
    $\beta \leftarrow$ overflow($B$)
    if the solver generates an input $I$ that satisfies $\beta$ then
        if the input $I$ triggers an overflow at label $\ell$ then
            return the input $I$
        else continue
    $\phi \leftarrow$ compress($\phi$)
    $\phi' \leftarrow$ relevant($\phi, \beta$)
while true do
    if the previous input $I$ satisfies $\phi$ then break
    $\phi' \leftarrow$ $\phi' \land$ (the first condition in $\phi$ that the previous input $I$
        does not satisfy)
    if the solver generates an input $I$ that satisfies $\phi' \land \beta$ then
        if the input $I$ triggers an overflow at label $\ell$ then
            return the input $I$
        else break
return not found
```

Figure 3-5: Goal-Directed Conditional Branch Enforcement

### 3.3 Constraint Representation

We support boolean and bitvector constraints. Bitvector constraints are defined A.1 for both signed and unsigned operations. Bit-width truncations and expansions can be specified with the Width constraint. MallocArg constraints allow one to specify a target expression. Boolean constraints, defined in A.2 are defined for boolean arguments as well as bitvector arguments to capture C truth value semantics.

### 3.4 Algorithm

Figure 3-5 presents the DIODE goal-directed conditional branch enforcement algorithm. Given a program $S$, an initial program state $\sigma$, and a target site $\ell$, the algorithm first extracts the symbolic target expression $B$ and the observed path $\phi$ (from the seed input) for that site.
FUNCTION compress($\phi$) =

begin

\[ \text{if } \phi \text{ is } \varepsilon \text{ then} \]

\[ \text{return } \varepsilon \]

\[ \text{else if } \phi \text{ is } \langle \ell, B \rangle \mapsto \phi \text{ then} \]

\[ B \leftarrow B \land (\bigwedge \langle \ell, B' \rangle \in \phi B') \]

\[ \phi \leftarrow \text{filter out all } \langle \ell, B' \rangle \text{ from } \phi \]

\[ \text{return } \langle \ell, B \rangle \mapsto \text{compress}(\phi) \]

Figure 3-6: Branch Condition Compression

(line 1). target($\langle S, \sigma \rangle, \ell$) is defined as follows:

\[ \text{target}(\langle S, \sigma \rangle, \ell) = \{ \langle \sigma_2(\rho(y)), \phi \rangle \mid \langle \sigma, \langle \ell, \rho, m, \phi \rangle \rangle \in \tau^*(S) \} \]

where $\ell = \text{before}(x = \text{alloc}(y))$

The function target($\langle S, \sigma \rangle, \ell$) is defined in terms of the reflexive transitive clousre $\tau^*(S)$ 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 $\beta$ (line 2). The overflow($B$) function returns a target constraint $\beta$ such that any input that satisfies the target constraint $\beta$ will trigger an overflow during the computation of the target expression $B$.

The algorithm next compresses the path $\phi$ to coalesce multiple occurrences of conditional branch constraints of a conditional statement into a single constraint (line 7 and Figure 3-6). This single constraint is the conjunction 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($\phi, \beta$) function takes a branch condition $\phi$ and a target constraint $\beta$ as its arguments, and removes conditions that are not relevant to the target constraint $\beta$ from the branch condition $\phi$. A condition $\langle \ell, B \rangle$ in a branch condition is relevant to a target constraint $\beta$ if the condition $B$ and the target constraint $\beta$ share the same input variable.

We described goal-directed branch enforcement, an alternative approach to path enu-
meration for uncovering integer overflow bugs. Next we describe the implementation of goal-directed branch enforcement, DIODE.
Chapter 4

System Design and Implementation

We next discuss the design of the implementation of our directed approach, DIODE. 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). First, we 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 [22] 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 program 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 (see 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 [1] 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. Transformations necessary to obtain the target constraint and preserve instruction semantics are covered in section 5. 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, there are no values that cause the following expression to overflow:

\[
((\text{width}_{16} \times \text{height}_{16}) \times 4)/\text{bbp}_8 > 2^{32} \text{ where } \text{bbp}_8 = \{8, 16, 32\}
\]

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

\[
((\text{width}_{16} \times \text{height}_{16}) \times 4)) > 2^{32}
\]

### 4.4 Test Input Generation

DIODE uses a combination of Hachoir [1] and Peach [3] 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 [24].
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 in 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. Our automated system first filters any errors that occur during the execution on the seed input.

We described the process used by DIODE to gather relevant constraints and minimize their size and its use of goal-directed branch enforcement to implement the directed approach. We next describe the constraint transformations necessary to model the overflow while preserving instruction semantics.

---

1For 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.
Chapter 5

Constraint Transformations

We next describe the transforms necessary for DIODE to correctly interpret constraints that describe the branch conditions and target constraints. We first present the algorithm used for constructing the target constraint which constrains inputs to those that cause the target value to overflow. We then describe the fixups needed for constraints to accurately model the semantics of the VEX intermediate representation.

5.1 Target Constraint Construction

In order to determine whether a target site is susceptible to overflow, each of the target’s subexpressions must be checked for overflow. To do this, we simply create a query for each subexpression which checks if the result can be greater the largest number its characteristic bit-width is capable of holding.

5.2 Constraint Fixups

DIODE constraints do not yield themselves to a straightforward conversion to Z3 input. One reason is DIODE must simultaneously model overflow by widening the bit-width of all variables (Z3 does allow formulae with mixed-bit-width terms) and maintain the signedness semantics and value ranges implied by the original bit-widths. Others arise due to friction between C semantics and the Z3 language.
**Input**: a constraint representation of an expression vulnerable to integer overflow and a path constraint \( p \)

**Output**: an assignment to variables that will cause some subexpression to overflow

```plaintext
for (subexpression c, bit-width b) in c(e) do
    \[ \beta \leftarrow c > 2^b \]
    if the solver generates an input \( I \) that satisfies \( \beta \land p \) then
        return the input \( I \)
    else continue
return not found
```

Figure 5-1: Constructing bug queries for integer overflows

### 5.2.1 Implementation

Constraints fixups are performed in two steps: 1) selection of constraints to fix and 2) constraint addition and replacement. Constraints are selected by performing a depth-first search through the constraint. Constraints that cause a specified predicate to return true have pointers to their reference in the parents argument list added to the selected constraint list. Subsequently, a fixup function is run over each of these entries. It can replace the old constraint with a new one and add additional constraints to the global formula set.

### 5.2.2 Bit-width Promotion

Integer overflows result from the value of an expression becoming too large for the data type that it is stored in. Thus in order to model this behavior, expression bit-widths must be expanded to take the larger values. Moreover, because Z3 cannot tolerate expressions with mixed bit widths, values of expressions that are not being checked for overflow must be constrained to maintain correctness.

In each expression, each variable is promoted to twice the bit width of the largest variable in the subexpression. To maintain correctness, each variable is constrained to take values that fit inside of their pre-fixup bit-widths and all expressions are AND-ed by the largest value their bit-width can take.
5.2.3 Signedness Constraints

Expanding the bit-widths of all of the variables can make sign extension functions fail as the position of the highest bit has changed. For this reason we introduce constraints to perform the sign extension. New fixup variables are introduced for each argument to a signed expression. This variable will take the original value if the replaced constraint takes an unsigned value. Otherwise, the replacement variable takes the original value sign-extended to the post-promotion bit-width.

1 def fixup(c)
2     create_variable(c');
3     add_constraint(BvAnd(c,UShl(1, c. bit-width)) = 0 ===> c' = c);
4     replace c with c';

Figure 5-2: Fixup function for signedness constraints

1 def predicate(c)
2     if parent(c) = None then
3         return False
4     if parent (c) has signed type then
5         return True
6     return False

Figure 5-3: Predicate for signedness fixups

5.2.4 Boolean Type Error Fixups

The C programming language as well as VEX IR and x86 assembly language, booleans have integer representations. true is represented as any value not equal to zero and false is represented as 0. However, the Z3 solver does not allow an integer expression to be used as a boolean or vise versa.

We replace integer expressions that reside inside of boolean expressions with a boolean variable that is true when the integer expression is not equal to zero and false when it is. To set the truth value properly, we introduce implications that implement the C programming
```python
def predicate(c):
    if parent(c) = None then
        return False
    if c has type Boolean then
        if parent(c) has type BitVector then
            return True
        return False

Figure 5-4: Predicate for boolean inside of bitvector fixup
```

```python
def fixup(c):
    create_variable(c');
    add_constraint(UGreaterThan(c, 0) ===> c' = True);
    add_constraint(c = 0 ===> c' = False);
    replace c with c';

Figure 5-5: Fixup function for boolean inside of bitvector type error
```

```python
def fixup(c):
    create_variable(c');
    add_constraint(c' = True ===> c' = 1);
    add_constraint(c' = False ===> c' = 0);
    replace c with c';

Figure 5-6: Fixup function for bitvector inside of boolean type error
```

```python
def predicate(c):
    if parent(c) = None then
        if c has type Boolean then
            return False
        else return True;
    if c has type BitVector then
        if parent(c) has type Boolean then
            return True
        return False

Figure 5-7: Predicate for bitvector inside of boolean fixup
```
language's boolean logic. Conversely, we rewrite boolean expressions inside of integer expressions with integer variables that are set to 1 if the boolean expression is true and 0 if it is false.

5.2.5 Preventing Divide-By-Zeros

Although many processor architectures will fault if one attempts to divide by zero, the Z3 solver does not attempt to avoid these. To avoid triggering divide-by-zero bugs that prevent us from exercising an integer overflow bug, we add constraints that prevent any divisor from evaluating to zero.

```python
1 def predicate(c)
2     if parent(c) = None then
3         return False
4     if parent(c) has type SDiv or UDiv then
5         if c is a divisor then
6             return True
7     return False
```

Figure 5-8: Predicate for divide-by-zero fixup

```python
1 def fixup(c)
2     create_variable(c');
3     add_constraint(c ≠ 0);
4     add_constraint(c = c');
5     replace c with c';
```

Figure 5-9: Fixup function for avoiding divide-by-zero
Chapter 6

Evaluation

We show that DIODE is capable of finding bugs in real world applications even in the presence of blocker constraints. We evaluate DIODE on five applications: Dillo 2.1 (description), 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.

<table>
<thead>
<tr>
<th>Application</th>
<th>Target Sites With Overflows</th>
<th>Inherently Safe Target Sites</th>
<th>Total Target Sites</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dillo</td>
<td>3</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>VLC</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>SwfPlay</td>
<td>3</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>CWEBP</td>
<td>1</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>ImageMagick 6.5.2</td>
<td>3</td>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>

Figure 6-1: Target Site Classification

6.1 Target Site Classification

We next present results that show that DIODE is complete — for each target memory allocation site, either 1) DIODE generates an input file that triggers an overflow at the site, or 2) no input file exists that will trigger an overflow for the target expression observed at the site.

Table 6-1 contains one row for each application. The first column (Application) identifies the application. The second column (Target Sites With Overflows) presents the number of target memory allocation sites for which DIODE was able to generate input files that
trigger the overflow. The third column (Inherently Safe Target Sites) presents the number
of target sites that are inherently safe (because the target constraint is unsatisfiable, which
indicates that, even in the absence of sanity checks, there are no input values that will trigger
an overflow for the observed target expression). The fourth column (Total Target Sites)
presents the total number of target sites.

Considering all of the columns together, for VLC 0.8.6h, SwfPlay 0.5.5, and CWebP,
DIODE found an overflow-triggering input for all sites that are not inherently safe. For
ImageMagick 6.5.2, DIODE found an overflow-triggering input for all but one of the sites
that are not inherently safe. Except for VLC 0.8.6h (which contains ineffective safety checks
that do not protect against overflows), we attribute these results to an absence of sanity
checks for overflows in these applications.

For Dillo 2.1, there are eight sites 1) that are not inherently safe, and 2) for which DIODE
did not find an input that triggered an overflow. A manual inspection of these sites indicates
that Dillo 2.1 does contain sanity checks and that these sanity checks are partially effective
in that they prevent overflows at five of the eight sites that are not inherently safe (for the
remaining three sites, of course, DIODE generates an input that triggers the overflow).

6.2 Overflow Characteristics

Table 6-2 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) identifies the source code file and
line that contains the memory allocation statement for which the overflow occurs. The third
column (Bug) identifies either the CVE number of the overflow (if the overflow was known)
or "New" if the overflow is new. We note that all but three of the 14 overflows are new.

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 program.
We detect these invalid reads and writes using the Valgrind memcheck tool [22], 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 (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 56 to 554 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 (see Section 6.5).

### 6.3 Inputs That Satisfy Target Constraint Alone

The sixth column (Target Success Rate) presents the results from the experiment in which DIODE generated 200 inputs that satisfied the target constraint alone. 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 does contain 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).

### 6.4 Target and Enforced Branch Success Rate

<table>
<thead>
<tr>
<th>Application</th>
<th>Target</th>
<th>Bug</th>
<th>Error Type</th>
<th>Enforced Branches</th>
<th>Target Success Rate</th>
<th>Target + Enforced Success Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dillo 2.1</td>
<td>png.c</td>
<td>CVE-2009-2294</td>
<td>SIGSEGV/InvalidRead</td>
<td>4/56</td>
<td>0/200</td>
<td>190/200</td>
</tr>
<tr>
<td>Dillo 2.1</td>
<td>ffmpegcore.c@39</td>
<td>New</td>
<td>SIGSEGV/InvalidRead</td>
<td>5/59</td>
<td>0/200</td>
<td>185/200</td>
</tr>
<tr>
<td>Dillo 2.1</td>
<td>image/cvxf@741</td>
<td>New</td>
<td>SIGSEGV/InvalidRead</td>
<td>5/59</td>
<td>0/200</td>
<td>190/200</td>
</tr>
<tr>
<td>VLC 0.8.6h</td>
<td>image.c@355</td>
<td>New</td>
<td>SIGSEGV/InvalidRead</td>
<td>2/17</td>
<td>0/200</td>
<td>100/200</td>
</tr>
<tr>
<td>VLC 0.8.6h</td>
<td>wnc@147</td>
<td>CVE-2008-2430</td>
<td>InvalidRead/Write</td>
<td>9/4</td>
<td>0/200</td>
<td>100/200</td>
</tr>
<tr>
<td>VLC 0.8.6h</td>
<td>doc.c@277</td>
<td>New</td>
<td>SIGSEGV/InvalidRead</td>
<td>142/57</td>
<td>0/200</td>
<td>190/200</td>
</tr>
<tr>
<td>VLC 0.8.6h</td>
<td>block.c@54</td>
<td>New</td>
<td>InvalidRead</td>
<td>0/554</td>
<td>200/200</td>
<td>N/A</td>
</tr>
<tr>
<td>SwfPlay 0.5.5</td>
<td>jpg_nibDecoder.c@237</td>
<td>New</td>
<td>SIGSEGV/InvalidRead</td>
<td>0/23</td>
<td>0/200</td>
<td>N/A</td>
</tr>
<tr>
<td>SwfPlay 0.5.5</td>
<td>jpg_nibDecoder.c@237</td>
<td>New</td>
<td>SIGSEGV/InvalidRead</td>
<td>0/23</td>
<td>0/200</td>
<td>N/A</td>
</tr>
<tr>
<td>SwfPlay 0.5.5</td>
<td>jpg.c@192</td>
<td>New</td>
<td>SIGSEGV/InvalidRead</td>
<td>0/23</td>
<td>0/200</td>
<td>N/A</td>
</tr>
<tr>
<td>CWebP 0.3.1</td>
<td>jpegbase.c@248</td>
<td>New</td>
<td>SIGSEGV/InvalidRead</td>
<td>0/18</td>
<td>0/200</td>
<td>N/A</td>
</tr>
<tr>
<td>ImageMagick 6.5.2</td>
<td>cjpeg.c@803</td>
<td>CVE-2008-1882</td>
<td>InvalidRead/Write</td>
<td>0/15</td>
<td>0/200</td>
<td>N/A</td>
</tr>
<tr>
<td>ImageMagick 6.5.2</td>
<td>display.c@439</td>
<td>New</td>
<td>SIGSEGV/InvalidRead</td>
<td>0/13</td>
<td>0/200</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Figure 6-2: Experimental results

The seventh 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. 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 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
6.5 Case: Dillo - png.c@203

We present an example where present concolic and symbolic execution tools would have great difficulty finding a bug despite the lack of sufficient sanity checks. To reach the target site, both approaches would include difficult to solve checksum constraints. Concolic execution tools would spend a considerable amount of time exercising the loop for different numbers of iterations if the seed input caused the loop to over-constrain the relevant inputs.

DIODE discovers figure B-2, the vulnerable target expression `png->rowbytes * png->height`. It enforces checks 1 and 2 when Z3 finds a solution to the overflow constraint that sets `width` and `height` to 32 to bit values instead of 31 bit values. Checks 3 and 4 are then enforced to constrain `width` and `height` to be less than an application supplied constant. Finally, check 5 is enforced - a check for integer overflow that fails due to its own susceptibility to overflow. After this set of checks are satisfied, a mutated file is created and then used to test the application. Valgrind detects an InvalidRead and a bug is reported.

DIODE is able to avoid the loop constraints due to its exclusion of constraints associated with loop-back edges. It excludes the checksum constraint because DIODE's ability to re-calculate checksums causes the checksum sanity check to always pass, rendering it ineligible for enforcement by the goal directed algorithm.

6.6 Case: ImageMagick Display - xwindow.c @ 5619

Figure B-4 details a vulnerable piece of code that will cause symbolic and concolic execution systems to explore other paths to reach the bug. Solving the path constraint and the overflow query is impossible due to the fact that the check on line 25 properly detects integer overflow. However, the check does not halt further processing of the input; the vulnerable `malloc` and data copy will still be exercised.

DIODE is able to find a vulnerability triggering input by solving the target constraint alone. It first feeds Display the seed input and discovers the target site that has a target ex-
pression of \texttt{cache\_info->columns*cache\_info->rows*sizeof(pixel\_packet)}. Subsequently it derives the target constraint, solves it, and produces a test input file based on the field assignments. Since the check does not exit the program or otherwise divert control away from the critical path upon failure, the test input triggers the bug. The Valgrind instrumented application ran with this input reports an \texttt{InvalidWrite}. DIODE then reports a bug because it was able to find an error. Because the error was triggered without any branch constraints the blocking constraint characteristic of line 25 was not added to the formula.

Our results show that DIODE is capable of finding bugs in real world applications. In the presence of blocker constraints, DIODE is still able to derive test inputs that trigger overflows without resorting to exponential time path exploration.
Chapter 7

Prior Work

The severity of the consequences of integer overflow vulnerabilities has lead to a variety of approaches to uncover them. We evaluate prior static analysis and automated testing tools and compare them with DIODE.

7.1 BuzzFuzz

Buzzfuzz [16] is an automated testing system that targets file consuming applications. It performs dynamic taint analysis on the target application ran with a seed file. It outputs a list of offsets that were used to calculate arguments to a list of specified functions. Many of the functions specified are memory allocation routines. To create a test input for the target application, Buzzfuzz places extremal values in the offsets found for the seed file. Subsequently, it runs the application with those malformed inputs and watches for program signals indicative of a fault, such as BADALLOC, SIGABRT, and SIGSEGV.

Buzzfuzz does not attempt to create constraints on the conditions which must be met in order for its targeted function instances to be exercised. For that reason, simple validity checks can stop Buzzfuzz from exercising the a potentially vulnerable condition, regardless of whether the assignment would have yielded a fault without those checks present.

Checks related to checksums are also not handled. Test files in formats that contain these will cause the application to abort early due to the expected checksum not matching the checksum present in the file.
Error checking could also be improved. Buzzfuzz only monitors signals thrown by the program and uses no form of binary instrumentation to detect errors. Thus inputs that cause small out of bounds memory reads and writes of a memory block will often be thought to be benign. For this reason, Buzzfuzz likely underreported the number of bugs it was able to induce.

7.2 SAGE

SAGE uses generational search to increase code coverage [18]. It produces path constraints via dynamic symbolic execution. Unlike other tools that are targeted whose goal is to develop exploits for certain vulnerability classes, SAGE’s main goal is code coverage. To this end, SAGE iteratively generates generations of files that maximize code coverage. It then analyzes the behavior induced by these files on the tested application using dynamic binary instrumentation.

Generational search is SAGE’s technique for gaining program coverage. It begins by feeding a seed file to the application. All branches are recorded that use file input in their calculations and a path constraint with those variables is generated. The solution is used to mutate the seed file to obtain a new file. Thus for each element of the path constraint, a new input is produced. The set of new inputs is called a generation. A new seed is selected and the process continues. The new seeds are selected by a path heuristic. In SAGE, it is the number of new basic blocks exercised.

SAGE has no heuristics for targeting bugs. It is merely a way to exercise a large portion of the logic of the target application. SAGE would likely yield more bugs if it did.

Generational search can also produce many inputs that do not result in more program coverage. Mainly, this happens when tainted branch constraints are a part of a loop. Should this occur, SAGE will create new files for every branch iteration, resulting in a potentially exponential number in the number of files generated.

SAGE detects faults by feeding the program inputs to the target application running in AppVerifier [5]. This strategy will detect error signals and out of bounds writes on heap allocated memory. However, it cannot detect out of bounds reads, which are often used to
disclose information which can later be used in malicious exploitation.

7.3 SmartFuzz

SmartFuzz [21] is an automated testing tool that implements two bug discovery strategies: 1) concolic execution and 2) mutation based fuzzing. Unlike SAGE, SmartFuzz detects signedness conversion bugs, bit-width conversion bugs and integer overflow bugs. It decides which constraints induce bugs by running the target application with the test input under the Valgrind Memcheck utility [22].

Unlike DIODE, SmartFuzz uses generational search to exercise new program paths rather than direct its search to specific target sites. So if SmartFuzz is unable to overflow a target value calculated along the path taken by the seed input, it will need to perform another exponential time search to discover an alternative path. Further, it does not make any attempt to filter loop-back edges. For many programs, this will mean a large number of test cases will simply result in executing a loop for a different number of iterations than the seed input. Furthermore, it makes no attempt to filter other types of blocker constraints from the branches it considers for a particular program path which makes the necessity of the exponential time search for the target site more likely to occur.

7.4 Automatic Patched Based Exploit Generation

Automatic patch based exploit generation (APEG) [8] discovers bug exercising inputs in pre-patch software versions based on post-patch versions. A binary differencing algorithm is run to determine the location of software patches. The locations of the patched code are then associated with their pre-patch versions. Subsequently, a weakest precondition to constrain inputs to reach the buggy program point is computed. The weakest precondition is then conjoined with a bug query - a simple negation on the characteristic constraint of the check in the post-patch code. This formula is then solved by the STP SMT solver [15] to obtain modifications to input needed to both exercise the buggy code path and trigger the bug.

APEG does not only discover bug inducing inputs for integer overflows - buffer overflows
and heap overflows have also been exercised with the tool. Further, Song et al. use a number of different methods: static chopping of the program's CFG, dynamic path constraint generation, and a combined approach to generate the weakest precondition. It uses a dynamic analysis tool to determine whether a generated input triggered a bug.

While APEG is can find inputs that exercise many different types of vulnerabilities, it is limited to those that have been patched. DIODE has no such limitations. It is also not clear that solutions to the negation of the sanity check introduced by the patch would trigger an overflow - the patch may deny many non-overflow inducing inputs. Problematic patches would include those that place a small upper bound on a relevant input.

7.5 KINT

KINT [25] is a static analysis tool that flags potential vulnerable target sites at compile time. Instead of performing symbolic execution, it considers constraints within functions and uses range propagation to approximate parameters. It relies on programmer annotations to describe the value ranges on input variables to the procedures analyzed and global variables. However, it incorrectly flags many functions as vulnerable due to imprecision from the use of range propagation.

Due to the imprecision of range propagation, KINT reports many false positives. DIODE requires no programmer annotations and reports no false positives as it tests every input before it is flagged as bug. KINT only works on LLVM [2] bytecode, which is at present limited to C source, whereas DIODE supports x86 binaries.

7.6 KLEE

KLEE [9] is a symbolic execution engine that operates on LLVM bytecode. It is capable of deriving inputs that violate assertions or cause out of bounds memory writes. To compensate for the exponential number of states required to model all program paths, KLEE uses heuristics to avoid impossible to exercise paths, simplify constraints, and keep its state representation small. It attempts to discover unexecuted code quickly by ranking each path
by weight. This allows KLEE to avoid spending significant amounts of time exercising loop branches.

While KLEE works to optimize symbolic execution, the path explosion problem still prevents it from reaching bugs deep in applications. Because it does not make use of seed inputs, it must consider all tainted branch conditions along a path and is thus unable to only consider constraints containing relevant inputs.

DIODE offers many benefits over existing approaches to uncover integer overflow bugs. It is able to detect integer overflow bugs in unmodified application binaries and has no false positives. It is able to use a very small number of constraints to uncover the overflow in comparison to concolic execution tools, which solve every tainted branch along the path reaching the target site. We conclude in the following section.
Chapter 8

Conclusion

We present goal-directed branch constraint enforcement and its implementation in DIODE. We show that DIODE is an effective tool for finding integer overflow errors in the presence of constraints that would block tools that consider every tainted branch along the program path reaching the target site. Our experimental results show that DIODE is effective at generating integer overflows, including previously unknown errors in widely used applications.
Appendix A

Tables
### Table A.1: Bitvector constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>UShl(x,y)</td>
<td>Unsigned left</td>
</tr>
<tr>
<td>SShl(x,y)</td>
<td>Signed shift left</td>
</tr>
<tr>
<td>UShr(x,y)</td>
<td>Unsigned shift right</td>
</tr>
<tr>
<td>SSshr(x,y)</td>
<td>Signed shift right</td>
</tr>
<tr>
<td>UDiv(x,y)</td>
<td>Unsigned divide</td>
</tr>
<tr>
<td>SDiv(x,y)</td>
<td>Signed divide</td>
</tr>
<tr>
<td>BvAnd(x,y)</td>
<td>Bitvector AND</td>
</tr>
<tr>
<td>BvOr(x,y)</td>
<td>Bitvector OR</td>
</tr>
<tr>
<td>Mul(x,y)</td>
<td>Multiplication</td>
</tr>
<tr>
<td>Sub(x,y)</td>
<td>Subtraction</td>
</tr>
<tr>
<td>Add(x,y)</td>
<td>Addition</td>
</tr>
<tr>
<td>Equals(x,y)</td>
<td>Equality</td>
</tr>
<tr>
<td>Width(x,y)</td>
<td>Addition</td>
</tr>
<tr>
<td>Overflow(x)</td>
<td>Integer overflow</td>
</tr>
</tbody>
</table>

### Table A.2: Boolean constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equals(x,y)</td>
<td>Equality</td>
</tr>
<tr>
<td>Implies(x,y)</td>
<td>Implication</td>
</tr>
<tr>
<td>Conjunction(x,...)</td>
<td>Logical AND</td>
</tr>
<tr>
<td>Disjunction(x,...)</td>
<td>Logical OR</td>
</tr>
<tr>
<td>UGreaterThan(x,y)</td>
<td>Unsigned &gt;</td>
</tr>
<tr>
<td>SGreaterThan(x,y)</td>
<td>Signed &gt;</td>
</tr>
<tr>
<td>ULessThan(x,y)</td>
<td>Unsigned &lt;</td>
</tr>
<tr>
<td>SLessThan(x,y)</td>
<td>Signed &lt;</td>
</tr>
<tr>
<td>ULessEqual(x,y)</td>
<td>Unsigned ≤</td>
</tr>
<tr>
<td>SLessEqual(x,y)</td>
<td>Signed ≤</td>
</tr>
</tbody>
</table>
Appendix B

Figures

```
png_check_IHDR(png_structp png_ptr,
    png_uint_32 width, png_uint_32 height, int bit_depth...) {
    ...  
    //Check 3: Height < 1000000L
    if (height > PNG_USER_HEIGHT_MAX) {
        png_warning(png_ptr,
            "Image width exceeds user limit in IHDR");
        error = 1;
    }
    //Check 4: Width < 1000000L
    if (width > PNG_USER_WIDTH_MAX) {
        png_warning(png_ptr,
            "Image width exceeds user limit in IHDR");
        error = 1;
    }
}
png_get_uint_31(png_structp png_ptr, png_const_bytep buf) {
    png_uint_32 uval = png_get_uint_32(buf);
    // Checks 1 & 2: Checks that width/height < 0x7fffffffL
    if (uval > PNG_UINT_31_MAX)
        png_error(png_ptr,
            "PNG unsigned integer out of range");
    return (uval);
}
```

Figure B-1: Code snipette from Dillo PNG parser
```c
#define PNG_ROWBYTES(pixel_bits, width) (((pixel_bits)>>3) > 0 ?
  (((width)*((png_uint_32)(pixel_bits))>>3)) :
  (((width)*((png_uint_32)(pixel_bits))>>7)>>3))

void png_handle_IHDR(png_structp png_ptr,
  png_infop info_ptr, ...) {
  ...
  // read individual png fields from input buffer
  width   = png_get_uint_31(png_ptr, buf);
  height  = png_get_uint_31(png_ptr, buf + 4);
  bit_depth = buf[8];
  ...
  png_ptr->width = width;
  png_ptr->height = height;
  png_ptr->bit_depth = (png_byte)bit_depth;
  ...
  png_ptr->pixel_depth = (png_byte)(
    png_ptr->bit_depth * png_ptr->channels);
  png_ptr->rowbytes = PNG_ROWBYTES(
    png_ptr->pixel_depth, png_ptr->width);
  }

// Dillo datainfo initialization callback
static void
Png_datainfo_callback(png_structp png_ptr, png_infop info_ptr)
{
  DilloPng *png;
  ...
  // Check 5: Incorrect check of max image size
  if (abs(png->width*png->height) > IMAGE_MAX_W + IMAGE_MAX_H) {
    MSG("suspicious image size request %ldx%ld
",
      png->width, png->height);
    return;
  }
  // Where the overflow happens
  png->image_data = (uchar_t *)dMalloc(png->rowbytes * png->height);
  ...
  for (i = 0; i < png->height; i++)
    png->row_pointers[i] = png->image_data + (i * png->rowbytes);
}
```

Figure B-2: Code for Dillo integer overflow vulnerability
// CRC Sanity check
png_crc_finish(png_structp png_ptr, png_uint_32 skip)
{
    png_size_t i;
    png_size_t istop = png_ptr->zbuf_size;

    for (i = (png_size_t)skip; i > istop; i -= istop)
    {
        png_crc_read(png_ptr, png_ptr->zbuf, png_ptr->zbuf_size);
    }

    if (i)
    {
        png_crc_read(png_ptr, png_ptr->zbuf, i);
    }

    if (png_crc_error(png_ptr))
    {
        if (((png_ptr->chunk_name[0] & 0x20) && /* Ancillary */
             !(png_ptr->flags & PNG_FLAG_CRC_ANCILLARY_NOWARN)) ||
            (! (png_ptr->chunk_name[0] & 0x20) && /* Critical */
             (png_ptr->flags & PNG_FLAG_CRC_CRITICAL_USE)))
        {
            png_chunk_warning(png_ptr, "CRC error");
        }
        else
        {
            png_chunk_benign_error(png_ptr, "CRC error");
            return (0);
        }
    }

    return (1);
}

return (0);

Figure B-3: Code for Dillo CRC Checksum
static inline void AcquirePixelCachePixels(CacheInfo *cache_info) {
    cache_info->mapped = MagickFalse;
    cache_info->pixels = (PixelPacket *) AcquireMagickMemory((size_t) cache_info->length);
    if (cache_info->pixels == (PixelPacket *) NULL) {
        cache_info->mapped = MagickTrue;
        cache_info->pixels = (PixelPacket *) MapBlob(-1, IOMode, 0, (size_t) cache_info->length);
    }
    cache_info->rows = image->rows;
    cache_info->columns = image->columns;
    cache_info->active_index_channel = ((image->storage_class == PseudoClass) ||
        (image->colorspace == CMYKColorspace)) ? MagickTrue : MagickFalse;
    number_pixels = (MagickSizeType) cache_info->columns * cache_info->rows;
    packet_size = sizeof(PixelPacket);
    if (cache_info->active_index_channel != MagickFalse) {
        packet_size += sizeof(IndexPacket);
        length = number_pixels * packet_size;
        columns = (unsigned long) (length / cache_info->rows / packet_size);
        if (cache_info->columns != columns) {
            ThrowBinaryException(ResourceLimitError, "PixelCacheAllocationFailed", image->filename);
        }
    }
    cache_info->length = length;
    status = AcquireMagickResource/AreaResource, cache_info->length);
    length = number_pixels * (sizeof(PixelPacket) + sizeof(IndexPacket));
    if ((status != MagickFalse) && (length == (MagickSizeType) (size_t) length)) {
        status = AcquireMagickResource/MemoryResource, cache_info->length);
        if (((cache_info->type == UndefinedCache) && (status != MagickFalse)) ||
            (cache_info->type == MemoryCache)) {
            AcquirePixelCachePixels(cache_info);
        }
    }
}

Figure B-4: Code for ImageMagick integer overflow vulnerability
Bibliography


