

Using type analysis in compiler to mitigate integer-overflow-to-buffer-overflow threat

Chao Zhang ^{a,b}, Wei Zou ^{a,b}, Tielei Wang ^{a,b}, Yu Chen ^{a,b} and Tao Wei ^{a,b,*}

^a *Institute of Computer Science and Technology, Peking University, Beijing, China*

^b *Beijing Key Laboratory of Internet Security Technology, Peking University, Beijing, China*

One of the top two causes of software vulnerabilities in operating systems is the integer overflow. A typical integer overflow vulnerability is the Integer Overflow to Buffer Overflow (IO2BO for short) vulnerability. IO2BO is an underestimated threat. Many programmers have not realized the existence of IO2BO and its harm. Even for those who are aware of IO2BO, locating and fixing IO2BO vulnerabilities are still tedious and error-prone. Automatically identifying and fixing this kind of vulnerability are critical for software security. In this article, we present the design and implementation of IntPatch, a compiler extension for automatically fixing IO2BO vulnerabilities in C/C++ programs at compile time. IntPatch utilizes classic type theory and a dataflow analysis framework to identify potential IO2BO vulnerabilities, and then uses backward slicing to find out related vulnerable arithmetic operations, and finally instruments programs with runtime checks. Moreover, IntPatch provides an interface for programmers who want to check integer overflows manually. We evaluated IntPatch on a few real-world applications. It caught all 46 previously known IO2BO vulnerabilities in our test suite and found 21 new bugs. Applications patched by IntPatch have negligible runtime performance losses which are on average 1%.

Keywords: Integer overflow to buffer overflow, compiler defense technique, static analysis

1. Introduction

The Integer Overflow to Buffer Overflow vulnerability (IO2BO for short), defined in Common Weakness Enumeration (CWE-680 [14]), is a kind of vulnerability caused by integer overflows. If an integer overflow occurs when a program performs a calculation to determine how much memory to allocate, a less than expected memory block will be allocated, and then a buffer overflow is triggered when the memory block is used.

For instance, Fig. 1 shows a typical IO2BO vulnerability in Faad2¹ version 2.6. The routine `mp4ff_read_int32(f)` at line 467 reads in an integer value (e.g., `0x80000001`) from external file `f` without any checks. This value is then used in a memory allocation function at line 469. If an overflow occurs, a smaller than ex-

* Corresponding author: Institute of Computer Science and Technology, Peking University, Beijing, China. Tel.: +86 10 82529656; Fax: +86 10 82529207; E-mail: weitao@icst.pku.edu.cn.

¹<http://www.audiocoding.com/faad2.html>.

```

458. static int32_t mp4ff_read_ctts(mp4ff_t *f)
459. {
460.     // sth. omitted ...
467.     p_track->ctts_entry_count = mp4ff_read_int32(f);
468.
469.     p_track->ctts_sample_count = (int32_t*) malloc (p_track->ctts_entry_count * sizeof(int32_t));
470.     p_track->ctts_sample_offset = (int32_t*) malloc (p_track->ctts_entry_count * sizeof(int32_t));
481.     for (i = 0; i < p_track->ctts_entry_count; i++)
482.     {
483.         p_track->ctts_sample_count[i] = mp4ff_read_int32(f);
484.         p_track->ctts_sample_offset[i] = mp4ff_read_int32(f);
485.     }
486.     return 1;
488. }

```

Fig. 1. A real-world IO2BO vulnerability in Faad2.

pected memory region (e.g., $0x80000001 * 4 = 4 \bmod 2^{32}$) will be allocated, leading to an IO2BO vulnerability. Further, at line 483, some unchecked values read from external file f will be written to the allocated small memory chunk, causing the heap corrupted and leading to arbitrary code execution attack [33].

IO2BO is an underestimated threat. In recent years, it has been witnessed that IO2BO is being widely used by attackers [8,33]. For example, Vreugdenhil, who won the computer hacking contest “Pwn2Own 2010”, exploited an IO2BO vulnerability [34] in IE8 to bypass the security-enhanced mechanisms provided by Windows 7 – Address Space Layout Randomization (ASLR [32]) and Data Execution Prevention (DEP [2]). Moreover, according to the statistical data collected in the National Vulnerability Database (NVD [28]), from April 1st 2009 to April 1st 2010, nearly half of all integer overflow vulnerabilities and one third of heap overflow vulnerabilities are IO2BO. Even worse, nearly 80% of these IO2BO vulnerabilities are in high risk, i.e. their CVSS (Common Vulnerability Scoring System, [13]) severity scores are greater than 7, and the remaining 20% are all in medium risk.

There are a number of studies focusing on detecting integer overflows, such as [6, 7,26,35]. They can be classified into two categories: static analysis and dynamic analysis methods. For those static analysis tools, false positives are non-negligible. The output of these tools need to be validated manually. For those dynamic analysis tools, the major disadvantage is their false negatives. Although many systems (such as KLEE [4], EXE [5], CUTE [31], DART [20]) have applied symbolic execution techniques to improve code coverage and reduce false negatives, the analysis results of these tools are still not sound due to the problem’s complexity.

Some compilers or compiler extensions can automatically fix integer overflow vulnerabilities. For example, with the `-ftrapv` compiler option, GCC can generate traps for each signed integer overflow in addition/subtraction/multiplication operations by inserting extra code after each signed arithmetic operation. When signed integer overflow occurs at runtime, the compiled program aborts. Brumley et al. have developed a static program transformation tool, called RICH [3]. RICH constructs formal semantics for safe C integer operations and instruments the target program with runtime checks against all unsafe integer operations.

However, there are some kinds of benign integer overflows. For example, when an integer is used as the sequence number of a HTTP message, whether it overflows does not matter. In some cases, integer overflow can be deliberately used in random number generators and message encoding/decoding or modulo arithmetic [3]. Methods which make full instrumentation like GCC and RICH cannot tell the differences between benign and harmful integer overflows and thus inevitably generate false positives, i.e. many overflows captured by these tools at runtime are not really vulnerabilities. Furthermore, the full instrumented programs usually suffer from a non-trivial performance overhead.

In this article, we present IntPatch, a tool targeting only IO2BO vulnerabilities. Integer overflows that exist in the context of IO2BO vulnerabilities usually are not benign. And thus IntPatch has fewer false positives. On the other hand, IntPatch applies conservative type analysis, and thus has few false negatives.

IntPatch is capable of identifying potential IO2BO vulnerabilities and fixing them automatically. First, we use a type analysis pass to detect potential IO2BO vulnerabilities. Then, for each candidate vulnerability, another analysis pass is made to locate related vulnerable arithmetic operations. Finally, runtime check code is inserted after these vulnerable arithmetic operations.

We implement IntPatch as an extension of LLVM (Low Level Virtual Machine [24,25]) and evaluate its performance on a number of real-world open-source applications. It shows that IntPatch is a powerful and lightweight tool which can efficiently fix IO2BO vulnerabilities. IntPatch could help programmers to accelerate software development and greatly promote programs' security.

1.1. Contributions

This article presents an automatic tool for efficiently protecting against IO2BO vulnerabilities. Specially, we:

- Survey 46 IO2BO vulnerabilities and compare them with publicly-available, patched versions, then provide an analysis of the reasons why existing methods of addressing IO2BO are tedious and error-prone.
- Construct a type system to model IO2BO vulnerabilities and present a framework for automatically identifying and fixing them at compile time, and thus freeing programmers from fixing IO2BO vulnerabilities.

- Provide an API for programmers who want to capture integer overflows manually.
- Implement a tool called IntPatch. It inserts dynamic check code to protect against IO2BO. The patched version's performance overhead is low, on average about 1%. Experiments also show that IntPatch is able to capture all 46 IO2BO vulnerabilities we surveyed.
- Identify 21 zero-day bugs (i.e., bugs unexposed before) in open-source applications with IntPatch.

1.2. Outline

We first describe the result of our survey and discuss the difficulties that programmers may face when fixing integer overflows in Section 2. Our type system which models IO2BO vulnerabilities and the framework for eliminating IO2BO threat are described in Section 3. In Section 4, we discuss the implementation of our tool IntPatch, including the extra interface provided for programmers. Section 5 evaluates our work, and shows the performance and false positives of IntPatch. Related work and conclusion are discussed in Sections 6 and 7.

2. Motivation

We have surveyed 46 IO2BO vulnerabilities consisting of 17 bugs found by IntScope [35] and 29 bugs reported in CVE [11], Secunia,² VUPEN,³ CERT⁴ and oCERT.⁵ In addition, 18 patches of these vulnerabilities were investigated. In this section, we will discuss in detail what we have learned from the survey and what difficulties programmers may face when fixing IO2BO vulnerabilities.

2.1. Input validation problem

Fixing integer overflows is essentially an input validation problem. Incomplete input validation is the origin of IO2BO vulnerability.

To check whether signed integer multiplication $a * b$ overflows, the widely used method in practice looks like:

```
if ( b != 0 && (a*b)/b != a )
    MSG("overflow occurs");
```

²<http://secunia.com>.

³<http://www.vupen.com/english/>.

⁴<http://www.cert.org/advisories>.

⁵<http://www.ocert.org/>.

This method is called *postcondition testing*. It depends on the computation of $a * b$. This method is efficient. But it may fail in some occasions, especially when compiler optimizations exist. Some aggressive optimizations may discard checks like “ $(a * b)/b = a$ ” and cause postcondition testing to be useless.

Another widely used method for checking overflow is *precondition testing*. It checks whether $a * b$ may overflow before the actual multiplication is calculated. Precondition testing method looks like:

```
if ( a>0 && b>0 && a > INT_MAX/b )
    MSG("overflow occurs");
else if ( a>0 && b<0 && b < INT_MIN/a )
    MSG("overflow occurs");
else if ( a<0 && b>0 && a < INT_MIN/b )
    MSG("overflow occurs");
else if ( a<0 && b<0 && b < INT_MAX/a )
    MSG("overflow occurs");
else
    c = a*b;
    // normal statements
```

Because the check is taken prior to the actual multiplication, this method can circumvent compiler optimization problem. In other words, whatever undefined behavior the compiler may take when signed integer overflow occurs, precondition testing is robust. However, detecting an overflow in this manner is expensive. For each signed integer operation, at least four extra statements are inserted. Furthermore, branches are expensive on modern hardware, and thus precondition testing introduces a non-trivial performance overhead.

Detecting integer overflow can also be done at the assembly code level. At this level, programmers can utilize hardware features to facilitate checking. For example, an x86 processor has two flags named an overflow flag and a carry flag, and then these flags can be used to determine whether an operation overflows. However, checking an integer overflow correctly is still difficult. On x86 architecture, methods for checking overflows in signed/unsigned multiplications/additions are different. Instructions `jo`, `jc` and `js` should be used in combination to check those overflows [22]. Furthermore, there are no general interfaces for C/C++ programmers to query these flags. Besides, no statements in high-level language are handy for programmers to efficiently invoke these assembler instructions. Thus, programmers who want to detect integer overflows at the assembly code level have to write compiler-specific and platform-specific inline assembler code, causing portability problems.

Even when programmers know how to detect integer overflows, protecting programs from integer overflows is still error-prone and tedious. In the following, we describe the difficulties programmers may face when fixing integer overflows.

2.2. Fixing IO2BO is tedious and error-prone

2.2.1. Awareness of IO2BO

First of all, many programmers have not yet realized the existence of integer overflow, not to speak of IO2BO. The following code snippet shows a patch released in Dillo⁶ version 2.1 against a vulnerability in version 2.0 whose ID is CVE-2009-2294. The original vulnerability is due to the product of `png->width` and `png->height` being too large. In this patch, the developers' intention is to check the upper bound of this product. However, the existence of integer overflow has changed the program logic assumed by programmers and thus causing this patch to be useless, i.e. the product of `png->width` and `png->height` is wrapped and the check is still not violated.

```
+ if (abs(png->width*png->height) > IMG_MAX_W*IMG_MAX_H) {
+     // handle overflow here
+ }
```

2.2.2. Fallibility of fixing IO2BO

For programmers who are aware of integer overflows, fixing integer overflow manually is still **error-prone**. The following code snippet illustrates an erroneous patch in CUPS⁷ version 1.3.9 against a vulnerability in revision 7434 whose ID is CVE-2008-1722. The original vulnerability is due to an overflow when calculating `img->xsize*img->ysize*3`. However, if `img->ysize*3` overflows, this check is useless.

```
+ bufsize = img->xsize * img->ysize * 3;
+ if (bufsize / (img->ysize * 3) != img->xsize) {
+     // handle overflow here
+ }
```

2.2.3. Complexity of fixing IO2BO

Even for programmers who are careful enough, fixing IO2BO is tedious. For example, in order to correctly check whether `img->xsize*img->ysize*3` overflows, we first need to check the product (denoted as `temp`) of `img->ysize` and 3, and then check the product of `temp` and `img->xsize`. Similarly, if a programmer wants to check whether the product of N operands overflows, at least $N - 1$ checks are needed.

2.2.4. Compiler problem

Finally, for programmers who are careful and patient enough, the method they use to check integer overflow may work in some cases while failing in others, especially when postcondition testing method is used.

⁶<http://www.dillo.org>.

⁷<http://www.cups.org>.

According to C99 standard [17], signed integer overflow is undefined and thus compiler-specific. For example, a wrapped value or a maximum value [12] may be returned as the result. Due to the non-determinacy of signed integer overflow, GCC developers argue that programmers should not make any assumptions on the result of the overflow. Specifically, programmers should detect an overflow before an overflow is going to happen, rather than using the overflowed result to check the existence of overflow. In other words, precondition testing should be applied to check integer overflow rather than postcondition testing. The detailed discussion between programmers and GCC developers can be found in [16].

As a result, postcondition testing statement such as `if (x>0 && x+x<0)` may be removed totally when the program is compiled with GCC, especially when it is compiled with optimization options. The Python interpreter is a victim of this issue [30].

So, fixing integer overflow is not an easy task for programmers. Programmers should first have the knowledge of IO2BO, and then be careful and patient enough, and finally choose to use the robust but expensive precondition testing when needed. Furthermore, even if programmers satisfy those requirements, the performance overhead of the code they write may be relatively high.

On the other hand, compilers can both access source code and generate efficient platform-specific assembler code, and thus it is a good choice to eliminate IO2BO vulnerabilities at compile time. With the help of compilers, IO2BO is transparent to programmers, i.e. they do not need to know what IO2BO is and how to fix IO2BO. IntPatch is such a compiler extension. In the following sections, we will describe IntPatch's method and its implementation in detail.

3. Methodology

In this section, we present a framework to automatically identify potential IO2BO vulnerabilities and fix them in three steps, as shown in Fig. 2.

First, tainted data are tracked along with the dataflow. In this process, arithmetic operations which have tainted operands are identified. These arithmetic operations are potential vulnerable operations which may trigger IO2BO vulnerabilities. If some of these operations' results flow into a memory allocation function, a potential IO2BO vulnerability is found.

Not all those identified arithmetic operations may influence a memory allocation function, i.e. the results of some arithmetic operations with tainted operands can never reach any memory allocation functions. IntPatch applies a backward slicing analysis in the second step to find out those vulnerable arithmetic operations that may trigger IO2BO vulnerabilities.

Finally, runtime check statements are inserted after those remaining vulnerable operations. If a runtime overflow occurs at check points, the program execution flow is directed to predefined error handler.

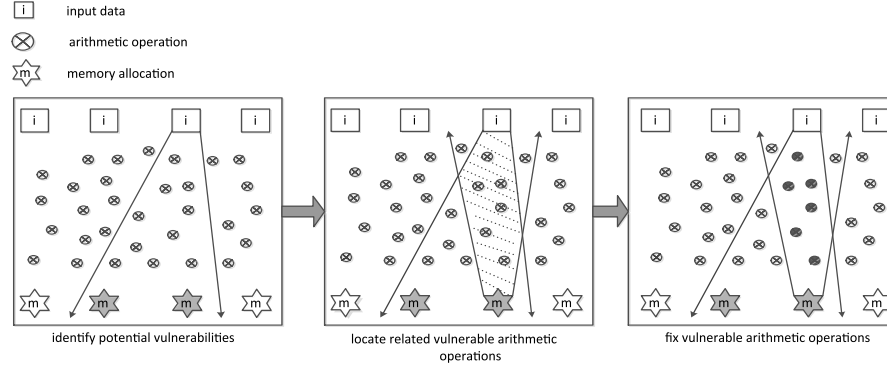


Fig. 2. Overview of IntPatch's method.

3.1. Identifying potential IO2BO vulnerabilities

IntPatch targets only IO2BO vulnerabilities. From the survey we made, IO2BO vulnerabilities have some common attributes, i.e. untrusted inputs were read in and then propagated to an arithmetic operation and finally used in a memory allocation function. Thus, a data flow analysis is suitable to identify potential IO2BO vulnerabilities.

Taking into account IO2BO vulnerabilities' characteristics, each variable's attributes `taint` and `overflow` are considered. A value is tainted (i.e., untrusted) if it originates from some program inputs. Meanwhile, a value is overflowed if it comes from an arithmetic operation which may overflow at runtime. If a tainted and overflowed value is used as a memory allocation size, a potential IO2BO vulnerability is found.

In order to track each variable's `taint` and `overflow` attributes, a type system is constructed. In this system, type inference rules are built according to each statement's semantic. Then a type analysis based on classic dataflow analysis is made to infer each variable's type.

3.1.1. Type system

Our type system is shown in Fig. 3(b). The type system forms a lattice. The bottom of the lattice is type T_{00} , representing values untainted and non-overflowed. Variables with this type are trusted, i.e. program inputs and overflowed variables never flow into these variables.

The top of this lattice is type T_{11} , representing values tainted and overflowed. Variables with this type originate from program input and some overflowed variables. If a variable with type T_{11} is used as a memory allocation size, a potential IO2BO vulnerability is found.

There are also two other types T_{10} and T_{01} , which represents for the `taint` and the `overflow` attribute respectively, in our type system.

$$\frac{\Gamma \vdash v_1 : \tau \quad v_2 := v_1}{\Gamma \vdash v_2 : \tau} \text{ (assignment)}$$

$$\frac{v := \text{if } v_{con} \text{ goto } loc_1, \text{ else goto } loc_2}{\Gamma \vdash v : T_{00}} \text{ (branch instruction)}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \quad \Gamma \vdash v_2 : \tau_2 \quad v := v_1 \geq v_2}{\Gamma \vdash v : ((\tau_1 \vee \tau_2) \wedge T_{10})} \text{ (compare ops)}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \quad \Gamma \vdash v_2 : \tau_2 \quad v := v_1 \otimes v_2}{\Gamma \vdash v : (\tau_1 \vee \tau_2 \vee T_{01})} \text{ (partial arithmetic ops)}$$

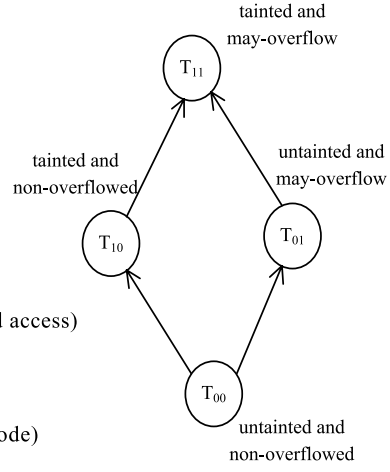
$$\frac{\Gamma \vdash v_1 : \tau \quad *v := v_1}{\Gamma \vdash *v : \tau \quad tp_v = tp_v \vee \tau} \text{ (store ops)}$$

$$\frac{v_1 \sim \mathbf{V} \quad v_2 := *v_1}{\Gamma \vdash v_2 : (\bigvee_{v \in \mathbf{V}} tp_v \vee tp_{v_1})} \text{ (load ops)}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \quad \Gamma \vdash offset : \tau_2 \quad v := v_1.offset}{\Gamma \vdash v : (\tau_1 \vee \tau_2)} \text{ (struct field access)}$$

$$\frac{\Gamma \vdash v_1 : \tau_1 \quad \dots \quad \Gamma \vdash v_n : \tau_n \quad v := \varphi(v_1, v_2, \dots, v_n)}{\Gamma \vdash v : (\tau_1 \vee \tau_2 \vee \dots \vee \tau_n)} \text{ (\varphi-node)}$$

(a)



(b)

Fig. 3. (a) Type inference rules in our system and (b) our type system.

3.1.2. Type initialization

Our type system is different from embedded type system of the C/C++ programming language. So, when applying our type system on programs, we must assign each variable to a new type.

If a variable is read from program inputs (called *sources*), type T_{10} is assigned to this variable. If a variable is used in memory allocation (called *sinks*), it may not be type T_{11} . In the process of type inference, type T_{00} is given to operands whose types have not been initialized.

File/socket reads or command line options are considered as program inputs (i.e., *sources*). Standard APIs which access program inputs are identified and related values are assigned with type T_{10} . If the target program uses third-party libraries which

wrap standard program input interfaces, a custom configuration file containing such APIs is needed. In a similar way, all sinks can be identified.

3.1.3. Type inference

In order to calculate each variable's type, type inference along with dataflow analysis is made. Each variable's type is inferred according each statement's semantic. The type inference rules for each statement are shown in Fig. 3(a).

We assume that the target program is represented in a SSA (Static Single Assignment [15]) form. In a SSA form program, each variable has only one definition. Thus, a variable can be used interchangeably with the operation which generates this variable.

A supergraph (i.e., an expanded CFG that is used for interprocedural analysis) of the target program is constructed for dataflow analysis. In the supergraph, each node corresponds to a statement in the program. Each function call statement is replaced by some assignment statements which assign the actual parameters to each formal parameter of the callee function. In addition, an edge from the call site to the callee's entry node is added. Similarly, an edge from each return instruction of the callee function to the caller is added. An statement which assigns the callee's return value (i.e., the operand of the return instruction) to the corresponding variable in the caller is added too.

Assignment statement. For each assignment, the operand's attributes are directly passed to the result. Thus, the right-hand side variable's type will be directly assigned to the left-hand side variable.

Statement which terminates a basic block. This kind of statements include direct/indirect branch instructions, direct/indirect function calls and return instructions.

For each branch instruction, its result usually is useless and can be assigned with type T_{00} . For each return instruction, it is expanded with an edge and an assignment statement in the supergraph. And thus, a similar rule as assignment rule is applied.

For each call instruction, it is expanded in the supergraph as if the callee function is an inlined function. The type inference rule can be simulated through analyzing the whole callee function statement by statement. Moreover, recursive or nested function calls will not introduce any troubles because they are represented as a big loop in the supergraph. Our dataflow analysis applies a fixed point algorithm and is able to handle loops.

However, for return instructions or indirect branch/call instructions, their jump targets may be undetermined due to the limitation of the alias analysis [21]. As a result, some type information flows are lost and some potential IO2BO vulnerabilities are ignored (i.e., there are false negatives).

Compare instruction. The result of a compare instruction is a Boolean value, i.e. True or False. If one of the operand is tainted, the result may be controlled by program input indirectly, and thus the result is tainted.

Table 1
Kinds of arithmetic operations leading to integer overflow

Arithmetic operations	May overflow?	Example (suppose register width is n)
$pos.signed + pos.signed$	✓	$(2^{n-1} - 1) + (2^{n-1} - 1) = -2$
$pos.signed - pos.signed$	×	none
$pos.signed * pos.signed$	✓	$(2^{n-1} - 1) * (2^{n-1} - 1) = 1$
$pos.signed + neg.signed$	×	none
$pos.signed - neg.signed$	✓	$(2^{n-1} - 1) - (-2^{n-1}) = -1$
$pos.signed * neg.signed$	✓	$(2^{n-1} - 1) * (-2^{n-1} + 1) = -1$
$neg.signed + pos.signed$	×	none
$neg.signed - pos.signed$	✓	$(-2^{n-1}) - (2^{n-1} - 1) = 1$
$neg.signed * pos.signed$	✓	$(-2^{n-1} + 1) * (2^{n-1} - 1) = -1$
$neg.signed + neg.signed$	✓	$(-2^{n-1}) + (-2^{n-1}) = 0$
$neg.signed - neg.signed$	×	none
$neg.signed * neg.signed$	✓	$(-2^{n-1} + 1) * (-2^{n-1} + 1) = 1$
$unsigned + unsigned$	✓	$(2^n - 1) + (2^n - 1) = 2^n - 2$
$unsigned - unsigned$	✓	$1 - (2^n - 1) = 2$
$unsigned * unsigned$	✓	$(2^n - 1) * (2^n - 1) = 1$

Arithmetic operation. Overflow could only occurs in some addition, subtraction, multiplication or left shift operations as listed in Table 1. So, the listed rule for arithmetic operation covers only these four kinds of operations. Results of arithmetic operations may overflow. Besides, if one of the operand is tainted, the result is also tainted. So, the result's type is joined by the two operands' types and T_{01} (i.e., overflowed).

Store operation. Type inference rule for memory store operation is a little more complex. In order to make a conservative analysis, for each pointer variable v , we record an additional type information tp_v , which represents the *possible Type of those memory chunks Pointed by v* . If variable v_1 with type τ is stored into a memory pointed by v , the target memory will be assigned with type τ , and the memory's type information will be joined into tp_v .

Load operation. If variable v_2 is loaded from memory pointed by v_1 , it may have a type same as any memory pointed by v_1 . Furthermore, if pointer v_1 is an alias of pointers in set V (denoted as $v_1 \sim V$), then variable v_2 's type may also be same as any memory pointed by any pointer v in V . Thus, variable v_2 's type is the upper bounds of tp_{v_1} and tp_v for each pointer v in V .

Struct field access. For instructions that access struct fields or array elements, the offset also have types. For example, when accessing an element of an array, the offset may be an expression related to program input, and thus it have the attribute `taint`. If the offset is a constant rather than a variable, then it will be treated as a variable

with type T_{00} . When the struct or the offset is tainted or overflowed, the result of this struct field access is tainted or overflowed too.

φ -node statements. A φ -node statement is a virtual statement introduced by SSA. For a basic block BB , if a variable v is defined in all BB 's predecessors (denoted as $B1, \dots, Bk$), a φ -node statement $v_{BB} = \varphi(v_{B1}, \dots, v_{Bk})$ is inserted at the beginning of BB , where v_{B1}, \dots, v_{Bk} are aliases of variable v in $B1, \dots, Bk$. For each φ -node statement $v_{BB} = \varphi(v_{B1}, \dots, v_{Bk})$, the result is one of its operands v_{Bi} , i.e. the definition of v in BB comes from Bi . Thus, the type of a φ -node statement's result may be same as any of its operands. To make a conservative analysis, the upper bound of each operand's type is assigned to the φ -node statement's result.

Miscellaneous. Remaining operations' inference rules are straightforward. Most operations' results' types are upper bounds of the operands' types. Thus, detailed rules are not listed here.

3.1.4. Data flow analysis

Type inference rules are applied along with a data flow analysis. First, for each application to be analyzed, a configuration file which defines sources (i.e., functions which read input) and sinks (i.e., memory allocation functions) is manually provided. This configuration file is read in and used to initialize each variable's type.

Then, a classic dataflow analysis is performed to calculate each variable's type. In order to handle loops and recursive function calls, we utilized a fixed point algorithm. This algorithm is well known in compiling [1], similar to the reaching definitions algorithm. And thus, the pseudo code of this algorithm is not listed here.

In addition, there is a premise for the reaching definition algorithm to halt [27], i.e. the transfer function should be monotonic. In our algorithm, the transfer function is the type inference rule. From the type system, we can conclude that this transfer function is monotonic. And thus, this algorithm will halt. However, if we consider the sanitization effect on taint attributes, i.e. tainted values are transformed to untainted, the transfer function of this algorithm is no longer monotonic and this algorithm may not halt. On the other hand, a sanitization routine itself is hard to detect. So, we do not consider sanitization in our type analysis process.

As explained above, type T_{11} should not be used as memory allocation size at sinks. If a sink's type inferred from the dataflow analysis is T_{11} , there is a type conflict, i.e. there is a potential IO2BO vulnerability.

3.2. Locating vulnerable arithmetic operations

During the type analysis, candidate IO2BO vulnerabilities are identified. In order to fix these vulnerabilities, runtime checks are inserted after arithmetic operations which may trigger IO2BO vulnerabilities.

In the previous type analysis step, arithmetic operations of type T_{11} are identified. Only these arithmetic operations may trigger real IO2BO vulnerabilities. However,

not all of these operations may reach a IO2BO vulnerability. Thus, a backward slicing [36] is used to filter out these unrelated arithmetic operations.

At each vulnerable sink, variables (i.e., memory allocation size) are being focused on. A backward slicing analysis is then used to find out other variables which may affect the focused variables. If a variable found by slicing is with type T_{11} and the corresponding statement is an arithmetic operation, this statement is thought as a vulnerable arithmetic operation.

3.3. Fixing vulnerable arithmetic operations

For vulnerable arithmetic operations, statements for checking overflow at runtime are inserted after each of them, i.e. postcondition testing is used to check overflow. Notably, IntPatch works at the link stage of a compiling process and all other optimizations have been finished yet. And thus, IntPatch can prevent runtime checks from being optimized, i.e. postcondition testing works safely.

In summary, our framework first identifies potential IO2BO vulnerabilities and ignores many unrelated arithmetic operations. Then it filters out arithmetic operations which may not trigger IO2BO vulnerabilities. Finally, runtime checks are inserted after those remaining vulnerable arithmetic operations.

4. Implementation

In this section, details of the implementation are discussed. We implement our system as a compiler extension IntPatch based on LLVM. Figure 4 shows the workflow of IntPatch.

The target program is first compiled with LLVM-GCC, a modified version of GCC for generating LLVM object code instead of ELF object code. All source files (i.e., *.c, *.cpp) are compiled into LLVM object files (i.e., *.bc). The body of IntPatch is implemented in the file IntPatchPass.cpp. This file is then compiled into a dynamic library.

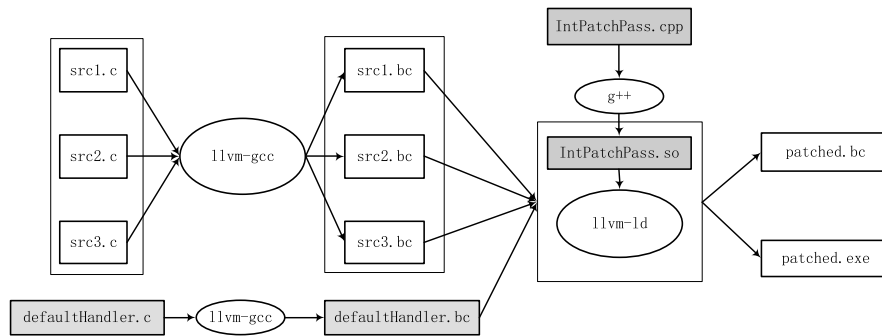


Fig. 4. Workflow of IntPatch.

The linker LLVM-LD then loads this library and invokes the analysis passes of IntPatch to analyze target programs. Vulnerable integer operations in the target program are located. The target programs are transformed. Specifically, runtime checks are inserted after each vulnerable integer operation, and the control flow is directed to a user-defined or default handler when an overflow occurs.

Then LLVM-LD links all the transformed object files and the default handler together. Finally, it generates another LLVM object file or a native executable file. The final object file or executable file is thus robust against IO2BO vulnerabilities.

The body of IntPatch consists of two analysis passes. The first pass makes a classic dataflow analysis to analyze each variable's type and identify potential IO2BO vulnerabilities. Then, for each potential vulnerability, the second pass makes a slicing to find related vulnerable arithmetic operations. Finally, check statements are inserted after those vulnerable operations to catch runtime bugs.

4.1. LLVM

LLVM [24,25] is a compiler infrastructure which supports optimization and analysis at compile time, link-time and run-time. IntPatch utilizes some useful features or interfaces provided by LLVM.

For example, LLVM provides us with an easy-to-use CFG which facilitates iterating over whole programs. All memory accesses are explicitly using load and store instructions in LLVM. Thus, our type inference rule for load and store operation is easy to apply. Moreover, LLVM's intermediate representation (IR) is in SSA (Static Single Assignment [15]) form, i.e. each use of a variable has one and only one definition, and thus facilitates our dataflow analysis. In addition, LLVM provides some intrinsic instructions for catching integer overflows at runtime. LLVM also provides some classic alias analysis pass for us to use, which helps us a lot when we perform the type analysis.

4.2. Type analysis

IntPatch uses a type analysis to identify potential IO2BO vulnerability. In LLVM, all kinds of instructions and operands are instances of class `llvm::Value`. A value which represents an instruction could be used as another instruction's operand, i.e. a value representing an instruction also represents the result of the instruction. In other words, an instruction can be used interchangeably with its result (i.e., a variable).

A predefined file which annotates what sources and sinks are is read in to initialize the mapping relationship between variables and types. IntPatch maintains a map from variables to types. Then the fixed point algorithm explained in Section 3.1.4 is performed. Along with the dataflow analysis, each variable's type is computed.

Type inference rules are applied on each instruction. Some rules are based on the result of the alias analysis, such as the rule for load operation. IntPatch utilizes

the `llvm::AliasSetTracker` interface provided by LLVM to get information about alias sets.

When the dataflow analysis reaches sinks, a type validation is performed. If variables at sinks are with type T_{11} , there is a type conflict, and thus a potential IO2BO vulnerability exists.

This type analysis process is implemented as a pass in LLVM and its result can be used by other passes. Because our analysis is interprocedural, our analysis pass needs to be invoked at link-time and is an instance of `llvm::ModulePass` which uses the whole program as a unit.

4.3. Locating vulnerable operations and patching

The type analysis can identify potential IO2BO vulnerabilities. The remaining task is to fix IO2BO vulnerabilities automatically. Fixing should be complete, i.e. if a bug is caught at runtime, it should be a real bug. In other words, a mechanism is needed to reduce false positive rates. Otherwise, users will complain about the quality of the program because the program is usually disturbed when a runtime overflow is caught.

To reduce false positives, another analysis pass is implemented to locate those vulnerable arithmetic operations and instrument runtime checks after those operations. This analysis uses classic slicing method to find variables which may flow into vulnerable memory allocation. If the related variable's type is T_{11} and the variable (i.e., instruction) is an integer arithmetic operation, a check statement is inserted after that instruction.

Intrinsic instructions provided by LLVM such as `llvm.sadd.with.overflow.*` are used by `IntPatch` to check integer overflow. These intrinsic instructions can bridge the gap between programmers and the underlying hardware. At the code generation stage, some hardware features are utilized by LLVM to implement the check in order to elevate performance. If an overflow occurs, the control flow is redirected to a predefined default handler or a user-supplied handler. The default handler that `IntPatch` provides blocks the program and waits for user debugging. Of course, in order to be user-friendly, the default handler can only send a message to the user and then exit.

Using these two analysis passes, `IntPatch` is able to automatically identify and fix IO2BO vulnerabilities in target program with a reasonable false positive rate.

4.4. An interface for programmers

At assembler code level, it is usually easy to detect overflow and redirect program's control flow, such as using `jo/jc` instructions on x86 platforms. But there are no interfaces for C/C++ programmers to query this information except writing compiler-specific inline assemble code. Fortunately, `IntPatch` can efficiently capture runtime overflows using intrinsic instructions provided by LLVM.

However, in some situations, programmers still want to capture overflows manually. In order to shield programmers from the tedious and error-prone fixing work, IntPatch also provides an easy-to-use interface. With this interface, programmers can specify what expressions to be monitored and what actions will be taken when overflow occurs in these expressions.

This interface, named `IOcheck(int exp, void (*f)())`, is implemented as an API. Programmers pass the expression to be monitored itself or any variable stored the result of the expression as the first argument, and pass the error handler function into the second argument. The second argument is by default set to `NULL`, which means we will use a handler predefined in IntPatch.

When IntPatch begins to work at the link stage, it scans all instructions in the program and finds out all call sites which call the function `IOcheck()`. IntPatch treats these function calls as memory allocation function calls. And then IntPatch works in the same way as fixing IO2BO vulnerabilities. Finally, runtime checks are inserted after vulnerable arithmetic operations to make sure the expression monitored by IntPatch does not overflow, and the control flow will be redirected to the user-defined handler if runtime overflow occurs.

However, after inserting runtime checks, instructions which call the function `IOcheck()` still exist in the program. To make sure it does not disturb the original program execution, we provide an implementation of the function `IOcheck()` which does nothing in fact. The library hosting the function `IOcheck()` will be linked by LLVM-LD.

5. Evaluation

We evaluate IntPatch with several real-world open-source applications, including *libtiff*,⁸ *ming*,⁹ *faad2*, *dillo*, *gststreamer*¹⁰ and some GNU applications and so on. The evaluation was performed on an Intel Core2 2.40 GHz machine with 2 GB memory and Linux 2.6.27.25 kernel.

5.1. Compile time

We first compiled applications with original LLVM and recorded the compile time. Then, applications were compiled with LLVM which is extended by IntPatch using the same compilation options and the time was recorded too. Figure 5 shows the recorded time result.

For applications which need to do a lot of memory allocations, such as *dillo*, *faad2*, *gststreamer*, *libtiff* and *ming*, IntPatch analysis takes much more time than original

⁸<http://www.libtiff.org/>.

⁹<http://www.libming.org/>.

¹⁰<http://gststreamer.freedesktop.org>.

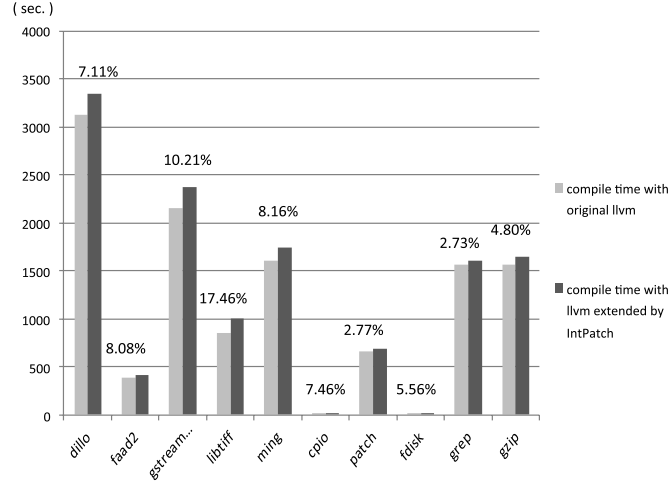


Fig. 5. Compile time with/without IntPatch (in seconds).

LLVM, about 9% more time is spent. For other applications, IntPatch consumes shorter time, which is about 5%.

Applications which allocates a lot of memory are more likely to suffer from IO2BO vulnerabilities. IntPatch tries to identify all potential IO2BO vulnerabilities and related vulnerable integer operations. Thus, compiling these applications takes more time than traditional applications.

Although IntPatch may introduce extra 5–10% time consumption when compile target programs, compile time usually is not so critical for end-user because programs usually are distributed in binary form.

5.2. Check density

We also measure how many checks IntPatch has inserted into programs. Table 2 shows, for each benchmark program, the number of total instructions in the program (in LLVM IR form), the number of arithmetic operations in the program, and the number of checks inserted by IntPatch. Then the checking ratios are calculated, i.e. (number of checks)/(number of arithmetic operations) and (number of checks)/(number of instructions).

Results show that, about every tenth arithmetic operations may trigger IO2BO vulnerabilities and are fixed by IntPatch. In fact, this ratio is a little bit higher than that in regular applications, because most of the test suites are image-related applications which needs to allocate a lot of memory. Compared to results of tools which captures all integer overflows, such as CQUAL [6] and RICH [3], the checking ratio

Table 2
Number of checks inserted

Application	#inst	#arith-ops	#checks	Checks/arith	Checks/inst
libtiff-3.8.2	781,212	20,739	1751	8.44%	0.22%
faad2-2.7	37,993	1189	150	12.6%	0.39%
ming-0.4.2	35,901	1375	241	17.5%	0.67%
dillo-2.0	641,574	8053	345	4.28%	0.05%
gststreamer-0.8.5	2,060,335	10,683	1067	9.98%	0.05%
cpio-2.9	39,153	831	28	3.37%	0.07%
patch-2.6	57,203	1035	47	4.54%	0.08%

Table 3
Performance overhead of IntPatch

Application	Original (s)	Patched (s)	Overhead
ming-0.4.2	236.143	239.549	1.44%
libtiff-3.8.2	127.571	129.123	1.01%
dillo-2.0	3.762	3.805	1.14%
faad2-2.7	361.163	364.478	0.91%

is very low. Although the checking ratio is low, most of the IO2BO vulnerabilities can be caught by IntPatch (i.e., there are few false negatives). The only false negatives are introduced by incomplete alias analysis, e.g. the targets of indirect jumps are missed.

With a low checking ratio, the patched programs' performance overheads are low. Moreover, fewer false positives exist in the patched programs if the checking ratio is low. Whereas, tools like RICH which instruments lots of checks may generate a lot of false positives. We will measure the performance and false positives in the following sections.

5.3. Performance overhead

In this section, we evaluate the performance overhead introduced by IntPatch. Each program is compiled both with original LLVM and LLVM extended by IntPatch with same compilation options. Then each program is run with a benchmark and its running time is recorded.

Our experiments show that the overhead is quite low, on average about 1%. Table 3 shows the overhead of applications patched by IntPatch relative to the uninstrumented versions (both compiled with the same options).

We test *ming*, a library for generating Macromedia Flash files (.swf), with the benchmark *PNGSuite*.¹¹ *PNGSuite* is a test-suite containing 157 different PNG for-

¹¹<http://www.schaik.com/pngsuite>.

mat images for PNG applications. These PNG files are converted into flash files using *ming* and the consumed time is recorded.

For *dillo*, we test its CSS rendering speed using a CSS benchmark devised by *nontropo*.¹² *Libtiff* is tested with a pack of TIFF format files distributed together with itself. These tiff files are compressed to JPEG format files using *libtiff* and the consumed time is recorded. For *faad2*, we use it to decode 100 MPEG-4 format videos randomly downloaded from *Mp4Point*.¹³

The performance overhead is about 1%, lower than that of RICH [3] due to the fact that fewer checks are inserted.

5.4. False positives and false negatives

IntPatch utilizes a type analysis to locate potential IO2BO vulnerabilities. The type system itself is sound according to previous work [6]. However, as explained earlier, the type analysis of IntPatch is based on a static dataflow analysis. Due to limitation of static analysis, such as a precise alias analysis is impossible, the dataflow analysis in IntPatch is not sound. For example, if the target of an indirect jump cannot be determined in static analysis, the supergraph of the target program is not complete. And thus, some type information are lost in the dataflow analysis, leading to false negatives (i.e., some potential IO2BO vulnerabilities are not identified).

However, false negative rates are hard to measure because we can hardly find out all potential vulnerabilities in real applications. In order to find out all potential vulnerabilities, usually a lot of human work are needed. As a result, we have not measured how many false negatives of IntPatch. But we believe the rates are low because the only false negatives introduced by IntPatch are due to incomplete alias analysis.

On the other hand, in order to evaluate the false positive rate of IntPatch, we test these applications instrumented by IntPatch with normal and malicious inputs. Each application is fed with 100 normal inputs, 5 known malicious inputs (i.e., those that can trigger known vulnerabilities in this program), and 10,000 malformed inputs which are generated by a custom fuzzer.

As shown in Table 4, all normal inputs do not trigger the runtime check. In other words, no false positives exist in this situation. When feeding each program with 5 malicious inputs, all of them trigger the alert. That is to say, no false positives and no false negatives exist in this situation too. In the third case, 10,000 malformed inputs are fed to each programs and false positives are generated. For example, when feeding *ming* with 10,000 random malformed inputs, 14 alerts are triggered. After manual validation, we found that 6 of them are not real vulnerabilities.

Vulnerable integer operations found by IntPatch have a type T_{11} which means the result is overflowed and untrusted. In other words, at least one of the operands is

¹²<http://nontropo.org/timer/csstest.html>.

¹³<http://www.mp4point.com>.

Table 4
False positives of IntPatch (alerts/false positives)

Application	100 normal inputs	5 malicious inputs	10,000 malformed inputs
ming-0.4.2	0/0	5/0	14/6
libtiff-3.8.2	0/0	5/0	5/0
dillo-2.0	0/0	5/0	9/9
faad2-2.7	0/0	5/0	10/0
gnash-0.8.5	0/0	5/0	31/22
InkScape-0.46	0/0	5/0	19/12
swftools-0.9.0	0/0	5/0	17/14

untrusted, i.e. it is propagated from program input. Thus, attackers can control the operand directly or indirectly and may cause the vulnerable operation to overflow. This overflowed and untrusted value will be used later in a memory allocation. Thus, attackers may exploit this IO2BO vulnerability. So, in the context of IO2BO vulnerability, overflows occurring in the related vulnerable integer operations usually cannot be benign. Thus, there are few false positives in IntPatch.

However, our type inference rules depend on third-party alias analysis results. The conservative alias analysis in LLVM we used brings some false positives, i.e. some variables that have nothing to do with program inputs may be marked as tainted by the conservative analysis. Moreover, our type analysis and slicing analysis are static analysis and path-insensitive, infeasible paths may also bring false positives to IntPatch. Even if tainted inputs trigger a runtime alert, the overflowed and tainted result can not flow into any memory allocation functions in the actual execution. During the analysis of these false positives, we figured out the existence of infeasible paths is the major cause of false positives.

For example, in *ming*, an integer overflow was caught when tested with these malformed inputs. However, the overflowed value is not passed to a memory allocation immediately. Instead, a lot of other statements include some branch instructions exist between the arithmetic operation and the memory allocation function. While we test *ming* with that malformed input, the overflowed arithmetic operation was caught at runtime. However, the program execution flows into another branch rather than the one which leads to the memory allocation. Thus, it is not a real IO2BO vulnerability (i.e., a false positive).

In addition, integer overflow checks (called sanitization routines) inserted by programmers will also lead to false positives because the sanitization routine will untaint the variable. However, the sanitization routine may cause IntPatch's fixed point algorithm not halt. So, our type analysis process does not take into account the sanitization effect in the process of type propagation. On the other hand, sanitization routines that operate at the semantic level are hard to be detected. We suggest programmers give up customized sanitization routines and use the interface `IOcheck()` provided by IntPatch.

Table 5
Zero-day bugs detected by IntPatch

Application	swftools	Inkscape	gnash	ming	faad2	libtiff
Version	0.9.0	0.46	0.8.5	0.4.2	2.7	3.8.2
# bugs	2	4	5	3	3	4

5.5. Zero-day bugs

The type analysis pass in IntPatch has generated many candidate IO2BO vulnerabilities. Of course, there are many false positives. With manual validation and dynamic testing, we can identify real vulnerabilities. During the time-consuming validation process, we discover 21 new IO2BO vulnerabilities in 6 applications, as shown in Table 5.

Some of them are exploitable and related proof-of-concept exploit payloads are constructed. However, these information is sensitive, and thus we are not going to discuss the detail here.

For example, we found a vulnerability in function `readPNG` in *ming-0.4.2*. Value `png.height` is read from an input PNG file. This value then multiplies a constant without any checks. The result of the multiplication is further used in function `malloc`. Finally, data from the input PNG file is read into the allocated memory. Attackers can control the program to allocate a smaller than expected memory. Then attacker-controlled file content are read in and written to the allocated memory buffer. The heap is then corrupted and arbitrary code execution attack may be launched. It is a typical IO2BO vulnerability.

Another example is a vulnerability in *Faad2*. This vulnerability also has the same features like other IO2BO vulnerabilities. But the statement where tainted input data is read in is far from the statement that allocates memory. They are not in the same function. Moreover, the statement which uses this allocated memory is far from the memory allocation too. They are even in different modules. Thanks to IntPatch's capacity of inter-procedure analysis, it helps us find out this kind of vulnerability.

We have submitted some of these zero-day vulnerabilities to security service providers such as Secunia and oCert. Some of the submissions, such as the vulnerability in *libtiff* (CVE-2009-2347), have been confirmed. Corresponding patches from vendors have been released or are in preparation.

5.6. Limitation

Our work is based on LLVM, which is still in the development stage. Therefore certain applications might have troubles compiling with LLVM. For example, some compiler options supported by GCC are discarded by LLVM-GCC. Furthermore, our analysis pass is a little time-consuming. These drawbacks limit the domain of IntPatch's applications.

In our implementation, IntPatch depends heavily on alias analysis. However, alias analysis is a well-known problem in static analysis. Its accuracy and performance will affect IntPatch's results.

Programmers' sanitization routines are not encouraged as mentioned above. This limitation is not friendly to programmers.

6. Related work

Many efforts have been made on integer overflow vulnerabilities.

Shuo Chen et al. [7] presented a FSM-based method uses finite state machines (FSM) to identify integer overflows. Experts summarize a finite state machine representing the integer overflow vulnerability first. Then a tool is used to check whether there are integer overflow vulnerabilities. It needs a lot of expert's effort and the FSM for applications may be different. Thus, it is not a general solution.

Ramkumar Chinchani et al. [9] describe each arithmetic operation formally and then utilize architecture characteristics to check each arithmetic operation and catch integer overflow at runtime [9]. This method does not pay much attention on distinguishing benign and unexpected overflows, thus there are a lot of false positives.

The sub-typing method presented by Brumley et al. [3] formalizes the semantics for safe integer operations in C. Overflow checks are inserted after each unsafe arithmetic operations to capture runtime overflows. It protects against many kinds of integer errors, including signedness error, integer overflow/underflow or truncation error. They implement a prototype called RICH and found several zero-day bugs too. However, benign and unexpected overflows are not distinguished either.

The method presented by Ceesay [6] utilizes type qualifiers theory [18] and a tool CQUAL [19] to detect type conflicts. Their work is implemented in the preprocessing step. They extend traditional type system with new type qualifier `trusted` similar to embeded type qualifier `const`. Then a type analysis is made and find all type conflicts. Each type conflict is reported as a potential vulnerability. Meanwhile, IntPatch focuses on the most typical integer overflow vulnerability only and tries to present a solution. IntPatch's type system is more complex and effective than CQUAL's.

GCC provides a `-ftrapv` compiler option that provides limited support for detecting integer overflows at runtime. Each overflow that occurs at runtime will cause a trap, i.e. function `abort()` will be called. David Chrisnall [10] implemented the `-ftrapv` option in another LLVM frontend—Clang. This implementation supports a user-defined handler rather than only `abort()` implemented in GCC.

Both methods target integer overflows, and suffer from the indistinguishability between benign overflows and unexpected overflows. Thus, their false positive rates are high.

David Keaton et al. presents an As-if Infinitely Ranged integer model (AIR [23]). This model either produces a value equivalent to one that would have been obtained using infinitely ranged integers or results in a runtime constraint violation. Unlike previous integer models, AIR integers do not require precise traps, and consequently do not break or inhibit most existing optimizations. This model can correctly repre-

sent integers whether they are overflowed and thus can prevent programs from being exploited by integer overflows. However, this model is not compatible with existing compilation systems and has not been implemented yet.

Yves Younan et al. presented PAriCheck [37], a novel method to reduce buffer overflows. However, IO2BO vulnerabilities are usually exploited through buffer overflow finally, and thus PAriCheck can protect IO2BO vulnerabilities from being exploited. Target programs are translated into their CIL [29] forms and then are analyzed by PAriCheck. PAriCheck's method is different from fat pointers (i.e., pointers with size and offset information). It modifies memory allocation functions and assigns each object (i.e., memory region) with a unique label. Each time a pointer arithmetic operation is met, PAriCheck looks up the operand pointer's and the result pointer's label (i.e., the labels of memory regions pointed by these pointers), and then checks whether these two labels match. If they do not match, a buffer overflow is detected. PAriCheck mainly focuses on buffer overflows and cannot handle conversions between pointers and integers, and thus there are still false negatives and false positives. In addition, PAriCheck has to modify memory allocation functions (i.e., object layout and label storage) and has to maintain a map between memory regions and labels. Thus, PAriCheck is more expensive than IntPatch.

7. Conclusion

This article surveys many IO2BO vulnerabilities, and presents a framework to model and automatically fix this kind of vulnerabilities. A prototype tool IntPatch is implemented based on LLVM. Experiments show that IntPatch is powerful and lightweight and can effectively defend against IO2BO vulnerabilities. Twenty-one zero-day vulnerabilities were found as a byproduct.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. This work is supported by the National Natural Science Foundation of China under the Grant No. 61003216 and the project "Mobile Internet Mal-behavior Detection Platform based on Cloud Computing (2010)" granted by the National Development and Reform Commission InfoSec Foundation.

References

- [1] A.V. Aho, M.S. Lam, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd edn, Addison-Wesley, Reading, MA, 2006.
- [2] S. Andersen and V. Abella, Data execution prevention. Changes to functionality in Microsoft Windows XP Service Pack 2, part 3: memory protection technologies, 2004.

- [3] D. Brumley, T. Chiueh, R. Johnson, H. Lin and D. Song, RICH: automatically protecting against integer-based vulnerabilities, in: *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'07)*, San Diego, CA, USA, 2007.
- [4] C. Cadar, D. Dunbar and D. Engler, KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs, in: *USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, San Diego, CA, USA, 2008.
- [5] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill and D.R. Engler, EXE: automatically generating inputs of death, in: *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, ACM, New York, NY, USA, 2006, pp. 322–335.
- [6] 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, in: *Detection of Intrusions and Malware and Vulnerability Assessment*, Lecture Notes in Computer Science, Vol. 4064, Springer, Berlin/Heidelberg, 2006, pp. 1–16.
- [7] S. Chen, Z. Kalbarczyk, J. Xu and R.K. Iyer, A data-driven finite state machine model for analyzing security vulnerabilities, in: *IEEE International Conference on Dependable Systems and Networks*, IEEE Computer Society, 2003, pp. 605–614.
- [8] S. Chen, J. Xu, E.C. Sezer, P. Gauriar and R.K. Iyer, Non-control-data attacks are realistic threats, in: *Proceedings of the 14th Conference on USENIX Security Symposium*, USENIX Association, Berkeley, CA, USA, 2005, p. 12.
- [9] R. Chinchani, A. Iyer, B. Jayaraman and S. Upadhyaya, Archerr: runtime environment driven program safety, in: *Proc. 9th European Symposium on Research in Computer Security*, Sophia Antipolis, French Riviera, France, 2004.
- [10] D. Chisnall, Adds -ftrapv compiler option to clang-cc, available at: <http://article.gmane.org/gmane.comp.compilers.clang.devel/4469>.
- [11] Common Vulnerabilities and Exposures, <http://cve.mitre.org>.
- [12] G.A. Constantinides, P.Y.K. Cheung and W. Luk, Synthesis of saturation arithmetic architectures, *ACM Transactions on Design Automation of Electronic Systems* **8** (2003), 334–354.
- [13] CVSS: Common Vulnerability Scoring System, Providing a universal open and standardized method for rating IT vulnerabilities, available at: <http://www.first.org/cvss/>.
- [14] CWE-680: IO2BO Vulnerabilities definition in Common Weakness Enumeration, <http://cwe.mitre.org/data/definitions/680.html>.
- [15] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman and F.K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, *ACM Transactions on Programming Languages and Systems* **13**(4) (1991), 451–490.
- [16] Discussion on signed integer overflow between programmers and GCC developers, available at: http://gcc.gnu.org/bugzilla/show_bug.cgi?id=30475#c2.
- [17] Draft of the C99 standard with corrigenda TC1, TC2, and TC3 included, <http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf>.
- [18] J.S. Foster, M. Fähndrich and A. Aiken, A theory of type qualifiers, in: *PLDI'99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ACM, New York, NY, USA, 1999, pp. 192–203.
- [19] J.S. Foster, T. Terauchi and A. Aiken, Flow-sensitive type qualifiers, in: *PLDI'02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ACM, New York, NY, USA, 2002, pp. 1–12.
- [20] P. Godefroid, N. Klarlund and K. Sen, Dart: directed automated random testing, in: *PLDI'05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, New York, NY, USA, 2005, pp. 213–223.
- [21] M. Hind, Pointer analysis: haven't we solved this problem yet?, in: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, ACM, New York, NY, USA, 2001, pp. 54–61.

- [22] Intel 64 and IA-32 Architectures Software Developer's Manuals, <http://www.intel.com/products/processor/manuals/>.
- [23] D. Keaton, T. Plum, R.C. Seacord, D. Svoboda, A. Volkovitsky and T. Wilson, As-if infinitely ranged integer model, Technical Report No. CMU/SEI-2009-TN-023, 2009.
- [24] C. Lattner, LLVM: an infrastructure for multi-stage optimization, Master's thesis, Department of Computer Science, Univ. Illinois at Urbana-Champaign, Urbana, IL, December 2002, available at: <http://llvm.cs.uiuc.edu>.
- [25] C. Lattner and V. Adve, LLVM: a compilation framework for lifelong program analysis and transformation, in: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, CA, USA, 2004.
- [26] 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 USENIX Security Symposium*, USENIX Association, San Diego, CA, USA, 2009.
- [27] S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, CA, USA 1997.
- [28] National Vulnerability Database, <http://nvd.nist.gov/>.
- [29] G. Necula, S. McPeak, S. Rahul and W. Weimer, CIL: intermediate language and tools for analysis and transformation of C programs, in: *Compiler Construction*, Lecture Notes in Computer Science, Vol. 2304, R. Horspool, ed., Springer, Berlin/Heidelberg, 2002, pp. 209–265.
- [30] Python interpreter suffers from GCC's undefined behavior taken on signed integer overflows, available at: <http://bugs.python.org/issue1608>.
- [31] K. Sen, D. Marinov and G. Agha, CUTE: a concolic unit testing engine for C, in: *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, New York, NY, USA, 2005, pp. 263–272.
- [32] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu and D. Boneh, On the effectiveness of address-space randomization, in: *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS'04)*, ACM, New York, NY, USA, 2004, pp. 298–307.
- [33] A. Sotirov, Heap feng shui in javascript, in: *Proceedings of Blackhat Europe*, Amsterdam, The Netherlands, 2007.
- [34] P. Vreugdenhil, Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit, <http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf>, 2010.
- [35] T. Wang, T. Wei, Z. Lin and W. Zou, IntScope: automatically detecting integer overflow vulnerability in x86 binary using symbolic execution, in: *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2009.
- [36] M. Weiser, Program slicing, in: *Proceedings of the 5th International Conference on Software Engineering*, San Diego, CA, USA, IEEE Press, Piscataway, NJ, USA, 1981.
- [37] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens and W. Joosen, Parichack: an efficient pointer arithmetic checker for C programs, in: *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS'10)*, ACM, New York, NY, USA, 2010, pp. 145–156.