IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time

Chao Zhang, Tielei Wang, Tao Wei, Yu Chen, and Wei Zou

Institute of Computer Science and Technology, Peking University
Key Laboratory of Network and Software Security Assurance (Peking University), Ministry of Education
{zhangchao,wangtielei,weitao,chenyu,zouwei}@icst.pku.edu.cn

Abstract. The Integer-Overflow-to-Buffer-Overflow (*IO2BO*) vulnerability is an underestimated threat. Automatically identifying and fixing this kind of vulnerability are critical for software security. In this paper, 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 dataflow analysis framework to identify potential IO2BO vulnerabilities, and then instruments programs with runtime checks. Moreover, IntPatch provides an interface for programmers to facilitate checking integer overflows. We evaluate IntPatch on a number of real-world applications. It has caught all 46 previously known IO2BO vulnerabilities in our test suite and found 21 new bugs. Applications patched by IntPatch have a negligible runtime performance loss which is averaging about 1%.

1 Introduction

The Integer Overflow to Buffer Overflow vulnerability (*IO2BO* for short), defined in Common Weakness Enumeration (CWE-680 [7]), is a kind of vulnerability caused by integer overflows, i.e. an integer overflow occurs when a program performs a calculation to determine how much memory to allocate, which causes less memory to be allocated than expected, leading to a buffer overflow.

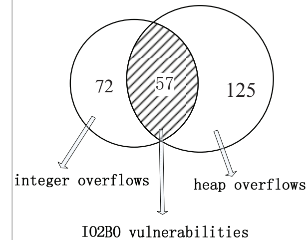
For instance, figure 1(a) shows a typical IO2BO vulnerability which existed in the old version of Faad2 [11]. In this code snippet, the argument `mp4ff_t *f` represents a mp4-file stream. Routine `mp4ff_read_int32(f)` at line 467 reads an integer value from external file `f` without any checks. The unchecked integer value (e.g. $0x80000001$) is then used in a memory allocation function at line 469. If an overflow occurs there, a smaller than expected memory (e.g. $0x80000001 * 4 = 4$) will be allocated. At line 483, some values read from external file without any checks will be written to the allocated memory chunk. Because the allocated memory is smaller than expected, these writes will corrupt the heap and may lead to arbitrary code execution [40].

D. Gritzalis, B. Preneel, and M. Theoharidou (Eds.): ESORICS 2010, LNCS 6345, pp. 71–86, 2010.
© Springer-Verlag Berlin Heidelberg 2010

```
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. }
```

(a)



(b)

Fig. 1. (a)A real-world IO2BO vulnerability in Faad2. (b)Number of vulnerabilities reported by NVD from April 1, 2009 to April 1, 2010. There are 129 (=72+57) integer overflows and 182 (=57+125) heap overflows. More than 44% (=57/129) of integer overflows are IO2BO vulnerabilities.

IO2BO is an underestimated threat. In recent years, we have witnessed that IO2BO is being widely used by attackers, such as bypassing the SSH authentication in [30] and the heap corruption attack in [40]. Moreover, according to the statistical data (from April 2009 to April 2010) in the National Vulnerability Database (NVD [17]), nearly a half of integer overflow vulnerabilities and one third of heap overflow vulnerabilities are IO2BO, as shown in Fig. 1(b).

The main reason that IO2BO is so popular is that many programmers have not yet realized the danger brought by integer overflows. Even for those who are aware of integer overflows, fixing these bugs is tedious and error-prone. For example, CUPS [4], a well-known open source printing system, has an IO2BO vulnerability in the function `_cupsImageReadPNG` [6]. CUPS first released a patch, but the initial patch failed to fix the vulnerability properly [5]. The developers had to release another patch to completely fix this vulnerability. Moreover, the C99 standard [12] specifies that signed overflow is considered as an undefined behavior, thus some patches that work properly in some compiler environments may fail in others.

Some compilers or compiler extensions such as RICH [25] have the ability to insert extra code to capture integer overflows at runtime. For example, with `-ftrapv` option, GCC can insert additional code to catch each overflow at runtime. However, there exists benign integer overflows deliberately used in random numbers generating, message encoding/decoding or modulo arithmetic [25], and thus such full instrumentation inevitably generates false positives. Furthermore, the instrumented programs usually suffer from a non-trivial performance overhead.

There are a number of integer overflow detection studies, such as [41] [38] [29] [28]. For the static-analysis-based tools, false positives are non-negligible.

Manually analyzing and patching the potential integer overflows is still error-prone. For the dynamic-analysis-based tools, the main disadvantage is their false negatives. Although many dynamic analysis systems (such as KLEE [26], EXE [27], CUTE [39], DART [35]) use symbolic execution techniques to improve code coverage and can be extended for detecting integer overflows, the analysis results are not sound.

In this paper, we present IntPatch, a tool capable of identifying potential IO2BO vulnerabilities and fixing them automatically. First, we use a type analysis to detect potential IO2BO vulnerabilities. Then, for each candidate vulnerability, another analysis pass is made to locate the points to fix at.

In the type analysis process, we consider each variable's taintedness and whether it overflows. If a tainted (thus untrusted) and maybe overflowed variable is used in a memory allocation function, there is a potential IO2BO vulnerability. In the locating and patching process, we use backward slicing [42] technique to identify those related vulnerable arithmetic operations and then insert check statements after them to catch vulnerability at runtime.

We implement IntPatch based on LLVM (Low Level Virtual Machine [36,37]) and evaluate its performance on a number of real-world open-source applications. Experiments show that IntPatch has caught all 46 previously known IO2BO vulnerabilities and it helps us find 21 zero-day bugs. These zero-day bugs are in the process of being submitted. Compared to their original versions, the patched applications have a negligible runtime performance loss which is averaging about 1%. Thus, IntPatch is a powerful and lightweight tool which can efficiently capture and fix IO2BO vulnerabilities. It could help programmers accelerate software development and greatly promote programs' security.

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

- Survey 46 IO2BO vulnerabilities and compare some of them with their patched versions. We figure out that fixing IO2BO is 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.
- Provide an API for programmers who want to fix IO2BO vulnerabilities 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 previously known IO2BO vulnerabilities.
- Identify 21 zero-day bugs in open-source applications with IntPatch.

Outline. We first describe what an IO2BO-type vulnerability is and how complicated it is when we try to fix it in Sect. 2. Our system overview and the type system we used to model IO2BO vulnerability are shown in Sect. 3. In Sect. 4,

we discuss our system's implementation, including the interface provided for programmers. Section 5 evaluates our work, and shows the performance and false positives. Related work and conclusion are discussed in Sect. 6 and Sect. 7.

2 Background

Although integer overflows may cause many other vulnerability types [23,25], the most typical case is IO2BO. In this section, we will discuss in detail what an IO2BO vulnerability is and what difficulties programmers may meet when they try to fix it.

2.1 What Is an IO2BO Vulnerability?

An IO2BO vulnerability, as defined in CWE [7], is a kind of vulnerability caused by integer overflow. Specifically, when an overflowed value (smaller than expected) is used as the size of a memory allocation, subsequent reads or writes on this allocated heap chunk will trigger a heap overflow vulnerability. A typical instance has been shown in the previous section.

Characteristics of IO2BO Vulnerabilities. We have surveyed 46 IO2BO vulnerabilities consisting of 17 bugs found by IntScope [41] and 29 bugs reported in CVE [2], Secunia [21], VUPEN [22], CERT [1] and oCERT [18].

According to the survey, we find that an exploitable IO2BO vulnerability has many significant features, similar to those presented in [41]. First, the program reads some user-supplied thus untrusted input. Then, the input value is used in an arithmetic operation to trigger an integer overflow. Finally, the overflowed value is propagated to the memory allocation function, and thus a smaller than expected memory is allocated.

Overflow in the Context of IO2BO Cannot be Benign. As mentioned in the introduction, it is difficult to distinguish integer overflow vulnerabilities from benign overflows. However, we argue that, in a context of IO2BO, the involved integer overflow cannot be benign.

More precisely, if an untrusted value triggers an integer overflow and then the overflowed result is used in memory allocation, the involved integer overflow is a real vulnerability. Usually, the overflowed result is smaller than its expected value. Besides, allocating a small memory chunk rather than a huge one doesn't cause any warnings or failures. Thus, programmers have no idea that the allocated memory is smaller than expected. It is note worthy that, further actions such as read/write will still be taken on the expected memory chunk, and then trigger buffer overflows. So, the involved integer overflow is a real vulnerability.

With this argument, we can conclude that, if an integer overflow in the context of IO2BO is caught at runtime, this overflow should be a real vulnerability. Thus, it is possible to construct a fixing mechanism with a low false positive rate for protecting against IO2BO.

2.2 How to Fix IO2BO Vulnerabilities?

Among the 46 IO2BO vulnerabilities, we investigate 18 patches of them. We find that manually fixing integer overflows is tedious. Even worse, some patches cannot fix integer overflows correctly.

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

The widely used method for checking integer overflow in practice looks like:

$$\text{if } (b \neq 0 \quad \&\& \quad (a * b)/b \neq a) \qquad \text{MSG("overflow occurs")};$$

However, this method has some problems when programs are compiled with GCC. We will discuss later.

On assembly language level, to check an integer overflow is also an annoying work. For example, 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 [13].

Fallibility and Complexity. Fixing integer overflow manually is **error-prone**. Figure 2 illustrates an erroneous patch in CUPS. Field `img->ysize` is propagated from the argument `height` which is read from external. If this field is given a big enough value, operation `img->ysize*3` may overflow first, then it will make the check in this patch useless. For example, let `img->xsize=2` and `img->ysize=0x60000000`, then `img->ysize*3` will be equal to $0x20000000$ (overflowed). Then the product of `img->xsize`, `img->ysize` and 3 overflows but this overflow cannot be caught by the check in this patch.

```
    png_get_IHDR(pp, info, &width, &height,         // untrusted source read from file
          &bit_depth, &color_type, &interlace_type, &compression_type, &filter_type);

    img->xsize = width;                              // propagate
    img->ysize = height;

-   in = malloc(img->xsize * img->ysize * 3);        // overflow occurs, and
                                                     // used in sensitive operation
+   {
+     bufsize = img->xsize * img->ysize * 3;
+     if ((bufsize / (img->ysize * 3)) != img->xsize)   // incorrect patch
+         fprintf(stderr, "…");
+   }
+   in = malloc(bufsize);
```

Fig. 2. Incorrect Patch in CUPS-1.3 for vulnerability whose ID is CVE-2008-1722 [6]

The correct method for checking overflow in this expression will take two steps. First, check whether expression `img->ysize*3` overflows. Then, check whether expression `product*img->xsize` overflows, where `product` is the product of `img->ysize` and 3.

Suppose we want to check an overflow in a long expression such as `a*b*c*d*e*f`, it follows that five sub-expressions should be checked separately. Since methods for checking each sub-expression are similar, it is too **tedious** for a programmer to manually fix integer overflows.

Compiler Problem. In this section, we will explain why the widely used method for checking integer overflow listed above will be useless when programs are compiled with GCC.

The C99 standard [10] specifies that signed overflow is considered as undefined behavior, thus implementation specific. And the GCC developers think that programmers should detect an overflow before an overflow is going to happen rather than using the overflowed result to check the existence of overflow. The detailed discussion between programmers and GCC developers can be found in [9].

As a result, the condition statement `if(a*b/b!=a)` in the widely used method 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 problem. Python developers use a check like `if(x>0 && x+x<0)` to test whether `x+x` (where `x` is a signed int variable) could overflow. However, the check may be optimized and discarded by GCC compiler [20], so that the code is still vulnerable. See [20] for more information.

So, freeing programmers from fixing integer overflows is necessary. Compilers should be responsible for fixing integer overflows.

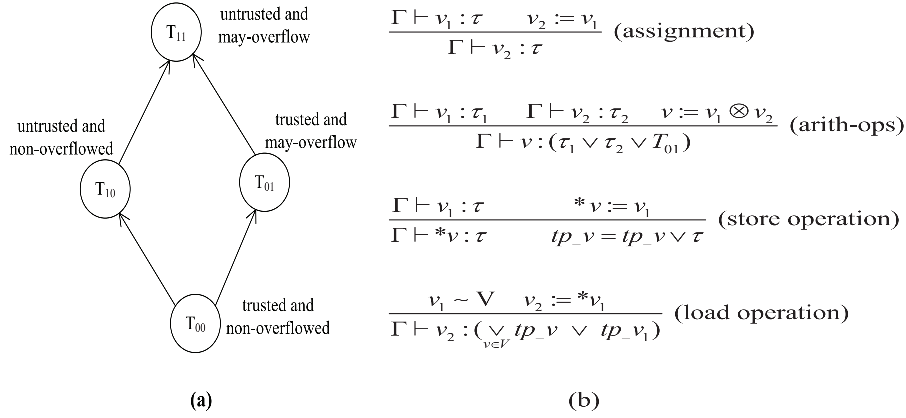
3 System Overview

In this section, we describe the overview of our system which is aimed at fixing IO2BO vulnerabilities automatically. To fix IO2BO vulnerabilities, we must identify them first. According to the features of IO2BO vulnerabilities, we use a type analysis to detect them. Then another analysis is made upon those candidate vulnerabilities to decide which points to fix at. Finally, runtime check statements are inserted at those points.

3.1 Identify Potential IO2BO Vulnerabilities

As mentioned above, an IO2BO vulnerability has some significant features. Thus, properties of variables, such as whether they are trusted and whether they may be overflowed, are considered. Then a type system is constructed and a type analysis to identify potential IO2BO vulnerabilities is made.

Type System. Figure 3(a) shows our type system. Our type system forms a lattice. The bottom of the lattice is type T_{00}. Variables with this type are trusted, i.e. their values are not from program input, and non-overflowed. The top of this lattice is type T_{11}, which represents for untrusted and may-overflow. Variables with this type origins from program input, and origins from some variables possibly overflowed. Our type system also has another two types T_{10} and T_{01} which respectively represents for untrusted and may-overflow.



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

$$\frac{\Gamma \vdash v_1 : \tau_1 \qquad \Gamma \vdash v_2 : \tau_2 \qquad v := v_1 \otimes v_2}{\Gamma \vdash v : (\tau_1 \vee \tau_2 \vee T_{01})} \text{ (arith-ops)}$$

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

$$\frac{v_1 \sim V \qquad v_2 := *v_1}{\Gamma \vdash v_2 : (\bigvee_{v \in V} tp_v \ \vee \ tp_v_1)} \text{ (load operation)}$$

Fig. 3. (a)Our type system, (b)type inference rules in our system

If a variable with type T_{11} is assigned to a variable which expects type T_{00}, there is a type conflict, which means there is a potential IO2BO. Due to the characteristics of IO2BO vulnerabilities, other type casting are allowed.

Type Initialization. Our type system is different from embeded type system of the C/C++ programming language. So, when applying our type system on programs, we must assign each variable with a type. It is impossible to assign each variable with a type manually. We just assign variables at key points with specific types. For example, if a variable is read from program input (called **sources**), then type T_{10} will be assigned to it. If a variable is used in memory allocation (called **sinks**), it will be assigned with type T_{00}. Then, following type inference rules are used to decide the remainder variables' types.

Type Inference. Figure 3(b) shows our type inference rules.

Assignment Statement. The right-hand side variable's type will be directly assigned to the left-hand side variable.

Arithmetic Operation. Overflow could only occurs in addition, subtraction, multiplication or left shift operation. So, the listed rule for arithmetic operation covers only these four kinds of operations. The result's type is joined by the

two operands' types and T_{01}. It means that, the result may overflow, and is untrusted if any one of its operand is untrusted.

Store Operation. Type inference rule for memory store operation is a little 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. Besides, if pointer v_1 alias to pointers in set V (denoted as $v_1 \sim \mathrm{V}$), 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.

Misc. Remaining operations' type inference rules are straightforward. Thus they are not listed here.

Type Analysis Process. 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 our type system. Then, a dataflow analysis applying our type inference rules is made. As explained above, type T_{00} is expected at sinks. If the type inferred from the dataflow analysis is T_{11}, there is a type conflict, i.e. there is a potential IO2BO vulnerability.

3.2 Locate Vulnerable Arithmetic Operations and Patch

After the type analysis, some candidate IO2BO vulnerabilities are generated. The type analysis is conservative, and thus it is sound (i.e. there is no false negatives). However, this type analysis is path-insensitive, thus there may be many infeasible paths which are reported as IO2BO vulnerabilities. Besides, the alias analysis in LLVM we used is conservative, it may also introduce additional false positives. Leaving all these candidate vulnerabilities for programmers to validate is terrible. In this section, we introduce an automatic fixing mechanism which can reduce false positives and protect programs against IO2BO vulnerabilities.

First, our approach identifies those related vulnerable arithmetic operations (i.e. overflow occurs here will further triggers the IO2BO vulnerability). Then, for each vulnerable arithmetic operation, statements for checking overflow at runtime are automatically inserted after it.

To locate vulnerable arithmetic operations, a backward analysis is made for each candidate IO2BO vulnerability. Variables at each vulnerable sink are focused. Techniques like backward slicing [42] are then used to find other variables which may affect the focused variable. 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. Finally, statements for checking overflow at runtime are inserted after those vulnerable arithmetic operations.

As argued in Sect. 2.1, integer overflows in the context of IO2BO are usually vulnerable. Thus, integer overflows caught by this fixing mechanism at runtime are real vulnerabilities, i.e. this fixing mechanism can reduce false positives.

4 Implementation

In this section, we present the implementation of our system. We implement our system as a tool IntPatch based on LLVM. Figure 4 shows the structure of IntPatch.

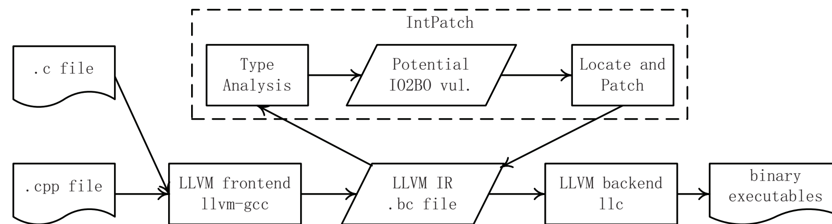


Fig. 4. Structure of IntPatch

IntPatch first makes an classic dataflow analysis to analyze each variable's type and identify potential IO2BO vulnerabilities. Then, for each potential vulnerability, it makes a slicing to find the vulnerable arithmetic operations. Finally, check statements are inserted after those vulnerable operations to catch runtime bugs.

4.1 LLVM

LLVM [36,37] is a compiler infrastructure which supports effective optimization and analysis at compile time, link-time, run-time and offline. IntPatch utilizes some useful features or interfaces provided by LLVM.

For example, LLVM provides us an easy-to-use CFG which facilitates iterating over whole program. 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 be applied. LLVM's intermediate representation (IR) is in SSA (Static Single Assignment [32]) form and thus facilitates our dataflow analysis. In addition, LLVM provides some intrinsic instructions for catching integer overflows. LLVM also provides some classic alias analysis pass for us to use, which helps us a lot when we make 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. That is to say, a value representing an instruction also represents the result of the instruction, thus can be thought as a variable.

We maintain a map from such variables to types. Because LLVM's IR is in SSA form, each variable has only one definition point. Thus, the type information of any variable won't change.

A predefined file which annotates what are sources and sinks is read in to initialize the mapping relationship between variables and types. Then we use classic dataflow analysis method [24] to analyze each variable's type. Type inference rules are applied on each instruction. At each basic block's entry, there may be some phi-nodes [32], which are introduced by SSA. For each of these phi-nodes, such as $v = \phi(v_1,\ v_2,\ ...,\ v_n)$, we join types of variable $v_1,\ v_2,\ ...,\ v_n$ together and assign it to variable v.

When the dataflow analysis analyzes variables at sinks, we do a type check here. If variables at sinks are with type T_{11} according to the analysis's result, 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 is an instance of `llvm::ModulePass` and needs to be invocated at link-time.

4.3 Locate Vulnerable Arithmetic Operations and Patch

The type analysis can identify potential IO2BO vulnerabilities. Our remainder 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 word, a mechanism is needed to reduce false positive rates. Otherwise, users will complain about the program's quality.

We implement another analysis pass to identify those vulnerable arithmetic operations. This analysis uses classic slicing method [42] to find related variables. If the related variable's type is T_{11} and the variable (i.e. instruction) is an arithmetic operation, a check statement is inserted after that instruction. We use intrinsic instructions provided by LLVM such as `llvm.sadd.with.overflow` to check integer overflow. If an overflow occurs, we redirect the control flow to a predefined function. By default, this function blocks the program and waits for user debugging. This function can also be specified by programmers.

Using these two analysis pass, IntPatch is able to automatically identify and fix IO2BO vulnerabilities over full programs with a reasonable false positive rate.

4.4 Another Compiler Interface

However, in some situations, programmers still want to fix IO2BO vulnerabilities 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 into the first argument, and pass the overflow handler function into the second argument. The second argument is default set to NULL, which means we will use a handler predefined in IntPatch.

In order to support this API, we need to make a few modifications to the original analysis. In the type analysis process, we just treat the first argument of function `IOcheck()` as sinks. And in the slicing process, we just need to change the inserted overflow handler function to the handler specified by programmers. Besides, we provide an library for function `IOcheck()` which does nothing in fact. This library will be linked by LLVM.

5 Evaluation

We evaluate IntPatch with several real-world open-source applications, including libtiff [14], ming [15], faad2 [11], dillo [8], gstreamer [12] and so on. The evaluation was performed on an Intel Core2 2.40GHz machine with 2GB memory and Linux 2.6.27.25 kernel.

5.1 Check Density

We first measure how many checks IntPatch inserts into programs. Table 1 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 the IntPatch. Then the checking ratio is calculated, i.e. (number of checks)/(number of arithmetic operations).

Table 1. Number of checks inserted

application	# inst	# arith-ops	# checks	ratio
libtiff-3.8.2	781212	20739	1751	8.44%
faad2-2.7	37993	1189	150	12.6%
ming-0.4.2	35901	1375	241	17.5%
dillo-2.0	641574	8053	345	4.28%
gstreamer-0.8.5	2060335	10683	1067	9.98%

Results show that, there are lots of arithmetic operations (about one tenth) which may affect memory allocations. 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 in [28] and [25], the checking ratio is very low.

5.2 Performance Overhead

In this section, we present the performance overhead of IntPatch. Our experiments show that the overhead is quite low, on average about 1%. Table 2 shows

Table 2. Performance 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%

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 benchmark PNGSuite [19]. PngSuite is a test-suite containing 157 different PNG format 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 *nontroppo* [3]. *Libtiff* is tested with a pack of TIFF format files distributed together with it. 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 [16].

5.3 False Positives and False Negatives

As mentioned above, our type analysis is conservative, and thus our analysis is sound (i.e. no false negatives). In other words, any vulnerability that satisfies IO2BO's features will be caught by the type analysis.

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 normal inputs described in Sect. 5.2 and with $2 \sim 3$ malicious inputs (e.g. crafted image files). Results show that all normal inputs don't trigger the runtime check and while malicious inputs both trigger the check. That is to say, no false positives exist. However, the test is not sufficient and the the code coverage rate is low, and thus IntPatch may still has false positives.

In fact, our type analysis and slicing analysis are path-insensitive, infeasible paths may bring false positives to IntPatch. The conservative alias analysis in LLVM we used also brings some false positives.

In addition, integer overflow checks (called sanitization routine) inserted by programmers will also lead to false positives. That is because the sanitization routine will untaint the variable, but our type analysis process hasn't considered this semantic effection on the type propagation. On the other hand, sanitization routines are at semantic level and hard to be detected. One possible solution is that programmers give up customized sanitization routines and use the interface `IOcheck()` provided by IntPatch only.

5.4 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, we can identify real vulnerabilities. During our unfinished time-consuming validation process, we discover 21 new IO2BO vulnerabilities in 6 applications, as shown in Table 3.

Table 3. 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

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. It is a typical IO2BO vulnerability.

We have submitted some of these zero-day vulnerabilities to security service provider such as Secunia [21] and oCert [18]. Some of the submissions, such as the vulnerability in libtiff (CVE-2009-2347), have been confirmed. Corresponding patches from vendors has been released or are in progress. Considering that other vulnerabilities are still in the process of being submitted or fixed, we do not want to provide further detailed information here.

5.5 Limitation

Our work is based on LLVM, which is still in development stage. So certain applications might have troubles being compiled with LLVM. Furthermore our analysis pass is 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. Followings are some representative works.

Shuo Chen et al. presented a FSM-based method [29] and 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 distinct applications may be different. Thus, it is not a general solution.

Ramkumar Chinchani et al. [31] describe each arithmetic operation formally and then utilize architecture characteristics to check each arithmetic operation and catch integer overflow at runtime [31]. This method doesn't pay much attention on distinguishing benign and unexpected overflows, thus there are lots of false positives.

The sub-typing method presented by Brumley et al. [25] formalizes the semantics for safe integer operations in C. Overflow checks are inserted after each arithmetic operations to capture runtime overflows. It protects against many kinds of integer errors, including signedness error, interger 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 [28] utilizes type qualifiers theory [33] and a tool CQUAL [34] 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.

Both of these methods treat all kinds of integer overflow vulnerabilities, and suffer from the indistinguishability between benign overflows and unexpected overflows. Thus, their false positive rates are high.

Our paper focus on the most typical integer overflow vulnerability and tries to present a sound solution. Our type system is more complex and effective than Ceesay's. The final result shows that our method is effective.

7 Conclusion

This paper surveys many IO2BO vulnerabilities, and presents a framework to model and automatically fix this kind of vulnerability. 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 are found as a byproduct.

References

1. Carnegie Mellon University's Computer Emergency Response Team, `http://www.cert.org/advisories/`
2. Common vulnerabilities and exposures, `http://cve.mitre.org`

3. Cssbench: a css benchmark devised by nontroppo, http://www.howtocreate.co.uk/csstest.html
4. CUPS: a standards-based, open source printing system developed by Apple Inc., http://www.cups.org/
5. Cups' erroneous patch, http://www.cups.org/str.php?L2974
6. CUPS Vulnerability, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1722
7. Cwe-680: Io2bo vulnerabilities, http://cwe.mitre.org/data/definitions/680.html
8. Dillo: a lightweight browser, http://www.dillo.org
9. Discussion between programmers and gcc developers, http://gcc.gnu.org/bugzilla/show_bug.cgi?id=30475#c2
10. Draft of the c99 standard with corrigenda tc1, tc2, and tc3 included, http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf
11. FAAD2: A MPEG-4 and MPEG-2 AAC Decoder, http://www.audiocoding.com/faad2.html
12. GStreamer: a framework for streaming media applications, http://gstreamer.freedesktop.org/
13. Intel 64 and ia-32 architectures software developer's manuals, http://www.intel.com/products/processor/manuals/
14. libtiff: TIFF Library and Utilities, http://www.libtiff.org/
15. Ming: a library for generating Macromedia Flash files, http://www.libming.org/
16. Mp4point: a source for free mp4 / mpeg-4 video movie clips, http://www.mp4point.com/
17. National vulnerability database, http://nvd.nist.gov/
18. oCERT: Open Source Computer Emergency Response Team, http://www.ocert.org/
19. Pngsuite: The "official" test-suite for png applications like viewers, converters and editors, http://www.schaik.com/pngsuite/
20. Python interpreter suffers from gcc's behavior, http://bugs.python.org/issue1608
21. Secunia: a Danish computer security service provider, http://secunia.com/
22. Vupen: a company providing security intelligence, http://www.vupen.com/english/
23. Ahmad, D.: The rising threat of vulnerabilities due to integer errors. IEEE Security and Privacy 1(4), 77–82 (2003)
24. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Princiles, Techniques, and Tools, 2nd edn. Addison-Wesley, Reading (2006)
25. Brumley, D., Chiueh, T.c, Johnson, R., Lin, H., Song, D.: Rich: Automatically protecting against integer-based vulnerabilities. In: Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS 2007) (2007)
26. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), San Diego, CA, USA (2008)
27. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006 (2006)

28. Ceesay, E., Zhou, J., Gertz, M., Levitt, K., Bishop, M.: Using type qualifiers to analyze untrusted integers and detecting security flaws in c programs. Detection of Intrusions and Malware & Vulnerability Assessment (2006)
29. Chen, S., Kalbarczyk, Z., Xu, J., Iyer, R.K.: A data-driven finite state machine model for analyzing security vulnerabilities. In: IEEE International Conference on Dependable Systems and Networks, pp. 605–614 (2003)
30. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: Proceedings of the 14th Conference on USENIX Security Symposium, p. 12 (2005)
31. Chinchani, R., Iyer, A., Jayaraman, B., Upadhyaya, S.: Archerr: Runtime environment driven program safety. In: 9th European Symposium on Research in Computer Security, Sophia Antipolis (2004)
32. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph (1991)
33. Foster, J.S., Fähndrich, M., Aiken, A.: A theory of type qualifiers. In: PLDI 1999: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, pp. 192–203. ACM, New York (1999)
34. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany, pp. 1–12 (2002)
35. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: PLDI 2005: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 213–223 (2005)
36. Lattner, C.: LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL (December 2002), `http://llvm.cs.uiuc.edu`
37. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO 2004), Palo Alto, California (March 2004)
38. Molnar, D., Li, X.C., Wagner, D.A.: Dynamic test generation to find integer bugs in x86 binary linux programs. In: Proceedings of the 18th USENIX Security Symposium (2009)
39. Sen, K., Marinov, D., Agha, G.: 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, pp. 263–272 (2005)
40. Sotirov, A.: Heap feng shui in javascript. In: Proceedings of Blackhat Europe (2007)
41. Wang, T., Wei, T., Lin, Z., Zou, W.: 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)
42. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering (1981)