IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution

Tielei Wang†, Tao Wei†*, Zhiqiang Lin‡, Wei Zou†

†Key Laboratory of Network and Software Security Assurance
Institute of Computer Science and Technology, Peking University
{wangtielei, weitao, zouwei}@icst.pku.edu.cn

‡Department of Computer Science
Purdue University
zlin@cs.purdue.edu

Abstract

The number of identified integer overflow vulnerabilities has been increasing rapidly in recent years. In this paper, we present a system, IntScope, which can automatically detect integer overflow vulnerabilities in x86 binaries before an attacker does, with the goal of finally eliminating the vulnerabilities. IntScope first translates the disassembled code into our own intermediate representation (IR), and then performs a path sensitive data flow analysis on the IR by leveraging symbolic execution and taint analysis to identify the vulnerable point of integer overflow. Compared with other approaches, IntScope does not run the binary directly, and is scalable to large software as it can just symbolically execute the interesting program paths. Experimental results show IntScope is quite encouraging: it has detected more than 20 *zero-day integer overflows (e.g., CVE-2008-4201, FrSIRT/ADV-2008-2919) in widely-used software such as QEMU, Xen and Xine.*

1. Introduction

Primitive types, including integers, typically have a fixed size (e.g., 32 bits) for a particular machine architecture. Thus, variables of these types have a maximum value (e.g., 2^{32}). Operations that result in a value greater than this maximum can cause the value to wrap-around: this well-known condition is called *overflow*. Integer overflows are particularly insidious: while the overflow itself is usually not exploitable, it often leads to other classes of vulnerabilities, including stack and heap overflows.

Recently, the number of identified integer overflow vulnerabilities has been increasing rapidly, as shown in Table 1. From 2000 to 2007, almost every year, 1.5 times more integer overflows are recorded by National Vulnerability Database (NVD [16]). Moreover, nearly 60% (219/364) of these vulnerabilities have the highest severity (with score 7 – 10) [16]. According to the 2006 CVE report [5], integer overflows rose to the second most common vulnerability in the advisories from operating system vendors. Furthermore, in the last 3 years, we have witnessed a number of integer overflow attacks, such as 0-day exploits against Adobe PDF [27], Flash [13], and the Microsoft WebViewFolderIcon ActiveX control [26].

Year	2000	2001	2002	2003	2004	2005	2006	2007
Num	1	2	5	29	40	66	96	125

Table 1. The reported number of integer overflows in NVD from 2000 to 2007.

It is important to identify integer overflows before an attacker does. Given program source code, there are several techniques and tools (e.g., RICH [32], EXE [36], KLEE [35]) that can perform static analysis or model checking to detect integer overflows. However, as source code is not always available to end users, the state-of-the-art techniques have to rely on dynamically running the software, exploring program paths (e.g., SAGE [44]), and generating test cases, to show the existence of a vulnerability. Such fuzzing techniques have been commonly used by underground attackers. However, to the best of our knowledge, none of the binary-based fuzz techniques (including SAGE [44]) focus specifically on integer overflows; and even if the fuzzing tool could explore all program paths, the vulnerability is still invisible if the fuzzing tool does not generate a desired input.

In this paper, we present a systematic static binary analysis based approach to particularly focus on detecting integer overflow vulnerabilities when given only a binary, with the goal of finally eliminating the vulnerabilities. Based on the observation that most known integer overflow vulnerabilities are caused by incomplete or improper input validation, we apply symbolic execution and data flow analysis (i.e., taint analysis) techniques to automatically detect the program

*Corresponding author

paths which lack proper input validation. Our approach is static in the sense that it does not run the software directly in a real environment, and instead it relies on a symbolic execution engine we developed to scan the program paths and identify integer overflows.

The key ideas of our approach are (1) symbolically executing the x86 binary on an SSA-like [28] intermediate representation; (2) using taint analysis, not only tracking the propagation of taint property, but also modeling accurate bounds of each tainted piece of data via symbolic execution; and (3) using a lazy checking, i.e., instead of checking each arithmetic operation, our approach only checks whether the tainted symbolic value used in sensitive points (e.g., memory allocation functions `malloc`, `alloca`) could overflow under path constraints. In short, our approach only reports a potential integer overflow vulnerability if the tainted symbolic value used in sensitive points could overflow.

We have implemented our system called IntScope. To enable our analysis, we first use IDA Pro [12], a Windows and Linux hosted multi-processor disassembler and debugger, to disassemble an executable, and then we translate the disassembled file into our own intermediate code, PANDA (Program ANalysis Dedicated to ASM), on which our analysis is performed. IntScope handles both Windows and Linux x86 binaries. We have evaluated it with a number of real-world programs from Microsoft Windows and Linux platform, and we have reached very encouraging experimental results: IntScope not only successfully detects all the known integer overflow bugs, but also identifies more than 20 new integer overflow vulnerabilities (e.g., CVE-2008-4201, FrSIRT/ADV-2008-2919) in widely used applications, such as QEMU [18], Xen [20], Xine [25], Mplayer [15], and VLC [23]. All of the new vulnerabilities (i.e., the 0-day ones) have been confirmed by our dynamic testing tool [46] and/or the developers. In addition, after we reported our results to the developers, some projects have already released patches (e.g., Xine [25], MPD [14], Hamsterdb [11], Faad2 [7]).

Our contributions include:

- We propose a systematic method of combining taint analysis and path-sensitive symbolic execution to specifically detect integer overflow vulnerabilities in executables.
- We devise an intermediate instruction representation, based on IDA Pro's disassembled code, and a symbolic execution engine.
- We implement a prototype called IntScope and use it to analyze real-world binaries. Experimental results show that our approach is highly effective and is able to detect 0-day integer overflow vulnerabilities.

2 Problem Statement

In this section, we first describe the common features of integer overflow vulnerabilities, and then discuss the challenges in binary level integer overflow detection and describe the problem scope of this paper.

2.1 Features of Integer Overflow

We have conducted more than 200 integer overflow case studies, and we found that integer overflow vulnerabilities usually have the following features:

I. Untrusted source – For most arithmetic operations where integer overflow occurs, there is an operand which is arithmetically derived from some tainted data. Tainted data is derived from untrusted input sources like network messages, input files, or command line options. There are several typical source functions such as `read`, `fread` and `recv` which could introduce tainted data.

II. Various types of sinks – Whether or not an integer overflow is harmful depends on where and how the program uses the overflowed value. It is very dangerous when an overflowed value is used in some sensitive points, since it may lead to other vulnerabilities. These sensitive points are called *sinks*. Particularly, from our case studies, we found the sinks for integer overflows are usually these points:

- **Memory allocation**: The overflowed value is used in memory allocation functions (e.g., `malloc`, `alloca`) as a size argument, and it usually results in an insufficient memory allocation, which may eventually become an attacker's springboard to a buffer overflow.
- **Memory access**: The overflowed value is used as an array index or a pointer offset, which may cause arbitrary bytes memory overwritten or read (e.g., an information leakage attack).
- **Branch statement**: The overflowed value is used in a branch statement, and the branch statement is not designed to catch the integer overflow. It could lead to a bypass of security checks (e.g., one of the cases in non-control-data attack [38]) or result in an undesirable execution.
- **Other program-dependent sensitive points**: There may be other program-dependent sensitive points which could be affected by the overflowed values. For instance, in the integer overflow vulnerability in NetBSD (CVE-2002-1490), its overflowed value is used as a structure reference counter, which causes a shared object (still in use) to be freed prematurely.

III. Incomplete or improper sanitization checks – Not all the tainted data can lead to integer overflows, because tainted values could be safely used in a program after careful checks. However, almost all the subtle integer overflow vulnerabilities are actually caused by incomplete or improper checks.

2.2 Challenges

As we aim to detect integer overflow vulnerabilities directly in x86 binaries without executing the program, there are a number of challenges to overcome.

- **Lack of type information**: Type information is usually not available in executables, and the only real information enforced is the operand size (i.e., 8, 16, 32, and 64 bit) in x86 instructions. Consider the instruction sequence:

```
mov eax, 0xffffffff;//eax = 0xffffffff
add eax, 2;         //eax = 0xffffffff+0x2
```

  we cannot determine whether `0xffffffff+0x2` is an integer overflow; if `0xffffffff` is interpreted as an `int` type value, `0xffffffff+0x2` is equivalent to `-1+2`, which is a normal addition operation; if `0xffffffff` is interpreted as an `unsigned int`, `0xffffffff+0x2` will certainly overflow. Since we are dealing with integer overflows, we have to recover all the possible use of integers in binary code, and make our analysis specifically focus on the data flow of these integers.

- **Differentiating benign integer overflow**: Even if we could recover type information, we still need to deal with benign integer overflow operations (i.e., harmless integer overflows). Benign integer overflow operations do exist in binary code. Programmers (even compilers) may use integer overflows deliberately. For example, if `x` is an `int` type variable, the statement `if(x >= -2 && x<= 0x7ffffffd)` will be translated into such a piece of assembly code by GCC-4.2.0 compiler:

```
mov eax, x;  // eax = x
add eax, 2;  // eax = eax+2
js target
```

  In this case, a large `x` such as `0x7fffffff` causes an overflow in above `add` instruction, but it is harmless as GCC actually uses this overflow to reduce a comparison instruction. We cannot treat benign integer overflow operations as vulnerabilities, and hence we have to differentiate benign integer overflows.

- **Path explosion**: As integer overflows are mainly caused by incomplete or improper sanitization checks, we need to analyze program paths to determine the existence of sanitization checks, and then to further identify the incomplete or improper checks. Therefore, our analysis is path sensitive. However, the number of paths in real-world software is too large (even infinite) to explore all paths. Although there are several methods to reduce the path explosion problem, such as the use of function summaries [45], it is still a major challenge.

2.3 Problem Scope

Integer overflows are one type of integer related bugs. There are other types of integer bugs, such as assignment truncation, integer underflow, and signedness errors [32]. In this paper, we focus on the first type of integer bugs, namely, integer overflows, which amount to nearly 70% of all the integer bugs in the study conducted by Brumley et al. [32]. We leave other types as future work.

3 System Design

In this section, we first give an overview of IntScope and then describe its detailed design.

3.1 System Overview

The intuition behind our approach is that most integer overflow vulnerabilities are caused by the misuse of overflowed values in sinks (e.g., memory allocation functions). Thus, we track the propagation of tainted data, collect path constraints on tainted data, and check whether a path has sufficient checks to prevent integer overflows when tainted values are used in sinks.

Instead of checking whether each arithmetic operation could overflow, we only check the tainted value used in sinks. We call it "lazy checking". Lazy checking can help differentiate real integer overflow bugs from benign ones, and also reduce the number of checks. Another advantage of lazy checking is that most sinks mentioned in Section 2 have already provided some hints on the type inference.

Unlike traditional static analysis, our approach only analyzes certain parts of a program to alleviate the path explosion problem. Most integer overflow vulnerabilities have a prominent feature, i.e., tainted data are introduced by a source function (e.g., `fread`, `recv`) and flow into a sink function (e.g., `malloc`, `alloca`). Hence, our approach only analyzes those program paths which pass through a source function and reach a sink function.

At a high level, IntScope takes an executable file as input and outputs the suspicious paths, along which some overflowed variables are used in sinks. Figure 1 shows the architecture of IntScope. Given a binary program *P* to be analyzed, IntScope works as follows:

- **Pre-processing the program.** IntScope first uses our Decompiler, which translates *P* into an SSA-like intermediate representation called PANDA [51]. The Decompiler also builds control flow graphs (CFG) *G* of

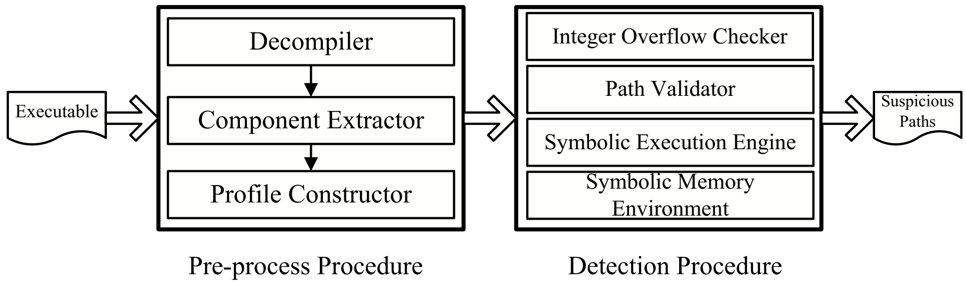


Figure 1. An Overview of IntScope.

functions in *P* and a call graph (CG) *C* of *P*. To alleviate path explosion, our Component Extractor and Profile Constructor are used to compute the “chop” graphs *G’* of *P*. *G’* includes those program paths which pass through a source function and reach a sink function. The output of this procedure is the “chop” graphs *G’*.

- **Detecting integer overflows.** In this procedure, IntScope traversals *G’* using a depth-first search. It maintains a symbolic memory environment, and symbolically executes the x86 binary on our PANDA representation. At the same time, IntScope tracks the propagation of tainted data. At each branch point, IntScope uses the Path Validator to check which branch is feasible under current path constraints. If both branches are feasible, IntScope will fork an additional execution and simulate each branch. At each sink (e.g., `malloc`, `alloca`) where a tainted value is used, the Integer Overflow Checker will enforce a constraint check on that value. Once the tainted value could overflow, IntScope outputs the path as a suspicious one.

3.2 Detailed Design

In this section, we describe the detailed design of IntScope. We first present the design of our PANDA intermediate representation in Section 3.2.1 and discuss how to compute the chop graphs of a program in Section 3.2.2. In Section 3.2.3 and Section 3.2.4, we introduce the design of the symbolic memory environment and execution strategies. Finally, we describe our lazy checking in Section 3.2.5.

3.2.1 The Design of PANDA

It is a significant challenge to directly analyze x86 instructions as the x86 instruction set is very complex. It has hundreds of instructions (many with side-effects), and there is no notion of variables in executables. Further, tainted data can be propagated between registers and memory locations. For example, after executing the following instructions:

```
mov [esp+4], eax // [esp+4] = eax
push 0
mov ebx, [esp+8] // ebx = [esp+8]
```

`ebx` is actually assigned `eax`; if `eax` is tainted, so is `ebx`. However, we cannot retrieve such information unless we can infer the two symbolic address `esp+4` and `esp+8` are actually referring to the same memory location.

Inspired by recent work such as Vine [22], Boomerang [40], and DIVINE [31], we devise an SSA-like IR, named PANDA, whose grammar is shown in Figure 2. We convert x86 assembly into our PANDA representation based on our previous work [51, 52].

statement	::=	$var :: \tau[e]$ | $val := var$ | $var = e$ | if $(e\Delta_b e)$ then l_1 else l_2 | goto l_1 | $var = function(e, e, ...)$ | return
e	::=	num| *val*| $e\Delta_{op}e$
τ	::=	qword|dword|word|byte
Δ_b	::=	$==, !=, <_s, \leq_s, >_s, \geq_s, <_u, \leq_u, >_u, \geq_u$
Δ_{op}	::=	+, -, *, /, &, |, ⊕, <<, >>
val, *var*, *function*	::=	string (value,variable and function’s name)

Figure 2. PANDA grammar.

There are six basic statements in PANDA: (1) *definition* statement: $var :: \tau[e]$ is used to define a variable, where *var* is the name of a variable, τ is *var*’s bit length (e.g., qword, dword, word, byte) and *e* is *var*’s memory address; (2) *reference* statement: $val := var$, similar to the Phi node in standard SSA form [28], it generates a new variable *val* when *var* has an unknown value or multiple possible values; (3) *assignment* statement: $var = e$ assigns variable *var* with value *e*; (4) *branch* statement: `if(`$e\Delta_b e$`) then` l_1 `else` l_2, where Δ_b stands for comparison operators; (5) *call* statement, *var* $=function(e_1,...)$ and (6) *return* statement, which means function exit.

In PANDA, we differentiate signed comparisons from unsigned comparisons. For example, `JG` is a condition jump instruction for signed comparisons, whereas `JA` is for unsigned comparisons; instructions `cmp eax, 1; JG label` are translated into `if(eax` $>_s$ `1) then label` in PANDA, whereas instructions `cmp eax, 1; JA label` are translated into `if(eax` $>_u$ `1) then` label, where $>_s$ and $>_u$ stand for signed greater and unsigned greater, respectively.

We model the semantics of the original x86 instructions, and translate x86 assembly into our PANDA representation. We build variable-like entities in binary code according to memory access expressions. There are two kinds of variables in PANDA: *mem* type and *val* type. A *mem* type variable, corresponding to a register or a memory location, could be assigned multiple times and only used as an l-value. A *val* type variable can only be used as an r-value and stands for the value in a memory location. A *reference* statement, a special statement in PANDA, will generate a *val* type variable when the program uses a *mem* type variable which has an unknown value or multiple possible values. Although a *mem* type variable could be assigned multiple times, each use of a *mem* type variable will be replaced by a specific value which is the latest assignment.

3.2.2 Component Extractor and Profile Constructor

To reduce path explosion, IntScope only analyzes certain parts of the program that may be relevant to integer overflow vulnerabilities. Based on the observation that most integer overflow vulnerabilities are caused by the misuse of tainted data in sinks without proper checks, IntScope only scans those program paths which pass through source functions (e.g., `read`, `recv`, `fread`) and reach a sink function (e.g., `malloc`, `alloca`, `LocalAlloc`).

The Component Extractor is responsible for selecting those candidate functions that may invoke both source functions and sink functions directly or indirectly. The call graph C is a directed graph $C = (N, E)$ that represents calling relationships between functions in a program. Each node $n_i \in N$ represents a function and each edge $(n_i, n_j) \in E$ indicates that function n_i calls function n_j. Since each function corresponds to a node in C, the Component Extractor selects the common ancestors of a taint source function node and a sink function node in C as candidates. For example, the Component Extractor would operate on the call graph in Figure 3 by selecting function `read_and_malloc` as a candidate since it may invoke both source function `fread` and sink function `malloc` indirectly.

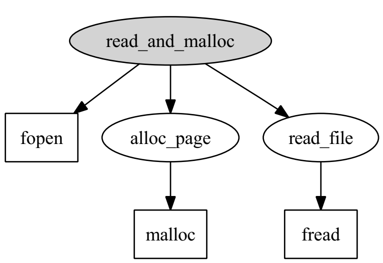


Figure 3. A simple call graph.

Given the candidate functions, the Profile Constructor is used to compute the "chop" graphs G'. A control flow graph is a directed graph $G = (N, E)$ where $n_i \in N$ represents a statement i and $(n_i, n_j) \in E$ represents a possible transfer of control flow from statement n_i to n_j. For the control flow graph of each candidate function, there is a unique entry point called n_{entry} and exit point called n_{exit}. Let Sr be the set of nodes that invoke a source function, and let Sk be the set of nodes that invoke a sink function. The Profile Constructor computes the nodes in G' by (a) computing the set Esr, which includes all nodes on the paths from n_{entry} to a node in Sr; (b) computing the set Esk, which includes all nodes on the paths from n_{entry} to a node in Sk; (c) computing the set Se, which includes all nodes on the paths from a node in Sr to n_{exit}; (d) if $(Se \cap Esk) \neq \phi$, the set of nodes in G' is: $Esr \cup (Se \cap Esk)$. Figure 4 shows the intuition graphically. There are three simple control flow graphs in Figure 4, and `read_and_malloc` is the candidate function. The Profile Constructor produces the chop graphs which only include the shaded nodes.

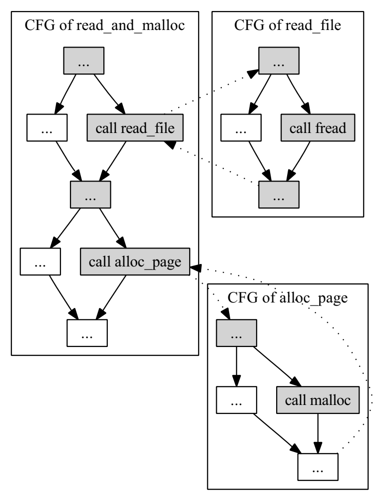


Figure 4. A graphical depiction of the chop graphs.

3.2.3 Modeling Memory

Because IntScope simulates program execution, it needs to build a symbolic memory space to maintain a large number of symbolic memory addresses and symbolic values. According to the design of PANDA, our virtual memory has three mapping relationships (shown in Figure 5):

M: a mapping from symbolic addresses to variable names.

ST: a mapping from variable names to symbolic values and other information, such as bit length (e.g., qword, dword, word, byte), and type (e.g., a *mem* type variable or a *val* type variable). For a *mem* type variable, its name also maps to its

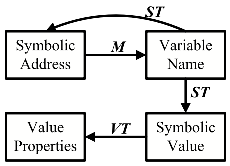


Figure 5. Memory mapping relationships.

symbolic address.

VT: a mapping from symbolic values to value properties (e.g., whether the value is tainted).

In our memory model, the taint property is bounded with a value, i.e., whether or not a variable is tainted depends on its current value. When multiple variables have the same tainted value, all the variables are trusted once a variable among them is completely checked. We take a source code example to illustrate it:

```
L1  x=y=read_from_net(); //x and y are tainted
L2  if(x==c)             //x is sanitized
L3    p = malloc(y);     //y is sanitized
```

Variable `x` and `y` share the same tainted value because of the assignment statement at `L1`. In our memory model, the use of `y` at `L3` is safe because `x` has been sanitized at `L2`.

3.2.4 Execution Strategies

Given the chop graphs *G'*, IntScope tries to explore each feasible path from the entry point. IntScope maintains the symbolic memory space and updates the symbolic memory according to the semantics of PANDA statements. In particular,

- At each branch point, IntScope will fork an additional execution. Along with the original "process", the two branch "processes" will check the feasibility of their current path. If the path is feasible, they will constrain themselves to their current path and continue to simulate the program execution.

- At each indirect jump statement such as `goto eax`, IntScope will evaluate the current value of `eax`. If the value is an address label, IntScope continues to simulate the code from that address; otherwise, if the value in `eax` is a symbolic value, IntScope cannot determine the target address and will terminate.

- At each call statement, if the target function is an internal function of the program, IntScope performs an interprocedural analysis. IntScope simulates a function call stack. Before entering a function, IntScope pushes the return address on the stack, and then simulates the target function; after exiting from the target function, IntScope reloads the return address from the stack and continues simulating.

With regards to the functions that are related to the propagation of taint data, IntScope will directly apply function summaries. For example, IntScope will mark the parameter `buffer` in functions `read`, `fread`, `recv` and `recvfrom` with a taint tag.

LOOPS. It is hard to model the number of loop executions in static analysis. In our approach, for a loop with a symbolic variable as a bound, IntScope traverses all branches in the loop only once; for a loop with a constant bound, IntScope tries to simulate the loop as accurately as possible, rather than simulate a fixed number of times. When a path re-enters the loop, IntScope checks whether the value of the loop variable is closer to the loop bound. If so, IntScope continues to simulate the path. For example, consider the instruction sequence shown below which is actually an idiomatic expression generated by Visual Studio C compiler.

```
lea  edi, [ebp+var_DC]
mov  ecx, 37h
mov  eax, 0h
rep  stosd
```

The prefix `rep` causes the instruction `stosd` to be repeated until `ecx` is decremented to 0. The idiomatic expression is used to initialize a large block of memory. In this case, the loop bound is a constant (37h), and IntScope will repeat the loop 37h times.

Block Memory Operation. Block memory operations (e.g., `strncpy`, `memcpy`, `memmove`) bring us a challenge if the parameter `size` in these functions is a symbolic value. For example, the function `memcpy(dst, src, n)` copies `n` bytes from memory area `src` to memory area `dst`. If `n` is a symbolic value, we are not sure how many bytes are copied. Subsequent memory accesses taking `dst` as a base address will be undecided. It is too expensive for static analysis to accurately simulate such functions. As such, IntScope only cares about the taint property propagation, but ignores the value propagation between `dst` and `src`. If `src` is a tainted buffer, IntScope will treat `dst` as a tained buffer as well, and all data from `dst` will be assigned a new tainted value.

3.2.5 Lazy Checking

The lack of type information and benign overflow operations in binary code significantly affect integer overflow detection. For example, a 32-bit memory location can be used as either an `int` type value or an `unsigned int` type value, and hence `0xffffffff+1` could be interpreted as `-1+1` or `4294967295+1`. Moreover, since benign overflow operations are prevalent in binaries, we cannot simply treat all overflow operations as vulnerabilities.

Whether or not an integer overflow is harmful depends on where and how the program uses the overflowed value. As a result, we do not check whether each arithmetic operation could overflow, but track how the program uses untrusted data; when a tainted value is used in sinks, we check whether it overflows or not. In addition, as described in Section 2, most sinks have already provided some hints on the type inference.

- For most memory allocation functions, since the type of parameter `size` is `unsigned int`, the value used as parameter `size` should not be greater than $2^{32} - 1$ (for 32-bit architectures).
- A symbolic value used as an array index should not be negative, i.e., if an expression of form `x-y` is used as an array index, IntScope will check whether the formula `x-y>=0` is valid under current path constraints; if not, IntScope generates an alarm.
- As an overflowed value used in a predicate may lead to a bypass of security checks or result in an unexpected execution, we should pay attention to the tainted value in predicates. In particular, at each branch point, if there is a tainted value used in the predicate, IntScope will query whether or not the tainted value could overflow under current path constraints.

We can get some type information from x86 conditional jump instructions, e.g., `JG`, `JNLE`, `JGE`, `JNL`, `JNGE`, `JLE`, `JNG`, `JE` and `JNE` are jump instructions for signed comparisons; `JA`, `JNBE`, `JAE`, `JNB`, `JB`, `JNAE`, `JBE`, `JNA`, `JE` and `JNE` are jump instructions for unsigned comparisons. We have preserved this information in PANDA. For example, $>_s$ and $>_u$ stand for signed greater and unsigned greater, respectively.

If tainted symbolic expressions of the form `x+y` or `x*y` appeared in unsigned comparisons, IntScope will check whether the expressions are greater than $2^{32} - 1$. Similarly, symbolic expressions of the form `x-y` that appear in an unsigned comparison should be greater than 0.

For tainted symbolic expressions of the form `x+y` that appear in signed comparisons, IntScope performs a check similar to `__addvsi3` (shown in Figure 6), a function in GCC run-time library to catch signed addition overflow errors. If a term (`x` or `y`) in `x+y` is positive/negative, we will check whether the sum is greater/less than the other term (`x` or `y`).

```
__addvsi3 (SItype a, SItype b){

const SItype w = a + b;
if (b >= 0 ? w < a : w > a)
    abort ();

return w;
}
```

Figure 6. Function `__addvsi3` in libgcc2.c of gcc-4.2.0

Note that IntScope does not immediately generate an alarm when it finds the tainted expression in a predicate could overflow, and instead it considers whether the predicate is designed to catch integer overflow errors. We have summarized several common patterns that make use of the erroneous result to catch integer overflow errors, such as `if((x+1)<x)`, `if((x !=(x*y)/y)`. In addition, we observe that if a program catches an integer overflow error, it is inclined to return soon or jump to a uniform error handling function.

Even if the tainted value used in an `if` statement could overflow, IntScope will not report it as a bug if the predicates of the `if` statement match the common patterns and/or one *successor* block of the `if` statement in the control flow graph includes a `return` statement. Because compound branch statements (i.e., multiple predicates in a statement such as `if(x>0&&(x+y)<y))` in source code will be compiled into multiple comparison instructions, we identify compound branch structures in executables using our previous work [51, 52].

4 Evaluation

We have implemented IntScope, which includes the following modules:

- **Decompiler** – We have implemented an in-house decompiler Bestar [51], which can be used for analyzing executables. Bestar makes use of IDA Pro[12] as a front-end, parses the disassemble result, identifies control flow structures and translates x86 assembly into PANDA. IDA Pro provides a lot of useful information: procedure boundaries and calls to library functions (hence we can identify many library functions by function name). Bestar can also build the call graph and control flow graphs of a program to present calling relationships between functions and abstracting the control flow behavior of each function.

- **Symbolic Execution Engine** – We implement our symbolic execution engine by leveraging GiNaC [8]. GiNaC is an open source framework for symbolic computation within the C++ programming language. It is very convenient for us to represent and manipulate arbitrary symbolic expressions with the help of GiNaC.

- **Path Validator and Integer Overflow Checker** – Both the Path Validator and the Integer Overflow Checker are built on top of STP [42], a decision procedure for bitvectors and arrays. STP implements all arithmetic operations (even non-linear operations such as multiplication, division and modulo), and bitwise boolean operations. It is very convenient for IntScope to translate path symbolic constraints into formulas accepted by STP. More importantly, STP supports sign-sensitive relational operations (signed/unsigned comparisons), which is very useful to model accurate symbolic variable bounds.

In this section, we present our evaluation results. We first evaluate the effectiveness in Section 4.1, and then measure the efficiency in Section 4.2.

4.1 Effectiveness

We have applied IntScope to two Microsoft programs and a number of widely used utility applications. IntScope successfully detected all the known integer overflow vulnerabilities, and also found more than 20 zero-day integer overflows in software such as QEMU [18], Xen [20], Xine [25], MPlayer [15] and VLC [23].

As IntScope is a static analysis tool, it may have false positives (the reasons are discussed in Section 5). To confirm the suspicious vulnerability is real, we rely on our previous dynamic vulnerability test case generation tool [46], which can mutate a benign input to drive program execution towards a particular point. If our tool happens to generate such a test case, we report that it is a real integer overflow vulnerability, otherwise we leave it as suspicious (it may be false positive, and we are just not sure).

4.1.1 Known-vulnerabilities

DSA_SetItem Integer Overflow Vulnerability [34]. Function `DSA_SetItem` in `comctl32.dll` is used to set the contents of an item in a dynamic structure array (DSA). To facilitate the bug explanation, we present a snippet of pseudo code from function `DSA_SetItem` in Figure 7. `DSA_SetItem` takes three parameters: `hdsa` is a pointer to a dynamic structure array, `index` is an index for the item in `hdsa` to be set, and `pItem` is a pointer to a new item data which will replace the item specified by `index`. If `index` is greater than the maximum item count (`hdsa->nMaxCount`) in `hdsa`, `DSA_SetItem` calls `ReAlloc` to allocate a new buffer. Although `index` is checked multiple times, a large `index` (such as `0x7ffffffe`) can trigger a multiplication overflow in `nNewItems*hdsa->nItemSize`, where `hdsa->nItemSize` is the size of an item, resulting in a smaller-than-expected returned pointer size.

To better understand how our system works, we present part of the disassembled code of `DSA_SetItem` in Figure 8(a), corresponding to the snippet of pseudo code in Figure 7. The assembly code in Figure 8(a) is converted into PANDA, shown in Figure 8(b).

For function `DSA_SetItem`, IntScope considers the second parameter (`arg2`) as tainted because it comes from user input, and all unknown variables will be assigned symbolic values.

Result. At each `if` statement, IntScope will fork an additional execution along the feasible branch. Let's focus on

```
int DSA_SetItem(HDSA hdsa, int index, void *pItem) {
    HLOCAL hMem;
    int nNewItems;
L1 if (index < 0)  return 0;
L2 if (index >=hdsa->nItemCount) {
L3   if(index + 1>hdsa->nMaxCount)
     {  nNewItems=((index + hdsa->cItemGrow)/
                   hdsa->cItemGrow)* hdsa->cItemGrow;
        hMem = ReAlloc(hdsa->hMemArrayData,
                 nNewItems * hdsa->nItemSize);
        //ignore some statements here
     }
L4   hdsa->nItemCount = index + 1;
   }
L5  memmove((void*)((index*hdsa->nItemSize)+
         (DWORD)hdsa->hMemArrayData),pItem, hdsa->nItemSize);
```

Figure 7. Pseudo code for DSA_SetItem.

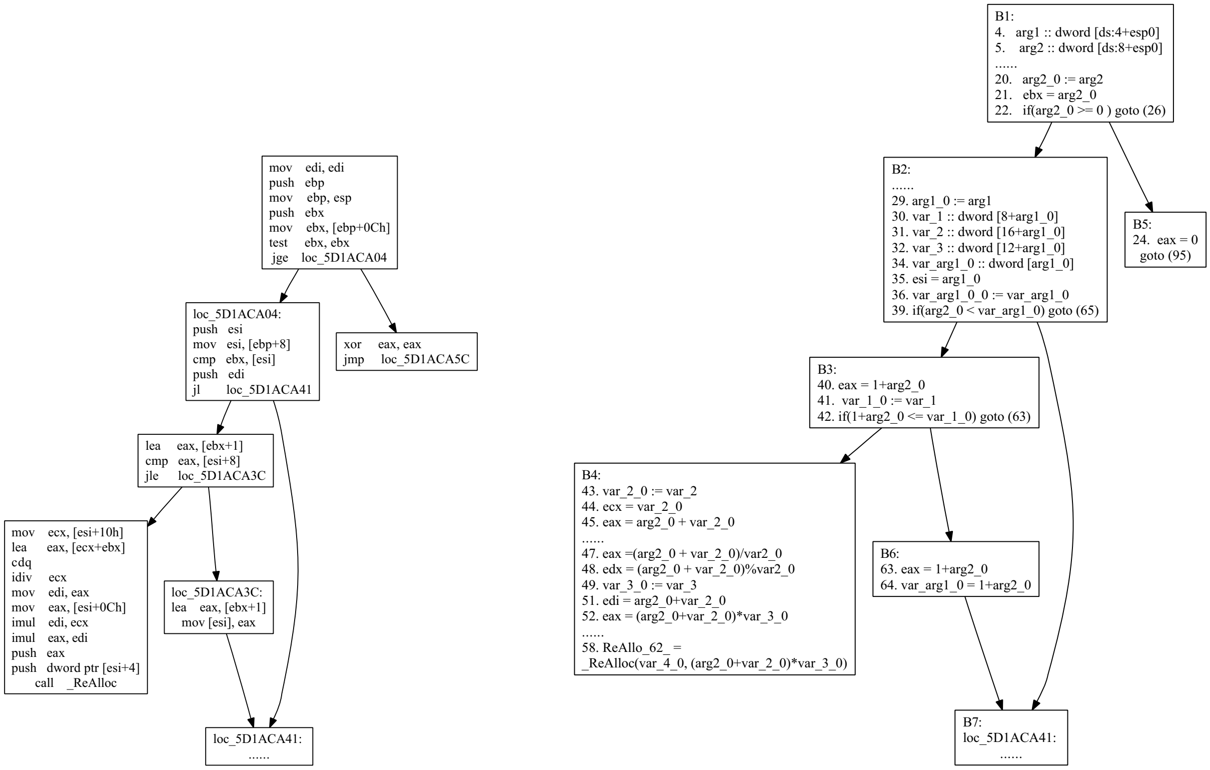
the "process" which reaches line 58 along the path ($B1$, $B2$, $B3$, $B4$) in Figure 8(b). Line 58 calls function `realloc` to allocate `(arg2_0+var_2_0)*var_3_0` bytes of memory. Because `arg2_0` is the tainted symbolic value of `arg2`, the whole expression is considered tainted. The Integer Overflow Checker will check whether the whole expression could overflow under current path constraints: `arg2_0>=0 && arg2_0>=var_arg1_0 && (1+arg2_0)>var_1_0`. In this case, a large `arg2_0` (e.g., `0x7ffffffe`) can pass program constraints and trigger an integer overflow. As a result, IntScope outputs this path as a detected one.

Besides the path we described above, IntScope reports the other two paths, corresponding to the paths (L1, L2, L5) and (L1, L2, L4, L5) in Figure 7, which terminate at function `memove` because of the potential multiplication overflow in `index * hdsa->nItemSize`. We do not have any prior knowledge about the DSA structure, and we assume all fields in `hdsa` have all possible values (e.g., `hdsa` can contain items of any size). If either `index` or `hdsa->nItemSize` is large enough, their product will overflow. However, we are missing some preconditions between the number of items (`nItemCount`) and the size of an item (`nItemSize`) in a normal DSA structure. Actually, neither `hdsa->nItemCount * hdsa->nItemSize` nor `hdsa->nMaxCount * hdsa->nItemSize` could overflow. Therefore, the product of `index` and `hdsa->nItemSize` cannot overflow under the constraint `index<hdsa->nItemCount`.

GDI AttemptWrite Integer Overflow Vulnerability (CVE-2007-3034). The integer overflow vulnerability in function `AttemptWrite` of `gdi32.dll` is a classic case. `AttemptWrite` tries to copy some data to a buffer (named `Buffer`), whose capacity is `Buffer_Capacity`. `AttemptWrite` performs memory management similar to the code below:

```
if(NumberOfBytesWritten + NumberOfBytesToWrite
        < Buffer_Capacity)
   memcpy(Buffer, file_data, NumberOfBytesToWrite);
```

where `NumberOfBytesWritten` stands for the number of bytes that has been written to `Buffer`, and



(a) Part of the disassembled result of DSA_SetItem.

(b) Part of the PANDA IR for DSA_SetItem.

Figure 8.

NumberOfBytesToWrite stands for the number of bytes still to be written to Buffer. To avoid copying too much data, AttemptWrite enforces a bounds check on NumberOfBytesToWrite. However, a large NumberOfBytesToWrite will cause an addition overflow and bypass the upper bounds check, resulting in a heap overflow in the subsequent call to memcpy.

AttemptWrite is invoked by many interface functions, such as CopyMetaFile. The function CopyMetaFile(hmfSrc, lpszFile) copies the content of a Windows-format metafile (hmfSrc) to the specified file (lpszFile). The integer overflow bug will be triggered if hmfSrc is a handle to a crafted metafile.

Result. Taking CopyMetaFile as the entry function and treating all data from hmfSrc as tainted data, IntScope identifies hundreds of suspicious paths and then analyzes each of them. These paths are on average more than $2,000$ statements (including more than 100 if statements) and approximately 20 function calls per path.

eEye Digital Security [24] has exposed the vulnerable code, shown in the Figure 9(a). Integer overflow occurs at the lea instruction. Figure 9(b) is the PANDA representation for the assembly instructions in Figure 9(a). When executing line 45 in Figure 9(b), IntScope detects that var_2_0+arg2_0 is a tainted value and invokes the Integer Overflow Checker to check whether var_2_0+arg2_0 could overflow. After further checking, IntScope determines that var_2_0+arg2_0 could overflow. Because using an overflowed value in a predicate is considered dangerous, IntScope outputs that path.

Besides successfully detecting the known vulnerability in function AttemptWrite, IntScope identified the other two suspicious vulnerable points. We found most paths terminate at a call statement LocalReAlloc (var_1_0,20+4*arg2_0,2) in a function named pmetalink16Resize, and 20+4*arg2_0 is reported as an integer overflow. We trace the origin of arg2_0 and find that arg2_0 comes from a global array in gdi32.dll. Missing global runtime information, IntScope assumes that global variables have tainted values, hence it deduces 20+4*arg2_0 could overflow and generates the alarms. The others paths terminate at a call statement LocalReAlloc(var_1,2*var_15_0, 2) in function CloseMetaFile because of a similar reason: var_15_0 also originates from the same global array, and IntScope considers 2*var_15_0 a suspicious overflow.

```
77F04271 mov   eax, [ebx+0Ch] // NumberOfBytesWritten
77F04274 mov   esi, [ebp+0Ch] // NumberOfBytesToWrite
77F0427D lea   ecx, [eax+ esi]
77F04280 cmp   ecx, [ebx+8]   // Buffer_Capacity
77F04283 ja    short loc_77F042B
```

```
eax =  var_2_0
esi =  arg2_0
ecx =  var_2_0+arg2_0
if(var_2_0+arg2_0 >u var_3_0)
```

Figure 9. (a) Vulnerable code in Function AttemptWrite. (b) PANDA IR for (a).

4.1.2 Zero-day Vulnerabilities

QEMU and Xen – We have detected 7 zero-day integer overflow vulnerabilities in QEMU. Six of them have been confirmed by our dynamic testing tool [46]; the remaining one is highly suspicious, but we cannot generate a test case to show its real existence yet. French Security Incident Response Team (FrSIRT) [19] has published a security advisory for these vulnerabilities (FrSIRT/ADV-2008-2919).

QEMU supports various disk image formats, such as `raw`, `qcow`, `qcow2`, `vmdk`, `vpc` and `cloop`. However, the block drivers for some image formats are vulnerable as detected by our system. Crafted disk images can trigger integer overflows and cause out-of-bound accesses when QEMU tries to open these images. We show the vulnerability in function `qcow_open` (in qemu-0.9.1/block-qcow2.c) to illustrate this.

```
194   if (bdrv_pread(s->hd, 0, &header, sizeof(header))
            != sizeof(header))
195       goto fail;
      ......
241    s->l1_size = header.l1_size;
246    if (s->l1_size < s->l1_vm_state_index)
247        goto fail;
249    s->l1_table=qemu_malloc(s->l1_size*sizeof(uint64_t));
250    if (!s->l1_table)
251        goto fail;
252    if (bdrv_pread(s->hd, s->l1_table_offset,
            s->l1_table,s->l1_size * sizeof(uint64_t)) !=
253         s->l1_size * sizeof(uint64_t))
254        goto fail;
255    for(i = 0;i < s->l1_size; i++) {
256        be64_to_cpus(&s->l1_table[i]);
257    }
```

Figure 10. Code snippet of function qcow_open (qemu-0.9.1/block-qcow2.c).

Function `qcow_open` in `block-qcow2.c` is used to open a `qcow2` [17] format disk image. Figure 10 shows the source code snippet of `qcow_open`. Function `bdrv_pread` in line 194 reads tainted data from a `qcow2` format image file to a structure `header`. All contents in `header` originate from the image file. After complex taint propagation, `s->l1_size` is tainted. Function `qemu_malloc` in line 134 directly calls function `malloc` to allocate `s->l1_size*sizeof(uint64_t)` bytes of memory. Although `s->l1_size` is checked in line 249, `s->l1_size * sizeof(uint64_t)` could still overflow, resulting in an array access out-of-bound in line 256. Block drivers for other disk image formats: `raw`, `vmdk`, `vpc`, `cloop` and the format used by Bochs have similar problems.

We detected the same integer overflow vulnerabilities in Xen. After a closer analysis, we found that Xen actually reuses QEMU's code, and hence all the integer overflow vulnerabilities in QEMU exist in Xen as well.

Media Players – We also applied IntScope to several multimedia players, including Xine [25], MPlayer [15], VLC [23] and Mpd [14].

Xine – Xine, a free multimedia player, has been downloaded nearly four million times from sourceforge. We detected three integer overflow vulnerabilities in Xine-lib 1.1.15 (the xine core engine), and an integer overflow vulnerability in Xine-ui 0.99.5 (an xlib-based GUI frontend). The two integer overflow vulnerabilities in function `process_commands` in Xine-lib allow remote attackers to compromise a server if the server uses the `cdda_server` routine (a component of xine) to play audio CDs over the network. Another integer overflow in function `ff_audio_decode_data` in Xine-lib could bypass a bounds check, resulting in a heap overflow. The three bugs were promptly confirmed by the developers.

A crafted PNG format picture could trigger an integer overflow vulnerability in function `_LoadPNG` in Xine-ui 0.99.5. `_LoadPNG` performs memory allocation similar to `malloc(width* height*3)`, where `width` and `height` are specified by the input PNG file. A crafted PNG file with very large `width` and `height` will cause integer overflow in `width*height*3`, resulting in a heap overflow when `_LoadPNG` tries to read data from the PNG file to the allocated memory.

Mplayer and VLC – Mplayer and VLC are two widely distributed media players. Both Mplayer and VLC can import Win32 codecs on ELF i386 platforms by building a DLL loader and emulating responses from necessary Win32 API calls, such as registry operations. However, we detected a malformed fake registry file could trigger an integer overflow in function `open_registry` in Mplayer and VLC, eventually causing a heap overflow.

We also detected another integer overflow vulnerability in `avisubdump`, an independent tool in the Mplayer package. `Avisubdump` dumps subtitle streams embedded in AVI files to `stdout`. The integer overflow in function `dumpsub_gab2` in `avisubdump` causes a bounds check to be bypassed, resulting in an unexpected execution.

MPD – Music Player Daemon (MPD) allows remote access for playing music and managing playlists. We detected that a crafted MPEG-4 format file can cause an integer overflow in function `mp4_decode` in MPD. Function `mp4_decode` does not correctly check data from the crafted input file, which causes a multiplication overflow, and the erroneous result is used in function `malloc`. The bug was fixed by the developers after we reported it.

Others – Besides examining those applications discussed above, we also checked Faad2 [7] (a portable MPEG-4 and MPEG-2 AAC decoder), hamsterdb [11] (a lightweight embedded database engine), Goom [10] (a visual effects generator for mp3 players), and Cximage [6] (an image processing and conversion library). IntScope detected two integer overflows in functions `mp4ff_read_stts` and `decodeMP4file` in Faad2, which could cause a heap overflow (CVE-2008-4201). The integer overflow in function `ConvertWmfFiletoEmf` in Cximage results in `malloc(0)`; a malformed local configure file could cause an integer overflow in function `gsl_read_file` in Goom, which could further lead to a heap overflow. An integer overflow vulnerability in function `btree_find_cursor` in hamsterdb, which could cause potential buffer overflows, has aleady been fixed by the developers in their latest version.

We summarize the experimental results in Table 2. The column "Paths" shows the number of suspicious paths generated by IntScope. Since IntScope performs a path-sensitive analysis, a vulnerable point in a program may cause many suspicious paths. The column "Total" indicates the number of vulnerable points. Note that Xen and QEMU have the same vulnerabilities. Totally, we detected 26 integer overflow vulnerabilities, and 20 of them have been confirmed by our dynamic testing tool [46] and/or by the developers. For the remaining 6 suspicious integer overflows, we cannot generate test cases to show their real existence yet.

4.2 Efficiency

In this section, we measure the performance and space overhead of our system. The evaluation is performed on an AMD Opteron Server (2.6 GHz) with 8GB memory running Linux Kernel 2.6.18. Table 3 shows the result of efficiency evaluation. We measured the time that IntScope spent translating x86 assembly into our PANDA representation (the column "Binary-to-IR time") and the time IntScope spent symbolically executing PANDA (the last column). We can see that translating the x86 binary into our PANDA IR is time-consuming part (varying from 1 seconds to nearly 1131 seconds). The "IR Size" shows the size of the target PANDA representation generated by our Decompiler, and it looks much bigger than the original binary mainly because for a single instruction, we may introduce many PANDA statements. For example, a simple `push ebp` will be translated into the statement sequence shown in Figure 11.

```
esp0 := esp
loc1 :: dword [-4+esp0]
esp  = -4+esp0
ebp0 := ebp
loc1 = ebp0
```

Figure 11. PANDA IR for "push ebp"

To summarize, the average size for these binaries is 320.3K bytes, and we find that IntScope takes 288.2s to translate them into the PANDA code, with a size of 5.46M bytes, and then takes 293.6 seconds to symbolically execute the interesting paths.

5 Discussion

Theoretically, IntScope may generate a test case for each identified vulnerable path by solving the path conditions with concrete values. In practice, however, the suspicious paths sometimes are not complete execution traces since IntScope only scans certain parts of a program, that is, the suspicious paths do not start from function `main`. This is why we use our dynamic vulnerability test case generation tool [46] to show the true existence for a particular vulnerability.

For those suspicious integer overflow vulnerabilities that we cannot generate test cases to confirm, we have to manually determine whether they are false positives or not. We examined the reasons and we sum up why IntScope may generate false positives as follows:

Missing of the constraints between inputs. IntScope assumes the input data could be "anything" and ignores the innate constraints between inputs. For example, when applying IntScope to function `DSA_SetItem` (see Section 4.1.1), IntScope reports the paths (L1, L2, L5) and (L1, L2, L4, L5) in Figure 7. The parameter `hdsa` in function `DSA_SetItem` is a dynamic structure array. The product of the number of items (`hdsa->nItemCount`) and the size of an item (`hdsa->nItemSize`) in `hdsa` should not overflow (i.e., `hdsa->nItemCount` $*$ `hdsa->nItemSize` $< 2^{32}$). Without this precondition, IntScope reports the two paths as suspicious paths.

Lack of global information. For example, besides the known integer overflow in `gdi32.dll`, IntScope reports the other two vulnerable points (see Section 4.1.1). We find the two vulnerable points are caused by the use of some global data. Actually, the use of this global data is safe. We need prior knowledge about the target program to identify such false positives.

Imprecise symbolic execution. Our symbolic execution is not perfect. IntScope does not accurately simulate block memory functions (`memmove`, `memcpy`, etc.) and some string functions (`strncmp`, `strchr`, etc.). However, IntScope is still able to find many integer overflow bugs in real applications, showing that our system is tolerant of such imprecise symbolic simulation.

Name	Version	Entry Function	Paths#	Total#	Confirmed #	Suspicious#
GDI32.dll	5.1.2600.2180	CopyMetaFile	452	3	1	2
comctl32.dll	5.82.2900.2180	DSA_SetItem	3	2	1	1
QEMU Xen	0.9.1 3.2.1	bochs_open	3	1	1	0
		cloop_open	1	1	1	0
		parallels_open	2	1	1	0
		qcow_open(for qcow2 format)	3	1	1	0
		vmdk_open	20	2	1	1
		vpc_open	1	1	1	0
Xine	1.1.15	ff_audio_decode_data	10	1	1	0
		process_commands	2	2	2	0
Xine-ui	0.99.5	_LoadPNG	4	1	1	0
MPlayer	1.0rc2	dumpsub_gab2	1	1	1	0
		init_registry	3	1	1	0
Mpd	0.13.2	mp4_decode	2	1	1	0
Goom	2k4	gsl_read_file	1	1	1	0
Cximage	600_full	ConvertWmfFiletoEmf	1	1	1	0
faad2	2.6.1	decodeMP4file	36	3	2	1
		mp4ff_read_stts	1	1	1	0
Hamstedb	1.0.4	btree_find_cursor	3	1	1	0

Table 2. Evaluation Result on Effectiveness

Name	Executable	File Size	Binary-to-IR time (seconds)	IR Size	Traversing Time (seconds)
GDI32.dll	GDI32.dll	271KB	614	7.61 MB	574
comctl32.dll	comctl32.dll	597 KB	1131	13.7 MB	0.1
QEMU	Qemu-img	341 KB	124	12.8 MB	358
Xine	cdda_server	14.5 KB	4	116 KB	26
	xine	966 KB	590	12.9 MB	327
Mplayer	avisubdump	14.2 KB	1	36.8 KB	0.3
MPD	mpd	243 KB	131	2.74 MB	667
GOOM	libgoom2.so	439KB	94	1.42 MB	445
faad2	faad	57.6 KB	29	693 KB	113
Hamstedb	libhamsterdb.so	260 KB	164	3.46 MB	426
Average		320.3KB	288.2	5.46MB	293.6

Table 3. Evaluation Result on Efficiency

6 Related Work

Integer Misuse Detection and Protection. To prevent integer based vulnerabilities, several techniques like compiler extensions and safe C++ integer classes have been proposed. With `-ftrapv` option, the GCC compiler will insert additional calls (e.g., `_addvsi3`) before signed addition operations to catch overflow errors. Similarly, RICH [32] inserts run-time checks with low overhead according to formal semantics for safe C integer operations. It is inevitable for such runtime check techniques to generate false positives because of benign integer overflows. Using other safe C++ class (such as SafeInt, IntSafe) or arbitrary precision arithmetic packages (such as GMP [9], CLN [3]) could relax integer security problems, however, the performance overhead is still non-trivial.

Using taint analysis to detect integer misuse vulnerabilities in C programs has proven to be highly effective [29, 37, 49]. For example, Ebrima N. Ceesay et al. [37] implemented a static analysis tool on top of Cqual [41] to track the untrusted data. The tool in [37] uses a dataflow-insensitive analysis and generates an alarm when an untrusted integer variable is used to access memory.

Ashcraft and Engler [29] presented a range checker to track whether an untrusted value is bounds checked before the value is used in trusting sinks. Because range checker only cares whether the untrusted value is bounds checked, untrusted data after bounds checking are considered safe data, which may miss subtle integer overflow bugs.

Sarkar et al. [49] proposed a constraint graph to describe sanitization checks. The nodes in the constraint graph represent the integer expressions, and the edges represent ordering relationships between the integer expressions. For an expression used in memory allocation, its sub-expressions are iteratively walked. If any sub-expression is unverified, it reports an alarm [49].

UQBTng [53] is a tool to automatically find integer overflows in Win32 binaries. UQBTng first makes use of UQBT [21] to translate binaries into C code; then UQBTng inserts `assert` statements before the calls to memory allocation functions; finally, UQBTng uses CBMC [39], a Bounded Model Checker, to verify the program property. UQBTng completely depends on the ability of the translator. However, the automatic decompilation of binary files to equivalent C code is still quite challenging. Unlike UQBTng, IntScope simulates program execution according to the semantics of low-level instructions.

Binary Analysis. Vine [22] is a static analysis component in the BitBlaze [2] project. Vine can accurately translate each x86 instruction into a RISC-like intermediate representation(IR) and it implements a dataflow and control flow analysis on that IR. We believe our techniques for detecting integer overflow can be implemented on top of Vine as well.

CodeSurfer/x86 [48, 47, 31, 30] is a binary analysis platform which makes use of both IDA Pro and the CodeSurfer system [4], a toolkit for building program analysis and inspection tools. CodeSurfer/x86 uses the Value-Set Analysis (VSA) algorithm [30] to recover variable-like entities in binaries, and translates x86 binary code into an IR which can be accepted by the CodeSurfer system.

Chevarista [1] is a project for automated vulnerability analysis on SPARC binary code. Chevarista demonstrates how to translate binary code into SSA form and model variable bounds by interval analysis to detect buffer overflows or integer overflows. Chevarista also mentions techniques to check race conditions based on model checking theory.

Symbolic Execution. Symbolic execution is widely used in many projects, such as EXE[36], CUTE[50], DART[43], SAGE [44], BitScope [33], and Archer [54]. Archer, EXE, CUTE and DART insert a symbolic execution engine into program source code and use a mixed execution to generate test inputs or find potential bugs on feasible paths. Unlike DART and CUTE, SAGE first runs the target program and collects a real trace. Then it virtually re-executes the recorded trace to collect input-related constraints and generates new inputs. BitScope implements a mixed execution to analyze malicious binaries.

7 Conclusion

In this paper, we have presented the use of symbolic execution to automatically detect integer overflow vulnerabilities in x86 binaries with the goal of identifying the vulnerabilities before an attacker does. Our approach simulates program execution, tracks the propagation of taint data, and checks whether a tainted symbolic expression used in a sink can overflow under path constraints. We have implemented our approach in a system called IntScope and applied it to analyze a number of real-world binaries. Experimental results show that our approach is highly effective and practical. We found more than 20 zero-day integer overflow vulnerabilities in several popular software packages, including QEMU, Xen and Xine.

8 Acknowledgements

We are grateful to our shepherd Michael Locasto, and the anonymous reviewers for their hard work, insightful comments and suggestions. We also thank Ryan Riley, and Jianwei Zhuge for their feedback in preparing this paper. This research was supported in part by the National High Technology Research and Development Program of China (No. 2006AA01Z402 and No. 2006AA01Z410) and a National Development and Reform Commission Project of China.

References

[1] Automated vulnerability auditing in machine code. *http://www.phrack.com/issues.html?issue=64&id=8*.

[2] BitBlaze: The BitBlaze Binary Analysis Platform Project. *http://bitblaze.cs.berkeley.edu/index.html*.

[3] CLN: Class Library for Numbers. *http://www.ginac.de/CLN/*.

[4] CodeSurfer: Automated Source-code Analysis Tool. *http://www.grammatech.com/products/codesurfer/*.

[5] Cve: Vulnerability type distributions. *http://cwe.mitre.org/documents/vuln-trends/index.html*.

[6] CXimage: Image Processing and Conversion Library. *http://www.xdp.it/cximage.htm*.

[7] FAAD2: A MPEG-4 and MPEG-2 AAC Decoder. *http://www.audiocoding.com/faad2.html*.

[8] GiNaC: A Free Computer Algebra Aystem. *http://www.ginac.de/*.

[9] GMP: GNU Multiple Precision Arithmetic Library. *http://gmplib.org/*.

[10] GOOM: Visual Effects Generator. *http://www.ios-software.com/*.

[11] Hamsterdb: A Lightweight Embedded Database. *http://hamsterdb.com/*.

[12] Ida pro. *http://www.hex-rays.com/idapro/*.

[13] Malware attack exploiting flash zero day vulnerability. *http://ddanchev.blogspot.com/2008/05/malware-attack-exploiting-flash-zero.html*.

[14] MPD:Music Player Daemon. *http://www.musicpd.org/*.

[15] MPlayer: The Movie Player. *http://www.mplayerhq.hu/*.

[16] National vulnerability database. *http://nvd.nist.gov/*.

[17] Qcow2: The QCOW2 Image Format. *www.gnome.org/ markmc/qcow-image-format.html*.

[18] QEMU: An Open Source Processor Emulator. *http://www.qemu.org/*.

[19] QEMU Buffer Overflow Vulnerability. *http://www.frsirt.com/english/advisories/2008/2919*.

[20] The Xen Hypervisor. *http://www.xen.org/*.

[21] UQBT: A Resourceable and Retargetable Binary Translator. *http://www.itee.uq.edu.au/ cristina/uqbt.html*.

[22] Vine: BitBlaze Static Analysis Component. *http://bitblaze.cs.berkeley.edu/vine.html*.

[23] VLC: Media Player and Streaming Server. *http://www.videolan.org/vlc/*.

[24] Windows Metafile AttemptWrite Heap Overflow. *http://research.eeye.com/html/advisories/published/AD20070814b.html*.

[25] Xine: A Free Video Player. *http://xinehq.de/*.

[26] Zero day exploit alert: Webviewfoldericon setslice vulnerability. *http://research.eeye.com/html/alerts/AL20061002.html*.

[27] Zero day exploit in pdf with adobe reader. *http://it.slashdot.org/article.pl?sid=07/09/22/1040225*.

[28] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools (Second Edition)*. Addison-Wesley, 2006.

[29] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May, 2002.

[30] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the 13th International Conference on Compiler Construction*, pages 5–23, 2004.

[31] G. Balakrishnan and T. Reps. Divine: Discovering variables in executables. In *Proceedings of Internation Conf. on Verification Model Checking and Abstract Interpretation (VMCAI)*, 2007.

[32] D. Brumley, T. cker 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)*, 2007.

[33] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Bitscope: Automatically dissecting malicious binaries, 2007. Technical Report CMU-CS-07-133, Carnegie Mellon University.

[34] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, May, 2008.

[35] 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, 2008.

[36] 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)*, pages 322–335, 2006.

[37] 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 & Vulnerability Assessment*, 2006.

[38] 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*, pages 12–12, 2005.

[39] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[40] M. V. Emmerik and T. Waddington. Using a decompiler for real-world source recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 27–36, 2004.

[41] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, pages 1–12, Berlin, Germany, 2002.

[42] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.

[43] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, pages 213–223, 2005.

[44] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA, February 2008.

[45] D. Gopan and T. Reps. Low-level library analysis and summarization. In *19th International Conference on Computer Aided Verification (CAV'07)*, Berlin, Germany, 2007.

[46] Z. Lin, X. Zhang, and D. Xu. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08)*, Anchorage, Alaska, USA, June 2008.

[47] T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, Charleston, South Carolina, USA, 2006.

[48] T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. A next-generation platform for analyzing executables. In *The Third Asian Symposium on Programming Languages and Systems*, Tsukuba, Japan, 2005.

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

[50] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, 2005.

[51] T. Wei, J. Mao, W. Zou, and Y. Chen. A new algorithm for identifying loops in decompilation. In *14th International Static Analysis Symposium (SAS'07)*, volume 4634 of *Lecture Notes in Computer Science*, pages 170–183. Springer, 2007.

[52] T. Wei, J. Mao, W. Zou, and Y. Chen. Structuring 2-way branches in binary executables. In *31st Annual International Computer Software and Applications Conference (COMPSAC'07)*, 2007.

[53] R. Wojtczuk. Uqbtng: a tool capable of automatically finding integer overflows in win32 binaries. In *22nd Chaos Communication Congress*, 2005.

[54] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. *SIGSOFT Softw. Eng. Notes*, 28(5):327–336, 2003.