Integer Overflow Detection with Delayed Runtime Test

Zhen Huang
zhen.huang@depaul.edu
DePaul University
Chicago, Illinois, USA

Xiaowei Yu
xyu43@depaul.edu
DePaul University
Chicago, Illinois, USA

ABSTRACT

Detecting integer overflow vulnerabilities is critical for software security. Many techniques have been proposed to dynamically detect integer overflow vulnerabilities by instrumenting integer overflow tests into target programs. Their major drawback is that they can produce many false positives. In this paper, we propose an approach to eliminate the false positives stemming from incorrectly or not considering the sanitization code in target programs that is designed by developers to catch integer overflows.

Unlike prior work that performs integer overflow test at arithmetic operations, our approach delays the test until the locations where the result of the arithmetic operation is about to be used by sensitive operations. This approach allows the sanitization code to filter out integer overflows before our integer overflow tests take place. As a result, it will not produce false positives for integer overflows that can be caught by the sanitization code.

We have implemented a prototype and our evaluation shows that it can effectively detect integer overflow vulnerabilities without producing false positives.

CCS CONCEPTS

• **Security and privacy → Software and application security**;

KEYWORDS

vulnerability detection; software vulnerability; integer overflow; static analysis

ACM Reference Format:
Zhen Huang and Xiaowei Yu. 2021. Integer Overflow Detection with Delayed Runtime Test. In *The 16th International Conference on Availability, Reliability and Security (ARES 2021), August 17–20, 2021, Vienna, Austria.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3465481.3465771

1 INTRODUCTION

Integer overflow is one of the most dangerous software weaknesses [1, 2]. Integer overflows often give rise to other dangerous software vulnerabilities such as buffer overflows and NULL-pointer dereference, which allows attackers to compromise computer systems or cause denial-of-service. Thus it is crucial to detect integer overflows in a timely manner.

ARES 2021, August 17–20, 2021, Vienna, Austria

ACM ISBN 978-1-4503-9051-4/21/08…$15.00
https://doi.org/10.1145/3465481.3465771

```
1 SIXELSTATUS sixel_encode_body(....) {
2 ....
3       size = y * width;
4+      if size > INT_MAX - x) {
5+          /* integer overflow */
6+          status = SIXEL_BAD_INTEGER_OVERFLOW;
7+          goto end;
8+      }
9       pix = pixels[size + x];
10 ....
11 }
```

Figure 1: The code extracted from the patched CVE-2014-9629 vulnerability in *libsixel*, an image library that processes sixiel images. An integer overflow can occur at line 3. The sanitization code at line 4–8 was added by developers to detect the integer overflow after the vulnerability was discovered.

For decades, many techniques have been proposed to detect integer overflows [3, 8, 12–15, 17–20]. These techniques can be roughly categorized as two types: static and dynamic. Static techniques rely on static analysis to detect integer overflows [14, 17, 18]. They usually cause many false positives due to the inherent approximations adopted by static analysis.

On the contrary, dynamic techniques check integer overflows at runtime [3, 8, 12, 13, 15, 19, 20]. They monitor the execution of integer operations that may cause integer overflows by instrumenting test to detect integer overflows. Typically they have fewer false positives than static techniques, at the cost of the runtime overhead introduced by the monitoring.

One main cause of false positives produced by dynamic techniques is that they fail to take into account existing sanitization code in target programs that are designed to detect integer overflows anticipated by developers. As illustrated in Figure 1, An integer overflow can occur during the calculation of `y * width` at line 3 but developers have added santization code at line 4—8 as a patch to detect such integer overflow.

Most dynamic techniques will instrument a test such as a check on whether `size/width` equals `y` immediately after line 3 to determine if `size` is an overflowed value as a result of the multiplication, because `size` will be used as the index to access an array at line 9. If the check fails, the test will determine that an overflow occurred and report it. However, this will cause a false positive because line 4–8 will catch the integer overflow and will prevent line 9 from being executed if the integer overflow happened.

Some prior work effectively reduces such false positives by changing the overflowed result of an arithmetic operation to a specifically-chosen dirty value and let the dirty value flow through the execution path until the result is about to be used in a dangerous

way, and reporting the integer overflow only if the dirty value is not detected by sanitization code [15]. Unfortunately the approach is not compatible with certain sanitization code and thus it still produces false positives.

To address this limitation, our work takes a different approach by instrumenting the integer overflow test code at the sites where an overflowed value is going to be used by a sensitive operation, e.g. as memory allocation size, rather than immediately after integer operations that can cause integer overflows, so that the test code will be executed after any sanitization code. Because the instrumented tests are executed after rather than before any sanitization code, the test code will report an integer overflow only when sensitive operations are about to use the overflowed value, in case sanitization code fails to catch the integer overflow or when there is no sanitization code. Because the integer overflow test instrumented immediately before the sensitive operations is the same as the test that could be instrumented before the arithmetic operation, it has the same detecting power without having the problem of using a dirty value to indicate integer overflows.

Using the code in Figure 1 as an example, our approach will instrument the test code to test the multiplication at line 3 immediately before line 9. When an integer overflow occurs at line 3, the instrumented test will only be executed if line 4–8 fails to catch the integer overflow.

There are two main challenge of our approach: 1) it must guarantee that the test code instrumented before a sensitive operation has the same effect as the test code instrumented immediately after the corresponding arithmetic operation; 2) it needs to cover all possible propagation flows from the arithmetic operation to the sensitive operation where the result of the arithmetic operation will be used.

For the first challenge, one straightforward solution is to use test code identical to the test code that can be instrumented right after an arithmetic operation. But this requires verifying whether all the variables used by the arithmetic operation stay the same on all the execution paths between the arithmetic operation and the sensitive operation. If that is not the case, such test code cannot be used at the sensitive operation.

On the contrary, our solution always provide such guarantee by saving a copy of the variables used by the arithmetic operation immediately before the arithmetic operation and constructs test code to test on the copy of the variables instead of on the original variables. This solution guarantees that the test code has identical effect regardless whether it is executed at the arithmetic operation or delayed until the sensitive operation.

For the second challenge, our solution uses inter-procedural static taint analysis to track all the data flows that propagate from arithmetic operations to sensitive operations.

We have implemented a prototype based on the approach, and evaluated the prototype on nine vulnerabilities in seven applications. Our evaluation shows that the approach can successfully detect real-world integer overflows without producing any false positives in the presence of sanitization code. The instrumented test code impose a runtime overhead of 1.75x on the SPEC CPU 2000 benchmarks.

In summary, our main contributions are as follows:

- We present our analysis on two types of integer overflow propagation: intra-procedural propagation and inter-procedural propagation.
- We propose an approach to accurately detect integer overflow vulnerabilities using delayed test code. This approach eliminates the false positives caused by the presence of sanitization code in target programs.
- We present a prototype implementation of the approach. The prototype uses static taint analysis to identify arithmetic operations whose result can be used by sensitive operations, and instruments test code at sensitive operations to check whether the operations will use overflowed values.
- Our evaluation on the prototype illustrates the effectiveness of our approach in detecting real-world integer overflow vulnerabilities. The test code instrumented into target programs does not produce any false positives.

2 PROBLEM DEFINITION

As discussed in prior work [8], the majority of integer overflows is benign. Many times developers intentionally use integer overflows for a variety of purposes, such as cryptography, hashing, and low-overhead floating point emulation. An integer overflow occurs in an integer operation becomes a vulnerability only when 1) one or more operands used in the integer operation are derived from user input, and 2) the result of the integer operation is used in a sensitive operation such as the size for memory allocation or the length for memory copy, which will lead to buffer overflows, and the condition that will trigger assertions, which will lead to denial-of-service.

Our work focuses on integer overflows vulnerabilities. In such an integer overflow, an arithmetic operation is executed first and causes an integer overflow. Then the result of the arithmetic operation, i.e. the overflowed value, is propagated to a sensitive operation. We make the following definitions in the paper.

- **arithmetic operation.** an integer addition, subtraction, multiplication, bitwise left shift, or integer type cast.
- **sensitive operation.** a memory allocation that has one integer parameter as the allocation size, a memory or string manipulation that has one integer parameter as the memory size or string length, or a memory dereference using a memory address that is a result of a pointer arithmetic operation.
- **def-site of an integer overflow.** an *arithmetic operation* that causes an integer overflow.
- **use-site of an integer overflow.** a *sensitive operation* that uses the result of an overflowed arithmetic operation.
- **sanitization code.** the code in a target program that is designed to detect an integer overflow.
- **integer overflow test.** a test that checks an overflow condition to determine whether an arithmetic operation causes an integer overflow.

We use the overflow condition listed in Table 1 for our integer overflow test. For an arithmetic operation, we deem an integer overflow occurred if its corresponding overflow condition is evaluated to be true.

Table 1: Overflow condition for integer operations.

Type	Arithmetic Operation	Overflow Condition
Signed	$c = a + b$	$(a > 0 \wedge b > 0 \wedge c < a) \vee$ $(a < 0 \wedge b < 0 \wedge c > a)$
Unsigned	$c = a + b$	$c < a$
Signed	$c = a - b$	$(a < 0 \wedge b > 0 \wedge c > 0) \vee$ $(a > 0 \wedge b < 0 \wedge c < 0)$
Unsigned	$c = a - b$	$a < b$
Signed	$c = a * b$	$a \neq 0 \wedge c/a \neq b$
Unsigned	$c = a * b$	$a \neq 0 \wedge c/a \neq b$
	$c = a \ll b$	$c \gg b \neq a$
	$c = type_c(a)$	$a \notin [min..max]$ of $type_c$

```
1 int ImagingLibTiffDecode(....) {
2 ....
3+    /* overflow check for realloc */
4+    if (INT_MAX / row_byte_size < tile_length) {
5+        state->errcode = IMAGING_CODEC_MEMORY;
6+        return -1;
7+    }
8     state->bytes = row_byte_size * tile_length;
9     new_data = realloc (state->buffer, state->bytes);
10 ....
11 }
```

Figure 2: The code extracted from the patched CVE-2020-5310 vulnerability in an image processing library called *libImaging*. An integer overflow can occur at line 8.

Our goal is to detect integer overflows that have a *use-site*, i.e. *integer overflow vulnerabilities*, and ignore 1) integer overflows that will be caught by sanitization code, which we refer to as *anticipated integer overflows* [15], and 2) integer overflows that do not have a use-site, i.e. *benign integer overflows*.

2.1 Sanitization Code

Sanization code can be classified as *precondition test* and *postcondition test* [8]. A precondition test checks whether an integer overflow can occur in an arithmetic operation without actually performing the arithmetic operation. It is often added before the arithmetic operation. As illustrated in Figure 2, the multiplication at line 8 can cause an integer overflow so developers added the sanitization code at line 3–7 to detect the integer overflow.

A postcondition test checks whether the result of an arithmetic operation is deemed to be overflowed. It is always added after the arithmetic operation. The code in Figure 1 is an example of postcondition test.

As we will discuss in Section 3, our approach instruments integer overflow test at the use-site of an integer overflow instead of the def-site of an integer overflow. Consequently it supports both forms of sanitization code.

2.2 Propagation of Integer Overflow

The propagation of an integer overflow refers to the data flow of the result of the integer overflow from the def-site of the integer overflow to the use-site of the integer overflow. We can categorize

```
1 char* at_bitmap_init(short width, short height) {
2     short size = width * height * sizeof(char);
3     return malloc(size);
4 }
5
6 char* input_pnm_reader (....) {
7     char* bitmap;
8     int xres, yres;
9 ....
10    bitmap = at_bitmap_init((short)xres, (short)yres);
11 ....
12 }
```

Figure 3: An inter-procedural propagation of integer overflow in the code adopted from an integer overflow vulnerability, CVE-2017-9156 in `autotrace`

the propagation into two types: *intra-procedural propagation* and *inter-procedural propagation*.

Intra-procedural propagation refers to the propagation in which both the def-site and the use-site are within the same function. The code shown in Figure 1 is a case of intra-procedural propagation in which the def-site is at line 3 and the use-site is at line 9.

Inter-procedural propagation refers to the propagation in which the def-site and the use-site are in different functions. An inter-procedural propagation can involve global variables, pointers, and function calls.

An example inter-procedural propagation across functions is presented in Figure 3. The integer overflow can happen at the type casting from `int` to `short` on variable `xres` and `yres` at line 10 in function `input_pnm_reader`, i.e. the def-site. The overflowed values in `xres` and `yres` are then passed to function `at_bitmap_init` as arguments at line 10. At last, the overflowed values are used to calculate the memory allocation size at line 2, which will be used by `malloc` to allocate memory at line 3, i.e. the use-site.

3 DESIGN

We describe our design in this section. It analyzes the source code of a target program and instruments the program with code to detect integer overflow vulnerabilities. To be able to bypass anticipated integer overflows, it delays the integer overflow test code till the use-site of integer overflows. There are two challenges that it needs to address:

- Because an integer overflow test is not to be performed right at the def-site of the integer overflow, it needs to guarantee that the delayed test has the same effect as if it were performed at the def-site.
- An integer overflow can be propagated from the def-site to the use-site via inter-procedural propagation. It must be able to identify inter-procedural propagation.

To address the first challenge, we design our integer overflow detection mechanism as two components: a *test-preparation code* that makes a copy of the value of the variables involved in an arithmetic operation at the def-site, and a *test code* that tests the copied values at the use-site. The test-preparation code is in charge of allocating memory space to save the values of involved variables, and copying the values of involved variables into the memory space. The test code is responsible of performing a check on the

overflow condition for the arithmetic operation. As a result, the test-preparation code and the test code work together to test whether an integer overflow occurs.

We note that our design requires that the test-preparation code dominates the test code, so that the test-preparation code always executes before the test code. Because we instrument the test-preparation code at the def-site and the test code at the use-site, this implies that the def-site needs to dominates the use-site.

Our design addresses the second challenge by using interprocedural static taint analysis to identify the pairs of def-site and use-site that exist in different functions.

The workflow of our design is illustrated in Figure 4. Our approach first performs static taint analysis on the target program to gather information on the use-site and def-site of arithmetic operations that may cause integer overflows. The information on the use-sites and the def-sites are then used to instrument test preparation code at the def-site and test code at the use-site.

3.1 Static Taint Analysis

Our static taint analysis is based on program dependency graph, or PDG, that embodies data dependency and control dependency information of a program [9]. It consists of two phases: 1) an intra-procedural analysis that identifies def-site and use-site within a same function and 2) an inter-procedural analysis that tracks the propagation across functions.

The intra-procedural analysis takes a PDG of a function as input and outputs a list of pairs of use-site and corresponding def-site for the function. For each function in a target program, the intra-procedural analysis first identifies all the arithmetic operations in it. Then it checks whether any sensitive operations invoked in the function is dependent on each arithmetic operation. If so, it identifies the arithmetic operation as a def-site and the sensitive operation dependent on it as a use-site corresponding to the def-site. Lastly it verifies that the def-site dominates the use-site. If that is not the case, it does not consider this pair of def-site and use-site for test instrumentation.

The inter-procedural analysis takes the list of pairs of use-site and def-site produced by the intra-procedural analysis, and appends pairs of def-site and use-site that can be reached across functions into the list.

For each use-site for a function, it finds the list of function calls that are dependent on the use-site. And it iterates the callees in the list of function calls and checks whether any sensitive operations in the callee is dependent on the parameters of the callee. If so, it considers the sensitive operation in the callee function as a def-site for the use-site in the caller function.

The result of our static taint analysis is a list of pairs of use-site and def-site for a target program.

3.2 Instrumenting Test-Preparation Code

For each integer overflow test, our approach needs to generate and instruments its corresponding test-preparation code. The instrumenting of test-preparation code iterates each pair of use-site and def-site in the list of pairs of use-site and def-site produced by the static taint analysis. It produces the code that allocates memory for saving a copy of the value for each variable on which the arithmetic operation corresponding to a def-site depends, as well as a copy of the result of the arithmetic operation, and the code to make a copy of the value for these variables and the value of the result. If the use-site is propagated from the def-site via intra-procedural propagation, it produces code to allocate local variables to save the copy of values. If the use-site is propagated from the def-site via inter-procedural propagation, it produces code to allocate global variables to save the copy of values.

To illustrate the test-preparation code, Figure 5 shows the code of our running example instrumented with test-preparation code, which consists of line 2, 4, 5, and 7. Line 2 allocates the memory for test-preparation. Line 4-5 saves a copy of the value of variable `y` and `width` on which the def-site at line 6 depends. Line 7 saves a copy of the result of the def-site.

The instrumenting of test-preparation code not only instruments test-preparation code into the target program, but also produces the information needed to instrument the test code, including the memory location allocated to save variable copies for each def-site.

3.3 Instrumenting Test Code

Our approach instruments test code at each use-site of the list of pairs of use-site and corresponding def-site generated from the static taint analysis.

For each use-site, it produces the code to check the overflow condition for the type of arithmetic operation of the corresponding def-site based on Table 1. It uses the mapping between the variables on which a def-site depends and the variables saving the copy of their values to generate code for the test. As illustrated in Figure 5, line 13 and 14 are the instrumented test code.

4 EVALUATION

We implement our prototype as an analysis pass of LLVM [16]. It works with C/C++ programs that can be compiled into LLVM bitcode. The prototype instruments integer overflow tests into target programs in the form of LLVM IR code.

We evaluate the accuracy of our prototype on real-world integer overflow vulnerabilities and the performance of our prototype on SPECINT 2000 benchmarks. All our experiments are conducted on a Intel i7-7700 3.60GHZ CPU workstation with 16GB memory. The workstation runs 64bit Ubuntu 16.04

We run the prototype on real world integer vulnerabilities and verify whether our prototype can detect the vulnerabilities without producing false positives.

4.1 Detecting Integer Overflow Vulnerability

We use real world integer overflow vulnerabilities to evaluate our prototype's capability in vulnerability detection. We randomly choose nine integer overflow vulnerabilities from five different types of programs, including IRC server, OCR tool, programming language interpreter, and image processing library, as listed in Table 2.

For each program, we run our prototype to instrument integer overflow tests. The prototype successfully instruments integer overflow tests for all but one vulnerabilities. It is unable to track the data flow from the def-site to the use-site for CVE-2016-5094 because the value produced at the def-site is passed to the use-site via a function

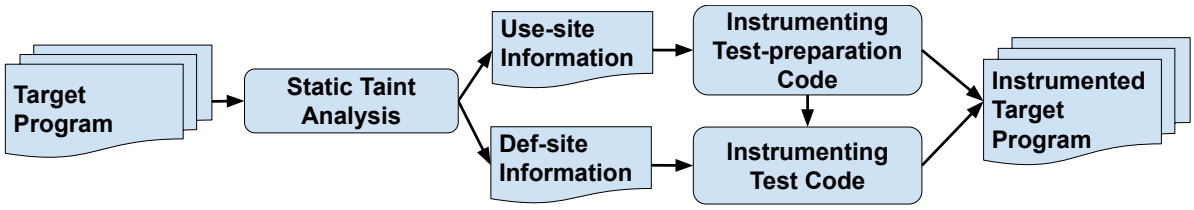

Figure 4: Workflow of Our Approach

```
1 SIXELSTATUS sixel_encode_body(....) {
2+     int y_copy, width_copy, size_copy;
3 ....
4+     y_copy = y;
5+     width_copy = width;
6      size = y * width;
7+     size_copy = size;
8      if size > INT_MAX - x) {
9          /* integer overflow */
10         status = SIXEL_BAD_INTEGER_OVERFLOW;
11         goto end;
12     }
13+    if (y_copy != 0 && size_copy/y_copy != width_copy)
14+       report_integer_overflow();
15     pix = pixels[size + x];
16 ....
17 }
```

Figure 5: The code instrumented with integer overflow test for the patched CVE-2014-9629 vulnerability in *libsixel*. The lines prefixed with '+' is instrumented code.

argument that is a pointer to a struct field, which is not identified by the static taint analysis. For the instrumented programs, we run the exploits for the vulnerabilities and check whether the instrumented tests can detect the exploits. Our results show that the vulnerabilities are effectively detected by the instrumented tests.

4.2 Bypassing Anticipated Integer Overflow

To evaluate our prototype's capability in bypassing anticipated integer overflows, we conduct vulnerability detection on the programs that are patched for the vulnerabilities in Table 2.

First, we run the exploits to verify that the vulnerabilities can no longer be triggered in the patched programs. Second, we instrument integer overflow tests for the vulnerabilities and re-run the exploits. This time we check whether our tests still report integer overflows.

We find that the tests instrumented by our prototype successfully bypass all the anticipated integer overflows. As a result, our approach produces no false positives for these vulnerabilities.

4.3 Test Density

We evaluate the density of instrumented integer overflow tests on SPECINT 2000 benchmarks. As shown in Table 3, all the instructions refer to LLVM instructions. On average, 4.73% of instructions are integer arithmetic operations and 1.25% of them are chosen by our prototype to instrument integer overflow tests.

Table 2: Detecting Integer Overflow Vulnerabilities.

CVE#	App.	Def-Site	Use-Site	Detected?
2005-0199	ngircd	$a - b$	string copy	Y
2005-1141	gocr	$a * b$	memory allocation	Y
2006-4812	php	$type_c(a)$	memory allocation	Y
2016-5094	php	$type_c(a)$	memory dereference	N/A
2019-11072	lighttpd	$a - b$	string copy	Y
2019-13109	exiv2	$a - b$	data copy	Y
2019-13110	exiv2	$a + b$	memory dereference	Y
2019-19746	fig2dev	$a * b$	memory dereference	Y
2019-20205	libsixel	$a * b$	memory allocation	Y

Table 3: Test Density: #I: number of instructions; #A: number of arithmetic operations; #S: number of sensitive operations; #T: number of instrumented integer overflow tests; Density is defined as #T / #A.

Benchmark	#I	#A	#A/#I	#S	#T	Density
164.gzip	11,604	954	8.22%	6	17	1.78%
175.vpr	38,647	1,249	3.23%	17	7	0.56%
181.mcf	5,979	151	2.53%	5	2	1.32%
186.crafty	45,428	2,748	6.05%	96	16	0.58%
197.parser	31,210	849	2.72%	17	18	2.12%
254.gap	197,099	9,405	4.77%	2	0	0
256.bzip2	9,100	676	7.43%	10	16	2.37%
Average	48,438	2,291	4.73%	22	11	1.25%

4.4 Runtime Overhead

To measure the runtime overhead, we ran SPECINT 2000 benchmarks with the *ref* dataset. Figure 6 presents the runtime overhead. The instrumented benchmarks run 1.75x slower than the original benchmarks on average. We believe the high overhead is mainly caused by the use of function calls in instrumented test code.

5 LIMITATIONS

The static taint analysis used by our prototype does not track the taint propagation via struct fields accessed using function parameters that are struct pointers. As a result, the prototype is unable to identify this type of data propagation from def-sites to use-sites.

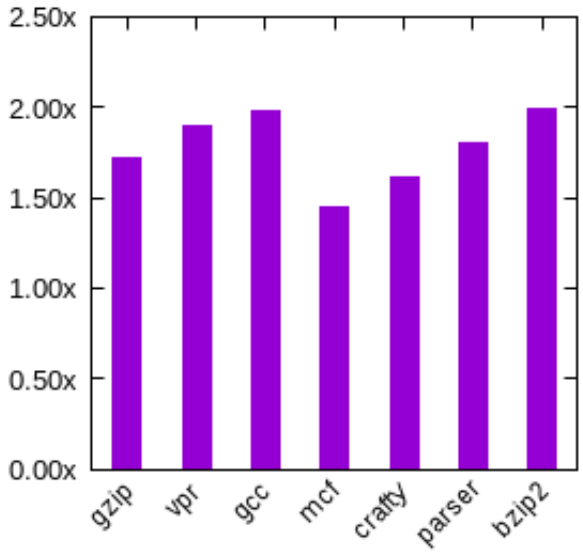

Figure 6: Runtime Overhead

Our approach requires that a def-site dominates its corresponding use-site. One way to loosen the requirement is to create pseudo test preparation code at other program paths leading to the use-site.

Because our prototype instruments test code that will make function calls to perform integer overflow test, the runtime overhead is high. We plan to inline test code to reduce the runtime overhead.

6 RELATED WORK

A large body of work has been proposed to detect or fix vulnerabilities [4–7, 10–12, 15, 17–20]. We focus on those detecting integer overflow vulnerabilities. IntScope performs symbolic execution and taint analysis on binaries to detect the program paths that can propagate integer overflows into exploitable vulnerabilities [17].

IntPatch automatically fixes Integer Overflow to Buffer Overflow vulnerabilities in C/C++ programs at compile time [19]. It utilizes type theory and static dataflow analysis to identify potential vulnerabilities and then instruments runtime checks on integer overflows. Instead of using static analysis, IntTracker adopts dynamic tracking technique to reduce false positive in detecting IO2BO vulnerabilities in C/C++ programs.

7 CONCLUSION

We present the design and implementation of our approach to detecting integer overflow vulnerabilities without producing common false positives caused by the presence of sanitization code in target programs. The approach eliminates such false positives by delaying integer overflow tests at the location where the result of integer overflows is to be used in sensitive operations. We test our prototype on real-world integer overflow vulnerabilities and find that it successfully detects the vulnerabilities without false positives.

REFERENCES

[1] 2019. 2019 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html. (2019).

[2] 2020. 2020 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html. (2020).

[3] David Brumley, T Chiueh, R Johnson, and H Lin. 2007. RICH: Automatically protecting against integer-based vulnerabilities. In *Ndss '07*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.73.7344

[4] David Brumley, Tzi cker Chiueh, and Robert Johnson. 2007. RICH: Automatically Protecting Against Integer-Based Vulnerabilities. In *Proceedings of NDSS Symposium 2007*. NDSS, New York, NY, USA.

[5] Xi Cheng, Min Zhou, Xiaoyu Song, Ming Gu, and Jiaguang Sun. 2017. IntPTI: Automatic integer error repair with proper-type inference. In *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 996–1001. https://doi.org/10.1109/ASE.2017.8115718

[6] Zack Coker and Munawar Hafiz. 2013. Program Transformations to Fix C Integers. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, 792âĂŞ801.

[7] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2012. Understanding Integer Overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, 760âĂŞ770.

[8] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2015. Understanding integer overflow in C/C++. In *the 34th International Conference on Software Engineering*, Vol. 25. https://doi.org/10.1145/2743019

[9] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319âĂŞ349. https://doi.org/10.1145/24039.24041

[10] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. 2014. Sound Input Filter Generation for Integer Overflow Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 439–452. https://doi.org/10.1145/2535838.2535888

[11] David Molnar, Xue Cong Li, and David A. Wagner. 2009. Dynamic Test Generation to Find Integer Bugs in X86 Binary Linux Programs. In *Proceedings of the 18th Conference on USENIX Security Symposium (SSYM'09)*. USENIX Association, USA, 67âĂŞ82.

[12] Raphael Ernani Rodrigues, Victor Hugo Sperle Campos, and Fernando Magno Quintao Pereira. 2013. A fast and low-overhead technique to secure programs against integer overflows. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013*. https://doi.org/10.1109/CGO.2013.6494996

[13] Julian Schütte. 2016. Osiris : Hunting for Integer Bugs in Ethereum Smart Contracts, Vol. D. 664–676.

[14] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and C. Martin Rinard. 2015. Targeted Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement. *ASPLOS* (2015), 473–486.

[15] Hao Sun, Xiangyu Zhang, Chao Su, and Qingkai Zeng. 2015. Efficient Dynamic Tracking Technique for Detecting Integer-Overflow-to-Buffer-Overflow Vulnerability. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIA CCS âĂŹ15)*. Association for Computing Machinery, New York, NY, USA, 483âĂŞ494. https://doi.org/10.1145/2714576.2714605

[16] The LLVM Compiler Infrastructure 2018. http://llvm.org/. (2018).

[17] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. 2009. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution.. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009*. The Internet Society. http://dblp.uni-trier.de/db/conf/ndss/ndss2009.html#WangWLZ09

[18] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Improving Integer Security for Systems with KINT. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 163–177. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/wang

[19] Chao Zhang, Tielei Wang, Tao Wei, Yu Chen, and Wei Zou. 2010. IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time. In *Computer Security – ESORICS 2010*, Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 71–86.

[20] Yang Zhang, Xiaoshan Sun, Yi Deng, Liang Cheng, Shuke Zeng, Yu Fu, and Dengguo Feng. 2015. Improving Accuracy of Static Integer Overflow Detection in Binary. In *Research in Attacks, Intrusions, and Defenses*, Herbert Bos, Fabian Monrose, and Gregory Blanc (Eds.). Springer International Publishing, Cham, 247–269.