*Int*egrity: finding integer errors by targeted fuzzing

Yuyang Rong[1], Peng Chen[2], and Hao Chen[1]

[1] University of California, Davis
{ptrrong, chen}@ucdavis.edu
[2] ByteDance Inc.
spinpx@gmail.com

Abstract. Integer arithmetic errors are a major source of software vulnerabilities. Since they rarely cause crashes, they are unlikely found by fuzzers without special techniques to trigger them. We design and implement *Int*egrity, which finds integer errors using fuzzing. Our key contribution is that, by targeted instrumentation, we empower fuzzers with the ability to trigger integer errors. In our evaluation, *Int*egrity found all the integer errors in the Juliet test suite with no false positive. On 9 popular open source programs, *Int*egrity found a total of 174 true errors, including 8 crashes and 166 non-crashing errors. A major challenge during error review was how to determine if a non-crashing error was harmful. While solving this problem precisely is challenging because it depends on the semantics of the program, we propose two methods to find potentially harmful errors, based on the statistics of traces produced by the fuzzer and on comparing the output of independent implementations of the same algorithm. Our evaluation demonstrated that *Int*egrity is effective in finding integer errors.

Keywords: Fuzzing · Integer Errors · Software Security

1 Introduction

Integer arithmetic errors are a significant source of security vulnerabilities [21]. Integer overflow and underflow[3] are undefined behavior in many languages, such as C/C++, and may cause security check bypass or malicious code execution. For example, on April 22, 2018, attackers created a massive number of Beauty Coins (BEC) in two transactions by exploiting an integer overflow in ERC20 [2], which forced the exchange platform OKEx to roll back all the transactions two days later [3]. Divide-by-zero causes the program to crash and so may be used to launch denial of service attacks. The number of reported integer arithmetic bugs has been increasing rapidly in recent years, which account for 104, 232,

[3] The term *underflow* sometimes refers to float point underflow. However, in accordance with Common Weakness Enumeration (CWE) [4], in this paper underflow means that the result of an integer arithmetic operation is smaller than the smallest value that the type can represent.

Table 1: Verified, unique arithmetic errors that *Int*egrity found in real world applications, compared with Angora + UBSan. Note that the total numbers of unique errors at the bottom are fewer than the sums of the rows above because some programs share the same library and therefore we removed these duplicate errors when calculating the totals.

Program	Errors found by *Int*egrity			Errors found by	Improvement
	Crashing	Non-crashing	Total(I)	Angora + UBSan(A)	(I - A)
cjpeg	1	12	13	0	+13
djpeg		17	17	14	+3
file		17	17	0	+17
img2txt	3	21	24	2	+22
jhead	2	4	6	4	+2
objdump		5	5	0	+5
readelf		38	38	0	+38
tiff2ps		27	27	1	+26
tiffcp	2	31	33	2	+31
Total	8	166	174	23	+151

and 635 Common Vulnerabilities and Exposures (CVE) in 2016, 2017, and 2018, respectively.

Prior work showed how to detect integer overflows *when* they happen. For example, Integer Overflow Checker (IOC)[16,15], which has been included in Undefined Behavior Sanitizer (UBSan) [8] since LLVM 3.5. However, they relied on the programmer to manually create test cases to trigger those bugs, which is laborious and unreliable. We face the challenge of how to generate these test cases automatically and efficiently.

Fuzzing is an automated approach for finding software bugs. Starting with AFL, graybox fuzzers have made great strides in finding bugs fast. They instrument programs with the code for recording program state during execution and use that information to guide input mutation. Fuzzers differ in their strategies for *exploration*, which aims at expanding branch coverage. Previous exploration strategies include matching magic bytes [28], finding sanity checks and checksums [24,35], measuring the distance between the input and target location [11,12], and solving constraints like Angora [13]. Besides exploration, another goal of fuzzing is *exploitation*. In the context of fuzzing, exploitation refers to triggering bugs, regardless if the bug may be used to launch attacks. It is difficult to find good exploitation strategies. As a result, most fuzzers randomly mutate the input to hope that some mutated input might trigger bugs. Given the huge space of input, the probability that a randomly mutated input will trigger a bug is low. Moreover, fuzzers have difficulty in detecting bugs that do not crash the program because they lack reliable signals that indicate bugs in those cases. For example, arithmetic errors cause a program to misbehave (e.g., to produce wrong results), but they rarely cause the program to crash.

Our goal is to allow fuzzers to exploit integer arithmetic errors efficiently. Our key technique is to provide fuzzers with critical information by targeted instrumentation such that the information can later be used to guide fuzzers to exploit potential bugs. For example, to detect overflow when adding two 32-bit signed integers, we extend both the operands to 64 bits, compute their sum (which cannot overflow), and, if the sum is out of the range of 32-bit signed integers, execute a special *guard branch* to send a signal to the fuzzer to indicate the error. This way, if the fuzzer can reach the guard branch, then an integer overflow occurs. The same idea can be used to check for other bugs, such as index out of range, null pointer dereference, etc.

In principle, the above idea works with any fuzzer. However, to find bugs efficiently, we need to overcome three challenges. First, we need to select a fuzzer that efficiently solves the constraints indicating arithmetic errors (Section 3.2). Second, the guard branches inserted by the fuzzer have much lower expected reachability than the original branches, because the guard branches indicate arithmetic errors but most arithmetic operations should not have such errors. Therefore, we need to redesign the fuzzer's scheduling algorithm to assign different priorities to the original and guard branches, respectively (Section 3.2). Finally, we need to send a unique signal to the fuzzer to indicate arithmetic errors if the guard branches are explored. The fuzzer should let the program continue exploring branches after receiving the signal, in contrast to when the signal indicates a memory violation (Section 3.2).

It might be tempting to implement the above idea by simply combining a sanitizer (e.g., UBSan [8]) with a fuzzer. However, because of the challenges described above, such a naive combination would result in poor performance, as we will show in Section 5.4. Instead, we implemented our approach in a tool called *Int*egrity. As we will show in Section 5, *Int*egrity is effective in finding integer arithmetic errors in both standard test suites and popular open source programs. On the Juliet Test Suite [9], *Int*egrity found all the bugs with no false positive (Table 2). Table 1 shows the bugs that *Int*egrity found on 9 popular open source programs from 6 packages. In total, *Int*egrity found 174 unique arithmetic errors, where 8 caused crash but 166 did not. We define a unique error by a unique (file name, line number, column number) tuple.

Fuzzing is attractive because it provides inputs that witness errors. When an error caused a crash, there is no doubt that the program misbehaved. However, when the error did not cause a crash, verifying whether the error caused the program to misbehave becomes difficult as the decision must take domain knowledge into consideration. We made progress on this problem by proposing two methods. The first method is based on the statistics of the traces generated by the fuzzer. If an integer arithmetic error occurred on most traces generated by the fuzzer where the arithmetic operation executed, then the error was likely benign, as long as the fuzzer had adequate path coverage. The other method is based on comparing the output of independent implementations of the same algorithm on the same input. If an integer error caused one implementation to misbehave, then the other independent implementation of the same algorithm will unlikely generate a similar output, as long as the output is a deterministic function of the input. These two approaches, when applicable, call attention to integer errors that are potentially harmful.

2 Background

2.1 Integer arithmetic errors

In statically typed languages such as C, the type of a variable is determined at compile time. An integer type has a fixed width and so can represent only a range of integers. For example, an unsigned 32-bit integer variable can represent only integers in $[0, 2^{32} - 1]$. When the result of an arithmetic operation exceeds the upper/lower bound of its type, overflow/underflow occurs. Another common arithmetic error is divide by zero.

Some compilers have the option to insert code that checks for integer arithmetic error at runtime. However, the checks cause runtime overhead. Moreover, some arithmetic errors are benign because they are intended by the programmer. For example,

```
v << (32 - b) >> (32 - b)
```

is a common idiom to extract the lower b bits from the unsigned 32-bit integer v. As long as b is in (0, 32][4], the implementation correctly achieved the programmer's goal, even though overflow might happen during the left shift. It would be undesirable to terminate the program upon detecting such benign overflows.

2.2 Fuzzing

To avoid runtime overhead or terminating programs upon benign arithmetic errors, we would like to find those errors during testing. Fuzzing is a popular technique for finding bugs automatically with Graybox fuzzing being particularly popular. It instruments programs with the code for recording program state during execution and uses that information to guide input mutation. However, integer overflow/underflow bugs rarely cause crashes, and most fuzzers cannot detect bugs that do not crash the program. In this paper, we propose an approach to instrument arithmetic operations to give the fuzzer critical information to help it find potential errors in arithmetic operations.

3 Design

Fuzzers mutate the input to find bugs in the program. They have two goals: (1) exploration: explore different paths; and (2) exploitation: trigger bugs (regardless whether they can be used to launch attacks). Previously, fuzzers were used

[4] It is undefined behavior when b is a constant 0. Some architectures only allow 5 bits for the second operand, making shift by 32 bits equivalent to shift by 0 bits, producing `v` as the result; yet compilers, when -O2 optimization is turned on, will optimize this line to `0` if b is compile-time known to be 0.

predominantly to find memory errors. To use fuzzers to find integer arithmetic errors effectively, we need to modify both their exploration and exploitation strategies.

3.1 Exploitation

Arithmetic operations We detect integer overflow and underflow during addition (`+`), subtraction (`-`), multiplication (`*`), shift left (`<<`), and divide by zero during division (`/`) and remainder (`%`). We instrument LLVM IR code to detect those errors as follows.

- `+`, `-`, `*`: We promote both the operands to the next longer type (e.g., from `int32_t` to `int64_t`, and from `uint32_t` to `uint64_t`), evaluate the expression in the longer type, and check if the result is out of the range of the original type. As long as the width of the next longer type is as least doubled (e.g., `int8_t`, `int16_t`, `int32_t`, `int64_t`), which is the case in C and most C-like languages, the operation in the longer type never overflows. For example, to check if `(int8_t)x + (int8_t)y` overflows, we compute `(int16_t)x + (int16_t)y` and check if the sum is out of the range of `int8_t`.
- `<<`: A left shift operation `x << n` overflows if and only if $\mathrm{hp}(x) + n$ is greater than or equal to the width of (number of bits in) the result type, where the function $\mathrm{hp}(x)$ is the position of the highest non-zero bit of x. For example, $\mathrm{hp}(0b00000001) = 0$, $\mathrm{hp}(0b10000000) = 7$.
- `/` and `%`: We check if the second operand is 0. For `/`, we also check if the operands are MININT and -1 because MININT / -1 = MAXINT + 1 overflows.

Range inference Integer types have different ranges. To infer the correct integer type, we must determine both the bit width and sign.

Bit width inference For each operation, LLVM promotes every operand shorter than 32 bits to 32 bits, executes the operation, and then truncates the result back to the destination type when necessary. Therefore, if a truncation follows the operation, then we use the destination type of the truncation to infer the bit width; otherwise, we use the left-hand side of the operation.

Sign inference LLVM IR does not distinguish between signed and unsigned variables. LLVM determines if an operation on 32 or more bits may have signed overflow or unsigned overflow using the sign information from abstract syntax tree (AST), and encodes that information as a tag in the arithmetic instructions. For example, `add nsw` (no signed wrap) and `add nuw` (no unsigned wrap). We use these tags to infer the sign. However, operations on integers shorter than 32 bits carry no such tag because they never overflow in the range of 32-bit integers. In those cases, we infer the sign of each operand using the cast operation before the arithmetic operation. When LLVM casts the shorter type to 32 bits, we examine if the cast is signed or unsigned. If both operands are cast, we take the sign of the operand of the longer type if the operands have different bit widths. If they have the same bit width, and if either operand undergoes an unsigned cast, we infer the sign of the destination type as unsigned; otherwise, we infer the sign as signed.

Instrumentation reduction When we instrument an integer arithmetic operation to check for arithmetic errors, we create new branches. When a program has many integer arithmetic operations, the instrumentation would create many new branches for the fuzzer to explore. However, these branches differ from the original branches in the program in a very important way for the fuzzer: we expect most original branches to be reachable but few instrumented branches to be reachable (because the latter represent arithmetic errors). Since unreachable branches waste the fuzzer's computing budget, during instrumentation we eliminate branches that are guaranteed unreachable as follows:

- While we need to check both overflow and underflow of signed operations, we need not check underflow of unsigned operations, because once promoted to a wider type, underflow becomes overflow. For example, when the original type is 8-bit unsigned int, `(uint8_t)0 - 1 = 0xff` causes underflow. However, when promoted to 16-bit unsigned int, `(uint16_t)0 - 1 = 0xffff` causes an overflow on the original type because the result 0xffff is larger than the upper limit of the original type, 0xff.
- We do not check shift operation on negative integers for the same reason as above.
- When an operation is square, we do not check for underflow because it cannot.
- When a value is added to a negative constant or is subtracted by a positive constant, we do not check for overflow; similarly, when a value is added to a positive constant or is subtracted by a negative constant, we do not check for underflow.

Section 5.5 will show that the above optimization significantly reduced the number of branches that the instrumentation added to the program, and hence the number of constraints that the fuzzer tries to solve.

3.2 Exploration

The instrumentation described in Section 3.1 reduces the problem of exploitation to the problem of exploration. At each operation with potential integer arithmetic errors, *Int*egrity inserts a conditional statement to check for integer arithmetic errors. When an error happens, the conditional statement executes a branch, called the *guard branch*. In principle, we can use any fuzzer to do the exploration. However, we desire to select a fuzzer that can explore arithmetic errors efficiently. Moreover, since the guard branches are inherently different from the branches in the original program (original branches), the fuzzer must

Algorithm 1 *Int*egrity's scheduling algorithm.

```
function POP                                    ▷ Returns the next branch to fuzz
    return priorityQueue.pop()
end function
function PUSH($b$)                ▷ Pushes a new or existing branch onto the queue
    if $b$ is a newly found branch then
        if $b.tag = Tag.Original$ then
            $b.priority \leftarrow MAX\_PRIORITY$
        else
            $b.priority \leftarrow GUARD\_INIT\_PRIORITY$
        end if
    else
        $b.priority \leftarrow b.priority - 1$
    end if
    priorityQueue.push(b)
end function
```

treat them differently: the fuzzer should triage between the original and guard branches when scheduling branches (Section 3.2), and should behave differently between when arithmetic errors occur and when other errors occur (Section 3.2).

Fuzzer choice Section 3.1 provides critical information to the fuzzer by instrumenting the guard branches that represent those errors. While we may use any fuzzer to take advantage of that information, we selected Angora [13] for its two beneficial properties.

First, Angora fuzzes individual branches and can prioritize different branches. With enough computing budget, Angora fuzzes every branch on a path at least once. Since we associate every potential arithmetic error with a guard branch, Angora exploits (tries to trigger) every arithmetic error on the path. Angora also allows us to triage different branches, which is handy because the original branches and guard branches have different expected reachability (Section 3.2).

Second, Angora's input mutation strategy fits our goal well. When fuzzing a branch, Angora uses byte-level taint tracking to identify the input byte offsets that flow into the predicate that guards the branch. Then, Angora considers the predicate as a blackbox function on those byte offsets and uses gradient descent to find an input that satisfies the predicate. When the blackbox function is linear or monotonic, this mutation strategy guarantees to find a solution quickly. `+` and `-` are linear functions, and `*` is a monotonic function. When their operands take their values directly from in the input, Angora can solve the predicates of those operations efficiently.

Branch triage As discussed in Section 3.1, original branches and guard branches have different expected reachability: we expect most original branches to be reachable but few guard branches to be reachable because few arithmetic operations have errors. Moreover, before the fuzzer can reach an original branch b, it cannot explore any guard branch that b dominates.[5] Therefore, we must let the fuzzer assign higher priority to the original branches than to the guard branches.

We replaced Angora's scheduling with the following algorithm:

– At compile time, instrument each branch with a tag to indicate whether it is an original branch or a guard branch.
– At run time, store all the branches to be fuzzed in a priority queue.
– When finding a new branch, assign the branch a priority according to the branch tag (original or guard branch), and then push the branch onto the priority queue (`PUSH` in Algorithm 1).
– When failing to solve a branch, lower the priority of the branch and push it onto the priority queue (`PUSH` in Algorithm 1).
– When ready to explore a new branch, call `POP` in Algorithm 1 to get the branch with the highest priority.

Signal of errors When the fuzzer receives a signal indicating an error in the program, it stops the program execution and records the input, and the error and its location. Memory access violation, such as segmentation fault, is the most common signal. To reuse this framework, *Int*egrity lets the instrumented branches send a pre-determined signal to the fuzzer to indicate arithmetic errors.

However, merely sending a signal would be inadequate. Fuzzers stop the program when receiving signals. It makes sense when the signal is triggered by a memory error because the program cannot continue anyway. However, when the signal is triggered by an arithmetic error, the fuzzer should let the program continue to explore more paths, particularly when the error is false positive (see Section 5.2 for examples). Without this ability, a false positive arithmetic error early in the program would prevent the fuzzer from exploring most paths because most paths descend from the location of that error. We implemented this desirable function in Angora.

4 Implementation

We implemented *Int*egrity as an LLVM pass in 924 lines of C++. We also modified Angora to do branch triage (Section 3.2) and to deal with the new signal of arithmetic errors (Section 3.2) in 3419 lines of Rust.

We found that some programs may use 64-bit types (`uint64_t`, for example). However, Angora supported only 64-bit constraints, which was inadequate to check the overflow of the arithmetic operation on two 64-bit integers. To tackle this problem, we extended Angora to support 128-bit constraints. We did so by using `u128` and `__uint128_t` in Rust and C, respectively. In the case of a 128-bit or higher precision integer operation, we created a new struct that has two (or more) 128-bit unsigned integers inside and implemented all the arithmetic traits (`Add`, `Sub`, `Mul`, etc.) for it.

[5] A node d dominates a node n if every path from the entry node to n must go through d.

5 Evaluation

We evaluated the performance of *Int*egrity on both the Juliet test suite [9] and popular open source programs. We also evaluated the impact of instrumentation reduction described in Section 3.1.

All our experiments ran on a Linux server with two Intel Xeon Gold 5118 CPUs and 256 GB RAM.

We set *MAX_PRIORITY* and *GUARD_INIT_PRIORITY* in Algorithm 1 to 65 535 and 65 534, respectively, to guarantee that the fuzzer will try to solve all the original branches at least once before solving the guard branches.

5.1 Juliet test suite

The Juliet test suite, developed by the National Security Agency (NSA), contains tests for errors listed in Common Weakness Enumeration (CWE) [4]. It organizes the tests in a hierarchy: at the top level, the suite contains one test set for each CWE. Then, each test set contains many subsets, and each subset contains many tests. Each test is a C or C++ program containing a carefully designed and inserted error. This test suite provides ground truth for evaluating the false positive and false negative of *Int*egrity.

We used Juliet Test Suite v1.3 and selected the following test sets relevant to integer arithmetic errors:

– `CWE190_Integer_Overflow`
– `CWE191_Integer_Underflow`
– `CWE194_Unexpected_Sign_Extension`
– `CWE197_Numeric_Truncation_Error`
– `CWE369_Divide_by_Zero`

We excluded the following tests in the above test sets:

– Deterministic errors: These errors always happen regardless of the input, e.g., overflow caused by constant integers.
– Floating point errors, since we focus on integer arithmetic errors only.
– C++ programs. As discussed in Section 3.2, we used Angora as the fuzzer, and currently it supports only C programs. This is not an inherent limitation of *Int*egrity.

Two CWEs related to integer arithmetic errors are worth mentioning. One of them is `CWE197_Numeric_Truncation_Error`. Integer truncation causes an error when the result is out of the range of the destination type. Therefore, to detect this error accurately, we must detect the destination type (both sign and width) accurately. For example, consider `x & 0x0000ffff`. If the destination type has more than 16 bits or if it is unsigned 16-bit integer, then no overflow can happen. In all the tests of `CWE197`, it is easy to infer the destination types accurately because of the way how those errors were injected. However, in real world programs, we found that accurately inferring the destination type in the

Table 2: Errors that *Int*egrity found on the Juliet test suite. A"-" cell means that the corresponding test set on the top contains no corresponding subset on the left. *Int*egrity found all the errors with no false positive. Every test contains one inserted arithmetic error except subset s02 of CWE197, where half of its inserted bugs contain two truncation errors each.

Set / Subset	CWE190		CWE191		CWE194		CWE197		CWE369	
	bugs added	bugs found	bugs added	bugs found	bugs added	bugs found	bugs added	bugs found	bugs added	bugs found
s01	114	114	76	76	304	304	152	152	112	112
s02	38	38	38	38	0	0	76	**114**	38	38
s03	190	190	114	114	-	-	-	-	-	-
s04	114	114	190	190	-	-	-	-	-	-
s05	114	114	190	190	-	-	-	-	-	-
s06	190	190	-	-	-	-	-	-	-	-
s07	190	190	-	-	-	-	-	-	-	-

```
1 short CWE_197_s02_trunc_twice(char* inputBuffer){
2   short data = 0;
3   if (fgets(inputBuffer, 14, stdin) != NULL) {
4     data = (short)atoi(inputBuffer);
5   }
6   return (char) data;
7 }
```

Fig. 1: A test in CWE197 s02, which contains two truncation errors on Line 4 and 6.

context of integer truncation was difficult. Therefore, we disabled this rule when checking real world programs in Section 5.2.

The other one is `CWE680_Integer_Overflow_to_Buffer_Overflow`. This error happens when calling the function `malloc(site_t)` and when `size_t` is defined by `uint32_t`, which occurs on only 32-bit platforms. Since the fuzzer that we used(Angora) ran only on 64-bit platforms, we did not test this error.

Table 2 shows that *Int*egrity found all the bugs in the test sets of the above five CWEs with no false positives. Every test case has one inserted arithmetic error except subset s02 of CWE197. This subset contains 76 tests, where half of the tests contains two truncation errors each as shown in Figure 1: first truncating the result of `atoi` into `short`, and then further into `char`, both of which cause truncation errors. Therefore, *Int*egrity found a total of $38 + 38 \times 2 = 114$ unique errors in this subset of 76 tests.

We tried Angora and Angora + UBSan on this test set, respectively. Neither of them found any bugs.

Table 3: Unique errors that *Int*egrity found in common open source programs. Note that the total numbers of unique errors at the bottom are fewer than the sums of the rows above because when calculating the totals we removed the duplicate errors in the libraries shared by different programs.

Package	Version	Program	Unique errors			
			Divide by zero	Overflow to crashing	Non-crashing	Benign
libjpeg-ijg	v9a	cjpeg	1		12	63
		djpeg			17	101
file	5.32	file			17	7
libcaca	0.99beta99	img2txt	1	2	21	36
jhead	3.00	jhead		2	4	4
binutils	2.29	objdump -x			5	11
		readelf -a			38	27
libtiff	4.0.7	tiff2ps			27	36
		tiffcp -i	2		31	49
Total			4	4	166	315

5.2 Real world applications

We evaluated *Int*egrity on popular real world applications. We selected 9 applications from 6 packages that have many integer operations, such as image processing and executable file parsing. Detailed version and command line arguments are shown in Table 3. On each program, we ran *Int*egrity on 12 cores for 72 hours.

Table 3 shows all the unique errors that *Int*egrity found. We identified a unique error by the (file name, line number, column number) tuple where the error occurs. We divide those errors into three categories. The first category contains all errors that caused crashes (Section 5.2). Then, we manually reviewed the remaining errors to identify benign ones. We determined an error to be benign when we found that the error did not cause the program to misbehave (Section 5.2). After excluding those benign errors, the remaining errors belong to the non-crashing error category (Section 5.3).

It is also worth mentioning that *tiff2ps* and *tiffcp* share the same underlying library(*libtiff*). As a result, *Int*egrity found 6 duplicate non-crashing errors and 19 benign errors in both program. We removed those duplicate errors from total error count in Table 1 and Table 3.

Benign errors An error is benign when we found strong evidence that the error had been expected by the programmer and therefore did not cause the program to misbehave. We classify all the benign errors found into two classes:

Intentional overflows The programmer intended to use the result of an overflown value. One example is `v << (32 - b) >> (32 - b)`, where the programmer intended to exact the lower b bits from the unsigned 32-bit integer v, and implemented it by shifting v by $32-b$ bits to the left and then shifting by $32-b$ bits to the right. As long as b is in $(0, 32]$, the implementation correctly achieved the programmer's goal, even though overflow might happen during the left shift.

Unused overflown values This class of benign errors is commonly introduced by compiler optimization.

```
while (i--) { /* loop body */ }
```

is an example, Figure 2 shows the compiled LLVM IR. The loop subtracts 1 from the loop variable (an unsigned integer) and saves the result in another variable just before checking the predicate that if the loop variable is not 0. When the loop variable is 0, the subtraction underflows, but its result will never be used because the loop finishes.

```
1 ; <label>:loop_head:
2 %loop_var = load i32, i32*%loop_ptr, align 4
3 %next_loop_var = add nsw i32 %loop_var, -1
4 store i32 %next_loop_var,i32*%loop_ptr,align 4
5 %cond = icmp ne i32 %loop_var, 0
6 br i1 %cond, label %loop_body, label %loop_end
7 ; <label>:loop_body: /* body */
8 br label %loop_head
9 ; <label>:loop_end:
```

Fig. 2: An example of benign integer overflow. After LLVM optimization passes, the C program was translated into the IR shown in the figure, the syntax slighted modified for readability. On Line 3, the `add` instruction overflows when the loop variable `%iter_var` is 0, but the overflown result will never be used.

Crashes Arithmetic errors may cause crashes in two different ways. Divide by zero causes a crash immediately, while overflown or underflown values may cause a crash when used as indices to arrays. *Int*egrity discovered eight crashes, among which four are divide by zero, and four are overflow.

Figure 3 shows a divide by zero error on Line 4 in the program *libjpeg-ijp*. *Int*egrity found an input that caused the parameter `samplesperrow` to become 0, which then caused divide by zero on Line 4.

5.3 Which non-crashing error is harmful?

An error is said to be *harmful* when it triggers unexpected behavior, e.g. to produce a wrong result. Harmful errors may or may not be exploitable in the

```
// jmemmgr.c:395~435
... alloc_sarray(..., unsigned samplesperrow, ... ) {
  ...
  ltemp = ... / ((long) samplesperrow * SIZEOF(JSAMPLE));
  ...
}
```

Fig. 3: Divide by zero error in *jmemmgr.c* of *libjpeg-ijg* happens when the parameter `samplesperrow` is zero.

context of software security, yet they still cause problems in software correctness and reliability. If an arithmetic error causes a crash, it is definitely a harmful error. However, when it does not cause a crash, it is non-trivial to validate whether it is harmful.

We manually inspected all the 481 non-crashing errors reported by *Int*egrity and determined that 315 (or 65 %) were benign. However, manual inspection is tedious and unscalable.

Automatically determining if an arithmetic error is harmful is challenging because it depends on the semantics of the application. Nevertheless, we made progress on this problem by proposing two methods, one based on statistics of the traces generated by the fuzzer, and the other based on comparing the output of independent implementations of the same algorithm on the same input. These two approaches, when applicable, call attention to integer errors that are potentially harmful.

By statistics of traces This method is based on the conjecture that a harmful bug in a popular open source program unlikely occurs during most executions, because otherwise it would have been noticed, reported, and fixed with high probability. By this conjecture, if an integer arithmetic error occurred on most traces generated by the fuzzer where the arithmetic operation executed, then the error was likely benign, as long as the fuzzer had adequate path coverage.

To implement the above idea, for each non-crashing arithmetic error, we measured its rate of occurrence on all the traces where the arithmetic operation occurred. When this rate is above a threshold, we consider this error to be benign. We used the benign errors that we manually determined in Table 3 as the ground truth. Then, at each threshold, we counted the number of benign errors using the rule above, and calculated precision and recall based on the ground truth. That is, let G be the set of benign errors that we manually determined, and S be the set of benign errors that we identified by the statistics of traces. Then precision is $\frac{|S \cap G|}{|S|}$ and recall is $\frac{|S \cap G|}{|G|}$.

Table 4 shows the number of benign arithmetic errors and their precision and recall with regard to the ground truth. The overall precision is 79.2% at the threshold of 0.95, and is 75.7% at the threshold of 0.70. The overall recall is 37.5% at the threshold of 0.95, and is 67.3% at the threshold of 0.70. On several

Table 4: Benign arithmetic errors determined by statistics of traces. We use the benign errors found by manual inspection as the ground truth when calculating the precision and recall of the benign errors determined by statistics of traces.

Program	Benign errors found by manual inspection	Benign errors determined by statistics of traces					
		Threshold=0.95			Threshold=0.70		
		Count	Precision	Recall	Count	Precision	Recall
cjpeg	63	8	100.0 %	12.7 %	48	87.5 %	66.7 %
djpeg	101	19	100.0 %	18.8 %	42	97.6 %	40.6 %
file	7	6	83.3 %	71.4 %	8	87.5 %	100.0 %
img2txt	36	18	88.9 %	44.4 %	39	59.0 %	69.9 %
jhead	4	4	100.0 %	100.0 %	5	80.0 %	100.0 %
objdump	11	12	83.3 %	90.9 %	12	83.3 %	90.9 %
readelf	27	28	71.4 %	74.1 %	36	72.2 %	96.3 %
tiff2ps	36	25	88.0 %	61.1 %	37	62.2 %	63.9 %
tiffcp	49	46	67.4 %	63.3 %	53	67.9 %	73.5 %
Total	315	149	79.2 %	37.5 %	280	75.7 %	67.3 %

programs, this method was quite accurate. For example, at the threshold of 0.95, this method achieved both 100% precision and 100% recall on *jhead*, and 100% precision on *cjpeg*. On 7 out of 9 programs the precision reaches above 80%, which indicates that our method can efficiently rule out part of benign error and thus reduce human labor.

By comparing independent implementations This method uses two independent implementations P and Q of the same algorithm to evaluate whether an arithmetic error is likely harmful. If P and Q (1) agree (have identical or similar output) on all the inputs that trigger no arithmetic errors but (2) disagree (have different outputs) on the inputs that trigger arithmetic errors in P, then the errors in (2) are likely harmful. This is based on the conjecture that when an input triggers a harmful arithmetic error in P, it unlikely also triggers an arithmetic error in Q, and even if it does, the two errors unlikely cause P and Q to generate similar output. Obviously, the first property above requires the output to be a deterministic function of the input, i.e., no randomness may affect the output.

We applied the above method on the program *djpeg* in the *libjpeg-ijg* package. A JPEG encoder compresses an image by (1) dividing the image into 8×8 matrices and applying discrete cosine transform (DCT) to each matrix, (2) suppressing the high-frequency signals by element-wise dividing each matrix by a predefined matrix and rounding the result to the nearest integer, and (3) discarding all the tailing zeros. The decoder reverses the above operations, where it can infer the number of discarded zeros based on the size of the small matrix and that of the image.

Since a JPEG decoder uses floating point arithmetic, two independent decoders may create slight different outputs on the same input. However, if the difference is large, then at least one decoder is misbehaving. We measured the difference as the average L^1 distance between two images. More precisely, let

- A and B: two images of dimension $m \times n$.
- $A_{i,j}$: a 3-channel vector representing the RGB values of the pixel at (i, j)
- $A_{i,j}^{(k)}$: the value of the kth channel. This value is in the range $[0, 255]$, and $k \in \{1, 2, 3\}$.

Definition 1. *The* average L^1 distance *between two images A and B of identical size is:*

$$D(A,B) = \frac{\sum_{c \in C(A,B)} \sum_{k \in [1,3]} \mid c^{(k)} \mid}{\mid C(A,B) \mid} \tag{1}$$

where

$$C(A,B) = \{A_{i,j} - B_{i,j} : i \in [1,m], j \in [1,n], A[i,j] \neq B[i,j]\}$$

To evaluate whether non-crash arithmetic errors in *libjpeg-ijg* are harmful, we selected *libjpeg-turbo* as an alternative, independent implementation. *libjpeg-turbo* has the same API as *libjpeg-ijg*; however, its decoder uses SIMD instructions to accelerate arithmetic operations while *libjpeg-ijg* does not.

We prepared two sets of JPEG images as input to the decoders:

- Normal images: We randomly picked 100 JPEG images from Android system images, LaTeX testing images, *libjpeg* testing images, and GNOME 3.28 desktop images. None of them triggered arithmetic errors on either decoder.
- Exploit images: We collected images produced by *Int*egrity that triggered arithmetic errors in the program *djpeg* in the package *libjpeg-ijg*, and then removed the following from the collection:
  - Broken images: *Int*egrity generated many images that are invalid JPEG and therefore cannot be rendered.
  - Images whose width or height is less than 8 pixels. Since JPEG encoder partitions images into 8×8 matrices, the decoder's behavior on those images may be implementation-dependent.
  - Images that triggered only the benign errors described in Section 5.2

  After filtering, we were left with 67 exploit JPEG images.

Figure 4 compares the cumulative distribution functions (CDF) of the average L^1 distance (Equation 1) between normal and exploit images. The figure cleanly separates the CDF of normal and exploit images with no overlap: the L^1 distance of normal images ranges from 0.0 to 6.0 with a median of 2.4, while the distance of exploit images ranges from 16.9 to 342.4 with a median of 217.2. This implies that those arithmetic errors that *Int*egrity found in *libjpeg-ijg* are harmful.

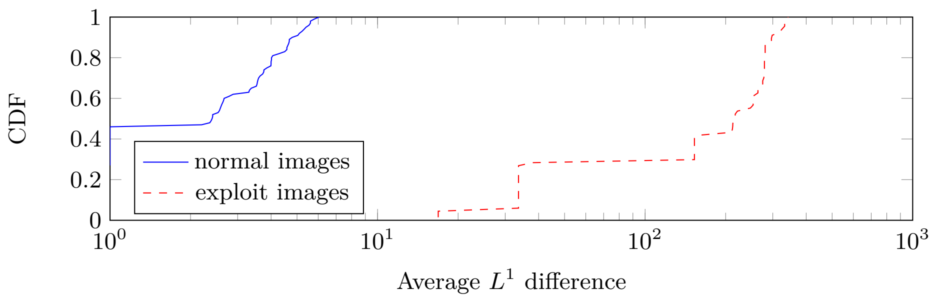

Fig. 4: Cumulative distribution function (CDF) of the average L^1 distance (Equation 1) between the output of two decoders on the same input JPEG image. The CDF of the normal images is cleanly separable from that of the exploit images.

5.4 Comparison with Angora + UBSan

We compared *Int*egrity with simple combination of Angora and UBSan. We ran Angora with UBSan in the same experimental configuration as we described in Section 5.2.

Table 1 compares the number of verified bugs found by *Int*egrity and Angora+UBSan, respectively. *Int*egrity found many more bugs than Angora on each program. On all program together, *Int*egrity found 174 bugs while Angora+UBScan found only 23 bugs. Angora+UBSan found no bug in *file*, *objdump* and *readlef*, but *Int*egrity found a total of 60 bugs in them. This result shows that *Int*egrity performs far superior than simple combination of Angora and UBSan. Without proper information sharing (Section 3.2 and Section 3.2), the fuzzer and the sanitizer cannot cooperate well because the fuzzer would not know where the potential bugs lie and divert computation power accordingly.

As a side note, we had to overcome engineering difficulties to combine Angora and UBSan. Angora compiles two binaries for each program: one uses Data Flow Sanitizer (DFSan) [6] to do taint tracking, and the other monitors the execution traces. DFSan instruments instructions to track data flow. If the program calls a function in third-party libraries, DFSan needs a modeled function to know how to propagate the taint. When we initially compiled the programs using UBSan and DFSan, it failed because DFSan could not find the modeled functions instrumented by UBSan. [31] also warned such issues when using multiple sanitizers. We applied a temporary hack to overcome the compilation problem: we enabled DFSan and disabled UBSan when compiling the binary for taint tracking, and enabled UBSan and disabled DFSan when compiling the binary for monitoring execution traces.

5.5 Instrumentation reduction

To evaluate the effect of instrumentation reduction described in Section 3.1, we instrumented five libraries with and without reduction and compared the number

Table 5: Number of instrumented arithmetic operations before and after instrumentation reduction

Library	# of instrumentation		Remaining
	after reduction	before reduction	instrumentation
libpng	2518	2773	90.80 %
binutils	16 432	18 203	90.27 %
libjpeg	14 335	15 312	93.62 %
libtiff	7383	8123	90.89 %
libpcap	714	887	80.50 %
Total	41 382	45 298	91.36 %

of instrumented arithmetic operations. Table 5 shows that overall this technique eliminated 9% instrumented arithmetic operations.

6 Related work

6.1 Detecting integer overflow

Integer overflow has been extensively studied [15,16,34,27,22,36]. IOC [15,16] instruments AST to test for overflow. It is now part of LLVM's UBSan [8].

IOC tends to generate many benign overflows. IntEQ [34] and IntFlow [27] intend to cut down reported benign overflows. Both use the assumption that an overflown value is benign unless it is used in a sink. IntFlow combines static and dynamic analysis to determine if any overflown value flows into a sink. IntEQ relies on symbolic execution to achieve this goal. It computes a value flown into a sink in both high and low precision and compares the two values. Both these tools rely on the user to provide input (test cases) for finding overflows. *Int*egrity overcomes this limitation by triggering arithmetic errors automatically through program instrumentation targeting arithmetic errors.

z3 [22] is a tool for solving integer-related symbolic constraints. IntScope [36] uses symbolic execution to detect integer overflow. Unlike IOC, IntScope does not rely on source code but translates x86 binary to an intermediate representation called PANDA first, then symbolically executes PANDA to detect possible arithmetic errors. Since *Int*egrity uses fuzzing, it inherits the advantages of fuzzing over symbolic execution, such as faster execution and tolerating obscure code (e.g., external libraries, system API, etc).

6.2 Coverage-directed fuzzers

A coverage-directed fuzzer mutates the input to explore paths in the hope to trigger bugs on some of these paths [13,14,1,5,33,37,28,10,30,11,12]. If a mutated input explores a new path, the fuzzer keeps the input as a seed. AFL [1] and LibFuzzer [5] employ evolutionary algorithms to mutate input. Driller [33] and QSYM [37] try to solve complex path constraints by concolic execution. VUzzer[28] and REDQUEEN [10] learn magic bytes and generate satisfying input without symbolic execution. AFLGo [11] and Hawkeye [12] direct fuzzing to a set of target program locations efficiently. Angora [13] models a path constraint as a black-box function, and uses optimization methods such as gradient descent to solve it. NEUZZ [30] also uses gradient descent to explore new paths and approximates the target program's branch coverage by a neural network.

Many coverage-directed fuzzers can turn on various sanitizers to detect bugs during exploration [31,29,32,18,7,8]. For example, Address Sanitizer [29], Memory Sanitizer [32], Thread Sanitizer [7], and Undefined Behavior Sanitizer [8] detect invalid memory addresses, use of uninitialized memory, data races, and undefined behavior, respectively. However, those fuzzers only passively detect those bugs when they are triggered by random mutation. By contrast, *Int*egrity instruments arithmetic operations with potential errors to triggers them actively.

6.3 Bug-directed fuzzers

Besides integer arithmetic errors, researchers developed fuzzers to exploit other vulnerabilities. SlowFuzz [26] targets algorithmic complexity vulnerabilities guided by resource usage. RAZZER [20] guides fuzzing towards potential data races in the kernel, then deterministically triggers a race. NEZHA [25] exploits the behavioral asymmetries between multiple test programs to focus on inputs that are more likely to trigger semantic bugs. Tensorfuzz [23] use coverage-guided fuzzing methods for neural networks to find numerical errors in a trained neural network. Dowser [17] determines "interesting" array accesses that likely harbor buffer overflow, and triggers overflow by taint tracking and symbolic execution. TIFF [19] infers input types by dynamic taint analysis, and sets input bytes with defined interesting values based on its type to maximize the likelihood of triggering memory-corruption bugs. Compared with those fuzzers, which were built to detect those specific bugs, *Int*egrity reduces the problem of exploitation to the problem of exploration, and therefore can work with most fuzzers and can benefit from the advances of exploration technologies.

7 Conclusion

We designed and implemented *Int*egrity for triggering integer arithmetic errors using fuzzing. By finding and instrumenting integer arithmetic operations with potential errors, *Int*egrity passes critical information to the fuzzer to help it trigger potential bugs. *Int*egrity found all the integer errors in the Juliet test suite with no false positive. On 9 popular open source programs, *Int*egrity found a total of 174 true errors, including 8 crashes and 166 non-crashing errors. To make progress on the challenge of determining if a non-crashing error is harmful, we proposed two methods to find potentially harmful errors, based on the statistics of traces produced by the fuzzer and on comparing the output of independent implementations of the same algorithm on the same input. Our evaluation demonstrated that *Int*egrity is effective in finding integer errors.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 1801751 and 1956364.

References

1. American fuzzy lop, `http://lcamtuf.coredump.cx/afl/`
2. Batchoverflow exploit creates trillions of ethereum tokens, major exchanges halt erc20 deposits | cryptoslate, `https://cryptoslate.com/batchoverflow-exploit-creates-trillions-of-ethereum-tokens/`
3. Beautychain (bec) withdrawal and trading suspended, `https://support.okex.com/hc/en-us/articles/360002944212-BeautyChain-BEC-Withdrawal-and-Trading-Suspended-Update-`
4. Cwe - common weakness enumeration, `https://cwe.mitre.org/`
5. libfuzzer – a library for coverage-guided fuzz testing., `https://llvm.org/docs/LibFuzzer.html`
6. LLVM dataflowsanitizer, `https://clang.llvm.org/docs/DataFlowSanitizer.html`
7. LLVM threadsanitizer, `https://clang.llvm.org/docs/ThreadSanitizer.html`
8. LLVM undefinedbehaviorsanitizer, `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`
9. Software assurance reference dataset, `https://samate.nist.gov/SARD/testsuite.php`
10. Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., Holz, T.: Redqueen: Fuzzing with input-to-state correspondence (2019)
11. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2329–2344. ACM (2017)
12. Chen, H., Xue, Y., Li, Y., Chen, B., Xie, X., Wu, X., Liu, Y.: Hawkeye: towards a desired directed grey-box fuzzer. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 2095–2108. ACM (2018)
13. Chen, P., Chen, H.: Angora: Efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 711–725. IEEE (2018)
14. Chen, P., Liu, J., Chen, H.: Matryoshka: fuzzing deeply nested branches. In: ACM Conference on Computer and Communications Security (CCS). London, UK
15. Dietz, W., Li, P., Regehr, J., Adve, V.: Understanding integer overflow in c/c++. In: 34th International Conference on Software Engineering, ICSE 2012 (2012)
16. Dietz, W., Li, P., Regehr, J., Adve, V.: Understanding integer overflow in c/c++. ACM Transactions on Software Engineering and Methodology (TOSEM) **25**(1), 2 (2015)
17. Haller, I., Slowinska, A., Neugschwandtner, M., Bos, H.: Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In: USENIX security. pp. 49–64 (2013)
18. Han, W., Joe, B., Lee, B., Song, C., Shin, I.: Enhancing memory error detection for large-scale applications and fuzz testing. In: Symposium on Network and Distributed Systems Security (NDSS). p. 148 (2018)
19. Jain, V., Rawat, S., Giuffrida, C., Bos, H.: Tiff: Using input type inference to improve fuzzing. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 505–517. ACM (2018)

20. Jeong, D.R., Kim, K., Shivakumar, B., Lee, B., Shin, I.: Razzer: Finding kernel race bugs through fuzzing. In: Razzer: Finding Kernel Race Bugs through Fuzzing. IEEE (2018)
21. Martin, B., Brown, M., Paller, A., Kirby, D., Christey, S.: 2011 cwe/sans top 25 most dangerous software errors. Common Weakness Enumer **7515** (2011)
22. Moy, Y., Bjørner, N., Sielaff, D.: Modular bug-finding for integer overflows in the large: Sound, efficient, bit-precise static analysis. Microsoft Research **11** (2009)
23. Odena, A., Goodfellow, I.: Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. arXiv preprint arXiv:1807.10875 (2018)
24. Peng, H., Shoshitaishvili, Y., Payer, M.: T-fuzz: fuzzing by program transformation. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 697–710. IEEE (2018)
25. Petsios, T., Tang, A., Stolfo, S., Keromytis, A.D., Jana, S.: Nezha: Efficient domain-independent differential testing. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 615–632. IEEE (2017)
26. Petsios, T., Zhao, J., Keromytis, A.D., Jana, S.: Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2155–2168. ACM (2017)
27. Pomonis, M., Petsios, T., Jee, K., Polychronakis, M., Keromytis, A.D.: Intflow: improving the accuracy of arithmetic error detection using information flow tracking. In: Proceedings of the 30th Annual Computer Security Applications Conference. pp. 416–425. ACM (2014)
28. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: VUzzer: application-aware evolutionary fuzzing. In: NDSS (Feb 2017)
29. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: A fast address sanity checker. In: USENIX ATC 2012 (2012)
30. She, D., Pei, K., Epstein, D., Yang, J., Ray, B., Jana, S.: Neuzz: Efficient fuzzing with neural program learning (2019)
31. Song, D., Lettner, J., Rajasekaran, P., Na, Y., Volckaert, S., Larsen, P., Franz, M.: Sok: sanitizing for security (2019)
32. Stepanov, E., Serebryany, K.: Memorysanitizer: fast detector of uninitialized memory use in c++. In: Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 46–55. IEEE Computer Society (2015)
33. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: augmenting fuzzing through selective symbolic execution. In: Proceedings of the Network and Distributed System Security Symposium (2016)
34. Sun, H., Zhang, X., Zheng, Y., Zeng, Q.: Inteq: recognizing benign integer overflows via equivalence checking across multiple precisions. In: Proceedings of the 38th International Conference on Software Engineering. pp. 1051–1062. ACM (2016)
35. Wang, T., Wei, T., Gu, G., Zou, W.: Taintscope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Security and privacy (SP), 2010 IEEE symposium on. pp. 497–512 (2010)
36. Wang, T., Wei, T., Lin, Z., Zou, W.: Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In: NDSS. Citeseer (2009)
37. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 745–761. USENIX Association, Baltimore, MD (2018)