IntFlow: Improving the Accuracy of Arithmetic Error Detection Using Information Flow Tracking

Marios Pomonis
Columbia University
mpomonis@cs.columbia.edu

Theofilos Petsios
Columbia University
theofilos@cs.columbia.edu

Kangkook Jee
Columbia University
jikk@cs.columbia.edu

Michalis Polychronakis
Columbia University
mikepo@cs.columbia.edu

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

ABSTRACT

Integer overflow and underflow, signedness conversion, and other types of arithmetic errors in C/C++ programs are among the most common software flaws that result in exploitable vulnerabilities. Despite significant advances in automating the detection of arithmetic errors, existing tools have not seen widespread adoption mainly due to their increased number of false positives. Developers rely on wrap-around counters, bit shifts, and other language constructs for performance optimizations and code compactness, but those same constructs, along with incorrect assumptions and conditions of undefined behavior, are often the main cause of severe vulnerabilities. Accurate differentiation between legitimate and erroneous uses of arithmetic language intricacies thus remains an open problem.

As a step towards addressing this issue, we present *IntFlow*, an accurate arithmetic error detection tool that combines static information flow tracking and dynamic program analysis. By associating sources of untrusted input with the identified arithmetic errors, IntFlow differentiates between non-critical, possibly developer-intended undefined arithmetic operations, and potentially exploitable arithmetic bugs. IntFlow examines a broad set of integer errors, covering almost all cases of C/C++ undefined behaviors, and achieves high error detection coverage. We evaluated IntFlow using the SPEC benchmarks and a series of real-world applications, and measured its effectiveness in detecting arithmetic error vulnerabilities and reducing false positives. IntFlow successfully detected all real-world vulnerabilities for the tested applications and achieved a reduction of 89% in false positives over standalone static code instrumentation.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.2.5 [**Software Engineering**]: Testing and Debugging

ACSAC '14, December 08–12 2014, New Orleans, LA, USA

http://dx.doi.org/10.1145/2664243.2664282 .

General Terms

Security, Reliability

Keywords

Static analysis, information flow tracking, arithmetic errors

1. INTRODUCTION

When developing programs in the C and C++ languages, programming practices that involve undefined arithmetic operations often constitute a well-established status quo. Compilers tolerate the use of undefined behavior, as this enables code optimizations that greatly increase the performance of critical code segments. Consequently, programmers often purposely rely on undefined language constructs due to empirical certainty for certain expected outcomes. This flexibility comes at a cost: arithmetic operations constitute a major source of errors, often leading to serious security breaches when erroneous values directly or indirectly affect sensitive system calls or memory operations.

The root of the problem is fundamentally bound to the differences between the mathematical and machine representations of numbers: although integer and floating point numbers are infinite, their machine representations are restricted by their respective type-specific characteristics (e.g., signedness and bit-length). Furthermore, not all arithmetic operations are well-defined by the language standards, to allow for a number of compiler optimizations. For instance, the standard does not specify what the value of a signed integer that overflows (or underflows) should be.

As it is non-trivial to determine whether a particular arithmetic operation resulting in undefined behavior is benign or not, bugs due to integer errors are prevalent. Integer errors are listed among the 25 most dangerous software bugs [3], and are often the root cause of various vulnerabilities such as buffer overflows [25] and memory disclosures [2]. During the past years, numerous attempts have been made towards their automatic detection and prevention. Such efforts include static [23] and dynamic [7] analysis solutions, tools based on symbolic execution and dynamic test generation [8, 17], as well as compiler extensions [25] that resolve ambiguities at compilation time.

Despite numerous suggested solutions, there is no generic tool that provides effective and complete detection and sanitization of integer bugs. One reason is that tools that focus on coverage typically generate a large amount of false positives (due to the inaccuracy of static analysis) [25, 23, 16],

while those that focus on accuracy provide poor coverage (as they rely on dynamic analysis or dynamic test generation) [10, 17, 19, 20]. Furthermore, existing tools typically focus only on certain integer error classes (mainly overflows and underflows) and as a result they do not provide broad spectrum protection. The most crucial reason, however, is the inherent difficulty of prevention mechanisms to differentiate between critical integer errors that may lead to exploitable vulnerabilities, from *intentional* uses of wrap-around behavior, type castings, bit shifts, and other constructs that serve application-specific purposes.

As a step towards addressing these issues, in this paper we propose an approach that combines static code instrumentation with information flow tracking to improve the accuracy of arithmetic error detection, focusing on reducing the number of false positives, i.e., developer-intended code constructs that violate language standards. Our tool, IntFlow, uses information flow tracking to reason about the severity of arithmetic errors by analyzing the information flows related to them. The main intuition behind this approach is that arithmetic errors are critical when i) they are triggered by or depend on values originating from untrusted locations, or ii) a value affected by an arithmetic error propagates to sensitive locations, such as the arguments to functions like `malloc()` and `strcpy()`.

To demonstrate the effectiveness of our approach, we evaluated IntFlow with real world programs and vulnerabilities and verified that it successfully identifies all the real world vulnerabilities for the applications of our testbed, generating 89% less false positives compared to IOC [10], our reference arithmetic error detection tool.

Our work makes the following contributions:

- We present an accurate arithmetic error detection approach that combines static information flow tracking and dynamic program analysis.
- We present IntFlow, our prototype implementation for this approach, which operates as an LLVM add-on. IntFlow is freely available as an open source project.[1]
- We evaluate IntFlow using real-world programs and vulnerabilities. Our results demonstrate that IntFlow achieves improved detection accuracy compared to previous solutions, as it suppressed more than 89% of the false positives reported by IOC [10].

2. BACKGROUND

To effectively deal with integer errors in real world applications, it is necessary to first define what is considered an error. Doing so is not trivial, as apart from the mere examination of conformance to the language standard, we must also examine whether pieces of seemingly erroneous code—from the perspective of the language specification—are explicitly written in that way due to the developer's intention, typically for performance, convenience, or other reasons.

In this section, we discuss how the C/C++ language standards define correctness for arithmetic operations, and examine why developers often write code that deviates from the language specification. We also present examples of exploitable vulnerabilities caused by integer errors, and demonstrate the importance of good programming practices.

[1]http://nsl.cs.columbia.edu/projects/intflow/

2.1 Integer Errors and Undefined Behavior

Although the C and C++ language standards explicitly define the outcome of most integer operations, a number of corner cases are left undefined. As an example, the C11 standard considers an `unsigned` integer overflow as a well-defined operation, whose result is the minimum value obtained after the wrap-around, while leaving `signed` integer overflows undefined. This choice facilitates compiler implementations to handle them in a way that produces optimized binaries [24]. For instance, signed integer overflows (or underflows) enable compiler developers to implement an optimization that infers invariants from expressions such as `i+1 > i` and replaces them with a constant Boolean value [5].

Table 1 lists special cases of integer operations and their definedness. It should be noted that although more instances of undefined behavior (not necessarily restricted to integer operations) are declared in the language specification, we only consider those relevant to this work.

Arithmetic Operation	Definedness
Unsigned overflow (underflow)	defined
Singed overflow (underflow)	undefined
Signedness conversion	undefined*
Implicit type conversion	undefined*
Oversized/negative shift	undefined
Division by zero	undefined

*if value cannot be represented by the new type

Table 1: Summary of defined and undefined arithmetic operations according to the C/C++ language specification.

As in practice not all cases of undefined behavior necessarily result in actual errors, the difficulty of dealing with these types of bugs lies in distinguishing *critical* integer errors from *developer-intended* violations of the standard. The intention of a developer, however, cannot be formally defined or automatically derived, as the code patterns used in a piece of code are deeply related to the author's knowledge, preference, and programming style.

Although writing code that intentionally relies on undefined operations is generally considered a bad programming practice (as the outcome of those operations can be arbitrary, depending on the architecture and the compiler), there are several cases in which the community has reached consensus on what is the expected behavior of the compiler in terms of the generated code, mainly due to empirical evidence. This explains why idioms that take advantage of undefined behavior are still so prevalent: although *according to the standard* the result of an operation is undefined, developers have an empirically derived expectation that compilers will always handle such cases in a consistent manner.

This expectation creates serious complications whenever developers check the validity of their code with state-of-the-art static analysis tools. These tools evaluate code based on strict conformance to the language specification, and consequently generate a large amount of false positives. Thus, the generated reports are often overlooked by developers who struggle to spot which of the reported bugs are actual errors. Unfortunately, tools based on dynamic code analysis also do not provide strong guarantees in these cases, as they suffer from low code coverage.

```
1  UINT_MAX = (unsigned) −1;
2  INT_MAX = 1 << (INT_WIDTH − 1) − 1;
```

Listing 1: Widely used idioms that according to the standard correspond to undefined behavior.

```
1  /* struct containing image data, 10KB each */
2  img_t *table_ptr;
3  unsigned int num_imgs = get_num_imgs();
4  ...
5  unsigned int alloc_size = sizeof(img_t) * num_imgs;
6  ...
7  table_ptr = (img_t*) malloc(alloc_size);
8  ...
9  for (i = 0; i < num_imgs; i++)
10         { table_ptr[i] = read_img(i); } /* heap overflow */
```

Listing 2: An unsigned integer overflow as a result of a multiplication (line 5), which results in an invalid memory allocation (line 7) and unintended access to the heap (line 10).

To further illustrate the complexity of this issue, in the following we present two characteristic integer error examples and discuss the complications introduced by the use of undefined operations.

2.2 Integer Error Examples

While the task of automatically detecting undefined arithmetic operations is relatively easy, the true difficulty lies in identifying the developer's intention behind the use of constructs that violate the language standard.

As an example, Listing 1 presents two C statements in which developers intentionally rely on undefined behavior, mainly to achieve persistent representation across different system architectures. Both are based on assumptions on the numerical representation used by the underlying system (two's complement). Line 1 shows a case of signedness casting in which the original value cannot be represented by the new type. In Line 2, a shift operation of `INT_WIDTH - 1` is undefined[2] but it conventionally returns the minimum value of the type, while the subtraction operation incurs a signed underflow which is also undefined. Although these cases are violations of the language standard, the desirable operation of an integer overflow checker would be to *not* report them, as they correspond to developer-intended behavior—otherwise, such cases are considered *false positives* [10].

In contrast, in the example of Listing 2, the `unsigned integer` variable (`alloc_size`) might overflow as a result of the multiplication operation at line 5. This behavior is well-defined by the standard, but the overflow may result in the allocation of a memory chunk of invalid (smaller) size, and consequently, to a heap overflow. An effective arithmetic error checker should be able to identify such potentially exploitable vulnerabilities as it is clear that the developer did not intend for this behavior.

3. APPROACH

The security community is still unsuccessful in completely eliminating the problem of integer errors even after years of effort [10, 17, 25, 23]. One of the main reasons is the difficulty in distinguishing *critical* errors, which may lead to reliability issues or security flaws, from uses of undefined constructs stemming from certain programming practices. The latter are regarded as errors by rigorous static checkers like IOC, as they strictly follow the language standard and report all violations found. In this work, we attempt to pinpoint critical, possibly exploitable arithmetic errors among all arithmetic violations, which include numerous less critical and often developer-intended uses of undefined behavior. Although many programming choices deviate from the language specification, IntFlow examines the conditions under which such constructs signify critical bugs, by focusing only on detecting errors that might break the functionality of the program or lead to a security flaw.

Before describing IntFlow's design, we first provide a concrete definition of what we consider critical arithmetic errors.

Definition 1. *An arithmetic error is potentially critical if it satisfies any of the following conditions:*

1. *At least one of the operands in an erroneous arithmetic operation originates from an* ***untrusted source****.*
2. *The result of an erroneous operation propagates to a* ***sensitive program location****.*

As capturing the intention of developers is a hard problem, IntFlow focuses on the detection of arithmetic errors that might constitute exploitable vulnerabilities or cause reliability issues. This is achieved not only by focusing on the identification of violations according to the language standard, which in itself is a tractable problem, but also by considering the information flows that affect the erroneous code. The rationale behind this definition is that *(1)* arithmetic errors influenced by external and potentially untrusted sources, such as sockets, files, and environment variables, may be exploited through carefully crafted inputs, and *(2)* arithmetic errors typically result in severe vulnerabilities when they affect sensitive library and system operations, such as memory allocation and string handling functions.

The two conditions of Definition 1 are reflected in IntFlow by two different modes of operation, *blacklisting* and *sensitive* (in addition to a third *whitelisting* mode), discussed in Section 4.3. Although the existence of either condition is an indication of a critical error, arithmetic violations for which both conditions hold are more severe, as they can potentially allow input from untrusted sources to misuse critical system functions—IntFlow's different modes of operation can be combined to detect such errors.

Figure 1 visualizes the definition with different types of information flows that may involve erroneous arithmetic operations. Critical errors are related to information flows that originate from untrusted inputs, or that eventually reach sensitive operations, such as system calls, through value propagation. In cases where the input of an arithmetic operation is untrusted (Definition 1.1) or a sensitive sink is reached (Definition 1.2), the error is flagged as critical. In contrast, arithmetic errors influenced only by benign inputs are considered less likely to be used in exploitation attempts.

The information flow based approach enables us to handle cases similar to the examples presented in Section 2.2. IntFlow can also silence error reports caused by statements similar to those presented in Listing 1, since it can trace that the

[2]According to the C99 and C11 standards. The C89 and C90 (ANSI C) standards define this behavior.

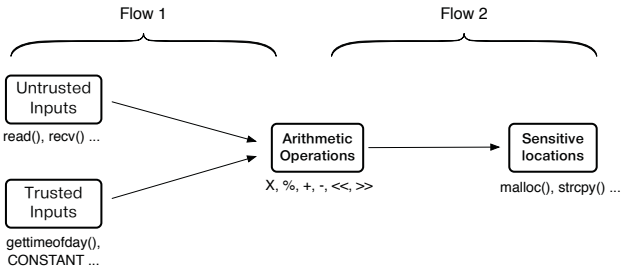

Figure 1: Information flows to and from the location of an arithmetic error.

origin of the value that leads to the undefined behavior is a constant initialization which is de facto developer-intended. The code snippet in Listing 2, contains two different types of flows. The first one connects `get_num_imgs()` (line 3) with the multiplication operation (line 5) while the second one connects the result of the multiplication with a memory allocation function (line 7), which is considered a sensitive program location. The former is a Type 1 flow because the value of the multiplication operand originates from an untrusted input, while the latter is a Type 2 flow since the result of the operation affects a sensitive function call. IntFlow would detect and report arithmetic errors caused by maliciously crafted inputs in both cases.

4. DESIGN AND IMPLEMENTATION

In this section we present the design and implementation of IntFlow, a tool that combines information flow tracking (IFT) [18] with a popular integer error checking tool [10] to improve the accuracy of arithmetic error detection. The main goal of IntFlow is to reduce the number of false positives produced by other static arithmetic error checkers. In this context, the term "false positive" refers to reporting developer-intended violations as critical errors. Although from the perspective of the language standard these correspond to erroneous code, the prevalence of such constructs makes reports of such issues a burden for security analysts, who are interested only in critical errors that may form exploitable vulnerabilities.

4.1 Main Components

IOC operates at the abstract syntax tree (AST) level produced by Clang [1], a C/C++ front-end of LLVM [14]. It instruments all arithmetic operations, as well as most of unary, casting, and type conversion operations. In contrast to previous tools that focus on a subset of integer errors (typically overflows and underflows), IOC provides protection against a broader range of integer errors. Even though it focuses mainly on errors with undefined behavior based on the language standards, it can also protect against errors that do not fall into this category, covering most of the integer error classes presented in Table 1.

IOC instruments *all* arithmetic operations that may lead to an erroneous result, and inserts checks accordingly. Essentially, for each integer operation inside a basic block, additional basic blocks that implement the error-checking logic are added and users are allowed to register callback functions for error handling. Similarly to other integer error detection systems, the fact that IOC instruments blindly all arithmetic operations is a major source of false positives, while IntFlow's active provisioning allows it to reduce false positives by eliminating checks for non-critical violations. IOC is a major component of our architecture, as it provides assurance that all potentially serious arithmetic errors can be checked. It is then up to the information flow analysis to identify and report only the critical ones.

For IntFlow's information tracking mechanism we employ `llvm-deps` [18], an LLVM compiler pass implementing static information flow tracking in a manner similar to classic data flow analysis [5]. It is designed as a context sensitive (inter-procedural) analysis tool that allows forward and backward slicing on source and sink pairs of our choice using the DSA [15] algorithm. DSA performs context-sensitive, unification-based points-to analysis that allows us to track data flows among variables referred by pointer aliases. It is important to note that the analysis scope of `llvm-deps` is limited to a single object file, as it is implemented as a compile-time optimization pass and not as a link-time optimization pass. Finally, due to the use of `llvm-deps`, IntFlow only examines explicit flows during its IFT analysis and ignores possible implicit flows.

4.2 Putting It All Together

Figure 2 illustrates the overall architecture of IntFlow: IOC adds checks to the integer operations that are exposed by Clang in the AST and then `llvm-deps` performs static IFT analysis on the LLVM intermediate representation (IR). To reduce unnecessary checks that may lead to false positives, IntFlow uses `llvm-deps` to examine only certain flows of interest. As discussed in the previous section, IntFlow examines only flows stemming from untrusted sources, or ending to sensitive sinks. Initially, IntFlow performs *forward* slicing: starting from a particular source used in a potentially erroneous arithmetic operation, it examines whether the result of the operation flows into sinks of interest. Once such a source is found, IntFlow performs *backward* slicing, to verify that the sink is actually affected by it. Since the flow tracking mechanism does not offer full code coverage, we employ this two-step process to gain confidence on the accuracy of the flow and verify its validity. Once the source is reached when using backward slicing starting from the sink, the flow is considered established.

After compiling and linking the IR, the resulting binary is exercised to identify critical errors, since the error checking mechanism triggers dynamically. Developers should execute the augmented binary with a broad range of inputs to exercise as many execution paths as possible, and identify whether they cause critical errors that can potentially lead to exploitable vulnerabilities.

4.3 Modes of Operation

As discussed in Section 3, IntFlow uses two different types of information flow to pinpoint errors. The first associates untrusted inputs with integer operations while the second associates the result of integer operations with its use in sensitive system functions. Once IOC inserts checks in all arithmetic operations that may lead to an error, IntFlow eliminates unnecessary checks by operating in one of the following modes:

- In *blacklisting* mode, IntFlow only maintains checks for operations whose operands originate from untrusted sources and removes all other checks.

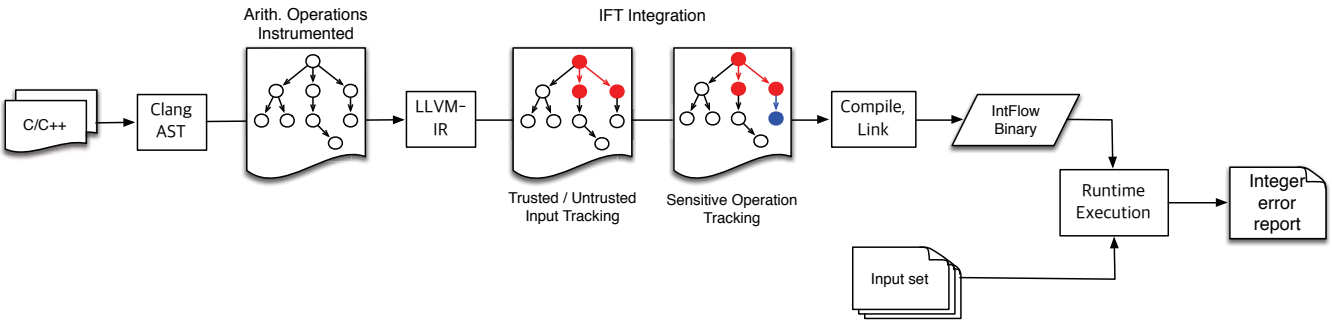

Figure 2: Overall architecture of IntFlow.

- In *sensitive* mode, IntFlow only maintains checks for operations whose results may propagate to sensitive sinks.

- In *whitelisting* mode, all checks for operations whose arguments come from trusted sources are removed.

4.3.1 Trusted and Untrusted Inputs

For each operation that may result in an arithmetic error, IntFlow's IFT analysis determines the origin of the involved operands. IntFlow then classifies the origin as either trusted or untrusted, and handles it accordingly, using one of the following two modes of operation.

Blacklisting: Input sources that can be affected by external sources are considered untrusted, since carefully crafted inputs may lead to successful exploitation. If any of the operands has a value affected by such a source, IntFlow retains the error checking instrumentation. System and library calls that read from untrusted sources, such as `read()` and `recv()`, are examples of this type of sources.

Whitelisting: Erroneous arithmetic operations for which all operands originate from trusted sources are unlikely to be exploitable. Thus, for those cases, IntFlow safely removes the error checks inserted by IOC at the instrumentation phase. Before an operation is verified as safe, IntFlow needs to examine the origin of *all* data flowing to that operation. Values derived from constant assignments or from safe system calls and library functions, e.g., `gettimeofday()` or `uname()`, are typical examples of sources that can be trusted, and thus white-listed.

Following either of the above two approaches, IntFlow selects the unsafe integer operations that will be instrumented with protection checks. These modes of operation can be complemented by IntFlow's third mode, which refines the analysis results for the surviving checks.

4.3.2 Sensitive Operations

In this mode, IntFlow reports flows that originate from integer error locations and propagate to sensitive sinks, such as memory-related functions and system calls. Moreover, in contrast to the previous modes, whenever an integer error occurs, the error is not reported at the time of its occurrence, but only once it propagates as input into one of the sensitive sinks. This is very effective in suppressing false positives, since errors that do not flow to a sensitive operation are not generally exploitable.

To report errors at sensitive sink locations, IntFlow performs the following operations:

- Initially, the tool identifies all integer operations whose results may propagate into a sensitive sink at runtime. Any checks that do not lead to sensitive sinks are not exploitable and thus are eliminated. A global array is created for each sensitive sink, holding one entry per arithmetic operation affecting it.

- Whenever an integer operation generates an erroneous result, its respective entries in the affected global arrays are set to `true`. If the operation is reached again but without generating an erroneous result, before reaching a sensitive location, the respective entry is set to `false`, denoting that the result of the sensitive operation will not be affected by this operation.

- If the execution reaches a sensitive function, the respective global array is examined. Execution is interrupted if one or more entries are set to `true`, as an erroneous value from any previous integer operation may affect its outcome.

Essentially, IntFlow keeps track of all the locations in the code that may introduce errors affecting a sensitive sink at compilation time. Once a sensitive sink is reached during runtime, IntFlow examines whether *any* of the error locations associated with the sink triggered an error in the current execution flow and in that case terminates the program. Although it is better to combine the two modes to establish end-to-end monitoring and detection of suspicious flows, each mode can also be used independently: the first mode to generally reduce the number of false positives, and the second mode to detect exploitable vulnerabilities.

4.4 Implementation

IntFlow is implemented as an LLVM [14] pass written in ~3,000 lines of C++ code. Briefly, it glues together its two main components (IOC and `llvm-deps`) and supports fine-tuning of its core engine through custom configuration files.

IntFlow can be invoked by simply passing the appropriate flags to the compiler, without any further action needed from the side of the developer. Although IOC has been integrated into the LLVM main branch since version 3.3, for the current prototype of IntFlow we used an older branch of IOC that supports a broader set of error classes than the latest one. IntFlow's pass is placed at the earliest stage of the LLVM pass dependency tree to prevent subsequent optimization passes from optimizing away any critical integer operations. During compilation, arithmetic error checks are inserted by IOC, and then selectively filtered by IntFlow. Subsequent compiler optimizations remove the filtered IOC basic blocks.

IntFlow offers developers the option to explicitly specify arithmetic operations or sources that need to be whitelisted or blacklisted. In addition, it can be configured to exclude any specific file from its analysis or ignore specific lines of code. Developers can also specify the mode of operation that IntFlow will use, as well as override or extend the default set of sources and sinks that will be considered during information flow analysis. Finally, they can specify particular callback actions that will be triggered upon the discovery of an error, such as activating runtime logging or exiting with a suitable return value. These features offer great flexibility to developers, enabling them to fine-tune the granularity of the generated reports, and adjust the built-in options of IntFlow to the exact characteristics of their source code.

5. EVALUATION

In this section, we present the results of our experimental evaluation using our prototype implementation of IntFlow. To assess the effectiveness and performance of IntFlow, we look into the following aspects:

- What is the accuracy of IntFlow in detecting and preventing critical arithmetic errors?
- How effective is IntFlow in reducing false positives? That is, how good is it in omitting developer-intended violations from the reported results?
- When used as a protection mechanism, what is the runtime overhead of IntFlow compared to native execution?

Our first set of experiments aims to evaluate the tool's ability to identify and mitigate critical errors. For this purpose, we use two datasets consisting of artificial and real-world vulnerabilities. Artificial vulnerabilities were inserted to a set of real-world applications, corresponding to various types of MITRE's Common Weakness Enumeration (CWE) [4]. This dataset provides a broad test suite that contains instances of many different types of arithmetic errors, which enables us to evaluate IntFlow in a well-controlled environment, knowing exactly how many bugs have been inserted, as well as the nature of each bug. Likewise, our real-world vulnerability dataset consists of applications such as image and document processing tools, instant messaging clients, and web browsers, with known CVEs, allowing us to get some insight on how well IntFlow performs against real-world, exploitable bugs.

In our second round of experiments, we evaluate the effectiveness of IntFlow's information flow tracking analysis in reducing false positives, by running IntFlow on the SPEC CPU2000 benchmark suite and comparing its reported errors with those of IOC. IOC instruments all arithmetic operations, providing the finest possible granularity for checks. Thus, by comparing the reports produced by IntFlow and IOC, we obtain a base case for how many non-critical errors are correctly ignored by the IFT engine.

Finally, to obtain an estimate of the tool's runtime overhead, we run IntFlow over a diverse set of applications of varying complexity, and establish a set of performance bounds for different types of binaries. All experiments were performed on a system with the following characteristics: 2× Intel(R) Xeon(R) X5550 CPU @ 2.67GHz, 2GB RAM, i386 Linux.

Applications	CWEs
Cherokee 1.2.101	CWE-190
Grep 2.14	CWE-191
Nginx 1.2.3	CWE-194
Tcpdump 4.3.0	CWE-195
W3C 5.4.0	CWE-196
Wget 1.14	CWE-197
Zshell 5.0.0	CWE-369
	CWE-682
	CWE-839

Table 2: Summary of the applications and CWEs used in the artificial vulnerabilities evaluation.

5.1 Accuracy

In all experiments for evaluating accuracy, we configured IntFlow to operate in *whitelisting* mode, since this mode produces the greatest number of false positives, as it preserves most of the IOC checks among the three modes. Thus, whitelisting provides us with an estimation of the worst-case performance of IntFlow, since the other two modes perform more fine-tuned instrumentation.

5.1.1 Evaluation Using Artificial Vulnerabilities

To evaluate the effectiveness of IntFlow in detecting critical errors of different types, we used seven popular open-source applications with planted vulnerabilities from nine distinct CWE categories.[3] Table 2 provides a summary of the applications used and the respective CWEs.

Each application is replicated to create a set of test-case binaries. In every test-case binary—essentially an instance of the real-world application—a vulnerability is planted and then the application is compiled with IntFlow. Subsequently, each test-case binary is executed over a set of benign and malicious inputs (inputs that exploit the vulnerability and result in abnormal behavior). A correct execution is observed when the binary executes normally on benign inputs or terminates before it can be exploited on malicious inputs.

Overall, IntFlow was able to correctly identify 79.30% (429 out of 541) of the planted artificial vulnerabilities. The 20.7% missed are due to the accuracy limitations of the IFT mechanism, which impacts the ability of IntFlow to correctly identify flows, and also due to vulnerabilities triggered by implicit information flows (i.e., non-explicit data flows realized by alternate control paths), which our IFT implementation (`llvm-deps`) is not designed to capture. We discuss ways in which accuracy can be further improved in Section 7.

5.1.2 Mitigation of Real-world Vulnerabilities

In our next experiment, we examined the effectiveness of IntFlow in detecting and reporting real-world vulnerabilities. For this purpose, we used four widely-used applications and analyzed whether IntFlow detects known integer-related CVEs included in these programs. Table 3 summarizes our evaluation results. IntFlow successfully detected *all* the exploitable vulnerabilities under examination. From this small-scale experiment, we gain confidence that IntFlow's

[3]The modified applications for this experiment were provided by MITRE for testing the detection of integer error vulnerabilities, as part of the evaluation of a research prototype [6].

Program	CVE Number	Type	Detected?
Dillo	CVE-2009-3481	Integer Overflow	Yes
GIMP	CVE-2012-3481	Integer Overflow	Yes
Swftools	CVE-2010-1516	Integer Overflow	Yes
Pidgin	CVE-2013-6489	Signedness Error	Yes

Table 3: CVEs examined by IntFlow.

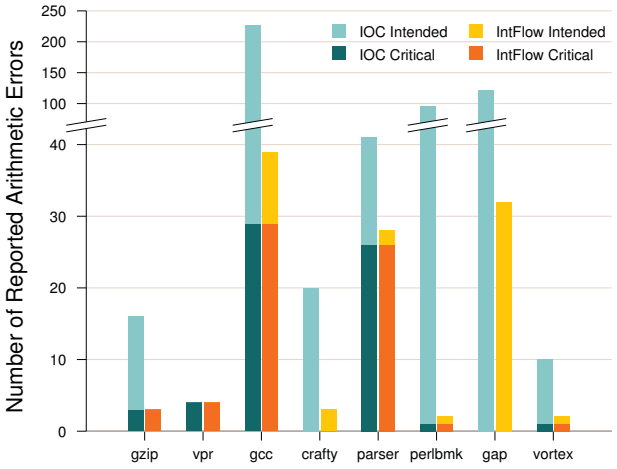

Figure 3: Number of critical and developer-intended arithmetic errors reported by IOC and IntFlow for the SPEC CPU2000 benchmarks. IntFlow identifies the same number of critical errors (dark sub-bars), while it reduces significantly the number of reported developer-intended violations.

characteristics are maintained when the tool is applied to real world programs, and therefore it is suitable as a detection tool for real-world applications.

5.1.3 *False Positives Reduction*

Reducing the number of false positives is a major goal of IntFlow, and this section focuses on quantifying how effective this reduction is. For our first measurement we used SPEC CPU2000, a suite that contains C and C++ programs representative of real-world applications. Since IOC is a core component of IntFlow, we chose to examine the same subset of benchmarks of CPU2000 that was used for the evaluation of IOC to measure the improvements of IntFlow's IFT in comparison to previously reported results [10]. We ran the SPEC benchmarks using the "test" data sets for both IOC and IntFlow, so that we could manually analyze all the reports produced by IOC and classify them as true or false positives, Once all reports were categorized based on Definition 1, we examined the respective results of IntFlow. We report our findings in Figure 3.

IntFlow was able to correctly identify all the critical errors (64 out of 64) triggered during execution, and reduced the reports of developer-intended violations by ∼89%.

5.1.4 *Real-world Applications*

In Section 5.1.2 we demonstrated how IntFlow effectively detected known CVEs for a set of real-world applications. Here, we examine the reduction in false positives achieved when using IntFlow's core engine instead of static instrumentation with IOC alone. To collect error reports, we ran

	Overall	Dillo	Gimp	Pidgin	SWFTools
IOC	330	31	231	0	68
IntFlow	82	26	13	0	43

Table 4: Number of False Positives reported by IOC and IntFlow for the real-world programs of Section 5.1.2.

each application with benign inputs as follows: for Gimp, we scaled the ACSAC logo and exported it as GIF; for SWFTools, we used the `pdf2swf` utility with a popular e-book as input; for Dillo, we visited the ACSAC webpage and downloaded the list of notable items published in 2013; and for Pidgin, we performed various common tasks, such as registering a new account and logging in and out. Table 4 shows the reports generated by IOC and IntFlow, respectively.

Overall, IntFlow was able to suppress 75% of the errors reported by IOC during the execution of the applications on benign inputs. Although this evaluation does not provide full coverage on the number of generated reports (for instance, we did not observe any false positives for pidgin with the tests we performed), it allows us to obtain an estimate of how well IntFlow performs in real world scenarios.

As the last part of our false positive reduction, we exercised vanilla versions of each application used in Section 5.1.1 over sets of benign inputs and examined the output. Since those inputs produce the expected output, we assume that all reported violations are developer-intended either explicitly or implicitly. With this in mind, we compared the error checks of IntFlow with those of IOC to quantify the ability of IntFlow in removing unnecessary checks. Overall, IntFlow eliminated 90% of the false checks (583 out of 647) when tested with the default set of safe inputs. This reduction was achieved due to the successful identification of constant assignments and the whitelisting of secure system calls, as discussed in Section 3.

It should be noted that the effectiveness in the reduction of false positives is highly dependent on the nature of each application, as well as on the level of the execution's source coverage. That is, the more integer operations occur throughout the execution, the greater the expected number of false positives. For instance, Gimp's functionality is tightly bound to performing arithmetic operations for a number of image processing actions, and thus IOC reports many errors, most of which are developer-intended, while Dillo does not share the same characteristics and as a result exhibits a smaller reduction in false positives.

5.2 Runtime Overhead

Although IntFlow was not designed as a runtime detection tool but rather as an offline integer error detection mechanism, one may wonder whether it could be customized to offer runtime detection capabilities. In this section, we seek to examine the performance of IntFlow for various applications, when running them with all the automatically inserted arithmetic error checks. For this purpose, we perform a set of timing measurements on the applications used in Section 5.1.1. For each run, we measured the time that was required to complete a series of tasks for each of IntFlow's modes of operation, and then normalized the running time with respect to the runtime of the native binary. Reported results are mean values over ten repetitions of each experiment, while the reported confidence intervals correspond to

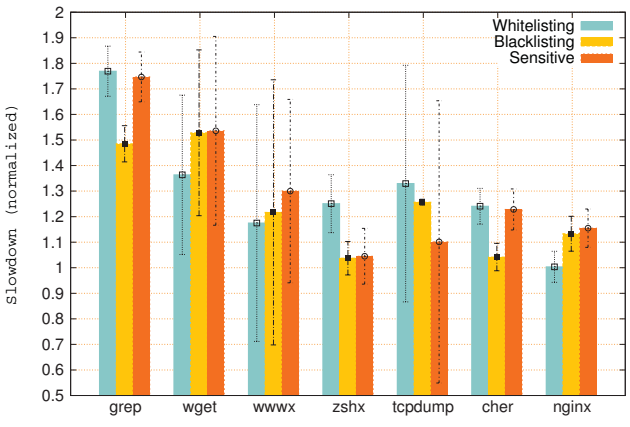

Figure 4: Runtime overhead for the applications of Section 5.1.1 (normalized over native execution).

95%. We ran all binaries natively, and measured user time with the `time` utility.

For `grep`, we search for strings that match a complex regular expression in a 1.2GB file. For `wget`, we download a 1GB file over a 1Gbit/s link from a remote server. For `zshx`, we execute a series of shell commands, and for `tcpdump` we examine packets from a 5.8GB pcap file. For the web servers, Cherokee was configured for IPv4 only, while for Nginx all configurations options were left to their default setting. We measured performance using Apache's `ab` benchmarking utility and static HTML files.

Figure 4 shows the results of our evaluation. IntFlow incurs high overhead in applications that involve a large number of integer operations, such as `grep`.[4] We also notice high performance deviation for `wget`, `wwwx`, and `tcpdump`, as they are I/O bound. Although in such applications the overhead is rather prohibitive, and cancels out the benefits of using a static mechanism, in other cases, such as for the `cher` and `nginx` servers, the overhead is within an acceptable 20%. Thus, it could be the case that IntFlow might be used as a runtime defense for certain types of applications, i.e., I/O-bound. As each of IntFlow's modes of operation targets different flows and can be fine-tuned by developers, customization can result in different overheads, as different flows dominate the execution of different applications. This is the reason we observe different slowdowns per mode: depending on whether the dominating flows involve sensitive calls (e.g., web servers), the sensitive mode will be slower or faster than the other two modes, and so on.

6. RELATED WORK

During the past years, as the protection mechanisms against buffer overflows became more mature, great focus was placed upon efficiently dealing with integer overflows. This section summarizes the main characteristics of the several approaches that have been followed so far for addressing integer overflows and outlines the connection between the current work and existing research on the field.

[4]Based on our experience with the SPEC CPU2000 benchmarks, the overhead on benchmarks with very frequent integer operations, such as `gzip`, is in the range of ∼x10, prohibiting IntFlow from being used as a generic runtime detection mechanism for such applications.

6.1 Static Analysis

Static analysis tools provide good coverage but generally suffer from a high rate of false positives. IntPatch [25] is built on top of LLVM [14] and detects vulnerabilities utilizing the type inference of LLVM IR. Similarly to our tool, IntPatch uses forward & backward analysis to classify sources and sinks as sensitive or benign. Each sensitive variable is located through slicing. If a variable involved in an arithmetic operation has an untrusted source and the respective sink may overflow, IntPatch will insert a check statement after that vulnerable arithmetic operation. If an overflow result is used for sensitive actions such as memory allocations, IntPatch considers it a real vulnerability. Contrary to the current work though, IntPatch does not deal with all types of integer overflows and also does not address programmer-inserted sanitization routines.

KINT [23] is a static tool that generates constraints representing the conditions under which an integer overflow may occur. It operates on LLVM IR and defines untrusted sources and sensitive sinks via user annotations. KINT avoids path explosion by performing constraint solving at the function level and by statically feeding the generated constraints into a solver. After this stage, a single path constraint for all integer operations is generated. Unfortunately, despite the optimization induced by the aforesaid technique, the tool's false positives remain high and there is a need for flagging false positives with manual annotations in order to suppress them. Moreover, contrary to this work, KINT attempts to denote all integer errors in a program and does not make a clean distinction between classic errors and errors that constitute vulnerabilities.

SIFT [16] uses static analysis to generate input filters against integer overflows. If an input passes through such filter, it is guaranteed not to generate an overflow. Initially, the tool creates a set of critical expressions from each memory allocation and block copy site. These expressions contain information on the size of blocks being copied or allocated, and are propagated backwards against the control flow, generating a symbolic condition that captures all the points involved with the evaluation of each expression. The free variables in the generated symbolic conditions represent the values of the input fields and are compared against the tool's input filters. A significant difference of this paper in comparison to SIFT is that the latter nullifies overflow errors but does not detect them nor examines whether they could be exploitable.

IntScope [21] decompiles binary programs into IR and then checks lazily for harmful integer overflow points. To deal with false positives, IntScope relies on a dynamic vulnerability test case generation tool to generate test cases which are likely to cause integer overflows. If no test case generates such error, the respective code fragment is flagged appropriately. This approach varies significantly from the one used in our tool as it relies on the produced test cases to reveal a true positive: if a test case does not generate an overflow, that does not guarantee that no overflow occurs. In addition, IntScope regards all errors as generic, without focusing particularly on errors leading to vulnerabilities.

Finally, RICH [7] is a compiler extension which enables programs to monitor their execution and detect potential attacks exploiting integer vulnerabilities. Although RICH is very lightweight, it does not handle cases of pointer aliasing and produces false positives in cases where developers inten-

tionally abuse the undefined behavior of C/C++ standards, both of which are basic components of IntFlow's design.

6.2 Dynamic & Symbolic Execution

Dynamic Execution tools use runtime checks to prevent unwanted program behavior. IOC [10] uses dynamic analysis to detect the occurrence of overflows in C/C++ programs. The tool performs a compiler-time transformation to add inline numerical error checks and then relies on a runtime handler to prevent any unwanted behavior. The instrumentation transformations operate on the Abstract Syntax Tree (AST) in the Clang front-end, after the parsing, type-checking and implicit type conversion stages. IOC checks for overflows both in shifting and arithmetic operations and makes a clear distinction between well-defined and undefined behaviors, that is, it does not consider all overflows malicious. Our tool adopts this perspective and complements IOC in the sense that it addresses the issue of high false positives by integrating static instruction flow tracking to the performed analysis.

Symbolic Execution tools provide low false positives but can't easily achieve full coverage. They usually use dynamic test generation [11] to detect violations. SmartFuzz [17] generates a set of constraints defining unwanted behavior and determines whether some input could trigger an unwanted execution path. This tool does not require source code and makes use of the Valgrind framework for its symbolic execution and scoring. Coverage and bug-seeking queries are explored in a generational search, whilst queries from the symbolic traces are solved generating a set of new test cases. Thus, a single symbolic execution feeds the constraint solver with many queries, which themselves generate new test cases etc. KLEE [8] uses symbolic execution to automatically generate tests that achieve high coverage for large-scale programs but it is not focused on integer errors, thus it does not achieve as good results against integer overflows as other tools that targeted towards integer operations.

7. DISCUSSION

7.1 Static information flow tracking

A core component of IntFlow is `llvm-deps` [18], which, as an implementation of static information flow tracking, is expected to provide good source code coverage with low runtime overhead. However, we should note that `llvm-deps` suffers from inherent inaccuracy issues, largely due to the limitations of its points-to analysis [15] and due to its data flow analysis mechanism. These limitations are amplified when one wishes to extend the scope of the technique by performing inter-procedural analysis. Fortunately, as our experience revealed, sources and sinks typically reside within a single function. This can be viewed as an instance of the classic trade-off between accuracy and performance: for cases where accuracy has the maximum priority, we may choose to incorporate dynamic IFT [13] and attempt to reduce any increased runtime overhead using techniques that combine static and dynamic analysis [12, 9].

7.2 IntFlow for Runtime Detection

While the primary use case of IntFlow is to help users analyze existing code during the development phase by reducing the amount of false positives reported by previous tools, another use case is to deploy it as a runtime defense against zero-day vulnerabilities. For this purpose, the two main issues that must be addressed are i) the increased runtime overhead due to the inserted checks, and ii) any remaining false positives after IntFlow's analysis.

The main source of runtime overhead, as shown in Section 5.2, can be attributed to IOC's checks, as it replaces each arithmetic operation with at least three basic blocks to perform the checking operation. Given the significance of the problem, there have been many previous proposals for implementing fast and efficient checking operations [22], which IntFlow could adopt to improve performance.

As shown in Section 5, IntFlow was able to identify a large portion of the developer-intended violations in the programs under examination, but still missed some cases. In order to provide as broad coverage of false positives as possible, IntFlow supports manual labeling of false positives. Developers can dedicate a separate off-line phase to apply IntFlow using a trusted input set over their application, and pinpoint the locations in which IntFlow falsely flags a benign operation as malicious. A more suitable solution for this use case would be the incorporation of dynamic IFT, which though would impose high runtime overhead, as already discussed.

7.3 Quality of the Produced Reports

Another advantage of IntFlow's design is the fact that its three different modes of operation offer an estimation of how critical a particular bug is. Errors reported by the sensitive mode have the highest risk, as they involve sensitive operations and are more likely to be exploitable. Likewise, in black-listing mode, IntFlow examines flows originating from untrusted locations and thus the produced reports are of moderate priority. Finally, the whitelisting mode is likely to generate the largest amount of warnings. Thus, if developers wish to examine as few locations as possible, e.g., due to limited available time for performing code auditing, they can first examine the reports generated by IntFlow in the sensitive mode, and if time permits then use the blacklisting mode, and so on.

Throughout our evaluation, we noticed that many of the reports generated by IntFlow follow a particular pattern, mainly due to code reuse from the side of developers. We believe that using simple pattern matching and lexical analysis of the source code, in combination with the reports of IntFlow, could further increase the accuracy with which IntFlow classifies errors as malicious or not—the more the occurrences of a particular error, the more likely for this error to be developer-intended. We will explore this approach as part of our future work.

8. CONCLUSION

We have presented IntFlow, a tool that identifies a broad range of arithmetic errors and differentiates between critical errors and developer-intended constructs that rely on undefined behavior, which do not constitute potential vulnerabilities. IntFlow uses static information flow tracking to associate flows of interest with erroneous statements, and greatly reduces false positives without removing checks that would prevent the detection of critical errors. The results of our evaluation demonstrate the effectiveness of IntFlow in distinguishing between the two types of errors, allowing developers and security analysts to detect and fix critical errors in an efficient manner, without the need to sift through numerous non-critical developer-intended violations. The

significant reduction in false positives that IntFlow achieves over IOC, which has been integrated into Clang since version 3.3, demonstrates the need for effective and accurate automated arithmetic error detection.

Acknowledgments

This work was supported by DARPA and the US Air Force through contracts DARPA-FA8750-10-2-0253 and AFRL-FA8650-10-C-7024, respectively, with additional support from Intel Corp. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, DARPA, the Air Force, or Intel.

9. REFERENCES

[1] Clang C language family frontend for LLVM. http://clang.llvm.org/.

[2] CVE - CVE-2006-3824. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3824.

[3] CWE - 2011 CWE/SANS top 25 most dangerous software errors. http://cwe.mitre.org/top25/.

[4] CWE - Common Weakness Enumeration. http://cwe.mitre.org/.

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[6] A. Benameur, N. S. Evans, and M. C. Elder. Minestrone: Testing the soup. In *Proceedings of the 6th Workshop on Cyber Security Experimentation and Test (CSET)*, 2013.

[7] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song. RICH: Automatically Protecting Against Integer-Based Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2007.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[9] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS)*, 2008.

[10] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[12] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.

[13] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2012.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization (CGO)*, 2004.

[15] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[16] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard. Sound input filter generation for integer overflow errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014.

[17] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[18] S. Moore. thinkmoore/llvm-deps. https://github.com/thinkmoore/llvm-deps. (Visited on 06/07/2014).

[19] E. Revfy. Inside the size overflow plugin. http://forums.grsecurity.net/viewtopic.php?f=7&t=3043.

[20] T. Wang, C. Song, and W. Lee. Diagnosis and emergency patch generation for integer overflow exploits. In *Proceedings of the 11th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2014.

[21] T. Wang, T. Wei, Z. Lin, and W. Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2009.

[22] X. Wang. Fast integer overflow detection. http://kqueue.org/blog/2012/03/16/fast-integer-overflow-detection/.

[23] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with KINT. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[24] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[25] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou. Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *Proceedings of the 15th European Symposium on Research in Computer Security (ESORICS)*, 2010.