Improving the Accuracy of Integer Signedness Error Detection Using Data Flow Analysis

Hao Sun, Chao Su, Yue Wang, Qingkai Zeng
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China
Email: {shqking, suchao1991}@gmail.com, wxywang89@163.com, zqk@nju.edu.cn

***Abstract*—Integer signedness error can be exploited by attackers to cause severe damages to computer systems. Despite of the significant advances in automating the detection of integer signedness errors, accurately differentiating exploitable and harmful signedness errors from unharmful ones still remains an open problem. In this paper, we present the design and implementation of *SignFlow*, an instrumentation-based integer signedness error detector to reduce the reports for unharmful signedness errors without sacrificing the completeness (i.e. no false negatives). SignFlow utilizes static data flow analysis to identify *unharmful integer signedness conversions* from the view of where the operands originate and whether the data after conversions can propagate to security-related operations, and then inserts security checks for the remaining conversions so as to accomplish runtime protection. We evaluated SignFlow on 7 real-world harmful integer signedness bugs, SPECint 2006 benchmarks together with 5 real-world applications. Experimental results show that SignFlow successfully detected all harmful integer signedness bugs and achieved a reduction of 41% in false positives over IntFlow, the state-of-the-art signedness error detector.**

***Keywords*—integer signedness error, data flow analysis, instrumentation, sanitization**

I. Introduction

The C/C++ programming language implements the *signedness* of integer types, including signed and unsigned. An *integer signedness error* occurs when a signed integer is interpreted as unsigned, or vice-versa. In two-complement representation, such conversions cause the sign bit to be interpreted as the most significant bit (MSB) or conversely, hence -1 and $2^{32}-1$ are misinterpreted to each other on 32-bit machines. Because such misinterpretation cannot overwrite memory directly, adversaries usually leverage *security-related operations* (e.g., bound checks, memory allocations and loops) to exploit integer signedness errors indirectly. For instance, Listing 1 shows a typical signedness error in Mumble [1]. Lines 4 to 5 are the patch for this bug. In the original buggy code, variable `decodedSamples` is used to denote the amount of decoded samples, and it would be assigned with small negative values when `opus_decode_float()` encounters an error. Note that such small negative integers indicate the error condition. Then `decodedSamples` is converted to unsigned integer, i.e. `inlen`, which becomes close to UINT_MAX. Later `inlen` is used as the buffer length in `speex_resample_process_float()` and buffer overflow occurs due to such inadvertently large buffer length.

(DOI reference number: 10.18293/SEKE2015-123)

Listing 1. Patched code for CVE-2014-0045 in Mumble

```
1   int decodedSamples = opus_decode_float(opusState
    ,NULL, ...);
2   ...
3   spx_uint32_t inlen = decodedSamples;

4+  if(inlen > 0x7fffffff)
5+      return error;

6   if (srs && bLastAlive)
7       speex_resampler_process_float(srs, 0,
            fResamplerBuffer, &inlen, ...);
```

During the past years, researchers have developed various techniques to address this problem. A classic approach is to insert *security checks* around integer signedness conversions to catch signedness errors at runtime. Instrumented programs can react to a signedness error by logging the event or terminating the execution. Many existing techniques, such as RICH [2], IOC [3], AIR [4] and RA [5], consider *all* integer signedness conversions in a subject program as potential signedness error sites and instrument all of them. The *instrument-all* techniques guarantee to detect all the runtime signedness errors; however, this safety has a price: they might report unharmful signedness errors as critical ones in real-world scenarios, i.e. producing false positives.

IntFlow [6] aims to eliminate the instrumentation for some integer signedness conversions if they originate from trusted source. The intuition is that such trusted integer signedness conversion cannot be controlled by attackers even if signedness error occurs. Hence, IntFlow could reduce a number of false positives produced by instrument-all techniques. However, IntFlow only considers where a signedness error originates, but doesn't track how it would be used afterward. IntFlow would produce false positives inevitably in the following cases: 1) the integer data after conversion is unused afterward, or propagates to uncritical program locations; 2) experienced developers often anticipate the possibility of signedness errors such that they add *sanitization routines* after these sites; 3) programmers use signedness errors intentionally for performance optimization or code compactness in many situations.

To further improve the accuracy of integer signedness error detection, we develop a novel runtime signedness error detector, named *SignFlow*, which features the capacities of detecting all harmful signedness errors (*complete*) and bypassing as many unharmful ones as possible (*precise*) with

acceptable performance loss (*practical*). In SignFlow, we not only consider where integer signedness conversions originate (as IntFlow does), but further track how they would be used afterward. The main intuition behind our approach is that integer signedness errors become unharmful if they are unused afterward, or used at uncritical program locations such as *printf()*, or get sanitized before flowing into security-related operations. As such, SignFlow could avoid the reports for more unharmful signedness errors, compared to IntFlow.

Our contributions are highlighted as follows.

- We define unharmful integer signedness errors from the perspective of where they originate and how they are used afterward;
- We propose and implement a novel instrumentation-based integer signedness error detector, SignFlow, as an extension of the GCC compiler [7], aiming to improve the accuracy of integer signedness error detection. SignFlow first exploits static data flow analysis to identify unharmful integer signedness conversions and then inserts security checks for the remaining potentially harmful sites;
- To demonstrate the effectiveness, we applied SignFlow to 7 harmful signedness bugs, SPECint 2006 benchmarks and 5 real-world applications. Experimental results show that SignFlow detected all 7 harmful errors and bypassed about 41% (46 out of 111) unharmful ones that the state-of-the-art integer signedness error detector, i.e. IntFlow, produced. Besides, as about 7.73% more integer signedness conversions were identified as unharmful at static analysis phase, SignFlow reduced the runtime overhead by 49.62% over IntFlow.

II. Approach

One key consensus in existing techniques is that an integer signedness error becomes harmful (or critical) if it satisfies both the following two conditions: *T1: the right-hand operand in signedness conversion originates from un-trusted source, i.e. controlled by users; T2: the misinterpreted data may propagate to security-related operations, i.e. sinks.*

T1 allows attackers to control this signedness error and determine the misinterpretation according to their own intentions, and T2 provides attackers an interface to exploit this signedness error so as to conduct malicious operations. Conversely speaking, we argue that *an integer signedness error can be treated as unharmful if it violates either T1 or T2*. In the following, we first present data flow patterns of unharmful signedness errors from the perspective of violating T1 or T2, and then introduce our approach briefly.

A. Unharmful Integer Signedess Errors

Definition 1: An integer signedness error is unharmful if it satisfies one of the following three conditions:

1. *the right-hand operand in signedness conversion is constant or originates from constant values;*
2. *the integer data after conversion is unused afterward, or propagates to uncritical program locations;*
3. *the integer data after conversion gets sanitized before it propagates to security-related operations.*

TABLE I. Post-condition test for signedness conversions

Signedness Conversion	Post-condition Test
int x; unsigned int y; x = (int) y;	x $<$ 0
unsigned int x; int y; x = (unsigned int) y;	x $>$ 0x7fffffff

TABLE II. Sanitization operator for signedness errors

Op Type	Basic Form	Influence
bitwise-and	res = a & b, and b $<$ 0x80000000	Erased
modulo	res = a % b, and b $<$ 0x80000000	Erased
left-shift	res = a $\ll$ b	Replaced
right-shift	res = a $\gg$ b	Replaced
bitwise-xor	res = a $\oplus$ b	Erased
bitwise-not	res = $\sim$ a	Erased

1) Trusted Source: The integer signedness error under condition 1.1 means that, the concrete value of source operand can be determined at compile-time. Hence, attackers cannot control this signedness error via providing crafted inputs. In other words, attackers have no chance to exploit such signedness errors. We call this data flow pattern of unharmful signedness errors as P_{cst} for short.

As shown below we adopt the code snippet from [6] to present an example, in which developers intentionally rely on signedness error mainly for performance reasons. It casts -1 into unsigned type to obtain the largest number that unsigned type can represent. Since the source operand is constant, satisfying P_{cst}, this signedness error is unharmful.

```
UINT_MAX = (unsigned int) -1;
```

2) Uncritical Program Location: The integer signedness error under condition 1.2 denotes that, the integer data after conversion is relatively not as critical as other integers. This shuts off the interface for attackers to jeopardize the whole program even if attackers can control this misinterpreted data. We call this data flow pattern of unharmful signedness errors as P_{uncrit} for short. Note that library function calls such as *printf()* and *fprintf()*, are typical uncritical locations.

3) Sanitization: According to our Definition 1, the paths of misinterpreted data to sinks can be cut off or stopped if there exist so-called sanitizations. Here, we give the following two kinds of sanitizations.

Experienced developers often anticipate the possibility of integer signedness errors such that they add *sanitization routines* after potential signedness error sites to prevent misinterpreted data from affecting further program execution. As shown in Table I, post-condition test [3][8] is the widely used sanitization routines for signedness errors. The patch for CVE-2014-0045, i.e. lines 4 to 5 in Listing 1, gives an example. Once signedness error occurs at line 3, the security check at line 4 would catch this misinterpretation, cutting off the flow to sink (i.e. line 7). As post-condition test is of fixed patterns, they can be identified statically. Furthermore, post-condition test is sound enough to catch signedness errors. Hence, we adopt post-condition test as one kind of sanitizations for signedness errors, and we call such data flow pattern as P_{post} for short.

Besides, many operators can remove or clean up the sign bit for signed type or the MSB for unsigned type. In these

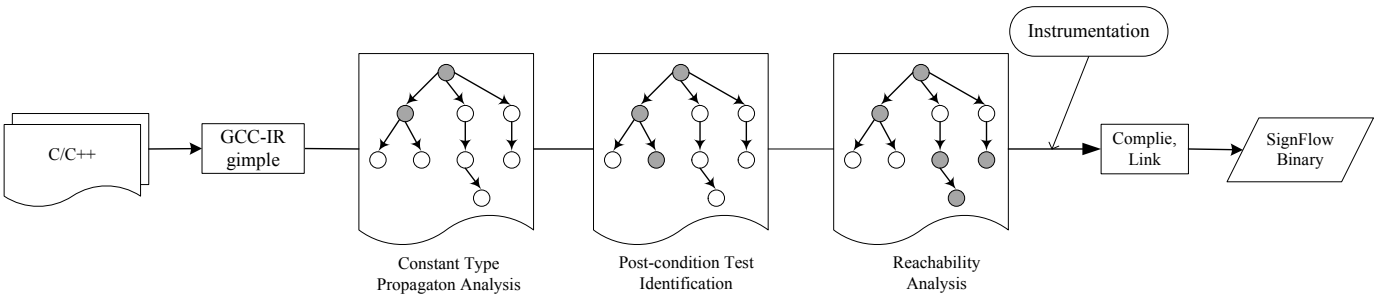

Fig. 1. Overall architecture of SignFlow

cases, the misinterpreted data gets sanitized since the bit, that signedness error matters, is *replaced by another benign bit or erased* after such operators. We call these operators *SantzOp* for short. Table II shows a serial of SantzOp, including the type, the basic form and the influence on the sign bit/MSB. Note that *a* denotes the misinterpreted integer. Here, we argue that *after going through SantzOp, misinterpreted data is less likely to be used in exploitation attempts by attackers, as the crucial bit, i.e. the sign bit/MSB, is replaced with another benign bit or erased by SantzOp*. We call such data flow pattern as P_{op} for short.

Take the following code snippet as an example. It is from *pp.c*, 400.perlbench in SPECint 2006. Implicit signedness conversion occurs at line 2332; however, the bitwise-and operators at line 2335 can erase the sign bit. Hence, such signedness error is unharmful. In fact the SantzOp at line 2335 is used here to achieve the effect of selecting specific bits of `flags`.

```
    struct STRUCT_SV{
       void * sv_any;
       U32 sv_refcnt;
       U32 sv_flags;
    }
    #define SVf_IOK 0x00100000
    #define SVp_IOK 0x01000000
    #define SVp_NOK 0x02000000

2332 int flags = sv->sv_flags;
2333 ...
2335 if((flags & SVf_IOK) ||
        ((flags & (SVp_IOK | SVp_NOK)) == SVp_IOK)){
2336 ...
```

B. Approach Overview

Based on the discussions above, we propose a novel approach to improve the integer signedness error detection using static data flow analysis. We at first identify *unharmful integer signedness conversions* based on the data flow characteristics, and then exclude them from further instrumentations. Specifically, we first assume that all integer signedness conversions in program are un-trusted and each of them should be instrumented with security check to guarantee the runtime safety. Then we conduct static data flow analysis to mark unharmful integer conversions in three aspects. 1) One taint-like analysis is employed to propagate the tag of '*constant*' starting from constant values and safe library function calls (e.g., *uname(), gettimeofday()*). An integer conversion is marked as unharmful if the source operand is tagged with '*constant*'; 2) An integer signedness conversion is also marked as unharmful if it is protected by post-condition test; 3) Another data flow tracking is deployed to compute an integer signedness conversion's reachability to security-related operations. Note that this process is accomplished through determining whether it will encounter uncritical program locations or SantzOp before sinks on each path of this signedness conversion. If so, we mark this conversion as unharmful. At last, security checks are inserted around the integer signedness conversions, which are not identified as unharmful, so as to gain runtime protection, as existing instrumentation-based detectors [2][3][8][6] do.

III. Design and Implementation

Figure 1 illustrates the overall architecture of SignFlow. It performs static data flow analysis on the GCC intermediate representation—*gimple*—to identify unharmful integer signedness conversions, and then instruments security checks for the remainder. At last, the inputs get further compilation and linking into binary. Specifically, SignFlow consists of four main components: 1) *constant type propagation analysis* is similar to taint-like analysis, aiming to identify all the integer signedness conversions under P_{cst}; 2) *post-condition test identification* checks whether an integer signedness conversion is followed by post-condition test; 3) *reachability analysis* aims to compute the reachability of an integer signedness conversion to security-related operations by determining whether there exists uncritical program location or SantOp along each path; 4) *instrumentation* part inserts security checks for integer signedness conversions except those, which have been identified as unharmful by previous analyses.

Similar to IntFlow [6], our constant type propagation analysis is implemented via a taint-like analysis, i.e. setting the trusted source as '*constant*', propagating this tag along the data flows, and marking the integer signedness conversion as unharmful if the source operand is '*constant*'. Moreover, the process of post-condition test identification can also be easily implemented since the post-condition test particularly for signedness errors is of fixed patterns, as shown in Table I. Hence, in the following we will discuss more about reachability analysis and instrumentation.

A. Reachability Analysis

In this section, we analyze the data flows out from an integer signedness conversion to decide whether this conversion satisfies condition 1.2 or condition 1.3 (in Definition 1). If so, this integer signedness conversion is treated as unharmful.

Whether an integer signedness conversion would be used at specific locations can be induced as a reachability problem. We

Algorithm 1 Reachability Analysis.

Input: Signedness conversion SC.
Output: $Sink, Unharm$;

```
1: Sink, Unharm are initialized with 0;
2: if SC is unused afterward then
3:    set Unharm;
4:    return ;
5: end if
6:
7: for each path of SC do
8:    for each node of path do
9:       if node is uncritical site or SantzOp then
10:         set Unharm;
11:      else if node is security-related operations then
12:         set Sink; break;
13:      else if node is assignment, unary and binary op then
14:         continue;
15:      else
16:         set Sink; break;
17:      end if
18:   end for
19:   if Sink then
20:      clear Unharm; break;
21:   end if
22: end for
```

compute the reachability of an integer signedness conversion to uncritical program locations and SantzOp to decide whether it satisfies condition 1.2, condition 1.3 or both. As shown in Algorithm 1, an integer signedness conversion can be treated as unharmful if 1) it is unused afterward (lines 2 to 5), 2) it ends up with uncritical program locations, or 3) it encounters SantzOp before security-related operations for each path. Note that security-related operations include *if*-statement, *while*-statement, array indexing and sensitive library routines such as *malloc()* and *memcpy()*. Lines 15 to 16 mean that our analysis is conservative for harmful signedness errors, i.e. an integer signedness conversion cannot be marked as unharmful if we are not certain of whether there will be sinks along some path.

B. Instrumentation

Through constant type propagation analysis, post-condition test identification and reachability analysis, a number of integer signedness conversions have been identified as unharmful. As the last step of SignFlow, security checks are inserted at the remaining signedness conversions for further runtime protection. We leverage pre-condition test [3][8], with the feature of testing whether misinterpretation will occur without actually performing the conversion. For instance, casting signed integer to unsigned type, i.e. `unsigned int a; signed b; a = (unsigned) b`, will cause signedness error if and only if the following expression is true: `(b < 0)`.

C. Implementation Details

We have implemented SignFlow for C/C++ programs based on GCC-4.5.0 [7]. Specifically, SignFlow is an optimization pass written in $\sim 4,000$ lines of C on *gimple*. *gimple* provides many interfaces for users to analyze the abstract syntax tree (AST), control flow graph (CFG) and call graph. Our constant type propagation analysis is accomplished by binding one tag '*constant*' with each variable node and updating this tag with the traversal of each statement in AST. Our reachability analysis utilizes the propagation analysis engine in GCC, which is widely used by optimizations such as the copy propagation analysis and value range propagation analysis, to traverse each potential path of an integer signedness conversion. At last, security checks are inserted for those integer signedness conversions which are not marked as unharmful. The runtime handler is linked into the compilers' output and takes actions when signedness errors are caught. It is worth noting that SignFlow works at *gimple* mainly because all implicit signedness castings are presented explicitly at this stage.

TABLE III. 7 HARMFUL INTEGER SIGNEDNESS BUGS IN REAL WORLD

CVE	Programs	Version	Sign Conv.	Sink
2008-1803	Rdesktop	1.5.0	u→s	bound check
2009-3743	GhostScript	8.70	s→u	memmove
2011-1471	PHP	5.3.6*	s→u	bound check
2012-3368	Dtach	0.8	s→u	bound check
2013-4927	Wireshark	1.10.0	u→s	loop
2013-6489	Pidgin	2.10.11*	s→u	malloc
2014-0045	Mumble	1.2.4	s→u	malloc

IV. Evaluation

In this section, we present the results of our experimental evaluation using our prototype implementation of SignFlow, and compare the results with instrument-all techniques and IntFlow. All experiments were performed on an Intel Dual Core 2.4 GHz machine with 4GB memory. The OS is Linux-3.5.0. GCC was ran under -O0 optimization level.

A. Detecting Harmful Integer Signedness Bugs

In order to evaluate the effectiveness of SignFlow in detecting and preventing harmful integer signedness bugs, we select 7 real-world signedness bugs published by CVE [9] as our test subjects, as shown in Table III[1]. Columns 1 to 3 describe the CVE number, vulnerable software and version. Column 4 refers to signedness error site. That is the specific conversion, where signedness error occurs. *u→s* denotes casting unsigned integer to signed integer and *s→u* denotes casting signed integer to unsigned. Column 5 describes the security-related operation where the misinterpreted data is exploited.

The evaluation result is that *SignFlow successfully instrumented all the signedness error sites, i.e. SignFlow didn't mark them as unharmful at the static data flow analysis phase*. To evaluate the runtime protection of SignFlow, we face the challenge that the corresponding signedness-error-inducing inputs are not available. We turn to extract the vulnerable conversion sites and their propagation paths from subject programs, and then execute the extracts with the self-designed signedness-error-inducing inputs. The result is *SignFlow reported warnings for all harmful signedness bugs*.

[1]Detailed information about these 7 bugs can be referred at CVE website [9] via the corresponding CVE number. For CVE-2011-1471, the vulnerable version should be 5.3.5 and below. However, these versions of PHP cannot be compiled successfully under our experimental environment due to certain reason. Therefore, we choose 5.3.6 version instead and remove the patch manually. So it is with CVE-2013-6489.

TABLE IV. INTEGER SIGNEDNESS ERRORS REPORTED BY ALL, SIGNFLOW$_{cst}$ AND SIGNFLOW

	#A	#S$_c$	#S	data flow pattern for each excluded error			
				P_{cst}	P_{uncrit}	P_{post}	P_{op}
400.perlbench	48	48	24			18	6
401.bzip2	15	14	7	1		4	4
445.gobmk	17	15	12	2		1	2
458.sjeng	3	3	0				3
462.libquantum	4	4	3		1		
464.h264ref	10	8	7	2			1
483.xalancbmk	9	7	6	2			1
Gzip	1	1	0				1
Dillo	5	5	4			1	
SWFTools	6	6	2			1	3
Total	118	111	65	7	1	25	21

TABLE V. CHECK DENSITY OF SIGNFLOW

	#SC	#U	data flow pattern for each conv. in #U			
			P_{cst}	P_{uncrit}	P_{post}	P_{op}
400.perlbench	5955	2173	287	20	199	1684
401.bzip2	1105	110	67	4	4	35
429.mcf	72	8	8	0	0	0
445.gobmk	2374	706	639	9	1	57
456.hmmer	5589	2991	2927	8	1	53
458.esjeng	332	116	94	5	0	19
462.libquantum	312	185	148	7	0	29
464.h264ref	8274	2907	2808	0	5	84
471.omnetpp	1715	1151	1144	2	1	2
473.astar	372	16	16	0	0	0
483.xalancbmk	6163	681	387	1	28	236
Total	32263	10998	8525	56	239	2199

Hence we gain confidence that SignFlow is suitable as a detection tool for real-world applications.

B. Reduction of False Positives

Reducing the number of false positives is the major goal of SignFlow, and this section quantifies how good SignFlow is in omitting unharmful signedness errors from the reported results by instrument-all techniques and IntFlow respectively. Here we implemented a prototype, named *ALL*, for instrument-all techniques by disabling our static data flow analysis, and a prototype, named *SignFlow$_{cst}$*, for IntFlow by only validating the constant type propagation analysis (i.e. disabling the post-condition test identification and reachability analysis).

We use SPECint 2006 benchmarks and 5 real-world applications as our testbed. We ran SPECint 2006 with the '*ref*' input set. For SWFTools-0.9.1, we used the pdf2swf utility with the call-for-paper of SEKE'2015 as input; for Dillo-3.0.4.1, we visited its homepage and downloaded the source code; for Pidgin-2.10.11, we registered a new account, logged in and out; for Gzip-1.4, we compressed the archive gcc-4.5.0.tar; and for *wget*-1.6, we downloaded a 70MB file from a remote server. As each program is run with benign inputs, the reported integer signedness errors are all unharmful, i.e. false positives. We applied ALL, SignFlow$_{cst}$ and SignFlow respectively on the testbed and calculated the reduction of false positives by SignFlow over ALL and SignFlow$_{cst}$. In addition, for each unharmful signedness error that SignFlow bypassed, we manually examined our static analysis results to find out which data flow pattern (as discussed in Section II) this signedness error belongs to. We report our findings in Table IV. For brevity, we only displayed the result of programs which have signedness errors.

Columns 2 to 4 show the number of integer signedness errors reported by ALL, SignFlow$_{cst}$ and SignFlow respectively. Overall, SignFlow was able to suppress about 45% (53 out of 118) integer signedness errors reported by ALL, and this reduction is about 41% (46 out of 111) over SignFlow$_{cst}$. These were achieved due to the data flow characteristics of unharmful signedness errors used by SignFlow.

Columns 5 to 8 show the distributions of different data flow patterns, to which each excluded unharmful signedness errors belong. From the result we can observe that 7 out of 53 unharmful signedness errors identified by SignFlow are under P_{cst} while the other 47 are under P_{uncrit}, P_{post} or P_{op}. Hence as SignFlow considers whether the data after conversions could propagate to security-related operations or not, it can identify more unharmful signedness errors than SignFlow$_{cst}$, i.e. SignFlow improved the accuracy of detecting integer signedness errors a lot over IntFlow. Besides, the distributions of data flow patterns vary over different programs. This is mainly because the effectiveness in the reduction of reporting unharmful signedness errors is highly dependent on the nature of each program as well as on the level of the execution's source coverage.

C. Performance

Check density refers to *the ratio of the number of instrumented integer signedness conversions over all*. Table V shows the experimental results of applying SignFlow to SPECint 2006 benchmarks. Column 2 shows the number of integer signedness conversions in *gimple*, and Column 3 indicates the number of integer signedness conversions identified as *unharmful* by SignFlow. In total, about 34% ($10,998$ out of $32,263$) integer signedness conversions are marked as unharmful and excluded from instrumentation by SignFlow. In another word, the rest 66% signedness conversions are instrumented for runtime protection, i.e. the check density of SignFlow.

Columns 4 to 7 present the distributions of different data flow patterns, to which each unharmful integer signedness conversion belongs. Note that as some signedness conversion can be under several different data flow patterns, the value of Column 3 might be less than the sum of Columns 4 to 7. Form the result, we can see P_{cst} is the main type of unharmful integer signedness conversions. About 26.42% ($8,528$ out of $32,263$) belongs to this pattern. It also denotes the number of signedness conversions excluded by IntFlow. Since SignFlow considers P_{uncrit}, P_{post} and P_{op} in addition, $2,494$ more signedness conversions are identified as unharmful and excluded from instrumentation, compared to IntFlow.

Then We executed the instrumented program with the '*ref*' input sets to test the overhead imposed by SignFlow. We ran each program 5 times and took the average value as the execution time. Note that the runtime handler is set as `nop` instruction when an integer signedness error is caught. We also have tested ALL and SignFlow$_{cst}$ in the same way so as to illustrate the reduction of performance overhead that SignFlow gained. Compared to original benchmarks, the runtime overhead of ALL, SignFlow$_{cst}$ and SignFlow are 6.19%, 5.35% and 2.70% respectively, which indicates that the performance loss of instrumentation-based techniques is quite low. On the other hand, SignFlow reduced the runtime overhead by

56.41% over ALL and 49.62% over SignFlow$_{cst}$ respectively. Such reduction is achieved by our static data flow analysis, especially for that about 26.42% all signedness conversions are excluded from instrumentation as they originate from trusted source (i.e. under P_{cst}), and another 7.73% are excluded in the view of how they are used afterward (i.e. under P_{uncrit}, P_{post} or P_{op}).

D. Limitations

We propose to improve the accuracy of integer signedness error detection using the data flow characteristics. As an initial step along this direction, SignFlow has a number of limitations. *First*, SignFlow in current implementation identifies bitwise-and and modulo operation as SantzOp by only checking whether `b` is a constant less than `0x80000000`. Precise integer range analysis is in need to identify more SantzOp. *Second*, SignFlow might fail to identify some data flow patterns due to the common challenges, such as pointer analysis and field-sensitiveness problem, faced by the static data flow analysis. *Third*, the scope of our static data flow analysis is limited to one single object file, as it is implemented as a compile-time optimization pass and not as a link-time optimization pass. This would also affect the detection accuracy of SignFlow. We are working on addressing these problems.

V. Related Work

During the past years, great focus was placed upon dealing with integer signedness errors. Safe library functions are used to wrap integer signedness conversions by adding check code, such as IntegerLib [10] and Ranged Integer [11]. SmartFuzz [12] utilizes symbolic execution to generate test cases to invoke integer errors. The key challenges are to construct test cases of high code coverage and to deal with the path explosion problem when applied to large scale software.

Instrument-all techniques, such as RICH [2], IOC [3], RA [5] and AIR [4] inserts security checks for all integer signedness conversions for runtime protection. RICH provides formal specifications for integer semantics in C, and applies sub-type theory from type safe languages into C language. IOC has been integrated into LLVM compiler. The main drawback of instrument-all techniques is that they produce many false positives as they instrument security checks blindly. As shown in Section IV-B, SignFlow reduced about 45% false positives produced by ALL, a prototype of instrument-all technique.

IntFlow [6] aims to improve the accuracy of integer arithmetic errors. The difference from SignFlow lies in: 1) IntFlow focuses on excluding the false positives for *developer-intended undefined behavior*, including not only signedness errors, but also signed integer overflows, oversized shift and division by zero error; 2) IntFlow identifies unharmful undefined behaviors by checking whether they originate from trusted source, i.e. IntFlow only consider P_{cst}. In total, IntFlow achieves a reduction of 89% in false positives for all these undefined behavior; however it doesnt provide detailed statistics on its effectiveness particularly for integer signedness error. Therefore in order to compare IntFlow and SignFlow, we implemented SignFlow$_{cst}$ as a prototype of IntFlow, and conducted a set of experiments on it. As SignFlow further considers how misinterpreted data is used afterward, i.e. the other three types of data flow patterns for unharmful signedness errors, not limited to P_{cst}, it describes richer features of unharmful signedness error than IntFlow. Experimental results showed that SignFlow excluded 7.73% more integer signedness conversions from instrumentation than IntFlow (Section IV-C) and reduced 41% false positives that IntFlow produced (Section IV-B).

VI. Conclusion

To improve the accuracy of integer signedness error detection, in this paper we first defined the data flow patterns for unharmful signedness error from the view of where they originate and whether they can propagate to security-related operations, and further proposed and designed an instrumentation-based runtime integer signedness error detector, which could improve the precision a lot without sacrificing the completeness. A prototype, SignFlow is implemented as an extension of GCC. Experiments demonstrated that our tool can detect all harmful signedness bugs from our testbed while reducing 41% reports for unharmful ones. In our future work, we will study more features of unharmful signedness errors so as to further improve the detection accuracy. In addition, we will extend SignFlow to handle other types of vulnerabilities.

Acknowledgment

This work has been partly supported by National NSF of China under Grant No. 61170070, 61431008, 61321491; National Key Technology R&D Program of China under Grant No. 2012BAK26B01; the Program B for Outstanding PhD candidate of Nanjing University.

References

[1] National Vulnerability Database, "Mumble Opus Voice Packet Handling Remote Buffer Overflow," http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0045.

[2] D. Brumley, D. X. Song, T. cker Chiueh, R. Johnson, and H. Lin, "Rich: Automatically protecting against integer-based vulnerabilities," in *NDSS'07*, 2007.

[3] W. Dietz, P. Li, J. Regehr, and V. S. Adve, "Understanding integer overflow in c/c++," in *ICSE'12*, 2012, pp. 760–770.

[4] R. B. Dannenberg, W. Dormann, D. Keaton, R. C. Seacord, D. Svoboda, A. Volkovitsky, T. Wilson, and T. Plum, "As-if infinitely ranged integer model," in *ISSRE'10*, 2010, pp. 91–100.

[5] R. E. Rodrigues, V. H. S. Campos, and F. M. Q. Pereira, "A fast and low-overhead technique to secure programs against integer overflows," in *CGO'13*, 2013, pp. 1–11.

[6] M. Pomonis, T. Petsios, K. Jee, M. Polychronakis, and A. D. Keromytis, "Intflow: improving the accuracy of arithmetic error detection using information flow tracking," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 416–425.

[7] "GCC, the GNU Compiler Collection," https://gcc.gnu.org/.

[8] H. Sun, X. Zhang, C. Su, and Q. Zeng, "Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability," in *AsiaCCS'2015*, 2015.

[9] MITRE Corporation, "Common vulnerabilities and exposures," http://cve.mitre.org/.

[10] CERT, "Integerlib, a secure integer library," 2006, http://www.cert.org/secure-coding/IntegerLib.zip.

[11] J. Gennari, S. Hedrick, F. Long, J. Pincar, and R. C. Seacord, "Ranged integers for the c programming language," Carnegie Mellon University, Technical Note CMU/SEI-2007-TN-027, 2007.

[12] D. Molnar, X. C. Li, and D. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *USENIX Security Symposium'09*, 2009, pp. 67–82.