Tolerating C Integer Error via Precision Elevation

Xi Cheng, Min Zhou, Xiaoyu Song, Ming Gu, and Jiaguang Sun

Abstract—In C programs, integer error is a common yet important kind of defect due to arithmetic operations that produce unrepresentable values in certain types. Integer errors are harbored in a wide range of applications and possibly lead to serious software failures and exploitable vulnerabilities. Due to the complicated semantics of C, manually preventing integer errors is challenging even for experienced developers. In this paper we propose a novel approach to automate C integer error repair by elevating the precision of arithmetic operations according to a set of code transformation rules. A large portion of integer errors can be repaired by recovering expected results (i.e., tolerance) instead of removing program functionality. Our approach is fully automatic without requiring code specifications. Furthermore, the transformed code is ensured to be well-typed and has conservativeness property with respect to the original code. Our approach is implemented as a prototype CINTFIX which succeeds in repairing all the integer errors from 7 categories in NIST's Juliet Test Suite. Furthermore, CINTFIX is evaluated on large code bases in SPEC CINT2000, scaling to 366 KLOC within 126 seconds while the transformed code has 10.5 percent slowdown on average. The evaluation results substantiate the potential of our approach in real-world scenarios.

Index Terms—Integer error, tolerability, code transformation

1 INTRODUCTION

In C programs, an integer numerical operation may yield a result that cannot be represented in a specific expression type, and such a value is somehow converted to fit into the target type. Some conversions are well-defined (e.g., unsigned wraparound) by the ISO C standard [1] while others are undefined (e.g., signed integer overflow). Integer errors originate from the misuse of defined behaviors and the use of undefined behaviors due to the developer's empirical certainty of the expected outcomes. Integer errors are classified into several kinds [2]: overflow, underflow, sign misinterpretation, unexpected truncation or sign extension. Integer errors are one of the major sources of system failures and vulnerabilities. A potential problem associated with the power system in Boeing 787 Dreamliners [3] was identified in 2015. A counter represented by a 32-bit signed integer overflows after 2^{31} centiseconds (approximately 248 days), which causes the power system to halt. Multiple integer overflows in the NVIDIA GPU kernel mode driver can be exploited to access out-of-bound memory [4], leading to possible information disclosure, crashes and escalation of privilege. According to a CVE report in 2007 [5], integer overflow error is the second most common vulnerability in the advisories for OS vendors. MITRE also placed integer overflow in the "Top 25 Most Dangerous Software Errors" in 2011 [6].

- *X. Cheng, M. Zhou, M. Gu, and J. Sun are with the School of Software, Tsinghua University, Beijing 100084, China. E-mail: chengxi09@gmail.com, {mzhou, guming, sunjg}@mail.tsinghua.edu.cn.*
- *X. Song is with the Department of Electrical and Computer Engineering, Portland State University, Portland, OR 97201. E-mail: songx@pdx.edu.*

Manuscript received 8 June 2018; revised 6 Aug. 2018; accepted 13 Aug. 2018. Date of publication 20 Aug. 2018; date of current version 22 Jan. 2019. (Corresponding author: Min Zhou.)
Recommended for acceptance by J.-M. Muller.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2018.2866388

The manipulation of integers in programming is error-prone due to the complicated semantics of C integers. The machine representation of an integer value is a fixed-size bit-vector restricted by type-specific characteristics: *width* and *signedness*. The *encoding* determines how a fixed-size bit-vector is interpreted as a numerical value. The most commonly used encodings are *binary encoding* for unsigned integers and *two's complement* for signed integers. Consider the n-bit bit-vector $x = \overline{a_{n-1} \ldots a_0}$, its interpretations under binary encoding and two's complement (denoted by $\langle x \rangle_U$ and $\langle x \rangle_S$, respectively) are as follows:

$$\langle x \rangle_U = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

$$\langle x \rangle_S = -2^{n-1} \cdot a_{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i.$$

The semantics over C integers and $\mathbb{Z}$ are inconsistent. For example, the formula $(x - y > 0) \Longleftrightarrow (x > y)$ holds over $\mathbb{Z}$ but no longer holds over fixed-size bit-vectors owing to the overflow in $x - y$. Even worse, not all integer arithmetic operations are defined in the language standard, such as signed integer overflow. Undefinedness may lead to unpredictable runtime behavior across different micro-processor architectures, compilers and optimization levels.

Researchers have shown great interest in addressing C integer errors automatically. A number of solutions have been proposed for automatic integer error detection. Numerical abstract domains, such as interval [7], octagon [8] and polyhedron [9], are used to reason on numerical program properties. KINT [10] finds integer errors in large C code bases by performing static analysis on a selected subset of variables and inferring user's annotations. LLBMC [11], a bounded model checker, uses a flat and bit-precise memory abstraction to support precise reasoning of arithmetic overflow and memory-based re-interpret casts. IOC [12]

```
void qsort(int arr[], int left, int right)
{
  int i = left, j = right;
  int pivot = arr[(left + right) / 2];
  /* partition and recursion */
  ...
}
```

Fig. 1. A simplified implementation of quick sort.

captures integer overflow errors at runtime via code instrument. SmartFuzz [13] derives test cases that trigger integer overflows by random fuzzing. Although sound reporting of integer errors with few false positives is possible in some domain-specific scenarios [14], detection tools provide few guides for developers to correct the buggy implementation. In practice, manual repair of integer errors is prone to additional errors, even for experts (Section 5.1).

Other approaches have been proposed to automate integer error repair. Coker and Hafiz [15] propose three code transformations based on design decisions. SIFT [16] inserts input filters in the program binary to guard the program from overflow errors at the sites of memory allocation and copy. SoupInt [17] repairs exploitable integer overflows in x86 binaries by leveraging existing error handlers for programmer-anticipated errors. There are several common drawbacks in the existing techniques: (1) the majority of them focus on a fragment of integer errors (e.g., integer overflows in memory operations); and (2) integer errors are repaired by functionality removal, which jeopardizes the robustness of the repaired programs.

Furthermore, recent interest in generic program repair has grown substantially, as evidenced by many individual research projects [18], [19], [20], [21]. The most representative approach is *generate-and-validate*, which heuristically generates many candidate patches by mutating program structures and then validates them until a patch that passes all the given test cases is found. However, generic repair techniques face several fundamental difficulties: (1) the effectiveness relies heavily on the quality of the test cases, which are often insufficient or incomplete; (2) the generated patches rarely ensure correctness [22]; and (3) generate-and-validate systems generally require hours of search for a real-world bug.

In this paper, we propose a novel approach to repair C integer errors automatically by elevating arithmetic precision, i.e., extending the range of integers where error may occur. This approach is inspired by the key observation that the root cause of integer errors is bound to the finite machine representation of integers. Precision elevation has several benefits. First, a substantial number of integer errors can be repaired by functionality recovery instead of removal. Consider the typical implementation of quick sort shown in Fig. 1. If the array to be sorted is very long, `left+right` possibly overflows while the index for `pivot` never exceeds the maximum value of `int`. After elevating the precision of `left` and `right` by changing their declared types to `long` (assuming that `long` has longer byte length than `int` on the certain platform), the program will run normally as expected. Second, precision elevation is conservative. In other words, the outputs of a program and its repaired version are the same given the input that does not trigger any integer errors or wraparounds in the original program. Third, it is possible to apply precision elevation to source code without bug detection or fault localization in advance, thus integer error repair can be run in an efficient manner.

Overall, this paper makes the following contributions:

- *Definition of* tolerability. We define the concept of tolerability for integer errors. Generally, tolerable errors can be repaired by recovering expected results via precision elevation.
- *An automatic approach for elevating integer precision in programs.* We propose a source-to-source transformation technique to repair integer errors in C programs. The input source code is transformed into code that utilizes multi-precision integer arithmetic to be compiled to the executable with integer errors eliminated.
- *A prototype CIntFix.* We implement our approach as a prototype CIntFix that supports integer error repair in a complete C project or some specified source files.
- *Experimental evaluation.* We evaluate our approach on 7 categories in the Juliet Test Suite, and the results show that all the integer errors in the benchmark can be repaired correctly. Moreover, we use the SPEC CINT2000 benchmark to evaluate the efficiency. The results show that (1) CIntFix processes the source code at 2.92 KLOC/s and (2) the transformed code introduces 10.5 percent runtime overhead. The results substantiate the potential of our approach for practical use.

Some of these points were previously presented in an early version of this work [23]. The major substantial extensions of this paper include:

- *A case study on the tolerability of integer errors.* We study the tolerability of real-world integer errors in the CVE database. The result shows that approximately 43.0 percent of integer errors are tolerable, which substantiates the importance of repairing integer errors with functionality recovery.
- *An overhauled formalization for code transformation.* A subset of C, namely CSub, serves as the vehicle for formalization. The semantics on integer arithmetic in CSub is strictly consistent with that defined in C11. We also extend CSub into CSubMP with the multi-precision integer type. Furthermore, the well-typedness of the transformed code and the conservativeness of the code transformation are formally established.
- *Critical-site-directed code transformation.* Precision elevation can be selectively applied to expressions closely associated with security-critical sites at which attacks are performed on the subject programs. This optimization substantially reduces the overhead of the transformed code.
- *Customizable code transformation.* Customization of transformation rules is supported to prevent introducing additional errors by corrupting intentional wraparounds.

The rest of this paper is organized as follows. Section 2 overviews C integer errors and defines their tolerability. Section 3 gives the details of the code transformation used to perform the precision elevation. Section 4 presents the design and implementation of CIntFix on standard C. The evaluation results are discussed in Section 5. Section 6 briefly surveys related research. Finally, Section 7 summarizes our work.

TABLE 1
The Definedness of Integer Operations in C11

Operation	Definedness	Ref.
$+_u, -_u, \times_u$	*defined*	6.2.5.9
$+_s, -_s, \times_s$	*undefined*	3.4.3.3
a/b or $a\%b$	a/b cannot be represented in the target type, or b is zero: *undefined*	6.5.5.6
	otherwise: *defined*	
to signed	the value cannot be represented in the target type: *implementation-defined*	6.3.1.3
	otherwise: *defined*	
to unsigned	*defined*	
$a \ll b$ or $a \gg b$	b is negative or is greater than or equal to the bit-width of the promoted type of a: *undefined*	6.5.7
$a \ll_s b$	$a \times 2^b$ is not representable in the promoted type of a: *undefined*	
	otherwise: *defined*	
$a \ll_u b$	*defined*	
$a \gg_s b$	a has a negative value: *implementation-defined*	
	otherwise: *defined*	
$a \gg_u b$	*defined*	

2 Background

2.1 C Integer Errors

C integer errors occur in arithmetic operations (including addition, subtraction, multiplication, division, remainder and bitwise shift) and conversion operations due to the finite machine representation of integers. C integer operations are not always defined, as shown in Table 1.[1] The notation $+_s$ represents addition over signed integers, and so forth for other similar notations. The signedness of a bitwise shift is consistent with that of the promoted type of the value to be shifted. Note that division-by-zero and shift-by-negative-value are not categorized as integer errors as they are irrelevant to representation issues.

There are two major categories of integer errors. The first involves undefined behaviors, which are behaviors upon the use of certain program constructs or data for which the language standard imposes no requirements. On the one hand, undefined behavior mechanisms grant compilers the freedom to generate high-performance code by exploiting specific properties of a target instruction set. On the other hand, a program with undefined behaviors may expose unpredictable behaviors across different micro-processor architectures, compilers or optimization levels.

Example 1. Signed overflow is undefined. On most x86 platforms, signed addition and subtraction silently wrap-around using two's complement. On MIPS, however, they trap on overflows at the instruction level [24].

Example 2. Consider the following code snippet:

```
int x = /* from user space */
if (x + 100 < x){
    /* do something */
}
```

The `if` condition checks for a possible overflow issue in `x + 100`. GCC 5.4.0 optimizes away the overflow check on the `-O2` level because it assumes that signed integer overflow is impossible due to undefinedness.

```
uint32 width = /* from user space */
uint32 rowsperstrip = /* from user space */
uint32 bytes_per_sample = /* from user space */
tsize_t rowstripsize;
...
rowstripsize = rowsperstrip * bytes_per_sample * (
    width + 1);
obuf = _TIFFmalloc(rowstripsize);
```

Fig. 2. The unsigned overflow error in libtiff.

The second category involves the misuse of defined behaviors, including those explicitly specified by the language standard and the documentation of the implementation in use. It is noteworthy that developers do not need to worry about implementation-defined behaviors unless portability is required, whereas undefined behaviors are unpredictable and thus generally undesired. Consider the code snippet from libtiff shown in Fig. 2. Attackers can use a crafted image with very large `rowsperstrip` and `bytes_per_sample` to make `rowstripsize` wraparound to a small value, leading to undersized allocation. The wrapped multiplication result, which complies with the language standard, does not meet the developer's expectation.

2.2 Tolerability of Integer Errors

A straightforward solution for representation issues is to use more bits to represent integers. If a sufficiently large bit-width is employed universally, representation issues can be eliminated. *Tolerating* an integer error is the process of recovering the user-anticipated program behavior by evaluating certain expressions at a properly elevated precision. Tolerance differs from sanitization in that program functionality vulnerable to target defects is *corrected* instead of *removed*. The major advantage of tolerance over sanitization is that the repaired program maintains the acceptability of continued execution, i.e., it has stronger robustness.

However, tolerance is not always feasible. For example, a multiplication overflow in the size for `malloc` cannot be tolerated because it is impossible to change the signature of `malloc` (which is not part of the subject program) to allocate a very large memory block. Therefore, an integer error is *intolerable* iff the correct value of the buggy expression must be stored in a memory location that has insufficient precision to represent the value and its precision cannot be elevated. In addition to library function parameters, expressions such as field accesses cannot have their precisions elevated because changing the definitions of data structures may hinder the correctness of the relevant pointer arithmetic. An intolerable error can be repaired by enclosing the buggy expression with a sanitizing routine that returns the value of its argument only if it is representable in the target type, and otherwise switches the program execution to an error handler.

In short, tolerability is a property for integer errors, and indicates whether the given integer error could have program functionality corrected by precision elevation.

3 Code Transformation

We present a source-to-source code transformation technique to convert the input source code into code that utilizes

1. The definedness shall be different under previous/future language standards. However, this does not make substantial differences in our discussion.

```
int square(int x) {
  return x * x;
}
int main(void) {
  int val = INT_MAX;
  int ret = square(val);
  printInt(ret);
  return 0;
}
```

(a) The original code

```
mp_int square(mp_int x) {
  return x * x;
}
int main(void) {
  mp_int val = INT_MAX;
  mp_int ret = square(val);
  printInt(__check_int_(ret
      ));
  return 0;
}
```

(b) The transformed code

Fig. 3. A motivating example of code transformation.

multi-precision integer arithmetic to be compiled to the executable with integer errors eliminated. We use a subset of C called CSub to serve as the vehicle for formalizing the code transformation. The design of CSub is partly inspired by Clight [25], and specifically focuses on the semantics of integer operations. The semantics of CSub complies with C11. The semantics of the implementation-defined behaviors is among the possible implementations. Before the formal discussion, we first present a small example to demonstrate how the code transformation works.

3.1 Motivating Example

Fig. 3a shows a simple program where `printInt` is a library function that prints the input integer value and an overflow error occurs at line 2. The transformed program by elevating the integer precision is shown in Fig. 3b. `mp_int` denotes multi-precision integer type while `_check_int_(ret)` returns the `int` value of `ret` only if `ret` is representable in `int`, and otherwise switches to an error handler. It is noteworthy that the return type of `main` and the parameter type of `printInt` are unmodifiable by the subject program, thus they are not elevated. The transformed program has the overflow error repaired because the multiplication result at line 2 is correctly recovered under multi-precision integer arithmetic.

3.2 The Language CSub

Notations. The notation se^* denotes 0,1 or multiple occurrences of the syntactic element se, and $se^?$ denotes the optional occurrence of the syntactic element se.

Types. Type syntax of CSub is shown in Fig. 4a. Supported types include integer types, structured types, array types, pointer types and function types. An integer type $\texttt{int}(n, \varsigma)$ is characterized by its byte-length n and signedness ς (`S` for signed and `U` for unsigned). In CSub, the legal values of n are predefined by the employed data model (i.e., data type width scheme). For example, in a typical x86-64 platform, the legal values of n are 1, 2, 4 and 8. An array type $\tau[n]$ carries the array size n which is determined at compile time. Array types with unknown sizes in C are modeled by the corresponding pointer types in CSub. A structured type carries an identifier id and a list of named types ϕ. For anonymous structured types in C, generated names are assigned at compile time. A recursive pointer type $\texttt{recur\_ptr}(id)$ models a pointer type to the nearest enclosing structured type with identifier id and is used for recursive data structures. For example, the structured type L for linked lists can be defined as follows:

$$\texttt{struct}(L, ((val, \tau), (next, \texttt{recur\_ptr}(L)))).$$

$$\begin{array}{lcl} \varsigma & ::= & \texttt{S} \mid \texttt{U} \\ \tau & ::= & \texttt{int}(n, \varsigma) \mid \texttt{void} \mid \tau[n] \mid \tau* \mid \tau(\tau^*) \mid \texttt{struct}(id, \phi) \mid \\ & & \texttt{union}(id, \phi) \mid \texttt{recur\_ptr}(id) \\ \phi & ::= & (id, \tau)^* \end{array}$$

(a) Types

$$\begin{array}{lcl} e & ::= & id \mid \texttt{*}e \mid e.id \mid \\ & & n \mid n\texttt{u} \mid n\texttt{l} \mid n\texttt{ul} \mid n\texttt{ll} \mid n\texttt{ull} \mid \texttt{sizeof}(\tau) \mid (\tau)e \mid \Diamond_u e \mid \\ & & e_1 \Diamond_b e_2 \mid \texttt{\&}e \mid e_1 \texttt{ ? } e_2 \texttt{ : } e_3 \\ \Diamond_u & ::= & \texttt{-} \mid \texttt{\~{}} \\ \Diamond_b & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \\ & & \texttt{<} \mid \texttt{<=} \mid \texttt{==} \\ & & \texttt{<<} \mid \texttt{>>} \mid \texttt{\&} \mid \texttt{|} \mid \texttt{\^{}} \end{array}$$

(b) Expressions

$$\begin{array}{lcl} s & ::= & \texttt{skip} \mid e_1 = e_2 \mid e_1 = e_2(e^*) \mid e_1(e^*) \mid s_1\texttt{;}s_2 \mid \\ & & \texttt{if}(e)\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}(e)\ s \mid \\ & & \texttt{break} \mid \texttt{continue} \mid \texttt{return}\ e^? \end{array}$$

(c) Statements

$$\begin{array}{lcl} d & ::= & \tau\ id \\ Fi & ::= & \tau\ id(d_1^*)\ \{d_2^*\texttt{;}s\} \\ Fe & ::= & \texttt{extern}\ \tau\ id(d^*) \\ F & ::= & Fi \mid Fe \\ P & ::= & d^*\texttt{;}F^*\texttt{;main} = id \end{array}$$

(d) Functions and programs

Fig. 4. The abstract syntax of CSub.

Recursive pointer types make CSub types finitely constructible (i.e., each CSub type can be constructed in finite steps based on abstract syntax of types). With a small abuse of notation, in the rest of the paper, we use $\tau*$ in place of $\texttt{recur\_ptr}(id)$, where τ is the type with the identifier id. CSub does not support type qualifiers in C as they can be erased at compile time.

Expressions. Fig. 4b shows the syntax of expressions. Each expression is either an l-value or r-value. Only l-values can refer to memory locations and thus can be assigned. L-values include identifiers (id), pointer dereferences ($*e$) and field accesses ($e.id$). It is noteworthy that (1) floating constants are unsupported and (2) one can specify the type of an integer literal by appending a suffix, which could be the combination of an unsigned-suffix `u`, a long-suffix `l` and a long-long-suffix `ll`. Expressions are side-effect-free. Some C expressions are not modeled as they can be expressed as syntactic sugars, as shown in Table 2.

Statements. Fig. 4c shows the syntax of statements. Supported statements include empty statements, assignments, function calls, statement sequences, branching statements, loops, and control statements for loops and functions.

Functions and Programs. As Fig. 4d shows, a CSub program P consists of global declarations, function definitions and the program entry `main`. Standalone type declarations and type aliases are not supported in CSub because types can be resolved at compile time in C.

TABLE 2
The Syntactic Sugars for Some C Expressions

C expression	CSub expression
$e_1[e_2]$	$\texttt{*}(e_1 + e_2)$
$e\texttt{->}id$	$\texttt{*}e.id$
e_1 && e_2	e_1 ? (e_2 ? 1 : 0) : 0
e_1 || e_2	e_1 ? 1 : (e_2 ? 1 : 0)
$\texttt{!}e$	e == 0
$e_1 > e_2$	$\texttt{!}(e_1 \texttt{<=} e_2)$
e_1 >= e_2	$\texttt{!}(e_1 < e_2)$
e_1 != e_2	$\texttt{!}(e_1 == e_2)$

$$\frac{\Gamma_g(id) = \tau \text{ or } \Gamma_f(id) = \tau}{\Gamma_g, \Gamma_f \vdash id : \tau} \ (1) \qquad \frac{\Gamma_g, \Gamma_f \vdash e : \tau* \text{ or } \Gamma_g, \Gamma_f \vdash e : \tau[n]}{\Gamma_g, \Gamma_f \vdash *e : \tau} \ (2)$$

$$\frac{\Gamma_g, \Gamma_f \vdash e : \texttt{struct}(\_, \phi) \text{ or } \Gamma_g, \Gamma_f \vdash e : \texttt{union}(\_, \phi) \qquad (id, \tau) \in \phi}{\Gamma_g, \Gamma_f \vdash e.id : \tau} \ (3) \qquad \frac{}{\Gamma_g, \Gamma_f \vdash n : \imath(4, \texttt{S}, n)} \ (4) \qquad \frac{}{\Gamma_g, \Gamma_f \vdash n\texttt{u} : \imath(4, \texttt{U}, n)} \ (5)$$

$$\frac{}{\Gamma_g, \Gamma_f \vdash n\texttt{l} : \imath(8, \texttt{S}, n)} \ (6) \qquad \frac{}{\Gamma_g, \Gamma_f \vdash n\texttt{ul} : \imath(8, \texttt{U}, n)} \ (7) \qquad \frac{}{\Gamma_g, \Gamma_f \vdash n\texttt{ll} : \imath(8, \texttt{S}, n)} \ (8) \qquad \frac{}{\Gamma_g, \Gamma_f \vdash n\texttt{ull} : \imath(8, \texttt{U}, n)} \ (9)$$

$$\frac{}{\Gamma_g, \Gamma_f \vdash \texttt{sizeof}(\tau) : \texttt{int}(8, \texttt{U})} \ (10) \qquad \frac{}{\Gamma_g, \Gamma_f \vdash (\tau)e : \tau} \ (11) \qquad \frac{\Gamma_g, \Gamma_f \vdash e : \tau \qquad \mathcal{I}(\tau)}{\Gamma_g, \Gamma_f \vdash \Diamond_u e :\uparrow (\tau)} \ (12)$$

$$\frac{\Gamma_g, \Gamma_f \vdash e_1 : \tau_1 \qquad \Gamma_g, \Gamma_f \vdash e_2 : \tau_2 \qquad \mathcal{I}(\tau_1) \wedge \mathcal{I}(\tau_2) \qquad \Diamond_b \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{\&}, \texttt{|}, \texttt{\^{}}\}}{\Gamma_g, \Gamma_f \vdash e_1 \Diamond_b e_2 : \tau_1 \Uparrow \tau_2} \ (13)$$

$$\frac{\Gamma_g, \Gamma_f \vdash e_1 : \tau_1 \qquad \Gamma_g, \Gamma_f \vdash e_2 : \tau_2 \qquad \mathcal{P}(\tau_1) \wedge \mathcal{I}(\tau_2) \qquad \Diamond_b \in \{\texttt{+}, \texttt{-}\}}{\Gamma_g, \Gamma_f \vdash e_1 \Diamond_b e_2 :\downarrow (\tau_1)} \ (14) \qquad \frac{\Gamma_g, \Gamma_f \vdash e_1 : \tau_1 \qquad \Gamma_g, \Gamma_f \vdash e_2 : \tau_2 \qquad \mathcal{I}(\tau_1) \wedge \mathcal{P}(\tau_2)}{\Gamma_g, \Gamma_f \vdash e_1 \texttt{+} e_2 :\downarrow (\tau_2)} \ (15)$$

$$\frac{\Gamma_g, \Gamma_f \vdash e_1 : \tau_1 \qquad \Gamma_g, \Gamma_f \vdash e_2 : \tau_2 \qquad \mathcal{P}(\tau_1) \wedge \mathcal{P}(\tau_2)}{\Gamma_g, \Gamma_f \vdash e_1 \texttt{-} e_2 : \texttt{int}(8, \texttt{U})} \ (16)$$

$$\frac{\Gamma_g, \Gamma_f \vdash e_1 : \tau_1 \qquad \Gamma_g, \Gamma_f \vdash e_2 : \tau_2 \qquad \mathcal{I}(\tau_1) \wedge \mathcal{I}(\tau_2) \qquad \Diamond_b \in \{\texttt{<<}, \texttt{>>}\}}{\Gamma_g, \Gamma_f \vdash e_1 \Diamond_b e_2 :\uparrow (\tau_1)} \ (17)$$

$$\frac{\Gamma_g, \Gamma_f \vdash e_1 : \tau_1 \qquad \Gamma_g, \Gamma_f \vdash e_2 : \tau_2 \qquad (\mathcal{I}(\tau_1) \wedge \mathcal{I}(\tau_2)) \vee (\mathcal{P}(\tau_1) \wedge \mathcal{P}(\tau_2) \wedge \tau_1 \approx \tau_2) \qquad \Diamond_b \in \{\texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{==}, \texttt{!=}\}}{\Gamma_g, \Gamma_f \vdash e_1 \Diamond_b e_2 : \texttt{int}(4, \texttt{S})} \ (18)$$

$$\frac{\Gamma_g, \Gamma_f \vdash e : \tau}{\Gamma_g, \Gamma_f \vdash \texttt{\&}e : \tau*} \ (19) \qquad \frac{\Gamma_g, \Gamma_f \vdash e_1 : \tau_1 \qquad \Gamma_g, \Gamma_f \vdash e_2 : \tau_2 \qquad \Gamma_g, \Gamma_f \vdash e_3 : \tau_3}{\Gamma_g, \Gamma_f \vdash e_1 \ \texttt{?} \ e_2 \ \texttt{:} \ e_3 :\Uparrow (\tau_1, \tau_2, \tau_3)} \ (20)$$

Fig. 5. The type inference rules for CSub expressions.

Static Semantics. The type judgment

$$\Gamma_g, \Gamma_f \vdash e : \tau,$$

denotes that given the typing context (Γ_g, Γ_f), the type of the expression e (in the function scope of f) is inferred as τ. Γ_g maps the identifiers of global variables to their declared types. Γ_f maps the parameter variables and local variables of the function f to their declared types. If e is in the global scope, we have $\Gamma_f = \emptyset$. CSub does not support variable overshadowing, thus we have $\mathrm{Dom}(\Gamma_g) \cap \mathrm{Dom}(\Gamma_f) = \emptyset$.

Fig. 5 gives type inference rules for CSub under the LP64 data model, where `int`, `long` and a pointer type have 4, 8 and 8 bytes, respectively. Implicit conversions other than integer promotion (C11 6.3.1.1) and usual arithmetic conversion (C11 6.3.1.8) are not supported in CSub, thus (1) both sides of an assignment have the same types, (2) a function argument has the same type as the corresponding parameter, and (3) the type of a return value is the return type of the enclosing function. Some auxiliary predicates and operations (shown in Fig. 6) are introduced to make type inference rules compliant with C11. The predicate $\mathcal{I}$ (resp. $\mathcal{P}$) checks whether a type is an integer type (resp. a pointer type or an array type). $\imath(sz, \varsigma, n)$ chooses a type for the given integer literal n restricted by the minimum byte length sz and the signedness ς. $\tau_1 \approx \tau_2$ holds iff τ_1 and τ_2 are compatible (C11 6.2.7). The operation $\downarrow$ flattens an array type to a pointer type. The operation $\uparrow$ models integer promotion, and $\Uparrow$ models usual arithmetic conversion. The dedicated operation for conditional expressions $\Uparrow\!\uparrow$ returns a valid type only if the types of the operands meet the constraints in C11 6.5.15,[2] and otherwise returns an invalid type $\oslash$. Once an invalid type is derived, type inference fails, which implies that the program has syntax errors.

Dynamic Semantics. We focus on the operational semantics of CSub expressions relevant to integer errors. The basic concepts are listed in Fig. 7. A memory location l is denoted with an abstract address $a \in \mathbb{N}$ and an offset δ which is non-negative. The null pointer points to the special memory location $(0, 0)$. An abstract memory state M maps a memory location l and a type τ to an abstract value v interpreted under τ. An abstract value can be an integer over $\mathbb{Z}$, a reference $\mathbb{P}(l)$ to a memory location l, or an undefined value $\oslash$ that makes the execution fail. An expression is evaluated based on a *program state* consisting of a global environment Δ_g that maps identifiers to abstract addresses and function definitions, a local environment Δ_f that maps parameters and local variables to their abstract addresses, and the memory state M that abstracts the contents in the memory. Since expressions are side-effect-free, (Δ_g, Δ_f, M) remains unchanged in the evaluation.

Fig. 8 lists the operational semantics of CSub expressions relevant to integer errors. Rules (21-24) evaluate l-values as memory locations. $\mathcal{F}(\phi, id)$ denotes the offset of id in the field list ϕ while the paddings between structure members are considered. Rule (25) loads an abstract value for the given l-value. Rules (26-30) evaluate r-values as abstract values. Rules (27-30) are relevant to integer errors. $\mathcal{C}(v, \tau)$ converts value v in type τ, such that

$$\begin{aligned}
&\mathcal{C}(\mathbb{Z}(n), \tau) &&= \mathbb{Z}((v - \underline{\tau}) \bmod (\overline{\tau} - \underline{\tau}) + \underline{\tau}) && \text{if } \mathcal{I}(\tau)\\
&\mathcal{C}(\mathbb{P}(a, \delta), \tau) &&= \mathbb{Z}(N_a + \delta) && \text{if } \tau ::= \texttt{int}(_, _) \wedge N_a + \delta \in [\underline{\tau}, \overline{\tau}]\\
&\mathcal{C}(\mathbb{Z}(0), \tau*) &&= \mathbb{P}(0, 0)\\
&\mathcal{C}(\mathbb{Z}(n), \tau*) &&= \mathbb{P}(l') && \text{if } n \neq 0\\
&\mathcal{C}(\mathbb{P}(l), \tau*) &&= \mathbb{P}(l)\\
&\mathcal{C}(v, \tau) &&= \oslash && \text{otherwise.}
\end{aligned}$$

2. In fact, there is one exception. In CSub, τ_1 is required to be an integer type while C11 allows it to be a pointer type. A pointer p serving as a condition expression can be expressed as the syntactic sugar $p \ \texttt{!} = 0$.

$$
\begin{array}{lcll}
\mathcal{I}(\tau) & = & \mathsf{true} & \text{if } \tau ::= \mathtt{int}(n,\varsigma)\\
\mathcal{I}(\tau) & = & \mathsf{false} & \text{otherwise}\\
\mathcal{P}(\tau) & = & \mathsf{true} & \text{if } \tau ::= \tau'* \text{ or } \tau ::= \tau'[n]\\
\mathcal{P}(\tau) & = & \mathsf{false} & \text{otherwise}\\
\downarrow(\tau) & = & \tau'* & \text{if } \tau ::= \tau'* \text{ or } \tau ::= \tau'[n]\\
\downarrow(\tau) & = & \oslash &\\
\uparrow(\tau) & = & \mathtt{int}(4,\mathtt{S}) & \text{if } \tau ::= \mathtt{int}(n,\_) \text{ and } n < 4\\
\uparrow(\tau) & = & \tau & \text{if } \mathcal{I}(\tau) = \mathsf{true}\\
\uparrow(\tau) & = & \oslash &
\end{array}
$$

$$
\begin{array}{lll}
\imath(sz,\varsigma,n) = & \mathtt{int}(8,\varsigma) & \text{if } \mathtt{int}(8,\varsigma) \text{ cannot represent } n\\
\imath(sz,\varsigma,n) = & \mathtt{int}(sz',\varsigma) & \text{where } sz' = \inf\{n' \mid n' \in \{4,8\},\\
 & & \mathtt{int}(n',\varsigma) \text{ can represent } n\}
\end{array}
$$

$$
\begin{array}{lll}
\tau_1 \Uparrow \tau_2 = & \uparrow(\tau_1) \Uparrow \uparrow(\tau_2) & \text{if } \uparrow(\tau_1) \neq \tau_1 \text{ or } \uparrow(\tau_2) \neq \tau_2\\
\tau_1 \Uparrow \tau_2 = & \oslash & \neg\mathcal{I}(\tau_1) \vee \neg\mathcal{I}(\tau_2)
\end{array}
$$

$$
\tau_a ::= \mathtt{int}(n_1,\varsigma_1) \qquad \tau_b ::= \mathtt{int}(n_2,\varsigma_2)
$$

$$
\begin{array}{lll}
\tau_a \Uparrow \tau_b = & \mathtt{int}(\max(n_1,n_2),\varsigma_1) & \text{if } \varsigma_1 = \varsigma_2\\
\tau_a \Uparrow \tau_b = & \mathtt{int}(n_i,\varsigma_i) & \text{if } n_1 \neq n_2 \text{ and } n_i = \max(n_1,n_2)\\
\tau_a \Uparrow \tau_b = & \mathtt{int}(n_1,\mathtt{U}) & \text{if } n_1 = n_2
\end{array}
$$

$$
\begin{array}{ll}
\tau \approx \tau & \\
\tau_1* \approx \tau_2* & \text{if } \tau_1 \approx \tau_2\\
\tau_1* \approx \tau_2[n] & \text{if } \tau_1 \approx \tau_2\\
\tau_1[n] \approx \tau_2* & \text{if } \tau_1 \approx \tau_2\\
\tau_1[n] \approx \tau_2[n] & \text{if } \tau_1 \approx \tau_2\\
\mathtt{struct}(id_1,\phi_1) \approx \mathtt{struct}(id_2,\phi_2) & \text{if } \phi_1 \approx \phi_2\\
\mathtt{union}(id_1,\phi_1) \approx \mathtt{union}(id_2,\phi_2) & \text{if } \phi_1 \sim \phi_2\\
\tau_{1r}(\tau_{1p}^*) \approx \tau_{2r}(\tau_{2p}^*) & \text{if } \tau_{1r} \approx \tau_{2r} \text{ and } \tau_{1p}^* \approx \tau_{2p}^*\\
\mathtt{recur\_ptr}(id_1) \approx \mathtt{recur\_ptr}(id_2) & id_1 \approx id_2
\end{array}
$$

$$
\begin{array}{rl}
\phi_1 ::= & \langle (id_{11},\tau_{11}),\ldots,(id_{1s},\tau_{1s})\rangle\\
\phi_2 ::= & \langle (id_{21},\tau_{21}),\ldots,(id_{2t},\tau_{2t})\rangle
\end{array}
$$

$$
\tau_{1p}^* ::= \langle \tau_{1p1},\ldots,\tau_{1ps}\rangle \qquad \tau_{2p}^* ::= \langle \tau_{2p1},\ldots,\tau_{2pt}\rangle
$$

$$
\begin{array}{ll}
\phi_1 \approx \phi_2 & \text{if } s = t, id_{1i} = id_{2i} \text{ and } \tau_{1i} \approx \tau_{2i}\\
\phi_1 \sim \phi_2 & \text{if a bijection } \mathcal{B} \text{ exists s.t. } \mathcal{B}((id_{1i},\tau_{1i})) = (id_{2j},\tau_{2j})\\
 & id_{1i} = id_{2j} \text{ and } \tau_{1i} \approx \tau_{2j}\\
\tau_{1p}^* \approx \tau_{2p}^* & \text{if } s = t \text{ and } \tau_{1pi} \approx \tau_{2pi}
\end{array}
$$

$$
\begin{array}{lll}
\Uparrow(\tau_1,\tau_2,\tau_3) = & \oslash & \text{if } \neg\mathcal{I}(\tau_1)\\
\Uparrow(\tau_1,\tau_2,\tau_3) = & \tau_2 \Uparrow \tau_3 & \text{if } \mathcal{I}(\tau_2) \wedge \mathcal{I}(\tau_3)\\
\Uparrow(\tau_1,\tau_2,\tau_3) = & \tau_2 & \text{if } \tau_2 = \tau_3\\
\Uparrow(\tau_1,\tau_2,\tau_3) = & \tau_2 & \text{if } \mathcal{P}(\tau_2) \wedge \mathcal{P}(\tau_3) \wedge \tau_2 \approx \tau_3\\
\Uparrow(\tau_1,\tau_2,\tau_3) = & \tau_i & \text{if } \mathcal{P}(\tau_i) \wedge \tau_j = \mathtt{void}* \wedge \{i,j\} = \{2,3\}\\
\Uparrow(\tau_1,\tau_2,\tau_3) = & \oslash & \text{otherwise}
\end{array}
$$

Fig. 6. The auxiliary predicates and operations.

An integer value is fit in a target integer type τ by being enlarged or reduced modulo $\overline{\tau} - \underline{\tau}$ where $\overline{\tau}$ (resp. $\underline{\tau}$) refer to the largest (resp. smallest) value that can be represented in the integer type τ. Such a conversion is strictly compliant with C11 for unsigned τ and is also a possible implementation for signed τ. A pointer reference $\mathbb{P}(a,\delta)$ is converted to

$$
\begin{array}{lcl}
a & \in & \mathbb{N}\\
l & ::= & (a,\delta)\\
v & ::= & \mathbb{Z}(n) \mid \mathbb{P}(l) \mid \oslash\\
\Delta_g & ::= & (id \mapsto a \cup F)\\
\Delta_f & ::= & id \mapsto a\\
M & ::= & a \mapsto (\delta, \tau \mapsto v)
\end{array}
$$

$$
\begin{array}{ll}
\Delta_g,\Delta_f,M \vdash e \Leftarrow l & \text{if } e \text{ is an l-value}\\
\Delta_g,\Delta_f,M \vdash e \Rightarrow v & \text{if } e \text{ is an r-value}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{L}(M,a,\delta,\tau) = & M(a)(\delta,\tau) & \text{if } \mathcal{I}(\tau) \text{ holds}\\
\mathcal{L}(M,a,\delta,\tau) = & M(a)(\delta,\mathtt{void}*) & \text{if } \tau ::= \tau'*\\
\mathcal{L}(M,a,\delta,\tau) = & \mathbb{P}(a,\delta) & \text{if } \tau ::= \tau'[n] \mid \tau'(\tau^*)\\
\mathcal{L}(M,a,\delta,\tau) = & \oslash & \text{otherwise}
\end{array}
$$

Fig. 7. Basic concepts of dynamic semantics.

$\mathbb{Z}(N_a + \delta)$ if it can be represented in the target type τ, where N_a is a specified positive integer for a. A non-zero integer $\mathbb{Z}(n)$ is converted to $\mathbb{P}(l')$, where l' may refer to an invalid memory location. $\blacklozenge_u(\Diamond_u, v, \tau)$ is the result of $\Diamond_u$ on v in type τ, such that

$$
\begin{array}{lcll}
\blacklozenge_u(\tilde{}, \mathbb{Z}(n), \tau) & = & \mathsf{bit\_comp}(n,\tau) & \text{if } \mathcal{I}(\tau)\\
\blacklozenge_u(-, \mathbb{Z}(n), \tau) & = & \mathcal{C}(\mathbb{Z}(-n),\tau) & \text{if } \tau ::= \mathtt{int}(\_,\mathtt{U})\\
\blacklozenge_u(-, \mathbb{Z}(n), \tau) & = & \mathbb{Z}(-n) & \text{if } \tau ::= \mathtt{int}(\_,\mathtt{S}) \wedge -n \in [\underline{\tau},\overline{\tau}]\\
\blacklozenge_u(\Diamond_u, \mathbb{Z}(n), \tau) & = & \oslash & \text{otherwise,}
\end{array}
$$

where $\mathsf{bit\_comp}(n,\tau)$ denotes the bitwise complement result of n in type τ. $\blacklozenge_b(\Diamond_b, v_1, v_2, \tau_1, \tau_2, \tau)$ is the result of $\Diamond_b$ over v_1 and v_2 with respect to the operand types τ_1, τ_2 and their combined type τ (which can be inferred from τ_1 and τ_2), and its full definition is shown in Fig. 9. The result of an undefined arithmetic operation (e.g., overflowed signed addition) is $\oslash$, whereas we specify a possible result for an implementation-defined operation (e.g., right-shift a negative value). The notation $\mathsf{bit\_and}(n_1,n_2,\tau)$ denotes the bitwise AND result of n_1 and n_2 in type τ, and $\mathcal{W}(n,\tau)$ refers to the bit-width of integer value n in type τ.

Integer Errors. Recall that there are two categories of integer errors. One involves undefined behaviors (Type I), and the other involves the misuse of defined behaviors (Type II). Formally, e has a Type I integer error under programs state (Δ_g,Δ_f,M) iff

(1) when $e ::= \Diamond_u e_1$, $\Delta_g,\Delta_f,M \vdash e_1 \Rightarrow v_1$, $v_1 \neq \oslash$ and $\Delta_g,\Delta_f,M \vdash e \Rightarrow \oslash$;

(2) when $e ::= e_1 \Diamond_b e_2$, suppose that for each $i = 1,2$, $\Delta_g,\Delta_f,M \vdash e_i \Rightarrow v_i$ and $v_i \neq \oslash$, then:

(a) $\Delta_g,\Delta_f,M \vdash e \Rightarrow \oslash$;

$$\frac{\Delta_f(id) = a \text{ or } a \notin \mathrm{Dom}(\Delta_f) \wedge \Delta_f(id) = a}{\Delta_g,\Delta_f,M \vdash id \Leftarrow (a,0)} \quad (21) \qquad \frac{\Delta_g,\Delta_f,M \vdash e \Rightarrow \mathbb{P}(l)}{\Delta_g,\Delta_f,M \vdash *e \Leftarrow l} \quad (22)$$

$$\frac{\Delta_g,\Delta_f,M \vdash e \Leftarrow (a,\delta) \qquad \mathcal{T}(e) = \mathtt{struct}(id',\phi) \qquad F(\phi,id) = \delta'}{\Delta_g,\Delta_f,M \vdash e.id \Leftarrow (a,\delta+\delta')} \quad (23) \qquad \frac{\Delta_g,\Delta_f,M \vdash e \Leftarrow l \qquad \mathcal{T}(e) = \mathtt{union}(id',\phi)}{\Delta_g,\Delta_f,M \vdash e.id \Leftarrow l} \quad (24)$$

$$\frac{\Delta_g,\Delta_f,M \vdash e \Leftarrow (a,\delta) \qquad \mathcal{L}(M,a,\delta,\mathcal{T}(e)) = v}{\Delta_g,\Delta_f,M \vdash e \Rightarrow v} \quad (25) \qquad \frac{\Delta_g,\Delta_f,M \vdash e \Leftarrow l}{\Delta_g,\Delta_f,M \vdash \&e \Rightarrow \mathbb{P}(l)} \quad (26) \qquad \frac{\mathcal{C}(\mathbb{Z}(n),\mathcal{T}(n)) = v'}{\Delta_g,\Delta_f,M \vdash n \Rightarrow v'} \quad (27)$$

$$\frac{\Delta_g,\Delta_f,M \vdash e_1 \Rightarrow v_1 \qquad \Delta_g,\Delta_f,M \vdash e_2 \Rightarrow v_2 \qquad \blacklozenge_b(\Diamond_b, v_1, v_2, \mathcal{T}(e_1), \mathcal{T}(e_2), \mathcal{T}(e_1 \Diamond_b e_2)) = v'}{\Delta_g,\Delta_f,M \vdash e_1 \Diamond_b e_2 \Rightarrow v'} \quad (28)$$

$$\frac{\Delta_g,\Delta_f,M \vdash e \Rightarrow v \qquad \blacklozenge_u(\Diamond_u, v, \mathcal{T}(\Diamond_u e)) = v'}{\Delta_g,\Delta_f,M \vdash \Diamond_u e \Rightarrow v'} \quad (29) \qquad \frac{\Delta_g,\Delta_f,M \vdash e \Rightarrow v \qquad \mathcal{C}(v,\tau) = v'}{\Delta_g,\Delta_f,M \vdash (\tau)e \Rightarrow v'} \quad (30)$$

Fig. 8. The operational semantics for selected CSub expressions. $\mathcal{T}(e)$ denotes the type of e.

$$
\begin{array}{lll}
\blacklozenge_b(+,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\tau_1,\tau_2,\tau) = \mathcal{C}(\mathbb{Z}(n_1'+n_2'),\tau) & \text{if } \tau \text{ is unsigned} & \text{analogous for - and *}\\
\blacklozenge_b(+,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\tau_1,\tau_2,\tau) = \mathbb{Z}(n_1'+n_2') & \text{if } \tau \text{ is signed and } n_1'+n_2' \in [\underline{\tau},\overline{\tau}] & \text{analogous for - and *}\\
\blacklozenge_b(/,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\tau_1,\tau_2,\tau) = \mathbb{Z}(\mathsf{trunc}(n_1'/n_2')) & \text{if } n_2' \neq 0 \text{ and } \mathsf{trunc}(n_1'/n_2') \in [\underline{\tau},\overline{\tau}] & \mathsf{trunc} \text{ denotes rounding towards } 0\\
\blacklozenge_b(\%,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\tau_1,\tau_2,\tau) = \mathbb{Z}(n_1'-n_2'\cdot\mathsf{trunc}(n_1'/n_2')) & \text{if } \blacklozenge_b(/,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\tau_1,\tau_2,\tau) \neq \oslash & \\
\blacklozenge_b(+,\mathbb{P}(a,\delta),\mathbb{Z}(n),\tau_1,\tau_2,\tau) = \mathbb{P}(a,\delta+n) & & \text{same as } \blacklozenge_b(+,\mathbb{Z}(n),\mathbb{P}(a,\delta),\tau_1,\tau_2,\tau)\\
\blacklozenge_b(-,\mathbb{P}(a,\delta),\mathbb{Z}(n),\tau_1,\tau_2,\tau) = \mathbb{P}(a,\delta-n) & & \\
\blacklozenge_b(-,\mathbb{P}(a,\delta_1),\mathbb{P}(a,\delta_2),\tau_1,\tau_2,\tau) = \mathbb{Z}(\delta_1-\delta_2) & & \\
\blacklozenge_b(<,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\tau_1,\tau_2,\tau) = \mathbb{Z}(1) & \text{if } n_1' < n_2' & \text{analogous for <= and ==}\\
\blacklozenge_b(<,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\tau_1,\tau_2,\tau) = \mathbb{Z}(0) & \text{if } n_1' < n_2' \text{ does not hold} & \text{analogous for <= and ==}\\
\blacklozenge_b(<,\mathbb{P}(a_1,\delta_1),\mathbb{P}(a_2,\delta_2),\tau_1,\tau_2,\tau) = \mathbb{Z}(1) & \text{if } a_1 = a_2 \wedge \delta_1 < \delta_2 & \text{analogous for <= and ==}\\
\blacklozenge_b(<,\mathbb{P}(a_1,\delta_1),\mathbb{P}(a_2,\delta_2),\tau_1,\tau_2,\tau) = \mathbb{Z}(0) & \text{if } a_1 = a_2 \wedge \neg(\delta_1 < \delta_2) & \text{analogous for <=}\\
\blacklozenge_b(==,\mathbb{P}(a_1,\delta_1),\mathbb{P}(a_2,\delta_2),\tau_1,\tau_2,\tau) = \mathbb{Z}(0) & \text{if } a_1 \neq a_2 \vee \delta_1 \neq \delta_2 & \\
\blacklozenge_b(\&,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\tau_1,\tau_2,\tau) = \mathsf{bit\_and}(n_1',n_2',\tau) & & \text{analogous for | and } \hat{}\\
\blacklozenge_b(<<,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\tau_1,\tau_2,\tau) = \mathcal{C}(\mathbb{Z}(n_1\times 2^{n_2}),\tau) & \text{if } \tau_1 \text{ is unsigned, } n_2 \geq 0 \text{ and } n_2 \leq \mathcal{W}(n_1,\tau_1) & \\
\blacklozenge_b(<<,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\tau_1,\tau_2,\tau) = \mathbb{Z}(n_1\times 2^{n_2}) & \text{if } \tau_1 \text{ is signed, } n_1 \geq 0,\ n_2 \geq 0,\ n_2 \leq \mathcal{W}(n_1,\tau_1) \text{ and } n_1\times 2^{n_2} \in [\underline{\tau},\overline{\tau}] & \\
\blacklozenge_b(>>,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\tau_1,\tau_2,\tau) = \mathbb{Z}(\lfloor n_1\times 2^{-n_2}\rfloor) & \text{if } n_2 \geq 0 \text{ and } n_2 \leq \mathcal{W}(n_1,\tau_1) & \\
\blacklozenge_b(\Diamond_b,n_1,n_2,\tau_1,\tau_2,\tau) = \oslash & \text{otherwise} &
\end{array}
$$

Fig. 9. The definition of $\blacklozenge_b$. $n_i' = \mathcal{C}(\mathbb{Z}(n_i),\tau)$ for $i = 1, 2$.

(b) if $\Diamond_b \in \{/, \%\}$, $v_2 \neq \mathbb{Z}(0)$;
(3) if $\Diamond_b \in \{ \ll, \gg \}$, $v_2 = \mathbb{Z}(n)$ where $n \geq 0$.

If e has a Type II integer error, then

$$\Delta_g, \Delta_f, M \vdash e \Rightarrow \mathbb{Z}(n) \wedge n \neq \mathcal{E}(\Delta_g, \Delta_f, M, e),$$

where $\mathcal{E}(\Delta_g, \Delta_f, M, e)$ denotes the expected value of e under the specified program state.

3.3 CSubMP: CSub with Multi-Precision Integers

CSubMP is a dialect of CSub that supports multi-precision integers. Let $\widehat{\tau}$ be the type variable in CSubMP, then we have $\widehat{\tau} ::= \tau \mid \widehat{\texttt{int}}$, where $\widehat{\texttt{int}}$ is the multi-precision integer type. The abstract syntax implies that $\widehat{\texttt{int}}$ cannot be used to construct derived types (e.g., pointer types).

Static Semantics. The type inference rules are all applicable for CSubMP on the premise that $\mathcal{I}$, $\imath$ and $\Uparrow$ are replaced by $\widehat{\mathcal{I}}$, $\widehat{\imath}$ and $\widehat{\Uparrow}$, respectively. Their definitions are as follows.

$$
\begin{array}{lll}
\widehat{I}(\tau) & = \mathsf{true} & \text{if } \tau ::= \widehat{\texttt{int}}\\
\widehat{I}(\tau) & = \mathcal{I}(\tau) & \text{otherwise}\\
\widehat{\imath}(sz,\varsigma,n) & = \widehat{\texttt{int}} & \text{if } n \text{ cannot be represented by } \texttt{int}(8,\varsigma)\\
\widehat{\imath}(sz,\varsigma,n) & = \imath(sz,\varsigma,n) & \text{otherwise}\\
\tau_1 \widehat{\Uparrow} \tau_2 & = \widehat{\texttt{int}} & \text{if either } \tau_1 \text{ or } \tau_2 \text{ is } \widehat{\texttt{int}}\\
\tau_1 \widehat{\Uparrow} \tau_2 & = \tau_1 \Uparrow \tau_2 & \text{otherwise.}
\end{array}
$$

Dynamic Semantics. The operational semantics of CSubMP is consistent with that of CSub only if $\mathcal{C}$, $\blacklozenge_u$ and $\blacklozenge_b$ are replaced by $\widehat{\mathcal{C}}$, $\widehat{\blacklozenge_u}$ and $\widehat{\blacklozenge_b}$, respectively.

$$
\begin{array}{lll}
\widehat{\mathcal{C}}(\mathbb{Z}(n),\widehat{\texttt{int}}) & = & \mathbb{Z}(n)\\
\widehat{\mathcal{C}}(v,\tau) & = & \mathcal{C}(v,\tau)\\
\widehat{\blacklozenge_u}(\tilde{},\mathbb{Z}(n),\widehat{\texttt{int}}) & = & \oslash\\
\widehat{\blacklozenge_u}(-,\mathbb{Z}(n),\widehat{\texttt{int}}) & = & \mathbb{Z}(-n)\\
\widehat{\blacklozenge_u}(\Diamond_u,\mathbb{Z}(n),\tau) & = & \blacklozenge_u(\Diamond_u,\mathbb{Z}(n),\tau).
\end{array}
$$

Fig. 10 shows the definition of $\widehat{\blacklozenge_b}$. Generally, the result of $\widehat{\blacklozenge_b}(\Diamond_b,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\tau_1,\tau_2,\widehat{\texttt{int}})$ is consistent with that of $\Diamond_b$ over n_1 and n_2 under mathematical arithmetic. Moreover, bitwise logical operations are undefined over multi-precision integers.

3.4 Transformation from CSub to CSubMP

The goal of code transformation is to utilize multi-precision integers in place of the bounded integers of built-in types such that certain arithmetic operations can have results consistent with those under mathematical integer arithmetic.

The code transformation rules are listed in Fig. 11. For each rule, the top part contains auxiliary computations and precondition checks. If successful, the transformation at the bottom is performed. The transformation operation is denoted by $\twoheadrightarrow$. For each syntactic element se, we use $\widehat{se}$ to denote its transformation. Two notations are introduced in the transformed code, $\texttt{BCH}(\widehat{e},\tau)$ and $\texttt{BCV}(\widehat{e},\tau)$ such that

$$
\begin{array}{ll}
\texttt{BCH}(\mathbb{Z}(n),\tau) = \mathbb{Z}(n) & \text{if } n \in [\underline{\tau},\overline{\tau}]\\
\texttt{BCH}(\mathbb{Z}(n),\tau) = \oslash & \text{otherwise}\\
\texttt{BCV}(\mathbb{Z}(n),\tau) = \mathcal{C}(\mathbb{Z}(n),\tau). &
\end{array}
$$

$$
\begin{array}{lll}
\widehat{\blacklozenge_b}(+,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\widehat{\tau}_1,\widehat{\tau}_2,\widehat{\texttt{int}}) = \mathbb{Z}(n_1+n_2) & & \text{analogous for - and *}\\
\widehat{\blacklozenge_b}(/,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\widehat{\tau}_1,\widehat{\tau}_2,\widehat{\texttt{int}}) = \mathbb{Z}(\mathsf{trunc}(n_1/n_2)) & \text{if } n_2 \neq 0 & \\
\widehat{\blacklozenge_b}(\%,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\widehat{\tau}_1,\widehat{\tau}_2,\widehat{\texttt{int}}) = \mathbb{Z}(n_1-n_2\cdot\mathsf{trunc}(n_1/n_2)) & & \\
\widehat{\blacklozenge_b}(+,\mathbb{P}(a,\delta),\mathbb{Z}(n),\tau,\widehat{\texttt{int}},\tau) = \mathbb{P}(a,\delta+n) & & \text{same as } \widehat{\blacklozenge_b}(+,\mathbb{Z}(n),\mathbb{P}(a,\delta),\widehat{\texttt{int}},\tau,\tau)\\
\widehat{\blacklozenge_b}(-,\mathbb{P}(a,\delta),\mathbb{Z}(n),\tau,\widehat{\texttt{int}},\tau) = \mathbb{P}(a,\delta-n) & & \\
\widehat{\blacklozenge_b}(<,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\widehat{\tau}_1,\widehat{\tau}_2,\tau) = \mathbb{Z}(1) & \text{if } n_1 < n_2 & \text{analogous for <= and ==}\\
\widehat{\blacklozenge_b}(<,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\widehat{\tau}_1,\widehat{\tau}_2,\tau) = \mathbb{Z}(0) & \text{if } n_1 < n_2 \text{ does not hold} & \text{analogous for <= and ==}\\
\widehat{\blacklozenge_b}(\&,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\widehat{\tau}_1,\widehat{\tau}_2,\widehat{\texttt{int}}) = \oslash & & \text{analogous for | and } \hat{}\\
\widehat{\blacklozenge_b}(<<,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\widehat{\texttt{int}},\widehat{\tau}_2,\widehat{\texttt{int}}) = \mathbb{Z}(n_1\times 2^{n_2}) & \text{if } n_2 \geq 0 & \\
\widehat{\blacklozenge_b}(>>,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\widehat{\texttt{int}},\widehat{\tau}_2,\widehat{\texttt{int}}) = \mathbb{Z}(n_1\times 2^{-n_2}) & \text{if } n_2 \geq 0 & \\
\widehat{\blacklozenge_b}(<<,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\tau,\widehat{\texttt{int}},\tau) = \mathcal{C}(\mathbb{Z}(n_1\times 2^{n_2}),\tau) & \text{if } \tau \text{ is unsigned, } n_2 \geq 0 \text{ and } n_2 \leq \mathcal{W}(n_1,\tau) & \\
\widehat{\blacklozenge_b}(<<,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\tau,\widehat{\texttt{int}},\tau) = \mathbb{Z}(n_1\times 2^{n_2}) & \text{if } \tau \text{ is signed, } n_1 \geq 0,\ n_2 \geq 0,\ n_2 \leq \mathcal{W}(n_1,\tau) & \\
\widehat{\blacklozenge_b}(>>,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\tau,\widehat{\texttt{int}},\tau) = \mathbb{Z}(\lfloor n_1\times 2^{-n_2}\rfloor) & \text{if } n_2 \geq 0 \text{ and } n_2 \leq \mathcal{W}(n_1,\tau) & \\
\widehat{\blacklozenge_b}(\Diamond_b,\mathbb{Z}(n_1),\mathbb{Z}(n_2),\widehat{\tau}_1,\widehat{\tau}_2,\widehat{\tau}) = \oslash & \text{otherwise} & \text{at least one of } \widehat{\tau}_1,\widehat{\tau}_2,\widehat{\tau} \text{ is of } \widehat{\texttt{int}}\\
\widehat{\blacklozenge_b}(\Diamond_b,v_1,v_2,\tau_1,\tau_2,\tau) = \blacklozenge_b(\Diamond_b,v_1,v_2,\tau_1,\tau_2,\tau) & \text{if none of } \tau_1,\tau_2,\tau \text{ are } \widehat{\texttt{int}} &
\end{array}
$$

Fig. 10. The definition of $\widehat{\blacklozenge_b}$.

$$\frac{}{d^*;F^*;\texttt{main}=id \twoheadrightarrow d^*;\widehat{F^*};\texttt{main}=id} \quad (31) \qquad \frac{}{\tau\ id(d_1^*)\ \{d_2^*;s\} \twoheadrightarrow \tau\ id(d_1^*)\ \{\widehat{d_2^*};\widehat{s}\}} \quad (32) \qquad \frac{id \in D_v}{\tau\ id \twoheadrightarrow \widehat{\texttt{int}}\ id';\tau\ id} \quad (33)$$

$$\frac{x \in D_v \qquad \mathcal{T}(x)=\tau}{x=e \twoheadrightarrow x'=(\widehat{\texttt{int}})\widehat{e};x=\texttt{BCV}(x',\tau)} \quad (34\text{A}) \qquad \frac{P(p)\cap D_v=\{x_1,\ldots,x_t\} \qquad \mathcal{T}(*p)=\tau \qquad \mathcal{I}(\tau)}{*p=e \twoheadrightarrow *\widehat{p}=\texttt{BCH}(\widehat{e},\tau);x_1'=(\widehat{\texttt{int}})x_1;\ldots x_t'=(\widehat{\texttt{int}})x_t} \quad (34\text{B})$$

$$\frac{e_1 \notin D_v \qquad \mathcal{T}(e_1)=\tau \qquad \mathcal{I}(\tau)}{e_1=e_2 \twoheadrightarrow \widehat{e_1}=\texttt{BCH}(\widehat{e_2},\tau)} \quad (34\text{C}) \qquad \frac{\text{cases other than (34A-34C)}}{e_1=e_2 \twoheadrightarrow \widehat{e_1}=\widehat{e_2}} \quad (34\text{D})$$

$$\frac{\mathcal{T}(e_1)=\tau_1 \qquad \mathcal{T}(e_2)=\tau_2 \qquad \mathcal{I}(\tau_1)\wedge\neg\mathcal{I}(\tau_2) \qquad P^*(e_2)\cap D_v=\{x_1,\ldots,x_t\}}{e_f(e_1,e_2) \twoheadrightarrow \widehat{e_f}(\texttt{BCH}(\widehat{e_1},\tau_1),\widehat{e_2});x_1'=(\widehat{\texttt{int}})x_1;\ldots x_t'=(\widehat{\texttt{int}})x_t} \quad (35) \qquad \frac{}{s_1;s_2 \twoheadrightarrow \widehat{s_1};\widehat{s_2}} \quad (36)$$

$$\frac{}{\texttt{if}(e)\ s_1\ \texttt{else}\ s_2 \twoheadrightarrow \texttt{if}(\widehat{e})\ \widehat{s_1}\ \texttt{else}\ \widehat{s_2}} \quad (37) \qquad \frac{}{\texttt{while}(e)\ s \twoheadrightarrow \texttt{while}(\widehat{e})\ \widehat{s}} \quad (38) \qquad \frac{\mathcal{T}(e)=\tau \qquad \mathcal{I}(\tau)}{\texttt{return}\ e \twoheadrightarrow \texttt{return}\ \texttt{BCH}(\widehat{e},\tau)} \quad (39\text{A})$$

$$\frac{\text{cases other than (39A)}}{\texttt{return}\ e \twoheadrightarrow \texttt{return}\ \widehat{e}} \quad (39\text{B}) \qquad \frac{id \in D_v}{id \twoheadrightarrow id'} \quad (40) \qquad \frac{}{*e \twoheadrightarrow *\widehat{e}} \quad (41) \qquad \frac{}{e.id \twoheadrightarrow \widehat{e}.id} \quad (42) \qquad \frac{\mathcal{I}(\tau) \qquad \mathcal{I}(\mathcal{T}(e))}{(\tau)e \twoheadrightarrow \widehat{e}} \quad (43\text{A})$$

$$\frac{\text{cases other than (43A)}}{(\tau)e \twoheadrightarrow (\tau)\widehat{e}} \quad (43\text{B}) \qquad \frac{}{\texttt{-}e \twoheadrightarrow \texttt{-}((\widehat{\texttt{int}})\widehat{e})} \quad (44) \qquad \frac{\mathcal{T}(e)=\tau}{\tilde{}e \twoheadrightarrow \tilde{}\texttt{BCV}(\widehat{e},\tau)} \quad (45)$$

$$\frac{\Diamond_b \notin \{\texttt{\&},\texttt{|},\texttt{\^{}}\} \qquad \mathcal{T}(e_1)=\tau_1 \qquad \mathcal{T}(e_2)=\tau_2 \qquad \mathcal{I}(\tau_1)\wedge\mathcal{I}(\tau_2)}{e_1\Diamond_b e_2 \twoheadrightarrow ((\widehat{\texttt{int}})\widehat{e_1})\Diamond_b\widehat{e_2}} \quad (46) \qquad \frac{\Diamond_b \in \{\texttt{\&},\texttt{|},\texttt{\^{}}\} \qquad \mathcal{T}(e_1)=\tau_1 \qquad \mathcal{T}(e_2)=\tau_2}{e_1\Diamond_b e_2 \twoheadrightarrow \texttt{BCV}(\widehat{e_1},\tau_1)\Diamond_b\texttt{BCV}(\widehat{e_2},\tau_2)} \quad (47)$$

$$\frac{\Diamond_b \in \{\texttt{+},\texttt{-}\} \qquad \mathcal{T}(e_1)=\tau_1 \qquad \mathcal{T}(e_2)=\tau_2 \qquad \mathcal{P}(\tau_1)\vee\mathcal{P}(\tau_2)}{e_1\Diamond_b e_2 \twoheadrightarrow \widehat{e_1}\Diamond_b\widehat{e_2}} \quad (48) \qquad \frac{e \notin D_v}{\texttt{\&}e \twoheadrightarrow \texttt{\&}\widehat{e}} \quad (49) \qquad \frac{}{e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \twoheadrightarrow \widehat{e_1}\ \texttt{?}\ \widehat{e_2}\ \texttt{:}\ \widehat{e_3}} \quad (50)$$

Fig. 11. The code transformation rules. D_v denotes the set of local integer variables, P maps an expression to its possibly points-to l-values. P^* maps an expression to l-values over one or more points-to relations, formally $P^*(e)=\{e_l \mid e_l \in P(e'),\text{ where } e'=e \text{ or } e'\in P^*(e)\}$.

BCH denotes a checked conversion that filters out values that are not representable in the given type, while BCV denotes a unchecked conversion that fits the given value in the specified type. The two notations can be regarded as macros, which are replaced with code at compile time.

We briefly explain the code transformation rules. Rule (31) is applied to programs, and rule (32) is applied to function definitions. As rule (33) shows, the declaration of the integer variable id is elevated by creating a multi-precision variable id' while maintaining the original one. The former is used for multi-precision arithmetic operations, and the latter is used for pointer manipulation. For the correctness of transformed programs, the consistency between an ordinary variable and its elevated form should be carefully maintained by *value update*. For the assignment to a local integer variable (i.e., $x=e$), we assign $\widehat{e}$ to $\widehat{x}$ and update x accordingly. The reasons for the design choice that x is updated with the unchecked conversion on $\widehat{x}$ are as follows: (1) if a checked conversion is employed instead, the program would halt once x overflows; (2) the ordinary variable is used only for transforming operations involving pointer dereference; (3) even if x and $\widehat{x}$ are inconsistent, their common lower bits are the same. For the assignment to an integer pointer (i.e., $*p=e$ in rule (34B)) where the points-to set of p possibly contains integer variables $x_1,\ldots,x_t$, we update each $\widehat{x_i}$ with respect to x_i because pointer manipulations only involve ordinary variables. For integer assignment where the left-hand side (LHS) is not a variable, we check if the elevated right-hand-side can fit into the type of the LHS. Rule (35) shows a special case of the rule for function calls. A function call is transformed by (1) inserting sanity checks on the elevated arguments of integer types; (2) performing value updates for local integer variables which are possibly modified via pointer arguments. We omit the rules for function call assignments because they are simply combinations of rules (34A-34D, 35). Rules (39A-39B) are applied to return statements. We add sanity checks on return expressions of integer types to ensure that their values can fit into the return types. Rules (45, 47) cast operands of bitwise logical operations to their original types. The other rules are self-explanatory. For the cases other than function call assignments not covered by the rules listed in Fig. 11, identity transformation is applied.

In a word, our approach elevates the arithmetic precision of a program by (1) replacing local integer variables with multi-precision variables; (2) replacing arithmetic operations with multi-precision ones. Each elevated variable has its ordinary variable retained for operations on the certain variable addressed through pointers. The consistency between elevated variables and ordinary variables are carefully maintained via value updates.

The listed rules imply that code transformation does not alter (1) signatures of functions, (2) types of global variables, and (3) definitions of structured types. This design decision (1) makes function-wise code transformation possible, as values spanning multiple function scopes have their precisions fixed; and (2) prevents incorrect pointer arithmetic over data structures.

We employ intra-procedural, Anderson's algorithm [26] based pointer analysis to infer points-to relations. Interprocedural analysis is unnecessary as only local variables are elevated. From the perspective of abstract interpretation, the abstract domain is $D: L \rightarrow 2^L$, where L is the set of l-values. For the soundness and efficiency of the analysis, when $|D(l)| \geq 4$, we set $D(l)$ as L, which is based on the observation that most of the points-to sets have sizes less than 4 [27]. When l points to an integer value, $D(l) \subseteq D_v$. As an optimization, if a local integer variable x does not

appear in any points-to sets, there is no need to maintain x's original declaration, thus no value updates are required.

In the following, we establish two important properties of code transformation. First, the transformed code is well-typed. Second, the transformed code behaves differently only on inputs that trigger integer errors and/or wraparounds in the original program.

Lemma 1. *Let e be an expression where $\tau = \mathcal{T}(e)$ and $\neg\mathcal{I}(\tau)$. If $e \twoheadrightarrow \widehat{e}$, then we have $\mathcal{T}(\widehat{e}) = \tau$.*

Lemma 2. *Let e be an expression where $\tau = \mathcal{T}(e)$ and $\mathcal{I}(\tau)$. If $e \twoheadrightarrow \widehat{e}$, then we have*

1) *$\mathcal{T}(\widehat{e}) = \tau$ when e is a non-variable l-value;*
2) *$\widehat{\mathcal{I}}(\mathcal{T}(\widehat{e}))$ otherwise.*

Proof Sketch. We use $P_{\mathsf{rl}}(e)$ and $P_{\mathsf{l}}(e)$ to denote Lemmas 1 and 2, respectively. $P_{\mathsf{rl}}(e)$ and $P_{\mathsf{l}}(e)$ are proved by induction over expression transformation rules (40-50). The full proof is presented in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2018.2866388. □

Theorem 3. *The transformed code is well-typed.*

Proof Sketch. We prove the following statements using Lemmas 1 and 2: (1) for each assignment, its LHS and RHS (right-hand side) have identical types; (2) the type of a function argument and the type of its corresponding parameter are identical; and (3) the type of a return value is identical to the return type of the enclosing function. The full proof can be referred in Appendix B, available in the online supplemental material. □

The program states of the transformed program are not directly comparable with those of the original program because of the elevated integer variables. To make them comparable, we propose the concept of *elevated program state*.

Definition 1. *Let P be a program and $P \twoheadrightarrow \widehat{P}$. (Δ_g, Δ_f, M) is a state for P while $(\widehat{\Delta_g}, \widehat{\Delta_f}, \widehat{M})$ is a state for $\widehat{P}$. We say $(\widehat{\Delta_g}, \widehat{\Delta_f}, \widehat{M})$ is an* elevated program state *of (Δ_g, Δ_f, M) if:*

(1) $\Delta_g = \widehat{\Delta_g}$;
(2) $\Delta_f \subseteq \widehat{\Delta_f}$, *for each* $(id', a) \in \widehat{\Delta_f} \setminus \Delta_f$, $\Delta_f(id)$ *is defined;*
(3) $M \subseteq \widehat{M}$, *and* $\widehat{M} \setminus M = \{(a, \{((0, \widehat{\texttt{int}}), v)\}) \mid (id', a) \in \widehat{\Delta_f} \setminus \Delta_f, v = M(\Delta_f(id))(0, \mathcal{T}(id))\}$.

Lemma 4. *Given a program state (Δ_g, Δ_f, M) and an expression e such that:*

1) *$\Delta_g, \Delta_f, M \vdash e \Rightarrow v$ where $v \neq \oslash$;*
2) *for each $\mathcal{C}(\mathbb{Z}(n), \tau)$ used in evaluating e where $\mathcal{I}(\tau)$ holds, we have $\mathcal{C}(\mathbb{Z}(n), \tau) = \mathbb{Z}(n)$.*

Suppose $e \twoheadrightarrow \widehat{e}$ and $(\widehat{\Delta_g}, \widehat{\Delta_f}, \widehat{M})$ is an elevated program state of (Δ_g, Δ_f, M), then:

1) *$\widehat{\Delta_g}, \widehat{\Delta_f}, \widehat{M} \vdash \widehat{e} \Rightarrow v$;*
2) *if e is an l-value other than a local integer variable, then we have $\Delta_g, \Delta_f, M \vdash e \Leftarrow l$ and $\widehat{\Delta_g}, \widehat{\Delta_f}, \widehat{M} \vdash \widehat{e} \Leftarrow l$.*

Proof Sketch. We use $P_v(e)$ and $P_l(e)$ to denote propositions (1) and (2), respectively. Let $\epsilon(e)$ and $\widehat{\epsilon}(e)$ be the evaluated values of e under the program states (Δ_g, Δ_f, M) and $(\widehat{\Delta_g}, \widehat{\Delta_f}, \widehat{M})$, respectively. $P_v(e)$ and $P_l(e)$ are proved by induction over expression kind. The full proof is presented in Appendix C, available in the online supplemental material. □

$$\frac{}{(\tau)e \twoheadrightarrow (\tau)\widehat{e}} \quad (43')$$

$$\frac{\lozenge_b \in \{+, -, *\} \qquad \mathcal{T}(e_1) = \tau_1 \qquad \mathcal{T}(e_2) = \tau_2 \qquad \mathcal{I}(\tau_1) \wedge \mathcal{I}(\tau_2) \qquad \mathcal{T}(e_1 \lozenge_b e_2) \text{ is unsigned}}{e_1 \lozenge_b e_2 \twoheadrightarrow \texttt{BCV}(\widehat{e_1}, \tau_1) \lozenge_b \texttt{BCV}(\widehat{e_2}, \tau_2)} \quad (46'\text{A})$$

$$\frac{\lozenge_b \in \{<<, >>\} \qquad \mathcal{T}(e_1) = \tau_1 \qquad \mathcal{T}(e_2) = \tau_2 \qquad \mathcal{I}(\tau_1) \wedge \mathcal{I}(\tau_2) \qquad \mathcal{T}(e_1 \lozenge_b e_2) \text{ is unsigned}}{e_1 \lozenge_b e_2 \twoheadrightarrow \texttt{BCV}((\widehat{\texttt{int}})\widehat{e_1} \lozenge_b \widehat{e_2}, \mathcal{T}(e_1 \lozenge_b e_2))} \quad (46'\text{B})$$

$$\frac{\mathcal{T}(e_1) = \tau_1 \qquad \mathcal{T}(e_2) = \tau_2 \qquad \mathcal{I}(\tau_1) \wedge \mathcal{I}(\tau_2) \qquad \text{cases other than (46'A,46'B)}}{e_1 \lozenge_b e_2 \twoheadrightarrow ((\widehat{\texttt{int}})\widehat{e_1}) \lozenge_b \widehat{e_2}} \quad (46'\text{C})$$

Fig. 12. New transformation rules.

Suppose P is a CSub program and $P \twoheadrightarrow \widehat{P}$, we assume that P and $\widehat{P}$ allocate the same abstract addresses for variables and function parameters in common. This assumption is plausible via careful abstractions on P and $\widehat{P}$. Theorem 4 implies that given the initial program state (Δ_g, Δ_f, M) for P and its elevated state $(\widehat{\Delta_g}, \widehat{\Delta_f}, \widehat{M})$ for $\widehat{P}$, after respective executions of P and $\widehat{P}$, the resultant state for $\widehat{P}$ is still the elevated state of the resultant state for P.

Theorem 5. *The transformed code has no Type I integer errors.*

Proof. According to rules (44-48) and the definitions of $\widehat{\blacklozenge_u}$ and $\widehat{\blacklozenge_b}$, Type I integer errors possibly occur in the transformed expressions by rules (44, 46).

For the expression $-((\widehat{\texttt{int}})\widehat{e})$ produced by rule (44), suppose that $\widehat{\epsilon}((\widehat{\texttt{int}})\widehat{e}) = \mathbb{Z}(n) \neq \oslash$, then $\widehat{\epsilon}(-((\widehat{\texttt{int}})\widehat{e})) = \mathbb{Z}(-n) \neq \oslash$.

For the expression $((\widehat{\texttt{int}})\widehat{e_1}) \lozenge_b \widehat{e_2}$ (denoted by $\widehat{e}$) produced by rule (46), suppose that $\widehat{\epsilon}((\widehat{\texttt{int}})\widehat{e_1}) = \mathbb{Z}(n_1) \neq \oslash$ and $\widehat{\epsilon}(\widehat{e_2}) = \mathbb{Z}(n_2) \neq \oslash$, according to the definition of $\widehat{\blacklozenge_b}$, we have:

1) when $\lozenge_b \in \{/, \%\}$ and $n_2 \neq 0$, $\widehat{\epsilon}(\widehat{e}) \neq \oslash$;
2) when $\lozenge_b \in \{ <<, >> \}$ and $n_2 \geq 0$, $\widehat{\epsilon}(\widehat{e}) \neq \oslash$;
3) otherwise, $\widehat{\epsilon}(\widehat{e}) \neq \oslash$. □

3.5 Intentional Wraparounds

Defined wraparounds (e.g., unsigned integer overflow) can be intentionally used for application-specific purposes, such as random generation, thus they are not always integer errors. In fact, it is generally impossible to determine whether a defined wraparound is misused without proper specifications. Although there are some proposed solutions for recognizing intentional wraparounds purely on source code [28], [29], they are neither sound nor complete. To prevent introducing errors on intentional wraparounds, code transformation rules are customizable to enable or disable precision elevation on explicit type casts and unsigned overflows. More specifically, when precision elevation on explicit type casts is disabled, rules (43A-43B) are replaced with rule (43') in Fig. 12; when precision elevation on unsigned overflows is disabled, rule (46) is replaced by rules (46'A-46'C) in Fig. 12. The new transformation rules ensure that Theorems 3, 4 and 5 hold.

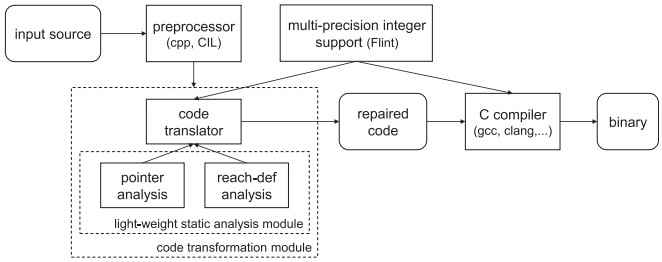


Fig. 13. Schematic overview of CIntFix.

4 Implementation Issues

4.1 Overview

We implemented a CIntFix prototype to automatically repair C integer errors via source-to-source code transformation. Fig. 13 illustrates the architecture. CIntFix takes C source code as input and uses cpp (the C preprocessor in GCC) to preprocess the code. Next, the code translator parses the preprocessed code using CDT (a C/C++ development tooling based on the Eclipse platform) and performs rule-based code transformation. The code translator works with the light-weight static analysis module, which consists of intra-procedural pointer analysis and critical distance analysis (Section 4.3). The former is indispensable for maintaining the consistency between elevated and ordinary integer variables. The latter is employed for selective code transformation close to security-critical program sites. The transformed code can be accepted by standard C compilers to produce binaries.

Flint [30], a fast number theory library, is chosen as the implementation of multi-precision integers. Compared to GMP [31], Flint elevates integer precision on demand and thus is more efficient for computations at low precision.

The source code of CIntFix and the experimental data are publicly available at https://git.io/vEnHz.

4.2 From CSubMP to C

It is generally straightforward to apply transformation rules on C code because CSub maintains the majority of C features. For a C expression with side effects (excluding function calls), we transform the expression into multiple CSub expressions or assignments separated by comma operators, then perform transformation on each operand. For an assignment with implicit conversion, we add an explicit cast on the RHS to the type of the LHS prior to transformation. For other implicit conversions unsupported by CSub, the transformation rules in CSub are directly applied as if there were no implicit conversions. It is trivial to transform C-specific statements such as `switch-case` statements. Finally, utilizing Flint integers underlies the major challenge in code transformation. First, Flint integers differ from the built-in integers in that they need to be created and destroyed explicitly. To prevent memory leak issues, the code translator moves local variable declarations to the beginning of the function body and inserts invocations of `fmpz_clear` to free the memory of the elevated variables before exiting the current function. Block-scoped variables are renamed if necessary to prevent name collisions.

Second, Flint APIs are procedure styled. Consider the addition function `fmpz_add()`:

$$\frac{\mathcal{I}(\tau) \qquad id \in I_v}{\tau\ id \twoheadrightarrow \widehat{\texttt{int}}\ id'; \tau\ id} \ (33'') \qquad \frac{id \in I_v}{id \twoheadrightarrow id'} \ (40'')$$

$$\frac{x \in I_v \qquad \mathcal{T}(x) = \tau}{x = e \twoheadrightarrow x' = (\widehat{\texttt{int}})\widehat{e}; x = \texttt{BCV}(x', \tau)} \ (34''\text{A})$$

$$\frac{P(p) \cap I_v = \{x_1, \ldots, x_t\} \qquad \mathcal{T}(*p) = \tau}{*p = e \twoheadrightarrow *\widehat{p} = \texttt{BCH}(\widehat{e}, \tau); \widehat{x_1} = (\widehat{\texttt{int}})x_1; \ldots \widehat{x_t} = (\widehat{\texttt{int}})x_t} \ (34''\text{B})$$

$$\frac{e_1 \notin I_v \qquad \mathcal{T}(e_1) = \tau \qquad \mathcal{I}(\tau)}{e_1 = e_2 \twoheadrightarrow \widehat{e_1} = \texttt{BCH}(\widehat{e_2}, \tau)} \ (34''\text{C})$$

$$\frac{\begin{array}{c}\mathcal{T}(e_1) = \tau_1 \qquad \mathcal{T}(e_2) = \tau_2 \\ \mathcal{I}(\tau_1) \wedge \neg\mathcal{I}(\tau_2) \qquad P^*(e_2) \cap I_v = \{x_1, \ldots, x_t\}\end{array}}{e_f(e_1, e_2) \twoheadrightarrow \widehat{e_f}(\texttt{BCH}(\widehat{e_1}, \tau_1), \widehat{e_2}); x_1' = (\widehat{\texttt{int}})x_1; \ldots x_t' = (\widehat{\texttt{int}})x_t} \ (35'')$$

Fig. 14. Rules for critical-site-directed transformation.

```
void fmpz_add(fmpz_t f,
      const fmpz_t g, const fmpz_t h);
```

The result of $\texttt{g} + \texttt{h}$ is stored in `f` instead of the function return value, thus several Flint arithmetic operations cannot be chained directly. To transform a compound expression, auxiliary variables need to be introduced to store intermediate results. For memory efficiency, the fewer auxiliary variables introduced the better. CIntFix numbers auxiliary variables and reuses them by always picking the available variable with the smallest number. Theorem 6 gives an upper bound for the number of required auxiliary variables.

Theorem 6. *Given a compound expression where the maximum arity of its containing operators is k and the depth of its abstract syntax tree is d, it is sufficient to introduce $\max\{1, (k-1)d\}$ auxiliary variables. Note that the call to an n-ary function $(n \geq 0)$ is regarded as an n-ary operation.*

Proof. Let e be the considered compound expression. If $k = 0$, one variable is required to store the value of e. Otherwise, assume $e ::= \Diamond(e_1, \ldots, n_t)$, where $\Diamond$ is the top-level operator with t arity. By induction on k and d, in the worst case, we need to use $(t-1)$ variables to store values of $e_1, \ldots, e_{t-1}$ and $\max\{1, (k-1)(d-1)\}$ variables to compute e_t. Thus, $N = (t-1) + \max\{1, (k-1)(d-1)\}$ auxiliary variables are sufficient. When $t = k$, we have $N = \max\{k, (k-1)d\} = \max\{1, (k-1)d\}$. □

4.3 Critical Distance Analysis

Although Flint is highly optimized, Flint arithmetic operations are still much more expensive than the built-in arithmetic operations because they involve branching instructions and heap memory management. CIntFix supports selectively elevating the precision for expressions close to the security-critical program sites at which attacks are performed to the subject programs. Security-critical sites include condition expressions, array subscripts, return expressions and function arguments, which follows the notions in related work [2], [32].

Let $G = (V, E)$ be the use-def graph. V is the set of defining nodes and usage nodes. A defining node $v_d(x, pc)$ refers to an assignment that assigns the l-value x at program location pc. Although function argument passing and function return involve value defining, they are excluded in the use-def graph because our code transformation is intra-procedural. A usage node $v_u(x, pc)$ refers to an assignment or an expression that

```
int square(int x) {
  fmpz_t __MP_x, __MP_INTERM_0;
  int __ret;
  fmpz_init(__MP_x);
  fmpz_init(__MP_INTERM_0);
  fmpz_mul(__MP_INTERM_0, __MP_x, __MP_x);
  __ret = __CHECK_MP_int(__MP_INTERM_0);
  fmpz_clear(__MP_x);
  fmpz_clear(__MP_INTERM_0);
  return __ret;
}
int main(void) {
  fmpz_t __MP_val, __MP_ret;
  fmpz_init(__MP_val);
  fmpz_init(__MP_ret);
  fmpz_set_si(__MP_val, INT_MAX);
  fmpz_set_si(__MP_ret,
      square(CHECK_MP_int(__MP_val)));
  printInt(CHECK_MP_int(__MP_ret));
  fmpz_clear(__MP_val);
  fmpz_clear(__MP_ret);
  return 0;
}
```

Fig. 15. The transformation of the example program by CIntFix.

uses the value of x defined at program location pc. For each $v_1 = v_d(x_1, pc)$ and $v_2 = v_u(x_2, pc)$, we have $(v_2, v_1) \in E$ if x_1 and x_2 possibly refer to the same memory location.

Definition 2. *Consider a use-def graph $G = (V, E)$ with $\mathcal{D} : V \to \mathbb{N} \cup \{+\infty\}$ which assigns each $v \in V$ to its* critical distance. *A* critical expression *is at a security-critical site. For the node of a critical expression $v_e \in V$, we have $\mathcal{D}(v_e) = 0$. For an assignment node $v_s \in V$, we have $\mathcal{D}(v_s) = k$ if $k = \min\{d(v_e, v_s) \mid \mathcal{D}(v_e) = 0\}$, where $d(v_e, v_s)$ is the geodesic distance from v_e to v_s in G. $d(v_e, v_s) = +\infty$ if v_s is unreachable from v_e.*

CIntFix employs backwards data-flow analysis to construct the use-def graph and annotate each node with the critical distance. We can specify a threshold κ ($\kappa \geq 0$) such that only the following nodes are marked for transformation: (1) critical expressions and (2) assignments with critical distances no larger than κ. Let I_v be the set of integer variables to be elevated, then $id \in I_v$ iff each $v_d(id, \_) \in V$ is marked. Fig. 14 introduces new rules for critical-site-directed transformation. For variable declarations, rule (33) is replaced with rule (33″). For the critical expression e on the site of target type τ (an array subscript has the target type `ptrdiff_t` which is an integer type alias defined in `stddef.h`) and $\mathcal{I}(\tau)$ holds, then $e \twoheadrightarrow \texttt{BCH}(\widehat{e}, \tau)$. For marked assignments, rules (34A-34C, 35, 40) are replaced by rules (34″A-34″C, 35″, 40″) accordingly, where each occurrence of D_v is replaced with I_v in the new rules. For other unmarked expressions and assignments, the transformation is identity.

In short, critical-site-directed code transformation selectively elevates precisions of expressions and assignments sufficiently close to security-critical sites with respect to geodesic distance in use-def graph. A local integer variable is replaced with a multi-precision variable iff each of its defining node is marked to be elevated.

Critical-site-directed transformation has three important properties. First, Theorems 3 and 4 still hold for the transformed code. Second, selective code transformation may miss some integer errors. Finally, when κ is sufficiently large such that $I_v = D_v$, critical-site-directed transformation is equivalent to global code transformation.

TABLE 3
The Case Study Results on the CVE Database

tolerability	#bugs (ratio)	category	#bugs (ratio)
tolerable	322 (43.0%)	overflow	229 (71.1%)
		signedness	74 (23.0%)
		length	19 (5.9%)
intolerable	426 (57.0%)	IO2BO	361 (84.7%)
		other sinks	65 (15.3%)

4.4 Running Example

Fig. 15 shows the transformation of the example code given by Fig. 3a when κ is 1. The critical expressions are underlined in Fig. 3a. In `main`, when $\kappa = 1$, all the definitions of `val` and `ret` are marked for elevation, thus $I_v = \{\texttt{val}, \texttt{ret}\}$. Since none of the integer variables appear in points-to sets, the ordinary variables are not retained. It is noteworthy that a return variable `_ret` is created in `square` because we have to store the return value before destroying multi-precision integers. `CHECK_MP_int` is defined in our sanity check library and used to examine whether the given multi-precision integer fits into `int`. If so, the converted `int` value is returned; otherwise, the execution switches to an error handler.

5 Experimental Evaluation

We evaluate CIntFix on NIST's Juliet Test Suite [33] and the SPEC CINT2000 benchmarks for both effectiveness and efficiency. All experiments are conducted on a PC running 64-bit Ubuntu 16.04, with an Intel Core i5-3470@3.20 GHz CPU and 32 GB memory. In the evaluation, we focus on the following questions:

- **Q1** Does it make sense to classify integer errors as tolerable and intolerable in real-world source code?
- **Q2** How accurate is CIntFix at repairing C integer errors?
- **Q3** How efficient is CIntFix at transforming the code?
- **Q4** How much overhead (in terms of runtime and source size) is introduced in the repaired code?

5.1 Tolerability of Real-World Integer Errors

Our work is motivated by a central observation that integer errors can be categorized by tolerability. Tolerable errors can be repaired by program functionality recovery via precision elevation. To substantiate this point, we investigate real-world integer vulnerabilities in the CVE database [34]. Our case study includes 748 CVE identifiers from 214 applications, which have sufficient details on source code.

As Table 3 shows, 322 of 748 integer vulnerabilities (43.0 percent) are tolerable. The most common type of tolerable error is general overflow (71.1 percent, including overflow and underflow), followed by sign error (23.0 percent) and unexpected truncation/sign extension (5.7 percent). It is remarkable that 62 of 322 tolerable errors (19.3 percent) are caused by faulty integer error preventions. Although developers are often aware of integer errors, manual prevention is error-prone.

TABLE 4
CWE Entries for Evaluation

CWE	description	#program	#file	KLOC
190	integer overflow	3,420	5,040	440.749
191	integer underflow	2,622	3,864	340.884
194	unexpected sign extension	912	1,344	108.742
195	signed to unsigned conversion error	912	1,344	106.006
196	unsigned to signed conversion error	18	18	1.547
197	numeric truncation error	684	1,008	67.629
680	integer overflow to buffer overflow	228	318	26.911

Approximately 56.8 percent of integer vulnerabilities are intolerable, and the majority of these errors (87.2 percent) are IO2BO (integer-overflow-to-buffer-overflow) errors. More specifically, an IO2BO error involves overflow on the size argument of a memory operation such as `malloc`. The remaining vulnerabilities involve other program sites that cannot have their precisions elevated.

Summary. For *Q1*, the classification of tolerability provides a better understanding for integer errors. A total of 43.0 percent of the studied real-world integer errors are tolerable, thus they can be repaired by precision elevation instead of other aggressive remedies, such as sanitization, which leads to program functionality removal.

5.2 Evaluation on the Juliet Test Suite

NIST's Juliet Test Suite [33] is a collection of test bench programs in C/C++ created by the NSA Center for Assured Software and developed specifically for assessing the capabilities of program analysis tools. The latest version (v1.3) released on Oct. 2017 contains 64,099 cases for 118 different CWEs. Our experiments are based on the latest version.

Experimental Design. There are 7 CWE entries related to integer errors, as shown in Table 4. A total of 8796 test programs are chosen for evaluation with C++ programs excluded. Each of test program contains one bad function with exactly one defect and multiple good functions that are mutations of the bad function without defects. To validate fixes for a program, we sample up to 20 inputs that trigger the target defect for the bad function and up to 20 inputs for each good function. Failures on validating good (resp. bad) functions contribute to false positives (resp. false negatives). The validation process is conducted on a virtual machine under 32-bit Ubuntu 16.04 with 4 GB memory. Time measures are the average time of 10 repeated executions. In CIntFix, κ is parameterized as 1, and precision elevation on explicit type casts and unsigned overflows is enabled.

```
1 unsigned int data;
2 data = -2;
3 printUnsignedInt(data);
```

Fig. 16. The kernel code of false negatives.

Experimental Results. Table 5 shows the evaluation results on the Juliet Test Suite. The accuracy results are shown in Columns 2-8, where Column *FN* reports false-negative rates, Column *G* lists the number of good functions, Column *CG* lists the number of corrupted good functions and Column *FP* reports the false-negative rates. The efficiency results are shown in Columns 9-10, where *TT* denotes the time cost of code transformation and Column *KLOC/s* reports the rate of code transformation in KLOC/s. The last 6 columns report the overhead results, where Column *INC* lists the blow-up rate in terms of source size, and Column *OH* lists the runtime slowdown of the repaired code.

The accuracy results show that CIntFix succeeds in repairing all the bad functions of the benchmark programs, whereas 459 of 20,383 good functions are corrupted (thus, the overall false-negative rate is 2.3 percent). No false positives occur when critical-site-directed code transformation with $\kappa = 1$ is performed, which substantiates that integer errors and critical program sites typically have strong locality along use-def chains. The false negatives all originate from CWE 191, and the kernel code is shown in Fig. 16. In line 2, the unsigned integer variable `data` is assigned with a negative value such that it is initialized with a large value. However, the transformed code throws an error message instead of printing the value of `data` in line 3 because -2 does not fit into `unsigned int`. If precision elevation on explicit type casts is disabled, the code transformation has no false positives, but none of the bad functions in CWE 197 can be correctly repaired (thus, the overall false-negative rate is 10.1 percent). The kernel code of false positives is shown in Fig. 17. The lossy truncation in line 3 is expected as an integer error, whereas the expected result of `charData` cannot be recovered without precision elevation.

The efficiency results show that, on average, CIntFix processes source code at a rate of 5.587 KLOC/s, which substantiates its scalability to large-scale code bases. CIntFix's efficiency can be explained as follows. First, CIntFix transforms the internal integer model of a C program

TABLE 5
Results on the Juliet Test Suite

CWE	#repaired/#total			FN	G	CG	FP	TT (s)	KLOC/s	KLOC			runtime (s)		
	tolerable	intolerable	Σ							old	new	INC	old	new	OH
190	3420/3420	0/0	3420/3420	0.0%	9360	0	0.0%	76.167	5.787	440.749	501.663	12.0%	0.063	0.067	6.3%
191	2622/2622	0/0	2622/2622	0.0%	7176	459	6.4%	59.350	5.744	340.884	387.835	13.5%	0.048	0.051	6.3%
194	0/0	912/912	912/912	0.0%	1272	0	0.0%	17.772	6.119	108.742	115.401	6.1%	0.024	0.026	8.3%
195	0/0	912/912	912/912	0.0%	1272	0	0.0%	18.403	5.760	106.006	114.375	7.9%	0.020	0.023	15.0%
196	18/18	0/0	18/18	0.0%	31	0	0.0%	0.876	1.766	1.547	1.664	7.6%	0.002	0.002	0.0%
197	684/684	0/0	684/684	0.0%	954	0	0.0%	15.143	4.466	67.629	74.609	10.3%	0.018	0.019	5.6%
680	0/0	228/228	228/228	0.0%	318	0	0.0%	7.824	3.440	26.911	29.233	8.6%	0.010	0.010	0.0%
Σ	6744/6744	2052/2052	8796/8796	0.0%	20383	459	2.3%	195.535	5.587	1092.468	1224.780	12.1%	0.185	0.198	7.0%

```
1 int data;
2 data = /* from user space */
3 char charData = (char) data;
4 printChar(charData);
```

Fig. 17. The kernel code of false positives when precision elevation on explicit type casts is disabled.

towards a multi-precision model in a syntax-directed manner. Second, static analyses are intra-procedural while state merge is employed at the join points of the program paths, thus the convergence can be reached faster.

The overhead introduced by code transformation is measured in terms of two aspects: blow-up in source size, and runtime slowdown. On the one hand, the source size increases by 12.1 percent on average. Additional lines of code are contributed by (1) the creation and destruction of Flint integer variables, (2) Flint arithmetic operations derived from splitting compound expressions, and (3) value updates between ordinary and multi-precision integer variables. On the other hand, the transformed code has an average slowdown of 7.0 percent. For each test program, we measure its average time cost given the sampled inputs for repair validation while excluding the inputs that make the program crash unexpectedly (i.e., buffer overrun). The average slowdown on CWE 196 is 0.0 percent because the timing has insufficient precision to distinguish the old and new time costs.

Summary. For *Q2*, CIntFix has no false negatives, and the overall false-positive rate is 2.2 percent. For *Q3*, CIntFix processes the source code at a rate of 5.02 KLOC/s. For *Q4*, the blow-up rate of the source size is 12.1 percent, and the runtime slowdown due to code transformation is 7.0 percent on average.

5.3 Evaluation on SPEC CINT2000

SPEC CINT2000 is an industry-standardized CPU-intensive benchmark suite consisting of a selected collection of C and C++ programs designed to be representative of a wide range of real-world programs. CINT2000 is a subcomponent of SPEC CPU2000 for measuring computationally-intensive integer performance to evaluate CIntFix.

The CINT2000 benchmarks are reported to contain a collection of integer errors [12]. Since we do not have complete knowledge on where the errors are or how they could be triggered, the CINT2000 benchmark is employed to evaluate only the efficiency and overhead (*Q3* and *Q4*). Our experiments are based on version 1.2 with necessary modifications.

Experimental Design. To measure the efficiency of the transformed code, we compare the execution time of the original and repaired programs given the same 'ref' data sets as inputs. All programs, except `252.eon` (a C++ program), are used for the evaluation. Clang 3.8 with LLVM 3.8 is used to compile all programs. As in the previous experiment, κ is parameterized as 1, and precision elevation on explicit type casts and unsigned overflows is enabled. To reduce random error in the timing, time measures are the average of 10 repeated runs.

Experimental Results. The evaluation results on SPEC CINT2000 are presented in Table 6. Columns 3-5 report the size of the original/transformed code and the blow-up rate of the source size. Column *RT* lists the reference runtime of each test program. The last 3 columns report the runtime of the original/transformed code and the runtime overhead.

Columns 6-7 list the efficiency results. On average, CIntFix processes source code at a rate of 2.92 KLOC/s, which is substantially slower than that on the Juliet Test Suite. This slowdown occurs because static analysis costs more time on the benchmark programs, which generally have higher complexity in terms of control-flows and memory manipulations.

The overhead is discussed in terms of source size and runtime. The transformed code expands by 36.5 percent in KLOC. The blow-up rate of the source size on CINT2000 is substantially larger than that on the Juliet Test Suite for two main reasons. First, compound expressions are widely harbored, thus a large number of expressions are split into multiple Flint arithmetic operations by the code transformation. Second, the employed pointer analysis has insufficient expressivity in analyzing complicated memory operations (such as accessing a statically indeterminable index of an array), thus value updates have to be performed for all the elevated local integer variables. Columns 9-11 show that the average runtime slowdown is 10.5 percent.

Summary. For *Q3*, CIntFix processes the source code at an overall rate of 2.92 KLOC/s. For *Q4*, the transformed code expands by 35.6 percent in KLOC and has 10.5 percent runtime slowdown on average.

TABLE 6
Results on SPEC CINT2000

name	#file	KLOC			TT (s)	KLOC/s	RT (s)	base time (s)		
		old	new	INC (%)				old	new	OH (%)
164.gzip	20	5.615	7.803	39.0	1.946	2.89	1400	64.292	69.895	8.7
175.vpr	41	11.301	15.366	36.0	4.341	2.60	1400	43.758	49.736	13.7
176.gcc	120	149.010	199.132	33.6	47.595	3.13	1100	24.978	27.023	8.2
181.mcf	25	1.482	1.880	26.9	1.300	1.14	1800	37.614	40.988	9.0
186.crafty	43	12.901	18.674	44.7	4.209	3.07	1000	21.267	24.439	14.9
197.parser	18	7.770	10.680	37.5	3.218	2.45	1800	68.681	75.207	9.5
253.perlbmk	91	72.095	100.368	39.2	23.560	3.06	1800	48.134	54.166	12.5
254.gap	63	35.696	47.512	33.1	12.512	2.85	1100	28.146	31.380	10.4
255.vortex	123	49.225	68.577	39.3	15.867	3.10	1900	65.705	74.285	13.1
256.bzip2	2	3.228	4.726	46.4	1.383	2.33	1500	49.008	54.171	10.5
300.twolf	85	17.821	24.908	39.8	9.559	1.86	3000	69.020	74.405	7.8
Σ	631	366.144	499.626	**36.5**	125.440	**2.92**	17800	520.873	575.695	**10.5**

```
1 int data;
2 data = /* from user space */
3 if (data < 0) {
4   char * dataBuffer = (char *)malloc(data);
5   memset(dataBuffer, 'A', data - 1);
6   dataBuffer[data - 1] = '\0';
7 }
```

Fig. 18. The kernel code of the defects on which Coker's approach fails.

5.4 Discussion

Comparison with the State-of-the-Arts. Among the existing tools [15], [16], [17], [35] for automatic integer error repair, we choose Coker and Hafiz [15] for comparison because (1) it works on source code only whereas SoupInt [17] works on x86 binary, TAP [35] needs seed inputs and SIFT [16] requires annotations on input statements; (2) it converts the internal integer model of a C program towards a safer model whereas other tools focus on patch generation or input filters; and (3) it supports various types of integer errors whereas the other three tools support only integer overflows in memory operations.

For Coker and Hafiz, we contacted the authors but did not succeed in obtaining the artifact, thus the comparison is based on the experimental results on the Juliet Test Suite reported in the paper. For *Q2*, although the paper claims that all the integer errors can be repaired, we found that 380 defects in CWE 194 and 195 cannot be correctly repaired by adding explicit casts on the condition expressions. Consider the kernel code of the target defects shown in Fig. 18. Coker's approach explicitly casts `data` in line 3 as `unsigned int` to make the type of `data` consistent with its usage context in `memcpy`. However, when `data` has a value of 0, `data-1` in `memset` still makes the program suffer from a buffer overflow error. Thus, Coker's fixes are incomplete for the target defects. Furthermore, among the 6,744 tolerable errors, only 702 (all the defects in CWE 196 and 197) are tolerated by Coker's, whereas CIntFix tolerates all the tolerable errors. This is because Coker's sanitizes overflows in arithmetic operations, whereas CIntFix attempts to elevate the arithmetic precision to recover the expected results. There is no sufficient data to answer *Q3* and *Q4*.

Limitations. CIntFix does not employ range analysis to detect potential integer errors in the first place, thus some introduced code transformations may unnecessarily elevate the precision of the expressions which have no integer errors. Currently range analysis is not integrated into our approach for the following reasons. First, since CIntFix focuses on converting the input source code into code that utilizes multi-precision integer arithmetic to be compiled to the executable with integer errors eliminated, code transformation is expected as an important step of the overall compilation process and thus emphasizes efficiency.

```
1 int i, j;
2 if (C) {
3   i = N1;
4   j = N2;
5 } else {
6   i = N3;
7   j = N4;
8 }
9 i = i - j;
```

A precise integer error detection can be too expensive. Second, if we fix the precision of the expressions and assignments without integer errors according to range information, issues of inconsistency between elevated and ordinary variables may arise. Consider the program on the left. `N1` and `N2` are random values that possibly cannot fit into `int`, whereas `N3` and `N4` fit into `int`. Suppose that we elevate the precision on the assignment at line 9 while fixing the precision on the assignments at lines 6-7, then the elevated forms of `i` and `j` have incorrect values when the `else`-branch of `C` is executed. Suppose that `i-j` at line 9 is guaranteed to fit into `int` and thus the precision of the assignment at line 9 is fixed, `i-j` would be wrong when the `if`-branch of `C` is executed because `i` and `j` are inconsistent with their elevated forms which store the values of `N1` and `N2`, respectively. This is because precision elevation fails to follow the def-use order. In other words, for an assignment to an ordinary (resp. elevated) integer variable, not all its subsequent usages are ordinary (resp. elevated). Third, it is acceptable that unnecessary code transformations introduce additional blow-up in source size because the transformed code is compiler-oriented. Moreover, the experimental results show that the average runtime slowdown of transformed code is 7.0 percent on the Juliet Test Suite and 10.5 percent on SPEC CINT2000, which is acceptable for a large number of general-purpose applications.

Our approach maintains the types of global variables, the definitions of structured types and function signatures, which leads to failures of tolerating relevant integer errors. This design decision (1) makes the function-wise code transformation possible and (2) facilitates maintaining the consistency between integer variables and their elevated forms. Tolerating integer errors involving data structures is a future direction.

Finally, utilizing Flint integers is the main source of non-negligible overhead. To overcome this limitation, CIntFix supports critical-site-directed code transformation to selectively elevate the expressions and assignments close to security-critical program sites. It is possible to utilize a large bounded integer type (e.g., `int128_t`) as the multi-precision integer type based on the intuition that the need for very high precision is not strong in practice. Furthermore, it is feasible to infer sufficiently elevated types by range analysis at compile time but it is non-trivial to guarantee that the transformed code is well-typed without unexpected implicit conversions. Consider the condition expression $e_1 < e_2$ where the sufficiently elevated types for e_1 and e_2 are `long` and `unsigned long`, respectively. If we have $e_1 < e_2 \twoheadrightarrow (\texttt{long})e_1 < (\texttt{unsigned long})e_2$, the possible implicit conversion in comparison may lead to unexpected program behaviors. A possible solution is to model the inference of sufficiently elevated types as constraint solving, which may substantially reduce the efficiency of code transformation.

6 Related Work

Coker and Hafiz. Three program transformations, namely, adding integer cast (AIC), replacing arithmetic operator (RAO) and changing integer type (CIT), are proposed to repair integer problems [15]. Both Coker's and CIntFix transform the internal integer model of a C program towards a safe model. However, they have two major differences. First,

arithmetic overflows are all sanitized in Coker's, whereas CIntFix elevates arithmetic precision if certain overflow issues are tolerable. Thus, the transformed code from CIntFix is more robust. Second, Coker's utilizes built-in integer types, whereas CIntFix supports multi-precision integers, which enable it to tolerate more integer errors, such as overflows over `long long int` values.

SIFT. SIFT [16] is a static input filter generation tool for integer overflows in determining sizes for memory allocations and memory block copy operations. It employs inter-procedural, weakest precondition static analysis to propagate safety constraints backwards against the control flow and it is proved to be sound [36]. This tool has several limitations. First, SIFT supports limited types of integer errors. Second, input filters simply remove the functionalities of programs. Third, the generation of effective input filters relies on correct input statement annotations, which requires considerable insight and elaborative manual effort for real-world applications, such as web servers.

SoupInt. Given an x86 binary, SoupInt [17] leverages existing error handlers for programmer-anticipated errors to repair the unanticipated integer overflows that are exploited to attack the subject programs. The limitations of this tool are as follows. First, only integer overflows at memory allocation sites are eligible to be repaired. Second, the deployment points of patches are restricted to existing error handlers, which are not always available. Third, generated patches may have false negatives, i.e., patches cannot capture the integer overflows triggered by different executions not given by certain attack instances.

TAP. Given an application and a seed input that makes the application run normally, TAP [35] analyzes the execution to the target memory allocation sites dynamically and employs target error discovery [37] to generate inputs that trigger integer/buffer overflow errors at certain program sites. Once an integer/buffer overflow error is detected, TAP generates a patch based on templates. Our work, however, focuses on transforming the internal integer model of a program instead of patch generation. Moreover, our work supports a broader scope of integer errors than only integer overflows in memory allocations.

CodePhage. CodePhage [38] is a horizontal code transfer system that transfers correct code from donor applications to recipient applications that process the same kind of inputs. It is based on the assumption that applications of the same type should have similar behavior given input of the same kind. CodePhage succeeds in eliminating integer overflows in real-world applications. Our work differs in that CIntFix is capable of repairing integer errors in a fully automatic manner without donor source code.

KCC. KCC [39], [40] interpreter translates C programs into a Maude [41] term which is then reduced by the rewrite rules of the formal semantics of C. A total of 2,155 rewrite rules are used to model (1) the positive semantics (i.e., the semantics of correct programs) and (2) the negative semantics (i.e., the rules identifying undefined behaviors). For a C program, KCC generates an execution script that has the identical behavior to that of the native code with a real C compiler. The major differences between KCC and our work are summarized below. First, the product of KCC is a formal representation of a C program instead of native code. Second, KCC traps the execution for undefined behaviors and even implementation-defined behaviors to filter portable, strictly-conforming programs. The goal of CIntFix is different as it attempts to recover the expected behaviors from integer errors. Third, the applicability of KCC is limited by the semantics modeling of library functions.

Generic Program Repair. Generic program repair is concerned with automatically correcting the implementation by patch generation with respect to the given specifications. The majority of existing approaches are generate-and-validate systems that heuristically produce candidate patches and validate them against test suites. GenProg [18] uses genetic programming (GP) to search for program variants that retain required functionality but are not vulnerable to the target defect. AE [19] leverages an approximate semantic equivalence relation to reduce the search space. RSRepair [42] replaces GP with random search. PAR [43] generates patches based on predefined templates summarized from successful human-written patches. SPR [20] deploys a set of parameterized transformation schemas and uses target value search to quickly filter out schemas where no parameter value exists to produce a successful repair. Prophet [21] learns a probabilistic, application-independent model for correct code from human-written patches to guide patch space exploration. Some approaches use program synthesis to generate patch code. SemFix [44] and MintHint [45] replace the buggy expression with a symbolic value, and then use symbolic execution [46] to synthesize a new expression that passes all test cases. DirectFix [47] leverage MaxSMT optimization to maximally preserve the structure of the original code. SPR [20] uses a condition synthesis algorithm to generate condition expressions that match the expected branch directions derived from the test suites.

Generic program repair techniques have some fundamental limitations. First, the quality of repair heavily relies on the completeness of the test suite which is usually insufficient in practice. Second, generated patches rarely ensure the correctness due to limitations in the patch validation infrastructure [22] and patch space. Third, generate-and-validate systems often require hours of search to find a plausible patch for a realistic bug. Our work solves a different problem as we transform the internal integer model of C programs, which requires global code transformation. Furthermore, our work has substantially stronger scalability to large code bases.

Integer Error Detection. A number of advances have been made in integer error detection. Many numerical abstract domains [7], [8], [9], [48], [49], [50], [51] have been proposed to support reasoning on numerical program properties and they vary in expressivity/cost trade-off. PREfix+Z3 [52] combines the static analyzer PREfix [53] and the bit-precise SMT solver Z3 [54] and is applied to find integer bugs in several Microsoft products. Kint generates constraints from source code and user annotations, and then feeds them to a constraint solver to determine whether an integer error possibly occurs. SmartFuzz [13], IntPatch [55] and DIODE [37] use symbolic execution to generate test cases that trigger integer errors. IntScope [32] performs path-sensitive dataflow analysis in a few interesting program paths by leveraging taint analysis. IntTracker [56] combines taint analysis and dirty value propagation analysis to detect harmful IO2BO vulnerabilities. RICH [2] captures integer errors based on safe integer

semantics derived by applying sub-typing theory. BRICK [57], RA [58] and IOC [12] instrument the program on either source or binary level to capture integer errors at runtime. Model checkers such as BLAST [59] and LLBMC [11] convert source code into logical representations that are fed to a SMT solver for bit-precise reasoning. Safe integer libraries such as SafeInt [60] and Ranged Integer [61] are used to wrap arithmetic operations. Our work differs in that CIntFix focuses on repairing integer errors by tolerance or sanitization, instead of simply reporting integer errors.

7 Conclusion

In this paper, we propose a novel approach to convert the input C source code into code that utilizes multi-precision integer arithmetic to be compiled to the executable with integer errors eliminated. Multi-precision integers can precisely simulate mathematical integers such that a large portion of integer errors can be tolerated (i.e., repaired by program functionality recovery). The code transformation employed in our approach is proved to (1) produce well-typed code and (2) be conservative such that program behaviors are preserved given inputs that do not trigger wraparound in the original program. Since our approach is integrated into the overall compilation process, efficiency is emphasized. We implement our approach as a prototype CIntFix, and the evaluation results substantiate its effectiveness in tolerating tolerable errors and sanitizing intolerable errors and its scalability to complex large-scale code bases.

Some future directions are summarized below. First, range analysis can be leveraged to generate more optimized code. Second, the properties of code transformation can be machine-checked with a theorem prover. Third, it would be interesting to leverage external specifications (e.g., test suites) to precisely handle the misuse of defined behaviors.

Acknowledgments

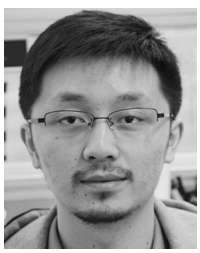
This research is sponsored in part by the NSFC Program (No. 61527812), the National Science and Technology Major Project of China (No. 2016ZX01038101), the MIIT Funds of China (Research and Application of TCN Key Technologies), the National Science and Technology Support Program of China (No. 2015BAG14B01-02) and the National Key R&D Program of China (No. 2016QY07X1402).

References

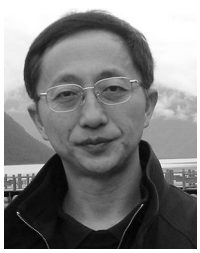
[1] ISO/IEC, ISO International Standard ISO/IEC 9899: 2011 - Information technology - Programming languages - C, ISO, Vernier, Geneva, Switzerland, 2011.
[2] D. Brumley, D. X. Song, T. Chiueh, R. Johnson, and H. Lin, "RICH: Automatically protecting against integer-based vulnerabilities," in *Proc. Annu. Netw. Distrib. Syst. Security Symp.*, 2007. pp. 1–13.
[3] N. Y. Times, "F.A.A. orders fix for possible power loss in boeing 787." [Online]. Available: http://www.nytimes.com/2015/05/01/business/faa-orders-fix-for-possible-power-loss-in-boeing-787.html, 2015.
[4] CVE, "CVE-2015-7869," 2015. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7869
[5] S. Christey and R. A. Martin, "Vulnerability type distributions in CVE," 2007. [Online]. Available: http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf
[6] S. Christey, B. Martin, M. Brown, A. Paller, and D. Kirby, "2011 CWE/SANS top 25 most dangerous software errors," 2011. [Online]. Available: http://cwe.mitre.org/top25/
[7] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Berlin, Germany: Springer, 1999.
[8] A. Miné, "The octagon abstract domain," *Higher-Order Symbolic Comput.*, vol. 19, no. 1, pp. 31–100, 2006.
[9] L. Chen, A. Miné, and P. Cousot, "A sound floating-point polyhedra abstract domain," in *Proc. Asian Symp. Program. Lang. Syst.*, 2008, pp. 3–18.
[10] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with KINT," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 163–177.
[11] F. Merz, S. Falke, and C. Sinz, "LLBMC: Bounded model checking of C and C++ programs using a compiler IR," in *Proc. Verified Softw.: Theories Tools Experiments*, 2012, pp. 146–161.
[12] W. Dietz, P. Li, J. Regehr, and V. S. Adve, "Understanding integer overflow in C/C++," *ACM Trans. Softw. Eng. Methodology*, vol. 25, no. 1, pp. 2:1–2:29, 2015.
[13] D. Molnar, X. C. Li, and D. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *Proc. 18th Conf. USENIX Security Symp.*, 2009, pp. 67–82.
[14] A. Miné and D. Delmas, "Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software," in *Proc. Int. Conf. Embedded Softw.*, 2015, pp. 65–74.
[15] Z. Coker and M. Hafiz, "Program transformations to fix C integers," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 792–801.
[16] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. C. Rinard, "Sound input filter generation for integer overflow errors," in *Proc. 41st ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, 2014, pp. 439–452.
[17] T. Wang, C. Song, and W. Lee, "Diagnosis and emergency patch generation for integer overflow exploits," in *Proc. Int. Conf. Detection Intrusions Malware Vulnerability Assessment*, 2014, pp. 255–275.
[18] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan./Feb. 2012.
[19] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2013, pp. 356–366.
[20] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 166–178.
[21] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, 2016, pp. 298–312.
[22] Z. Qi, F. Long, S. Achour, and M. C. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 24–36.
[23] X. Cheng, M. Zhou, X. Song, M. Gu, and J. Sun, "Automatic fix for C integer errors by precision improvement," in *Proc. IEEE 40th Annu. Comput. Softw. Appl. Conf.*, 2016, pp. 2–11.
[24] C. Price, *MIPS IV Instruction Set*. Sunnyvale, CA, USA: MIPS Technologies, 1995.
[25] S. Blazy and X. Leroy, "Mechanized semantics for the clight subset of the C language," *J. Automated Reasoning*, vol. 43, no. 3, pp. 263–288, 2009.
[26] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, DIKU, Univ. Cophenhagen, Copenhagen, Denmark, 1994.
[27] W. Lee, H. Oh, and K. Yi, "A progress bar for static analyzers," in *Proc. Int. Static Anal. Symp.*, 2014, pp. 184–200.
[28] 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 *Proc. 30th Annu. Comput. Security Appl. Conf.*, 2014, pp. 416–425.
[29] H. Sun, X. Zhang, Y. Zheng, and Q. Zeng, "IntEQ: Recognizing benign integer overflows via equivalence checking across multiple precisions," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 1051–1062.
[30] W. B. Hart, "Fast library for number theory: An introduction," in *Proc. Int. Congress Math. Softw.*, 2010, pp. 88–91.
[31] GNU, "The GNU MP Bignum Library." 2018. [Online]. Available: https://gmplib.org
[32] T. Wang, T. Wei, Z. Lin, and W. Zou, "IntScope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution," in *Proc. Annu. Netw. Distrib. Syst. Security Symp.*, 2009. pp. 1–14.
[33] NIST, "Juliet test suite," 2017. [Online]. Available: https://samate.nist.gov/SRD/testsuite.php

[34] CVE, "CVE list master copy." 2018. [Online]. Available: https://cve.mitre.org/cve/cve.html
[35] S. Sidiroglou-Douskos, E. Lahtinen, and M. Rinard, "Automatic discovery and patching of buffer and integer overflow errors," MIT CSAIL, Cambridge, MA, Tech. Rep. MIT-CSAIL-TR-2015–018, 2015.
[36] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard, "Sound input filter generation for integer overflow errors," MIT CSAIL, Cambridge, MA, Tech. Rep. MIT-CSAIL-TR-2013-018, 2013.
[37] S. Sidiroglou-Douskos, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. C. Rinard, "Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement," in *Proc. 20th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2015, pp. 473–486.
[38] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2015, pp. 43–54.
[39] C. Ellison and G. Rosu, "An executable formal semantics of C with applications," in *Proc. 39th Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Lang.*, 2012, pp. 533–544.
[40] C. Hathhorn, C. Ellison, and G. Rosu, "Defining the undefinedness of C," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2015, pp. 336–345.
[41] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada, "Maude: Specification and programming in rewriting logic," *Theoretical Comput. Sci.*, vol. 285, no. 2, pp. 187–243, 2002.
[42] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 254–265.
[43] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 802–811.
[44] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 772–781.
[45] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "MintHint: Automated synthesis of repair hints," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 266–276.
[46] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 209–224.
[47] S. Mechtaev, J. Yi, and A. Roychoudhury, "DirectFix: Looking for simple program repairs," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, 2015, pp. 448–458.
[48] A. Simon, A. King, and J. M. Howe, "Two variables per linear inequality as an abstract domain," in *Proc. Int. Workshop Logic-Based Program Synthesis Transformation*, 2002, pp. 71–89.
[49] R. Clarisó and J. Cortadella, "The octahedron abstract domain," in *Proc. Int. Static Anal. Symp.*, 2004, pp. 312–327.
[50] B. Jeannet and A. Miné, "Apron: A library of numerical abstract domains for static analysis," in *Proc. Int. Conf. Comput. Aided Verification*, 2009, pp. 661–667.
[51] G. Singh, M. Püschel, and M. T. Vechev, "Fast polyhedra abstract domain," in *Proc. 44th ACM SIGPLAN Symp. Principles Program. Lang.*, 2017, pp. 46–59.
[52] Y. Moy, N. Bjørner, and D. Sielaff, "Modular bug-finding for integer overflows in the large: Sound, efficient, bit-precise static analysis," Microsoft Research, Redmond, WA, Tech. Rep. MSR-TR-2009-57, 2009.
[53] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Softw. Practice Experience*, vol. 30, no. 7, pp. 775–802, 2000.
[54] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.
[55] C. Zhang, W. Zou, T. Wang, Y. Chen, and T. Wei, "Using type analysis in compiler to mitigate integer-overflow-to-buffer-overflow threat," *J. Comput. Security*, vol. 19, no. 6, pp. 1083–1107, 2011.
[56] H. Sun, X. Zhang, C. Su, and Q. Zeng, "Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability," in *Proc. 10th ACM Symp. Inf. Comput. Commun. Security*, 2015, pp. 483–494.
[57] P. Chen, Y. Wang, Z. Xin, B. Mao, and L. Xie, "BRICK: A binary tool for run-time detecting and locating integer-based vulnerability," in *Proc. Int. Conf. Availability Rel. Security*, 2009, pp. 208–215.
[58] 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 *Proc. IEEE/ACM Int. Symp. Code Generation Optimization*, 2013, pp. 33:1–33:11.
[59] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast," *Int. J. Softw. Tools Technol. Transfer*, vol. 9, no. 5/6, pp. 505–525, 2007.
[60] D. LeBlanc, "SafeInt," 2014. [Online]. Available: https://safeint.codeplex.com/
[61] J. Gennari, S. Hedrick, F. Long, J. Pincar, and R. C. Seacord, "Ranged integers for the c programming language," CMU SEI, Pittsburgh, PA, Tech. Rep. CMU/SEI-2007-TN-027, 2007.

Xi Cheng received the BS and PhD degrees in software engineering from Tsinghua University, Beijing, China, in 2013 and 2018, respectively. He is currently working as a member of Technical Staff at VMware, Inc. His main research interests include constraint solving, static analysis, automatic program repair and software-defined network.



Min Zhou received the BS degree in mathematics and the PhD degree in computer science from Tsinghua University, Beijing, China, in 2007 and 2014, respectively. He is currently working as a lecturer with the School of Software, Tsinghua University, Beijing, China. His research interests include model checking, program analysis and testing.



Xiaoyu Song received the PhD degree from the University of Pisa, Italy, in 1991. From 1992 to 1998, he was on the faculty with the University of Montreal, Canada. He joined the Department of Electrical and Computer Engineering, Portland State University, in 1998, where he is now a professor. He was an editor of the *IEEE Transactions on VLSI Systems* and the *IEEE Transactions on Circuits and Systems*. He was awarded an Intel Faculty Fellowship from 2000 to 2005. His research interests include formal methods, design automation, embedded systems and emerging technologies.

Ming Gu received the BS degree in computer science from the National University of Defense Technology, China, in 1984, and the MS degree in computer science from the Chinese Academy of Science, China, in 1986. She is currently a professor with the School of Software, Tsinghua University, China. Her research interests include software formal methods, software trustworthiness and middleware technology.

Jiaguang Sun received the BS degree in automation science from Tsinghua University, China, in 1970. He is currently a professor with the School of Information Science and Technology, Tsinghua University, China. He is also a member of the Chinese Academy of Engineering and is director of the Tsinghua National Laboratory for Information Science and Technology, China. His research interests include computer graphics, computer-aided design, formal verification and software engineering.