2016 IEEE 40th Annual Computer Software and Applications Conference

Automatic Fix for C Integer Errors by Precision Improvement

Xi Cheng[1,2,3], Min Zhou[1,2,3], Xiaoyu Song[4], Ming Gu[1,2,3], Jiaguang Sun[1,2,3]
[1]*Key Laboratory for Information System Security, MoE, China*
[2]*Tsinghua National Laboratory for Information Science and Technology (TNList), China*
[3]*School of Software, Tsinghua University, China*
[4]*Electrical and Computer Engineering, Portland State University, USA*
chengxi13@mails.tsinghua.edu.cn, {mzhou,guming,sunjg}@tsinghua.edu.cn, song@ece.pdx.edu

***Abstract*—Integer errors in C program may lead to serious failures and vulnerabilities. They are harbored in a wide range of programs including mature software such as Linux kernel. Code reviewing is laborious and cannot guarantee reliable fixes for errors. Addressing potential errors in the development phase is error-prone even for experts and essentially hinders developing efficiency. In this paper we propose a novel approach to automate fix for C integer errors. Our approach directly replaces original C integers with dynamic-precision integers to fix potential errors without detecting them in advance. Many errors can be fixed by precision improvement without changing the design of application. We implement a tool CIntFix to automatically fix C integer errors. CIntFix succeeds in fixing all 5414 programs in NIST's Juliet test suite from 7 weakness categories. Meanwhile, on Juliet test suite and SPEC CINT2000 benchmarks, CIntFix processes C source code at the rate of 0.157s/KLOC and the fixed programs have 18.0% slowdown on average. The results show that CIntFix is capable to fix integer errors in real-world C programs.**

***Keywords*-integer error, tolerability, code transformation**

I. Introduction

Integer errors in C program include overflow, underflow, sign misinterpretation and lossy truncation. These errors originate from bounded representation in memory, where an integer is stored with a fixed number of bits (bit-vector). C integer errors can lead to serious software failures. The failure of the Ariane 5 launcher in 1996 [1] was caused by a lossy truncation from 64-bit floating point to 16-bit signed integer. A potential problem of unexpected total power loss in Boeing 787 Dreamliners [2] reported in 2015 was caused by an overflow error of a 32-bit signed integer. Moreover, C integer errors can also become a source of serious vulnerabilities. For example, if an overflow occurs in calculation of determining how much memory to allocate, the program can possibly access more memory than expected which results in buffer overflow. Such errors can be found in many widely-used software such as Firefox [3] and Flash [4], and they allow attackers to obtain critical information, escalate privilege and execute arbitrary malicious codes. Integer overflow error is one of the main threats to security of software applications. According to a CVE report in 2007 [5], integer overflow error is the second most common vulnerability in the advisories for OS vendor. MITRE also places integer overflow in the "Top 25 Most Dangerous Software Errors" in 2011 [6].

It is challenging to reliably address integer errors in C program and the reasons are twofold. First, overflow

```
void qsort(int arr[], int left, int right)
{
  int i = left, j = right;
  int pivot = arr[(left + right) / 2];

  /* partition and recursion */
}
```

Listing 1. A simplified implementation of quicksort. `(left + right)` overflows when both `left` and `right` are assigned with large values.

and inconsistent conversion are insufficient to be witnesses of integer errors because they are language features. For example, overflow of unsigned integer is well-defined and particularly common in device drivers and cryptographic applications. Second, developers are often aware of possible errors in program and protect critical sites such as memory allocations with sanitization. If such protection works, suspicious integer behaviors are anticipated and should not be regarded as errors. Furthermore, fixing integer errors correctly is quite difficult due to complicated semantics of C integers. A sign misinterpretation error in gzip was addressed incorrectly for multiple times, and was finally fixed correctly 17 years after its introduction [7].

Due to its importance, researchers have shown great concern for addressing C integer errors. Many tools and techniques are proposed for detection and repair of integer errors in source code [7]–[10] or binaries [11]–[13]. However, they have several drawbacks in general. First, most of them focus on overflow error while sign misinterpretation and lossy truncation are seldom discussed in detail. Second, most techniques are limited on detecting and preventing integer errors while they are unable to guide developers to revise faulty code correctly. Third, existing dynamic techniques are too aggressive for tolerable errors. Consider the code snippet shown in Listing 1. If the array to be sorted is very long, `(left + right)` possibly overflows while the index never exceeds the range of `int`. This issue can be addressed by rewriting expression or elevating types of operands, but unfortunately many approaches would capture this error and directly halt the program.

In this paper, we propose a novel approach to fix C integer errors automatically by elevating precision of integer, i.e. enlarging the range of integer where error may occur. This approach brings several benefits. First, an integer variable improves its precision when assigned with the value out of its range. As a result, this variable can avoid storing an inconsistent value due to wraparound or implicit conversion. Second, integer errors are addressed in a lazy manner. All overflows and problematic conversions are tolerated with high precision integer operations until we reach a critical site such as function call and memory operation. Laziness makes our approach failure-oblivious, i.e. having a reasonable acceptability of continued execution. Third, potential errors can be fixed without being detected by employing sophisticated and expensive analysis in advance.

Our contributions are highlighted as follows.

1) We define the concept of *tolerability*. The *tolerable* error can be fixed by elevating integer precision but the *intolerable* ones cannot.
2) We propose a code transformation technique to fix faulty programs. Built-in integer types are replaced with multi-precision type where the error is tolerable. For intolerable errors, sanitization is employed to ensure that the error does not propagate elsewhere.
3) We implement a prototype tool CIntFix to automate the above fix technique. Experiment results show that it can correctly fix all errors in 5414 test programs from 7 categories in Juliet test suite. It takes less than 7 seconds to process every one thousand lines of code and fixing induces about 18.0% runtime overhead. The results indicate that our method is promising for practical use.

The rest of paper is organized as follows. Section II defines integer error and its tolerability. Section III outlines code transformation technique. Section IV describes design of CIntFix and implementation issues. Section V evaluates CIntFix using standard benchmarks. Section VI surveys related work. Finally, Section VII summarizes our work.

II. Tolerability of Integer Errors

In C language, an integer is represented by a fixed-size bit-vector. In bit-vector logic, arithmetic semantics depends on *width* and *encoding* while the former refers to the length of bit-vector and the latter refers to the method of interpreting a bit-vector as numerical value. For a n-bit bit-vector $x = \overline{a_{n-1} \dots a_0}$, *binary encoding* interprets it as an unsigned integer $\langle x \rangle_U$ while *two's complement* interprets it as a signed integer $\langle x \rangle_S$, as follows.

$$\begin{aligned} \langle x \rangle_U &= \sum_{i=0}^{n-1} a_i \cdot 2^i \\ \langle x \rangle_S &= -2^{n-1} \cdot a_{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i \end{aligned}$$

Due to the finite encoding of integers, arithmetic over fixed-size bit-vector is different from that over $\mathbb{Z}$. For example, $\forall x, y.(x - y > 0) \Leftrightarrow (x > y)$ does not hold when x, y are n-bit bit-vectors with signed encoding and one of the counterexamples is $x = \overline{00 \dots 0}$ and $y = \overline{11 \dots 1}$. Developers have to deal with complex bit-vector semantics in every programming, which is quite error-prone.

A. A Kernel Language: IKL

To facilitate our discussion of C, we introduce an imperative kernel language IKL. It is an abstraction of C's subset (shown in Figure 1). A IKL program is a collection of functions where the main one initiates the execution. Its arguments are supplied from input stream and the value returned is appended to the output stream. Functions take any number of arguments and return a single value. The function body consists of statements and a `return` operation. The `output` statement writes an 64-bit signed integer to the output stream. The `vari` statement declares a 64-bit signed integer and initialized as 0 while `varp` statement declares an empty pointer. Expressions in IKL may have side effects. The `input` expression reads a 64-bit signed integer from input stream. The comparison operation yields 0 for false and 1 for true. `malloc` allocates a single heap cell with the size of 64 bits and returns its pointer. We also model three kinds of conversions: (1) cast from the signed to unsigned (`s2u`) and vice versa (`u2s`); (2) `trunc` operation to truncate an integer by a bit; (3) `ext` operation to extend an integer by a bit. IKL is well-typed with the following restrictions: (1) binary operators are applied to integers only; (2) the value of a controller expression must be 0 or 1; (3) `*` operator only applies to pointers; (4) only functions can be called.

We introduce the concept of *program state*. Given a program P, we use $\mathbb{X}$ to denote the set of qualified names of identifiers. A state of P is denoted by a binary tuple (Δ, σ) where $\Delta : \mathbb{X} \to \mathbb{Z}$ maps names to integers and $\sigma \in 2^{\mathbb{X} \times \mathbb{X}}$ stores point-to relations. Initially, all names in $\mathbb{X}$ are undefined in Δ and $\sigma = \emptyset$. A variable appears in Δ and σ after it is created and is removed after the program counter leaves its scope. Program state depicts execution context determining evaluation result along with side effects of an expression and execution consequence of a statement.

The operational semantics of IKL is shown in Figure 2. An expression can be either an integer or a pointer. An integer expression is evaluated as a numerical value while a pointer expression is evaluated as a pointer. We explain the rule of function call here since it is the most complicated one. To evaluate a function call, we first evaluate its arguments of which results are passed to corresponding parameters. In the scope of function x, we use qualified names of parameters in Δ and σ. By executing function body and evaluating return expression, we obtain the final evaluation result along with the modified program state. Notice that we omit the case of calling a function pointer. In this case, we first look up the actual function to be called in σ. Figure 2 also lists rules of all statements. Assignments (including x = e and $*x$ = e) are discussed by the type of left-hand expression because they change program state in different manners.

B. Integer Error and its Tolerability

Generally speaking, an integer error occurs when a certain computation in program produces a result inconsistent with that under arithmetic over $\mathbb{Z}$.

Definition 1 (Integer error)**.** Given a program P and its state ϵ, an *integer error* occurs in evaluating expression e:

- if $e ::= e_1 \lozenge_b e_2$ such that $\langle e_1, \epsilon \rangle \to \langle n_1, \epsilon' \rangle$, $\langle e_2, \epsilon' \rangle \to \langle n_2, \epsilon'' \rangle$, $\langle e, \epsilon \rangle \to \langle n, \epsilon'' \rangle$ and we have $n_1 \blacklozenge_b n_2 \neq n$ where $\blacklozenge_b \in \{+, -, \times, /, \mathrm{mod}, <, \leq, =\}$, $\lozenge_b$ and $\blacklozenge_b$ are corresponding operators;
- if $e ::= \texttt{cast}(e_1)$ such that $\langle e_1, \epsilon \rangle \to \langle n_1, \epsilon' \rangle$, $\langle e, \epsilon \rangle \to \langle n, \epsilon' \rangle$ and we have $n \neq n_1$ where `cast`

$$
\begin{array}{ll}
E ::= & n \mid x \mid E \Diamond_b E \mid \texttt{(}E\texttt{)} \mid \texttt{input} \mid \texttt{\&}x \mid \texttt{malloc} \mid \texttt{*}E \mid \texttt{null} \mid E(E,\ldots,E) \mid \texttt{u2s(}E\texttt{)} \mid \texttt{s2u(}E\texttt{)} \mid \texttt{trunc(}E\texttt{)} \mid \texttt{ext(}E\texttt{)} \\
S ::= & \texttt{skip;} \mid x \texttt{ = } E\texttt{;} \mid \texttt{*}x \texttt{ = } E\texttt{;} \mid \texttt{output } E\texttt{;} \mid S\ S \mid \texttt{if(}E\texttt{)}\ \{S\}\ \texttt{else}\ \{S\} \mid \texttt{while(}E\texttt{)}\ \{S\} \mid \texttt{vari } x\texttt{;} \mid \texttt{varp } x\texttt{;} \\
F ::= & x(x,\ldots,x)\ \{S\ \texttt{return } E\texttt{;}\} \\
P ::= & F\ldots F
\end{array}
$$

E: expression S: statement F: function P: program $x \in \mathbb{I}$ $n \in \mathbb{Z}$ $\Diamond_b \in \{\texttt{+},\texttt{-},\texttt{*},\texttt{/},\texttt{\%},\texttt{<},\texttt{<=},\texttt{==}\}$

Figure 1. A imperative kernel language IKL. $\mathbb{I}$ and $\mathbb{Z}$ denote the set of identifiers and integer values, respectively.

NOTATIONS:

$\langle e, \epsilon\rangle \to \langle n, \epsilon'\rangle$: evaluation of e in state ϵ yields the result n along with new state ϵ'

$\langle s, \epsilon\rangle \to \epsilon'$: execution of statement s in state ϵ terminates in the final state ϵ'

EXPRESSIONS:

$$
\frac{}{\langle n,\epsilon\rangle \to \langle n,\epsilon\rangle}
\qquad
\frac{}{\langle x,\epsilon\rangle \to \langle \Delta(x),\epsilon\rangle}\ \textsc{Int}
\qquad
\frac{}{\langle x,\epsilon\rangle \to \langle x,\epsilon\rangle}\ \textsc{Ptr}
\qquad
\frac{(e,x)\in\sigma}{\langle \texttt{*}e,\epsilon\rangle \to \langle \Delta(x),\epsilon\rangle}\ \textsc{Int}
\qquad
\frac{(e,p)\in\sigma}{\langle \texttt{*}e,\epsilon\rangle \to \langle p,\epsilon\rangle}\ \textsc{Ptr}
$$

$$
\frac{\begin{array}{c}\langle e_1,\epsilon\rangle \to \langle n_1,\epsilon'\rangle \\ \langle e_2,\epsilon'\rangle \to \langle n_2,\epsilon''\rangle \\ \langle n_1\Diamond_b n_2,\epsilon''\rangle \to \langle n,\epsilon''\rangle\end{array}}{\langle e_1\Diamond_b e_2,\epsilon\rangle \to \langle n,\epsilon''\rangle}
\qquad
\frac{\begin{array}{c}\langle e,\epsilon\rangle \to \langle n,\epsilon'\rangle \\ \langle c\rangle_U = n \\ \langle c\rangle_S = n'\end{array}}{\langle \texttt{u2s(}e\texttt{)},\epsilon\rangle \to \langle n',\epsilon'\rangle}
\qquad
\frac{\begin{array}{c}\langle e,\epsilon\rangle \to \langle n,\epsilon'\rangle \\ \langle c\rangle_S = n \\ \langle c\rangle_U = n'\end{array}}{\langle \texttt{s2u(}e\texttt{)},\epsilon\rangle \to \langle n',\epsilon'\rangle}
\qquad
\frac{\begin{array}{c}\langle e,\epsilon\rangle \to \langle n,\epsilon'\rangle \\ \langle c_{[l]}\rangle = n \\ \langle c_{[l-1]}\rangle = n'\end{array}}{\langle \texttt{trunc(}e\texttt{)},\epsilon\rangle \to \langle n',\epsilon'\rangle}
\qquad
\frac{\begin{array}{c}\langle e,\epsilon\rangle \to \langle n,\epsilon'\rangle \\ \langle c_{[l]}\rangle = n \\ \langle ext_{[l+1]}(c_{[l]})\rangle = n'\end{array}}{\langle \texttt{ext(}e\texttt{)},\epsilon\rangle \to \langle n',\epsilon'\rangle}
$$

$$
\frac{\langle e,\epsilon\rangle \to \langle n,\epsilon'\rangle}{\langle \texttt{(}e\texttt{)},\epsilon\rangle \to \langle n,\epsilon'\rangle}
\qquad
\frac{}{\langle \texttt{\&}x,\epsilon\rangle \to \langle \texttt{\&}x,\epsilon\rangle}
\qquad
\frac{}{\langle \texttt{malloc},\epsilon\rangle \to \langle \texttt{malloc},\epsilon\rangle}
\qquad
\frac{}{\langle \texttt{null},\epsilon\rangle \to \langle \texttt{null},\epsilon\rangle}
$$

$$
\frac{\begin{array}{c}
e_{m_1},\ldots,e_{m_u} \text{ are integers, } e_{m_{u+1}},\ldots,e_{m_t} \text{ are pointers, } \{m_1,\ldots,m_t\} \text{ is a permutation of } \{1,\ldots,t\} \\
\langle e_{m_1},\epsilon\rangle \to \langle n_{m_1},\epsilon_1\rangle \quad \cdots \quad \langle e_{m_u},\epsilon_{u-1}\rangle \to \langle n_{m_u},\epsilon_u\rangle \quad \Delta' = \Delta_u[x.x_{m_1} \leftarrow n_{m_1},\ldots,x.x_{m_u} \leftarrow n_{m_u}] \\
\sigma' = \sigma_u \cup \left(\bigcup_{i=u+1}^{t} \bigcup_{(e_{m_i},y)\in\sigma_u \text{ or } e_{m_i}=\texttt{\&}y} (x.x_{m_i},y)\right) \quad \epsilon' = (\Delta',\sigma') \quad x(x_1,\ldots,x_t)\{s\ \texttt{return}\ e\} \quad \langle s,\epsilon'\rangle \to \epsilon'' \quad \langle e,\epsilon''\rangle \to \langle n,\epsilon'''\rangle
\end{array}}{\langle x(e_1,\ldots,e_t),\epsilon\rangle \to \langle n,\epsilon'''\rangle}
$$

STATEMENTS:

$$
\frac{}{\langle \texttt{skip;},\epsilon\rangle \to \epsilon}
\qquad
\frac{\Delta' = \Delta[x \leftarrow 0]}{\langle \texttt{vari } x\texttt{;},(\Delta,\sigma)\rangle \to (\Delta',\sigma)}
\qquad
\frac{}{\langle \texttt{varp } x\texttt{;},\epsilon\rangle \to \epsilon}
\qquad
\frac{\begin{array}{c}\langle s_1,\epsilon\rangle \to \epsilon' \\ \langle s_2,\epsilon'\rangle \to \epsilon''\end{array}}{\langle s_1\ s_2,\epsilon\rangle \to \epsilon''}
\qquad
\frac{\langle e,\epsilon\rangle \to \langle n,\epsilon'\rangle}{\langle \texttt{output } e\texttt{;},\epsilon\rangle \to \epsilon'}
$$

$$
\frac{\langle e,\epsilon\rangle \to \langle n,\epsilon'\rangle \quad \Delta'' = \Delta'[x \leftarrow n]}{\langle x \texttt{ = } e,\epsilon\rangle \to (\Delta'',\sigma')}\ \textsc{Int}
\qquad
\frac{\sigma' = \sigma \cup (x,p) \text{ where } (e,p)\in\sigma \text{ or } e = \texttt{\&}p}{\langle x \texttt{ = } e,\epsilon\rangle \to (\Delta,\sigma')}\ \textsc{Ptr}
$$

$$
\frac{\langle e,\epsilon\rangle \to \langle n,\epsilon'\rangle \quad (x,y)\in\sigma' \quad \Delta'' = \Delta'[y \leftarrow n]}{\langle \texttt{*}x \texttt{ = } e,\epsilon\rangle \to (\Delta'',\sigma')}\ \textsc{Int}
\qquad
\frac{\sigma' = \sigma \cup (y,p) \text{ where } (x,y)\in\sigma, (e,p)\in\sigma \text{ or } e = \texttt{\&}p}{\langle \texttt{*}x \texttt{ = } e,\epsilon\rangle \to (\Delta,\sigma')}\ \textsc{Ptr}
$$

$$
\frac{\langle e,\epsilon\rangle \to \langle 1,\epsilon'\rangle \quad \langle s_1,\epsilon'\rangle \to \epsilon''}{\langle \texttt{if(}e\texttt{)}\ \{s_1\}\ \texttt{else}\ \{s_2\},\epsilon\rangle \to \epsilon''}\ \textsc{If-T}
\qquad
\frac{\langle e,\epsilon\rangle \to \langle 0,\epsilon'\rangle \quad \langle s_2,\epsilon'\rangle \to \epsilon''}{\langle \texttt{if(}e\texttt{)}\ \{s_1\}\ \texttt{else}\ \{s_2\},\epsilon\rangle \to \epsilon''}\ \textsc{If-F}
$$

$$
\frac{\langle e,\epsilon\rangle \to \langle 1,\epsilon'\rangle \quad \langle s\texttt{;while(}e\texttt{)}\ \{s\},\epsilon'\rangle \to \epsilon''}{\langle \texttt{while(}e\texttt{)}\ \{s\},\epsilon\rangle \to \epsilon''}\ \textsc{While-T}
\qquad
\frac{\langle e,\epsilon\rangle \to \langle 0,\epsilon'\rangle}{\langle \texttt{while(}e\texttt{)}\ \{s\},\epsilon\rangle \to \epsilon'}\ \textsc{While-F}
$$

Figure 2. The operational semantics of IKL.

$\in \{\texttt{u2s}, \texttt{s2u}, \texttt{trunc}, \texttt{ext}\}$.

We denote by $\epsilon \vdash e^T$ indicating that integer errors occur in evaluation of expression e in current program state ϵ and $\epsilon \vdash e^F$ vice versa.

Integer error is *propagative*. Consider an expression e and its sub-expression e', if $\epsilon \vdash e'^T$, then $\epsilon \vdash e^T$ probably holds. For example, given $a = \overline{11111111}, b = \overline{00000001}, c = \overline{00000001}$ with unsigned encoding, we have $a + b = \overline{00000000}$. The overflow of $a + b$ leads to incorrect evaluation of $a+b+c$ though addiction of c and the result of $a+b$ produces correct result. On the other hand, integer error is *accumulative*. It is possible that an erroneous evaluation can neutralize another one to yield a correct result finally. Consider the expression $a + b - c$ on the same context as above. $a + b$ overflows and $\overline{00000000} - c = \overline{11111111}$ underflows, but the final evaluation result is correct since $\langle a+b-c\rangle_U = 63$. Hence, the relation between integer error and program bug is quite hard to determine.

Aside from analyzing integer errors by employing sophisticated techniques, a straightforward solution is to universally utilize the arithmetic of ∞-bit bit-vector with two's complement encoding in program. By this setting, arithmetic operations always yield desired results under arithmetic over $\mathbb{Z}$ while type casts are no longer required. In principle, this solution could fix most integer errors, but is infeasible for immutable parts of program. For example, passing an out-of-range value as function argument is prohibited because value of argument is bounded by the signature of corresponding function. Hence, not all kinds of integer errors could be *tolerated*, i.e. fixed by improving precision of integers. Overflow errors in function argument or value for `output` involve violation of program constraints.

Definition 2. Given a program P and its expression e, we say e is *externally constrained* if its evaluation result ranges in a specific set E, which is irrelevant to current program state. Formally, for any program states ϵ of P, we have $\langle e, \epsilon\rangle \to \langle n, \epsilon'\rangle$ and $n \in E$. We use the notation $e \Downarrow E$ to denote that e is externally constrained by the set E.

Definition 3 (Tolerability)**.** Given a program P and its state ϵ, $\epsilon \vdash e^T$ and $e \Downarrow E$. The integer error in evaluating e is *intolerable*:

- if $e ::= e_1 \lozenge_b e_2$ such that $\langle e_1, \epsilon\rangle \to \langle n_1, \epsilon'\rangle$ and $\langle e_2, \epsilon'\rangle \to \langle n_2, \epsilon''\rangle$ and we have $n_1 \blacklozenge_b n_2 \notin E$ where $\blacklozenge_b \in \{+, -, \times, /, \mathrm{mod}, <, \le, =\}$, $\lozenge_b$ and $\blacklozenge_b$ are corresponding operators;
- if $e ::= \mathtt{cast}(e_1)$ such that $\langle e_1, \epsilon\rangle \to \langle n_1, \epsilon'\rangle$ and we have $n_1 \notin E$ where $\mathtt{cast} \in \{\mathtt{u2s}, \mathtt{s2u}, \mathtt{trunc}, \mathtt{ext}\}$.

Otherwise this error is *tolerable*.

Consider three unsigned 64-bit integers x, y, z and a program state ϵ which maps x, y, z to $2^{64} - 1$,1 and 1 respectively. For the expression $x + y - z$ we have $\epsilon \vdash (x + y)^T$ and this error is tolerable. However, for statement `output` $(x + y)$ we also have $\epsilon \vdash (x + y)^T$ but this error is intolerable since $(x + y) \Downarrow [0, 2^{64} - 1]$.

An intolerable error is not intractable. Developers should be aware of external constraints on certain expression and prevent them from having an illegal value. For example, to address the issue of illegal argument in function call, one can sanitize certain argument expression to prohibit out-of-range value by halting the program with error message. Obviously, such fix jeopardizes continued execution.

III. Code Transformation

This paper describes a code transformation technique to fix C integer errors. The general idea is to utilize integers of infinite size with two's complement encoding in place of original bounded integers. Expression in critical sites such as I/O operation and function call are sanitized such that its value satisfies corresponding external constraint. In fact, we cannot precisely represent an integer of infinite size on machine. A practical solution is to use dynamic-precision value, which has limited size at initial and is extended on demand. In the following, we use the identifier with a hat (e.g. $\widehat{x}$) to denote a dynamic-precision integer value.

To simplify the presentation, we extend IKL with `fail` and conditional expression E `?` E `:` E. The expression `fail` makes the program halt when it is being evaluated and it can also be a standalone statement. The conditional expression has three operands while the first one is controller expression determining which operand is evaluated as the value of the whole expression. The operational semantics of conditional expression is formally defined in Figure 3.

$$\frac{\langle e_0, \epsilon\rangle \to \langle 1, \epsilon'\rangle \qquad \langle e_1, \epsilon'\rangle \to \langle n_1, \epsilon''\rangle}{\langle e_0 \ \texttt{?}\ e_1 \ \texttt{:}\ e_2, \epsilon\rangle \to \langle n_1, \epsilon''\rangle}\ \textsc{Cond-T}$$

$$\frac{\langle e_0, \epsilon\rangle \to \langle 0, \epsilon'\rangle \qquad \langle e_2, \epsilon'\rangle \to \langle n_2, \epsilon''\rangle}{\langle e_0 \ \texttt{?}\ e_1 \ \texttt{:}\ e_2, \epsilon\rangle \to \langle n_2, \epsilon''\rangle}\ \textsc{Cond-F}$$

Figure 3. The operational semantics of conditional expression.

A new predicate $P(e, E)$ is introduced to indicate whether the value of expression e is in the interval E. $P(e, E)$ is true under program state ϵ if and only if $\langle e, \epsilon\rangle \to \langle n, \epsilon'\rangle$ and $n \in E$ hold. $P(e, E)$ can be represented in IKL expression denoted by $\mathtt{inRange}(e, E)$.

The transformation rules for IKL program are presented in Figure 4. For each rule, the top part contains auxiliary computations and checks, if successful the transformation at bottom is performed. Given a program, we apply transformation to its syntactic elements in post-order of its abstract syntax tree. The basic idea of transformation is that, for each integer variable x in the original program, we create x and its lifted variable $\widehat{x}$ in the new program. $\widehat{x}$ is used for dynamic-precision integer operations while x is used for pointer manipulation. To keep the consistency of x and $\widehat{x}$, we synchronize their values when either of them is changed. For directly assignment (e.g. x `=` e`;`), we directly assign the value of $\widehat{e}$ to $\widehat{x}$ and update x accordingly. It is possible that $\widehat{x}$ is not representable in the original type of x, we do not regard it as an error instantly because the inconsistency matters only when x is directly used for pointer manipulation. For indirect assignment (e.g. `*`p `=` e`;`), value of $\widehat{e}$ is checked to ensure that it can be correctly stored in `*`p. If the check passes, the target of p is required for updating associated dynamic-precision variable consistently. We infer point-to relations with $\Sigma : \mathbb{X} \to 2^{\mathbb{X}}$ on the fly. Σ maps a pointer to the set of all possible targets. If $\Sigma(p) = \{q\}$ and q is an integer then we update $\widehat{q}$ accordingly because p *must* point to q. If $\Sigma(p) = \{v_1, \ldots, v_n\}$ and $n > 1$, then we synchronize each $\widehat{v_i}$ with v_i because p *may* point to any v_i.

Table I
Four patterns of intolerable errors in IKL. All boxed expressions are integers.

pattern	constraint
$e(\ldots \boxed{e_i}, \ldots)$	E_i (type of e_i)
`*`x `=` $\boxed{e}$`;`	E_x (type of `*`x)
`output` $\boxed{e}$`;`	E_o (range of valid output value)
$x(x_1, \ldots, x_t)$`{s return` $\boxed{e_r}$`;}`	E_R (type of x's return value)

By observing rules in Figure 4, we summarize four patterns of intolerable errors shown in Table I. The expression enclosed by a box is called *suspicious operand* because its error is intolerable. In transformed code, sanitizations are used to protect suspicious operands against illegal values. We implement sanitizations with a series of `check` functions. Each `check` function accepts a dynamic-precision integer, returns it unchanged if it is in the specified range and halt the program otherwise. For example, `check_E_o()` checks if accepted integer is in the range of E_o and its implementation is presented at the bottom of Figure 4.

Code transformation based on rules in Figure 4 is *conservative*. Roughly speaking, transformed code behaves differently only on inputs that trigger integer errors in the original program. To formally prove this property, we introduce a definition of sanity state for expression.

Definition 4 (Sanity state)**.** An expression e is evaluated on the state $\epsilon = (\Delta, \sigma)$. If no integer errors occur in every step of evaluation, we say ϵ is a *sanity state* for e. A statement s is executed on the state ϵ and if no integer errors occur in

$$\frac{}{n \longrightarrow n} \qquad \frac{x \text{ is an integer}}{x \longrightarrow \widehat{x}} \qquad \frac{x \text{ is a pointer}}{x \longrightarrow x} \qquad \frac{}{e_1 \Diamond_b e_2 \longrightarrow \widehat{e_1} \Diamond_b \widehat{e_2}} \qquad \frac{}{\texttt{(}e\texttt{)} \longrightarrow \texttt{(}\widehat{e}\texttt{)}} \qquad \frac{}{\texttt{input} \longrightarrow \texttt{input}} \qquad \frac{}{\texttt{malloc} \longrightarrow \texttt{malloc}}$$

$$\frac{}{\texttt{*}e \longrightarrow \texttt{*}e} \qquad \frac{}{\texttt{null} \longrightarrow \texttt{null}} \qquad \frac{}{\texttt{cast(}e\texttt{)} \longrightarrow \widehat{e}} \qquad \frac{x \text{ is an integer, its type constraint is } E_x}{\texttt{\&}x \longrightarrow \texttt{inRange(}\widehat{x}, E_x\texttt{) ? \&}x\texttt{ : fail}} \qquad \frac{x \text{ is a pointer}}{\texttt{\&}x \longrightarrow \texttt{\&}x}$$

$$\frac{e_{m_1}, \ldots, e_{m_u} \text{ are integers, their type constraints are } E_{m_1}, \ldots, E_{m_u} \text{ respectively, other arguments are pointers} \qquad 1 \le i \le u \text{ and } j > u}{e(e_1, \ldots, e_t) \longrightarrow e(\ldots \texttt{conv(check\_E\_mi(}\widehat{e_{m_i}}\texttt{))} \ldots \widehat{e_{m_j}} \ldots)}$$

$$\frac{}{\texttt{skip;} \longrightarrow \texttt{skip;}} \qquad \frac{}{s_1\ s_2 \longrightarrow \widehat{s_1}\ \widehat{s_2}} \qquad \frac{}{\texttt{vari } x\texttt{;} \longrightarrow \texttt{vari } x\texttt{;vari } \widehat{x}\texttt{;}} \qquad \frac{}{\texttt{varp x;} \longrightarrow \texttt{varp } x\texttt{;}}$$

$$\frac{e \text{ is an integer}}{x\ \texttt{=}\ e\texttt{;} \longrightarrow \widehat{x}\ \texttt{=}\ \widehat{e}\texttt{;}\ x\ \texttt{=}\ \texttt{conv(}\widehat{x}\texttt{);}}\ \textsc{Int} \qquad \frac{e \text{ is a pointer}}{x\ \texttt{=}\ e\texttt{;} \longrightarrow x\ \texttt{=}\ e\texttt{;}}\ \textsc{Ptr} \qquad \frac{E_o \text{ is the constraint on the value to be output}}{\texttt{output } e\texttt{;} \longrightarrow \texttt{output conv(check\_E\_o(}\widehat{e}\texttt{));}}$$

$$\frac{\Sigma(x) = \{y\} \qquad \texttt{*}x \Downarrow E_x \qquad y \text{ is an integer}}{\texttt{*}x\ \texttt{=}\ e\texttt{;} \longrightarrow y\ \texttt{= conv(check\_E\_x(}\widehat{e}\texttt{));}\ \widehat{y}\ \texttt{= conv(}y\texttt{);}}\ \textsc{Int-Single} \qquad \frac{e \text{ is a pointer}}{\texttt{*}x\ \texttt{=}\ e\texttt{;} \longrightarrow \texttt{*}x\ \texttt{=}\ e\texttt{;}}\ \textsc{Ptr}$$

$$\frac{\Sigma(x) = \{v_1, \ldots, v_n\} \qquad n > 1 \qquad \texttt{*}x \Downarrow E_x \qquad v_1, \ldots, v_n \text{ are integers}}{\texttt{*}x\ \texttt{=}\ e\texttt{;} \longrightarrow \texttt{*}x\ \texttt{= conv(check\_E\_x(}\widehat{e}\texttt{));}\ \widehat{v_1}\ \texttt{= conv(}v_1\texttt{);}\ \ldots\ \widehat{v_n}\ \texttt{= conv(}v_n\texttt{)}}\ \textsc{Int-Multi}$$

$$\frac{}{\texttt{if(}e\texttt{)}\{s_1\}\ \texttt{else}\ \{s_2\} \longrightarrow \texttt{if(}\widehat{e}\texttt{)}\{\widehat{s_1}\}\ \texttt{else}\ \{\widehat{s_2}\}} \qquad \frac{}{\texttt{while(}e\texttt{)}\{s\} \longrightarrow \texttt{while(}\widehat{e}\texttt{)}\{\widehat{s}\}}$$

$$\frac{e_r \text{ is an integer constrained by } E_R}{x(x_1, \ldots, x_t)\{s\ \texttt{return}\ e_r\texttt{;}\} \longrightarrow x(x_1, \ldots, x_t)\{\widehat{s}\ \texttt{return conv(check\_E\_R(}\widehat{e_r}\texttt{));}\}}\ \textsc{Int}$$

$$\frac{e_r \text{ is a pointer}}{x(x_1, \ldots, x_t)\{s\ \texttt{return}\ e_r\texttt{;}\} \longrightarrow x(x_1, \ldots, x_t)\{\widehat{s}\ \texttt{return}\ \widehat{e_r}\texttt{;}\}}\ \textsc{Ptr}$$

$$\texttt{check\_E\_o (}\widehat{e}\texttt{)}\ \{\ \texttt{vari}\ \widehat{i}\texttt{;}\ \widehat{i}\ \texttt{=}\ \widehat{e}\texttt{; if(inRange(}\widehat{i}\texttt{,}E_o\texttt{))}\{\ \texttt{skip;}\ \}\ \texttt{else}\ \{\ \texttt{fail;}\ \}\ \texttt{return}\ \widehat{i}\texttt{;}\ \}$$

Figure 4. Rules for transforming a IKL program. `conv` is a notation representing a conversion from an integer to a new one of expected type to keep well-typedness. A `check` function returns a valid numerical value only if its arguments values in the specified range. An implementation example of `check_E_o()` is shown at the bottom of figure.

every step of s's execution, ϵ is a *sanity state* for s.

We also introduce the concept of *equivalent states*. Given a program P with a state $\epsilon = (\Delta, \sigma)$ and its transformation $\widehat{P}$ with a state $\overline{\epsilon} = (\overline{\Delta}, \overline{\sigma})$, ϵ and $\overline{\epsilon}$ are equivalent under transformation if (1) for each $x \in \mathbb{X}$ with integer value, we have $\Delta(x) = \overline{\Delta}(\widehat{x}) = \overline{\Delta}(x)$; (2) $\sigma = \overline{\sigma}$. Without ambiguity, we simply say ϵ and $\overline{\epsilon}$ are equivalent, which is denoted by $\epsilon \sim \overline{\epsilon}$. The following theorems show that given a sanity state and its equivalent one: (1) the execution of a statement and its transformation results in equivalent final states; (2) an expression and its transformation are evaluated to the same results along with equivalent final states.

Theorem 1 (Conservativeness)**.** *A statement s has a sanity state ϵ and its transformation is $\widehat{s}$. $\langle s, \epsilon \rangle \to \epsilon'$ and $\langle \widehat{s}, \overline{\epsilon} \rangle \to \overline{\epsilon'}$ where $\epsilon \sim \overline{\epsilon}$. Then we have $\epsilon' \sim \overline{\epsilon'}$.*

Two pointers p_1, p_2 are equal if one of the following cases holds: (1) Both are `null` or `malloc` (2) p_1 and p_2 point to pointers p_1', p_2' respectively and $p_1' = p_2'$; (3) p_1 and p_2 point to the same integer variable.

Theorem 2. *An expression e has sanity state ϵ and its transformation is $\widehat{e}$. $\langle e, \epsilon \rangle \to \langle n, \epsilon' \rangle$ and $\langle \widehat{e}, \overline{\epsilon} \rangle \to \langle \overline{n}, \overline{\epsilon'} \rangle$ where $\epsilon \sim \overline{\epsilon}$. Then we have $n = \overline{n}$ and $\epsilon' \sim \overline{\epsilon'}$.*

Proof Sketch of Theorem 1 and Theorem 2: Propositions of Theorem 1 and Theorem 2 are denoted by $Q(s)$ and $R(e)$ respectively. To prove $\forall s.Q(s) \wedge \forall e.R(e)$, we perform structure induction jointly on statements and expressions. ■

Program execution can be regarded as a function call of its main function. Therefore, executing a program and its transformation on the equivalent sanity states yields the same results along with equivalent final states. In other words, transformed code behaves differently only on the state causing errors in the original program.

IV. CIntFix: A Tool to Fix C Integer Errors

A. Overview

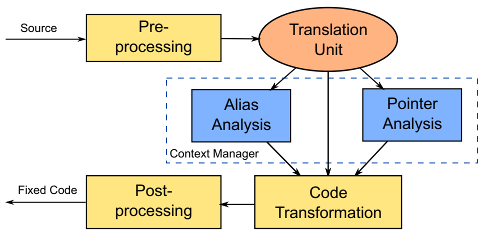

Figure 5. The workflow of CIntFix.

We implement a prototype tool CIntFix to automatically fix C integer errors by dynamic precision improvement. Figure 5 illustrates the architecture of CIntFix. It takes a source file as input, and outputs the fixed one. The first step is to preprocess input source code to obtain a translation unit where all preprocessor directives are eliminated. Then, code transformation rules are applied to translation unit. Context manager conducts alias and pointer analysis on the fly. The former is required to correctly recognize integer variables while the latter is important for value consistency in pointer manipulations. Finally, we restore preprocessor directives and make beautification for transformed code.

To implement dynamic-precision integer arithmetic in fixed C program, we use GMP[1] which is a carefully designed and highly optimized library supporting arbitrary precision arithmetic. To simplify the fixed code, we also create a library to support sanity check for integer operation including various `check` functions. Our tool can be applied to existing C project by slightly modifying the `Makefile`.

All source code of CIntFix is publicly available[2].

B. Implementation Issues

Real world C programs involve a richer set of C features. They are handled by CIntFix in the following ways.

External variables: CIntFix does not elevate any external variable because its definition is out of the scope of program to be analyzed.

Data structures: C supports data structures including array, structure and union. Their memory layouts have specific patterns instead of amorphous structures. CIntFix does not alter definitions of data structures and the reasons are twofold. First, changing memory layout hinders the correctness of program especially for pointer manipulations. Second, definition of data structure is part of developer's design. Developer is responsible for a correct design while CIntFix is responsible for fixing faulty implementation.

Library calls: Library calls are common in C programs and their sources are generally inaccessible. After a library call, we synchronize local integer variables used directly or indirectly in arguments. We assume that other local variables are unmodified because most real-world library functions do not require a specific calling context, such as a specific sequence of variable definitions.

Variadic functions: C supports variadic function which accepts a variable number of arguments. In principle, it is impossible to infer types of all arguments because of insufficient information. CIntFix contains predefined annotations for library functions such as `printf()` and is able to infer types of arguments by parsing the format string. For other functions, CIntFix will not check values of arguments.

Bit manipulations: C supports various bit manipulations including bitwise shift and bitwise logical operation. They are based on bit-vector logic and does not require precision elevating. Without loss of generality, consider a bitwise AND expression $x \& y$. We first convert dynamic-precision integers $\widehat{x}$, $\widehat{y}$ into bounded integers $\widetilde{x}$, $\widetilde{y}$ while the type of $\widetilde{x}$ is consistent with the type of x in original program and so is the type of $\widetilde{y}$. Then, the resultant expression is $\widetilde{x} \& \widetilde{y}$.

It is worth mentioning that unsigned wraparound is widely utilized in cryptographic applications and device drivers. These wrapped values are finally propagated to (1) bit manipulations; (2) critical sites such as function arguments. For the former case, CIntFix lazily truncates dynamic-precision integer to obtain the wrapped value when bit manipulation is reached. For the latter case, wrapped value is sanitized because CIntFix is unable to automatically recognize an intentional wraparound without developer's hints.

[1]https://gmplib.org

[2]http://git.io/vEnHz

Five more patterns of intolerable error are summarized for C programs and they are shown in Table II. Array index is bounded by `SIZE_MAX` because it is the upper bound of `size_t` which is the type for memory manipulation.

Table II
NEW INTRODUCED PATTERNS OF INTOLERABLE ERRORS. ALL BOXED EXPRESSIONS ARE INTEGERS.

pattern	constraint
`x =` $\boxed{\texttt{e}}$`;`, `x` is global	E_x (type of `x`)
`o.i =` $\boxed{\texttt{e}}$`;`	$E_{\texttt{o.i}}$ (type of `i` in object `o`)
`p->i =` $\boxed{\texttt{e}}$`;`	$E_{\texttt{p.i}}$ (type of `i` in object pointed to by `p`)
`p[`$\boxed{\texttt{e}}$`]` or `p +` $\boxed{\texttt{e}}$	[`-SIZE_MAX`, `SIZE_MAX`]
`p[i] =` $\boxed{\texttt{e}}$`;`	$E_{\texttt{p}}$ (type of elements in array `p`)

To discuss the conservativeness, we revise the definition of state equivalence since some variables could not be elevated. $\epsilon = (\Delta, \sigma)$ and $\overline{\epsilon} = (\overline{\Delta}, \overline{\sigma})$ are equivalent if: (1) for each liftable $x \in \mathbb{X}$ we have $\Delta(x) = \overline{\Delta}(\widehat{x}) = \overline{\Delta}(x)$; for other integers $x \in \mathbb{X}$, we have $\Delta(x) = \overline{\Delta}(x)$; (2) $\sigma = \overline{\sigma}$. $\mathbb{X}$ denotes the set of all identifiers in program, including names of fields and array elements. By structural induction on new syntactic elements, conservativeness can be proved.

C. A Running Example

```
Pairs *pairs;
int count, in_count;
...
count = in_count / 2;
pairs = palloc(count * sizeof(Pairs));
...
```

Listing 2. Possible wraparound on the argument of `palloc()` leads to potential buffer overflow, which is a serious security issue.

```
Pairs *pairs;
mpz_t __HIGHPREC_count, __HIGHPREC_in_count;
mpz_t __HIGHPREC_INTERM_5;
mpz_init(__HIGHPREC_count);
mpz_init(__HIGHPREC_in_count);
mpz_init(__HIGHPREC_INTERM_5);
...
mpz_tdiv_q_ui(__HIGHPREC_count, __HIGHPREC_in_count, 2);
mpz_mul_ui(__HIGHPREC_INTERM_5, __HIGHPREC_count, sizeof(Pairs));
pairs = palloc(__CHECK_GMP_UINT(__HIGHPREC_INTERM_5));
...
```

Listing 3. Fixed code in `hstore_from_array()` function. Integer variables are elevated with additional prefix `__HIGHPREC_`. Identifiers with the prefix `__HIGHPREC_INTERM_` are new introduced intermediate variables. `__CHECK_GMP_UINT` returns `unsigned int` value of the input GMP integer if possible and halts the program otherwise.

PostgreSQL is a popular open-source database management system. PostgreSQL 9.0.x (before 9.0.16) has multiple integer overflow bugs in `contrib/hstore/hstore_io.c` that can be exploited [14]. Listing 2 shows a code snippet from `hstore_from_array()`, one of the problematic functions. If `count` is a large integer, the argument of `palloc()` will be wrapped to a small positive value leading to buffer overflow. Fixed code is shown in Listing 3. If the product of `__HIGHPREC_count` and `sizeof(Pairs)` overflows, `__CHECK_GMP_UINT()` prevents overflown value for memory allocation by halting the program with error message. This function is defined in our sanity check library.

D. Advantages and Limitations

CIntFix has several advantages. First, the fixing procedure is syntax-directed and rule-based, which avoids sophisticated and imprecise analysis. Second, CIntFix works in function-wise manner and it is possible to apply transformation to a portion of program code and scale to very large program. However, CIntFix possibly misses some integer errors. CIntFix cannot automatically infer constraints on arguments of library function without annotations. CIntFix cannot handle C program mixed with assembly code. CIntFix cannot address errors caused by wrong design, such as faulty function signatures and data structures. Finally, CIntFix is unable to tolerate intentional wraparounds which directly propagate to critical sites.

V. Experimental Evaluation

We evaluate CIntFix on NIST's Juliet test suite[3] and SPEC CINT2000 benchmarks for both effectiveness and efficiency. In CIntFix, implementation of dynamic-precision integer is based on GMP 6.0.0. All experiments are conducted on a PC under 64-bit Ubuntu 15.10, using Intel Core i5-3470@3.20GHz CPU and 20GB memory. In evaluation, we focus on the following questions:

Q1 Could CIntFix fix both tolerable and intolerable integer errors in C programs effectively?
Q2 How efficient does CIntFix transform C source code?
Q3 How much overhead is introduced by using dynamic-precision integers in fixed programs?

A. Evaluation on Juliet Test Suite

NIST's Juliet test suite is a collection of test bench programs in C/C++ created by NSA Center for Assured Software and developed specifically for assessing the capabilities for program analysis tools. The latest version is 1.2 released on May 2013, contains 61387 cases for 118 different CWEs. Our evaluation is based on version 1.2.

1) Experiment Design: We choose C programs in 7 CWEs shown in Table III. CWE190 and CWE191 refer to overflow and underflow respectively. They are collectively known as overflow. CWE194-197 involve with various kinds of conversion errors such as lossy truncation and sign error. CWE680 defines a special kind of integer overflow, namely *integer overflow to buffer overflow* (IO2BO). It occurs when an integer overflow causes less memory to be allocated than expected leading to a buffer overflow.

Before evaluation, test programs should be processed to facilitate automated experiments. We filter out some programs that heavily depend on blocking I/O, then specify inputs for remaining programs of which inputs are usually from system calls such as `fscanf()` and `rand()`. We specify an input for a program by replacing existing input statement with assignment of this input. For each test program, the range of bad inputs can be derived from annotation of code. Thus, we can sample a group of bad input candidates as *reference inputs*. By default we sample 20 different reference inputs for each test program. If bad inputs are insufficient, we just take all of them.

[3]https://samate.nist.gov/SRD/testsuite.php

Table III
Chosen CWEs related to integer errors. C++ test programs are excluded.

CWE	Description	#program	#file	KLOC
190	Integer Overflow	2052	3024	263.38
191	Integer Underflow	1254	1848	163.65
194	Unexpected Sign Extension	912	1344	108.20
195	Signed to Unsigned Conversion Error	912	1344	105.46
196	Unsigned to Signed Conversion Error	18	18	1.55
197	Numeric Truncation Error	684	1008	67.63
680	Integer Overflow to Buffer Overflow	228	336	26.37

In evaluation, a test program P and its transformation P' are executed with reference inputs. P produces incorrect results or halts unexpectedly. We say that P' is a correct fixed version of P if and only if P' can produce correct results or halt expectedly for all reference inputs.

2) Experimental Results: We use GCC 5.2.1 to compile all test programs. All time measures are valued based on the average time of 10 repeated executions. The experimental results are presented in Table IV.

Column 2 and 7 show the number of programs and their files used in evaluation. We can notice that a small portion of test programs are not evaluated because they depend heavily on I/O and unsuitable for automatic evaluation. Columns 8-11 show the performance of code transformation. CIntFix processes source code at the rate of 0.181s/KLOC, i.e. about 5500 code lines per second. Moreover, fixed code expands by 23.5% on the measurement of KLOC.

Columns 3-6 present the effectiveness of CIntFix in fixing integer errors. Notice that errors in CWE680 can be triggered only if `size_t` is equivalent to `unsigned int`. However on our x64 platform `size_t` is an alias of `unsigned long int`. To trigger anticipated errors, we evaluate CWE680 programs under 32-bit Ubuntu 15.10 in QEMU virtual machine with 4 CPU cores and 4GB memory. In total, CIntFix succeeds in fixing every integer error of either tolerable or intolerable in all 5414 benchmark programs.

The last 3 columns show the efficiency of fixed code. We discard runtime of some test programs since their executions are interrupted with segmentation faults given bad inputs. For normal runs, the average slowdown by fixing is 10.8%. Notice that for CWE197, fixed programs are even more efficient than the original ones. This is because sanitizations work to prevent taint values from further propagation and thus the fixed programs can terminate early.

B. Evaluation on SPEC CINT2000

The SPEC CPU2000 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 software. CINT2000 is one of subcomponents of SPEC CPU2000 for measuring compute-intensive integer performance and it is used for evaluating CIntFix. CINT2000 benchmark programs have a collection of integer errors [7]. Since we do not have full and definite knowledge on where they are and how to trigger them, this

Table IV
RESULTS ON JULIET TEST SUITE. SEG REFERS TO SEGMENTATION FAULT DURING EXECUTION.

CWE	#program	#fixed program				#file	GT (s)	KLOC			runtime (s)		
		T	IT	Σ	fix rate (%)			old	new	expansion (%)	old	new	overhead (%)
190	1938	1938	0	**1938**	**100.0**	2856	39.800	230.06	289.12	25.7	0.037	0.043	16.2
191	1178	1178	0	**1178**	**100.0**	1736	22.389	141.39	193.56	36.9	0.022	0.025	13.6
194	760	0	760	**760**	**100.0**	1120	12.267	78.96	92.38	17.0	SEG	0.022	N/A
195	760	0	760	**760**	**100.0**	1120	13.953	78.40	92.11	17.5	SEG	0.025	N/A
196	18	18	0	**18**	**100.0**	18	0.876	1.55	1.9	22.6	0.002	0.002	0.0
197	570	570	0	**570**	**100.0**	840	10.453	49.24	49.43	0.4	0.013	0.012	-7.7
680	190	0	190	**190**	**100.0**	280	6.324	19.60	22.90	16.8	SEG	0.010	N/A
Σ	5414	3704	1710	**5414**	**100.0**	7970	112.109	618.81	764.30	23.5	0.074	0.082	10.8

T/IT: tolerable/intolerable; GT: code generation time

Table V
RESULTS ON SPEC CINT2000.

name	#file	KLOC			code generation time (s)	reference time (s)	base time (s)		
		old	new	increment (%)			old	new	overhead (%)
164.gzip	20	5.62	12.71	126.4	1.536	1400	64.292	75.950	18.1
175.vpr	41	11.30	27.31	141.7	2.353	1400	43.758	52.366	19.7
176.gcc	120	149.01	395.53	165.4	11.549	1100	24.978	29.104	16.5
181.mcf	25	1.48	2.75	85.7	1.124	1800	37.614	43.646	16.0
186.crafty	43	12.90	41.79	223.9	2.884	1000	21.267	26.153	23.0
197.parser	18	7.77	24.54	215.8	1.942	1800	68.681	81.207	18.2
253.perlbmk	91	72.10	304.18	321.9	8.085	1800	48.134	55.781	15.9
254.gap	63	35.70	106.84	199.3	4.517	1100	28.146	33.633	18.4
255.vortex	123	49.23	92.04	87.0	4.428	1900	65.705	77.362	17.7
256.bzip2	2	3.23	7.95	146.1	0.982	1500	49.008	58.460	19.3
300.twolf	85	17.82	46.34	160.0	3.154	3000	69.020	81.206	17.7
total	631	366.14	1061.97	190.0	42.554	17800	520.873	614.868	18.0

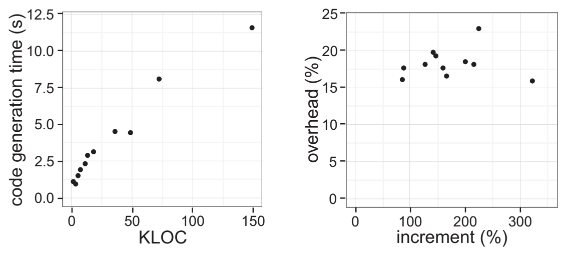

Figure 6. Correlation analysis of experimental results on SPEC CINT2000. The left is the correlation of code size and fixing time ($\rho = 0.978$); the right is the correlation of increment rate and runtime overhead ($\rho = 0.066$). ρ denotes the correlation coefficient.

benchmark is for evaluating fixing efficiency only. Our evaluation is based on version 1.2 with necessary modifications.

1) Experiment Design: To measure the efficiency of fixed code, we compare the execution time of original program and the fixed one given the same "ref" data sets as inputs. In order to reduce random error in timing, all time measures are valued based on the average of 10 repeated runs. We use all programs except the only one C++ program 252.eon.

2) Experimental Results: We use Clang 3.6.2 with LLVM 3.6.2 to compile test programs and the results are shown in Table V. Columns 3-6 show the performance of fixing. On average, CIntFix processes C source code at the rate of 0.116s/KLOC. Fixed code expands aggressively by 190.0% on average. In particular, the size of 253.perlbmk increases by more than 2 times after fixing. Columns 8-10 present the runtime efficiency of fixed code. The average slowdown is about 18.0% while the highest is 23.0% for 186.crafty.

Furthermore, we analyze correlations between key factors in experimental results. Figure 6 shows that code size and time cost of fixing has strong linear dependence. However, code increment rate and runtime overhead by fixing are unlikely correlated while the correlation coefficient is 0.066. Hence, a large increment rate does not imply a significant slowdown and vice versa, because runtime overhead is mainly related to complexity of control flow.

C. Discussion

Questions proposed at the beginning of this section involve two main criteria: effectiveness and efficiency. The former refers to the ability of correctly fixing integer errors (**Q1**). The latter has two aspects: the fixing efficiency (**Q2**) and runtime performance of fixed code (**Q3**). In the sequel we answer these questions with experimental results.

Effectiveness. We evaluate the effectiveness of CIntFix on Juliet test suite. Experimental results show that CIntFix succeeds in fixing all 5414 errors in test programs, which is a strong evidence of CIntFix's effectiveness. Moreover, CIntFix generates fixed code without detecting potential errors in advance. By observing that the nature of integer errors includes: (1) inconsistent arithmetics over fixed-length bit-vectors and $\mathbb{Z}$, (2) type unsafe property of C, we directly elevate precision of program integers to fix potential errors.

Efficiency. Efficiency of CIntFix is evaluated on both Juliet test suite and SPEC CINT2000 benchmark.

Efficiency of code transformation: On one hand, by the experimental results, we estimate that CIntFix can process 1 million lines of C source code using about 150 seconds, which means our tool can be applied to a large software system (typically has millions lines of code) with relatively acceptable time cost. On the other hand, code increment rates are quite different on two benchmarks. Fixed code expands by 23.5% on Juliet test suite but 190.0% on SPEC CINT 2000. The difference mainly derives from the complexity of code structure. Programs in Juliet test suite have relatively simple structures while SPEC CINT2000 bench programs obtained from real-world software have complicated control structures and many compound expressions.

Runtime efficiency of fixed code: Fixed code has moderate runtime overhead on two benchmarks (10.8% on Juliet test suite and 18.0% on SPEC CINT2000) and the reasons are twofold. First, dynamic-precision integer is extended on demand and it is rare to yield an integer of very large length in computation. In practice, overhead brought by manipulating dynamic-precision integers is limited. Second, fixed program basically keeps the design of original program except for additional sanitizations which are usually cheap. All in all, the overhead introduced by fixing is non-negligible but acceptable for many applications. Moreover, runtime overhead has no necessary (or very weak) correlation with code increment rate by fixing. Therefore a significantly expanded fixed program is not necessarily unacceptable on runtime slowdown and vice versa.

Memory efficiency of fixed code: Memory efficiency is not discussed in detail since in most scenarios time resource is more critical than memory. Generally speaking, the fixed program consumes more memory than the original because of introduction of dynamic-precision integers. To reduce memory overhead, we apply several optimizations in generating fixed code including controlling number of introduced temporary variables for carrying intermediate results and freeing dynamic integers timely when they are no longer needed in sequel computation. In experiments, fixed programs run normally without memory issues.

VI. Related Work

Static analysis: One class of static analysis is symbolic execution, which systematically explores all possible paths by treating inputs as symbolic values. Symbolic execution based techniques attempt to find inputs exposing integer errors following a specific control flow. Example tools include PREfix [15], SmartFuzz [11], IntScope [12] and KINT [8]. In particular, KINT performs symbolic analysis in function-wise manner and succeeded in identifying more than 100 integer errors in Linux kernel. In general, these techniques achieve high coverage but suffer from path explosion and inability of solving complicated path constraints.

Abstract interpretation [16] based techniques [8], [17] soundly approximate semantics of program by employing range analysis [18] or taint analysis [19] , and locate potential errors accordingly. They have acceptable scalability for large programs but report many false positives.

Given error condition, a program can be model checked by verification tools such as BLAST [20], CBMC [21] and LLBMC [22]. Model checking is sound and complete for many practical models and specifications, but generally has poor scalability for verifying large programs.

Dynamic analysis: Dynamic techniques insert checks to capture integer problems at runtime. Some techniques are source-oriented. RICH [9] captures errors based on safe integer semantics derived by applying sub-type theory. IOC [7] detects overflow errors in compile-time integrated with Clang. The AIR [23] model provides well-defined semantics for integer-related undefined behavior and guarantees that an integer operation either produces a representable result or trap. RA [17] reduces unnecessary checks by employing range analysis. IntPatch [24] and IntTracker [25] focus on detecting IO2BO vulnerabilities. Some techniques are binary-oriented. BRICK [13] emulates binaries using a Valgrind [26] based virtual machine and detects errors dynamically. Overall, dynamic analysis reports *real* errors without applying sophisticated approximations on control flow of program, while the main drawback is low code coverage and non-negligible overhead at runtime.

Safe library support: To prevent integer errors in developing phase, safe libraries such as CERT's SafeInt [27] and Ranged Integer [28] are available to support trusted integer operations. These libraries, however, are mostly unverified and reported to contain integer errors in implementation [7]. Moreover, library functions have low tolerability for errors, which makes developed program highly failure-sensitive.

Error nullification: Integer errors can be nullified by discarding or changing inputs that possibly trigger integer errors in programs. SIFT [10] employs an interprocedural, weakest precondition static analysis to propagate critical constraints backwards against control flow to generate input filter and it is proved to be sound [29]. Anomaly detection technique [30]–[32] generates input filters by learning properties from benign and bad inputs. These techniques are generally incomplete and incapable to provide sufficient information on faulty program data and structures.

Integer error repair: Various automatic program repair techniques are proposed for generic bug fixing. Generate-and-validate techniques heuristically generates candidate patches and searches the appropriate one by validating given test suite. Example tools include GenProg [33], AE [34], SPR [35] and Prophet [36]. Semantics based synthesis techniques solve constraint of correctness conditions and synthesis new expressions conforming the specification accordingly. Example tools include SemFix [37] and DirectFix [38]. Generic bug fixing techniques have two main drawbacks. First, the completeness of generated patch relies on given specification, which is usually insufficient in practice. Second, generated patches rare ensure correctness according to a study [39]. Some techniques are proposed specifically for integer errors. Z. Coker et al. [40] introduce three transformations rules to generate fixed program. However, this method relies on limited information from type inference and wraps integer operations with safe integer functions which is not failure-oblivious.

VII. Conclusion

In this paper we proposed a novel approach to automatically fix C integer errors by elevating precision. Dynamic-precision program integers can precisely simulate mathematical integers such that many integer errors can be tolerated. To fix a program, we directly apply rule-based code transformation on the source code without employing sophisticated and expensive analysis to find possible errors. This technique is proved to be conservative which ensures that all sanity behaviors are kept unchanged in fixed program. We also introduce the design and implementation of the tool CIntFix and evaluate it using standard benchmarks. The results show that (1) CIntFix can effectively fix various kinds of integer errors; (2) CIntFix can efficiently generate fixed code; (3) fixed code has non-negligible but acceptable slowdown.

In our future work, we intend to evaluate CIntFix with more programs and analyze fixed code in instruction level in order to better understand the effectiveness and efficiency of our approach. Moreover, we aim to enhance CIntFix to support repairing of data structures and function signatures instead of simply sanitizing them.

Acknowledgment

This research is sponsored in part by National Natural Science Foundation of China (Grant No. 91218302, 61402248) and the Ministry of Industry and Information Technology IT funds (Research and application of TCN key technologies) of China.

References

[1] Wikipedia, "Ariane 5 Flight 501," https://en.wikipedia.org/wiki/Ariane_5_Flight_501.

[2] N. Y. Times, "F.A.A. Orders Fix for Possible Power Loss in Boeing 787," http://www.nytimes.com/2015/05/01/business/faa-orders-fix-for-possible-power-loss-in-boeing-787.html.

[3] "CVE-2010-2753," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2753.

[4] "CVE-2015-865," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-865.

[5] S. Christey and R. A. Martin, "Vulnerability Type Distributions in CVE," http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf, 2007.

[6] S. Christey, B. Martin, M. Brown, A. Paller, and D. Kirby, "2011 CWE/SANS Top 25 Most Dangerous Software Errors," http://cwe.mitre.org/top25/, 2011.

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

[8] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with KINT," in *OSDI*, 2012, pp. 163–177.

[9] D. Brumley, D. X. Song, T. Chiueh, R. Johnson, and H. Lin, "RICH: automatically protecting against integer-based vulnerabilities," in *NDSS*, 2007.

[10] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. C. Rinard, "Sound input filter generation for integer overflow errors," in *POPL*, 2014, pp. 439–452.

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

[12] T. Wang, T. Wei, Z. Lin, and W. Zou, "Intscope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution," in *NDSS*, 2009.

[13] 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 *ARES*, 2009, pp. 208–215.

[14] "CVE-2014-2669," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2669.

[15] Y. Moy, N. Bjørner, and D. Sielaff, "Modular bug-finding for integer overflows in the large: Sound, efficient, bit-precise static analysis," Tech. Rep. MSR-TR-2009-57, 2009.

[16] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, 1977, pp. 238–252.

[17] R. E. Rodrigues, V. H. S. Campos, and F. M. Q. Pereira, "A fast and low-overhead technique to secure programs against integer overflows," in *CGO*, 2013, pp. 33:1–33:11.

[18] W. H. Harrison, "Compiler analysis of the value ranges for variables," *IEEE Trans. Software Eng.*, vol. 3, no. 3, pp. 243–250, 1977.

[19] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *IEEE S&P*, 2010, pp. 317–331.

[20] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast," *STTT*, vol. 9, no. 5-6, pp. 505–525, 2007.

[21] D. Kroening and M. Tautschnig, "CBMC - C bounded model checker - (competition contribution)," in *TACAS*, 2014, pp. 389–391.

[22] S. Falke, F. Merz, and C. Sinz, "The bounded model checker LLBMC," in *ASE*, 2013, pp. 706–709.

[23] R. B. Dannenberg, W. Dormann, D. Keaton, R. C. Seacord, D. Svoboda, A. Volkovitsky, T. Wilson, and T. Plum, "As-if infinitely ranged integer model," in *ISSRE*, 2010, pp. 91–100.

[24] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, "Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time," in *ESORICS*, 2010, pp. 71–86.

[25] H. Sun, X. Zhang, C. Su, and Q. Zeng, "Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability," in *ASIACCS*, 2015, pp. 483–494.

[26] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007, pp. 89–100.

[27] D. LeBlanc, "SafeInt," https://safeint.codeplex.com/, 2014.

[28] J. Gennari, S. Hedrick, F. Long, J. Pincar, and R. C. Seacord, "Ranged integers for the c programming language," Tech. Rep. CMU/SEI-2007-TN-027, 2007.

[29] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard, "Sound input filter generation for integer overflow errors," Tech. Rep. MIT-CSAIL-TR-2013-018, 2013.

[30] D. Gao, M. K. Reiter, and D. X. Song, "On gray-box program tracking for anomaly detection," in *USENIX Security Symposium*, 2004, pp. 103–118.

[31] C. Krügel and G. Vigna, "Anomaly detection of web-based attacks," in *CCS*, 2003, pp. 251–261.

[32] W. K. Robertson, G. Vigna, C. Krügel, and R. A. Kemmerer, "Using generalization and characterization techniques in the anomaly-based detection of web attacks," in *NDSS*, 2006.

[33] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE*, 2009, pp. 364–374.

[34] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *ASE*, 2013, pp. 356–366.

[35] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *SIGSOFT/FSE*, 2015, pp. 166–178.

[36] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *POPL*, 2016, pp. 298–312.

[37] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: program repair via semantic analysis," in *ICSE*, 2013, pp. 772–781.

[38] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *ICSE*, 2015, pp. 448–458.

[39] 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 *ISSTA*, 2015, pp. 24–36.

[40] Z. Coker and M. Hafiz, "Program transformations to fix C integers," in *ICSE*, 2013, pp. 792–801.