IntPTI: Automatic Integer Error Repair with Proper-Type Inference

Xi Cheng*, Min Zhou*, Xiaoyu Song†, Ming Gu*, Jiaguang Sun*
*School of Software, TNLIST, KLISS, Tsinghua University, China
†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/C++ are caused by arithmetic operations yielding results which are unrepresentable in certain type. They can lead to serious safety and security issues. Due to the complicated semantics of C/C++ integers, integer errors are widely harbored in real-world programs and it is error-prone to repair them even for experts. An automatic tool is desired to 1) automatically generate fixes which assist developers to correct the buggy code, and 2) provide sufficient hints to help developers review the generated fixes and better understand integer types in C/C++. In this paper, we present a tool IntPTI that implements the desired functionalities for C programs. IntPTI infers appropriate types for variables and expressions to eliminate representation issues, and then utilizes the derived types with fix patterns codified from the successful human-written patches. IntPTI provides a user-friendly web interface which allows users to review and manage the fixes. We evaluate IntPTI on 7 real-world projects and the results show its competitive repair accuracy and its scalability on large code bases. The demo video for IntPTI is available at: https://youtu.be/9Tgd4A_FgZM.**

***Index Terms*—integer error, type inference, fix pattern**

I. Introduction

In C/C++ programs, integer arithmetic operations (e.g. addition and assignment) may produce results that the certain expression type cannot represent, and such values are converted somehow to fit the target type. Some conversions are well-defined (e.g. unsigned wraparound) by the language standard but others are undefined (e.g. signed overflow). Integer errors are generally caused by misuse of well-defined conversions or undefined behaviors due to developer's empirical certainty of expected outcomes. Integer error is known to be one of the main threats to the safety and security of software system. A potential total power loss in Boeing 787 Dreamliners [1] was caused by the signed overflow of a 32-bit counter. Multiple integer errors in Linux kernel can be exploited for denial-of-service attacks [2] or privilege escalations [3]. A CVE report in 2007 [4] suggests that integer overflow error is the second most common vulnerability in the advisories for OS vendor.

Challenge. It is error-prone to correctly repair integer errors even for experts due to the complicated semantics of integers in C/C++. The machine representation of an integer is a fixed-size bit-vector restricted by its type-specific characteristics: signedness and width. Generally, the semantics over fixed-size bit-vectors and $\mathbb{Z}$ are inconsistent. For example, $(x - y > 0) \iff (x > y)$ 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 well-defined. Although undefinedness grants compilers freedom to generate efficient code by exploiting specific properties of a certain instruction set, it could lead to unexpected runtime behaviors across different architectures or optimization levels. For example, an overflow in signed addition silently wraparounds on x86 but traps on MIPS [5].

Related Work. Numerous automatic solutions for integer errors have been proposed, but they have various limitations in real-world applicability. One thread of the related work focuses on integer error detection by symbolic execution [6], [7], [8], static analysis [9] or code instrumentation [10], [11], [12]. These tools produce reports on where integer errors are and how to trigger them, but they are unable to guide developers to correct the buggy implementation. Generic program repair techniques are proposed to automatically correct the implementation with its specifications. They generate patches that address certain defects by, typically, validating heuristically generated patches with test suites [13], [14], [15], [16], [17], [18], or synthesizing desired expressions with respect to constraints derived from test suites [19], [20], [21], [17]. The effectiveness of these tools, however, heavily relies on specifications which are often insufficient in practice. Moreover, even the state-of-the-art generate-and-validate systems do not scale to large software systems with thousands of potential defects as they generally require hours to find a plausible patch for one real-world bug. Some tools are designed for integer errors specifically [22], [23]. They transform the internal integer model of a program towards a safer model but an excessive number of unnecessary changes are made in the program.

Approach. We present IntPTI, an automatic tool that generates and applies fixes for integer errors in C programs. It aims to assist developers and testers to improve code quality against integer errors. First, IntPTI preprocesses the source files on the fly in the building process. Next, IntPTI computes the appropriate types (i.e. *proper-types*) for variables and expressions to eliminate representation issues and generates fixes by utilizing proper-types. Then, users interact with IntPTI via a web interface to review fixes. Finally, accepted fixes are collected and applied to the source code. Users can benefit from IntPTI as it proposes fixes for possible integer errors with proper explanations, which helps users to (1) locate the new integer errors in code and repair them correctly, (2) better understand the integer types in C language.

Our key approach is *proper-type inference*, which finds appropriate types for expressions and variables such that each expression has the type that covers all its possible values. The goal of proper-type inference is achieved by static value analysis (§II-B, which approximates possible values of ex-

$$
\begin{array}{lcl}
\tau & ::= & \varsigma \mid \tau* \mid \texttt{void} \mid \texttt{struct}\{\tau c_1; \ldots \tau c_n\} \mid \tau(\tau, \ldots \tau) \\
e & ::= & n \mid x \mid e.c \mid e \Diamond e \mid (\tau)e \mid *e \mid \&e \mid e := e \mid f(e, \ldots e)
\end{array}
$$

Fig. 1. The core language syntax.

pressions) and type inference (§II-A, which computes types for expressions and variables with respect to the proper-type property and the well-typedness of program). Inferred types are utilized to generate fixes (§II-C) by common fix patterns codified from the real world: sanity check, explicit type casting and declared type changing. For the scalability in real-world projects, IntPTI adopts *multi-entry analysis* (§II-D) to run proper-type inference in the compositional manner.

To demonstrate the accuracy and runtime efficiency of IntPTI, we apply it to 7 widely-used open-source projects with known integer vulnerabilities. The results show that IntPTI succeeds in repairing 23 out of 25 defects. Furthermore, IntPTI substantially addresses the limitations of existing tools as it: 1) does not rely on specifications such as test suites, 2) generates fixes with proper explanations of why the fix is generated and how it transforms the code, 3) reduces false-positives (i.e. fixes that correspond to no genuine bugs) on the *critical program sites* (where attacks are typically performed on the subject programs) by 93.3% compared to the state-of-the-art approach [22], and 4) scales to large code bases as it spends no more than 11 minutes on Vim with over 244 KLOC.

Contribution. Main contributions are summarized as:

1) We propose a novel approach that automatically generates fixes under appropriate patterns via type inference.
2) We implement our approach as a tool IntPTI. It is designed for complex realistic C code bases and provides a user-friendly interface for users who are unfamiliar with program analysis techniques.
3) We evaluate IntPTI on 7 open-source projects. The results show its competitive repair accuracy and runtime efficiency, which substantiates that our tool is of practical concern.

II. Approach Overview

Generally, IntPTI repairs C integer errors by three main steps: 1) static value analysis, 2) proper-type inference and 3) fix generation. For the scalability of IntPTI, multi-entry analysis is employed to divide-and-conquer the fix generation task on the whole program.

A. Proper-Type Inference

We present a C-like kernel language shown in Fig. 1 to formalize our discussion. The language models expressions including integer literals, variables, structure members, binary expressions (the operator $\Diamond$ can be arithmetic or logical), cast expressions, pointer dereferences, address-of expressions, assignments and library calls. There can be multiple kinds of integer types varying on length and/or signedness. Each expression has a type which can be integer type (ς), pointer type, structured type and function type. A program consists of variable declarations and expressions. Let $[\![\cdot]\!] : E \to 2^{\mathbb{Z}}$ map an expression to its possible values and $(\!|\cdot|\!) : T \to 2^{\mathbb{Z}}$ map a type to values that it can represent. We say that τ is the *proper-type* of an arithmetic expression e if $[\![e]\!] \subseteq (\!|\tau|\!)$.

To derive proper-types, we scan the program and collect constraints on the types of expressions and variables by proper-type inference rules. All non-arithmetic expressions keep their original types. Rules for arithmetic expressions are listed in Fig. 2. The type judgment $\Gamma, \Theta, \Upsilon \vdash e : \tau \mapsto C$ denotes that given the context $(\Gamma, \Theta, \Upsilon)$, the type of e is inferred as τ along with the constraint set C. The context consists of the typing hypothesis Γ which maps variables to their declared types, Θ that assigns arithmetic expressions with their enforced types, and $\Upsilon = ([\![\cdot]\!], (\!|\cdot|\!))$ where the former is computed by value analysis (§II-B) and the latter is given by C language data model (e.g. data type width schemes, including LP32, ILP32, LP64, etc.) in use. The notation $\varsigma_1 \preceq \varsigma_2$ denotes that the byte length of ς_2 is no less than that of ς_1 (namely ς_2 *elevates* ς_1). $\mathcal{E}$ maps expressions to their original types.

We give brief explanations for some rules. In the Binary-Arith rule, the type ς is required to 1) be eligible to represent possible values of $e_1 \Diamond e_2$, 2) be the common type of ς_1 and ς_2 (namely $\varsigma_1 \uparrow \varsigma_2$) to preserve well-typedness. In the Binary-Logical rule, however, operands of logical operation are enforced to have their common type in order to prevent 1) implicit conversion in comparison and 2) overflow bug in each operand. The Assign-Var rule elevates the declared type of the variable to be assigned with respect to the right operand, while the Assign-NonVar rule enforces the non-variable L-value to have its original type. In the Library-Call rule, argument expression is enforced to fit its parameter type. The operand of an address-of operation keeps its original type to prevent memory issues after repair.

The collected constraints need further processing. First, for each arithmetic variable x we add a constraint $\mathcal{D}(x) \preceq \Gamma(x)$ denoting that the new type of x elevates its original type (where $\mathcal{D}$ maps variables to their original declared types). Second, constraints of value inclusion such as $n \in (\!|\varsigma|\!)$ and $[\![e]\!] \subseteq (\!|\varsigma'|\!)$ are allowed to be violated with penalty w_v since built-in integer types can only represent a limited range of integers. Third, for each type elevation constraint $\varsigma_1 \preceq \varsigma_2$, we add a matching equality constraint $\varsigma_1 = \varsigma_2$ with violation penalty w_p to minimize changes on types. Generally, w_v should be substantially larger than w_p to not overshadow necessary type elevations. The conjunction of processed constraints is a partial weighted MaxSMT problem [24]. A MaxSMT solver such as Z3 [25] finds a solution that satisfies all hard constraints (i.e. ones with no penalties) while minimizing the total penalty of violated constraints (albeit a local minima is derived in general), or reports unsatisfiable otherwise. There always exists a solution $\mathcal{I}$ such that $\mathcal{I}(\Gamma) = \mathcal{D}$, $\mathcal{I}(\Theta)(e) = \mathcal{E}(e)$ and $\mathcal{I}(\varsigma_e) = \mathcal{E}(e)$ where ς_e is the type variable for e.

B. Static Value Analysis

Static value analysis approximates values of expressions and gives $[\![\cdot]\!]$ for proper-type inference. Three analyses are mainly used: interval analysis (which captures lower and upper bounds of expressions), pointer analysis (which analyzes points-to relations) and reaching definition analysis (which builds use-def chains). Analyses share the derived information to achieve better precision. For example, points-to relations are exploited to approximate pointer dereferences more precisely. In interval analysis, we extend the basic interval arithmetic [26] by 1)

$$\textsc{Var}\quad \frac{}{\Gamma,\Theta,\Upsilon \vdash x : \varsigma \mapsto \{\Gamma(x) \preceq \varsigma\}}$$

$$\textsc{Const}\quad \frac{}{\Gamma,\Theta,\Upsilon \vdash n : \varsigma \mapsto \{n \in \llparenthesis\varsigma\rrparenthesis, \mathcal{E}(n) \preceq \varsigma\}}$$

$$\textsc{Binary-Arith}\quad \frac{\Gamma,\Theta,\Upsilon \vdash e_1 : \varsigma_1 \mapsto C_1 \qquad \Gamma,\Theta,\Upsilon \vdash e_2 : \varsigma_2 \mapsto C_2 \qquad \Diamond \text{ is an arithmetic operator}}{\Gamma,\Theta,\Upsilon \vdash e_1 \Diamond e_2 : \varsigma \mapsto C_1 \cup C_2 \cup \{\llbracket e_1 \Diamond e_2 \rrbracket \subseteq \llparenthesis\varsigma\rrparenthesis, \varsigma_1 \uparrow \varsigma_2 = \varsigma\}}$$

$$\textsc{Binary-Logical}\quad \frac{\Gamma,\Theta,\Upsilon \vdash e_1 : \varsigma_1 \mapsto C_1 \qquad \Gamma,\Theta,\Upsilon \vdash e_2 : \varsigma_2 \mapsto C_2 \qquad \Diamond \text{ is a logical operator}}{\Gamma,\Theta,\Upsilon \vdash e_1 \Diamond e_2 : \varsigma \mapsto C_1 \cup C_2 \cup \{\varsigma = \mathcal{E}(e_1 \Diamond e_2), \varsigma' = \varsigma_1 \uparrow \varsigma_2, \llbracket e_1 \rrbracket \not\subseteq \llparenthesis\varsigma'\rrparenthesis \iff \Theta(e_1) = \varsigma', \llbracket e_2 \rrbracket \not\subseteq \llparenthesis\varsigma'\rrparenthesis \iff \Theta(e_2) = \varsigma'\}}$$

$$\textsc{Assign-Var}\quad \frac{x \coloneqq e \qquad \Gamma,\Theta,\Upsilon \vdash e : \varsigma \mapsto C}{\Gamma,\Theta,\Upsilon \vdash x : \varsigma' \mapsto C \cup \{\Gamma(x) = \varsigma', \llbracket e \rrbracket \subseteq \llparenthesis\varsigma'\rrparenthesis\}}$$

$$\textsc{Assign-NonVar}\quad \frac{e_1 \coloneqq e_2 \qquad \Gamma,\Theta,\Upsilon \vdash e_2 : \varsigma \mapsto C \qquad e_1 \text{ is not a variable}}{\Gamma,\Theta,\Upsilon \vdash e_1 : \varsigma' \mapsto C \cup \{\varsigma' = \mathcal{E}(e_1), \llbracket e_2 \rrbracket \not\subseteq \llparenthesis\varsigma'\rrparenthesis \iff \Theta(e_2) = \varsigma'\}}$$

$$\textsc{Deref}\quad \frac{\Gamma,\Theta,\Upsilon \vdash e : \varsigma_1 * \mapsto C}{\Gamma,\Theta,\Upsilon \vdash *e : \varsigma_2 \mapsto C \cup \{\varsigma_1 \preceq \varsigma_2\}}$$

$$\textsc{Address-Of}\quad \frac{\Gamma,\Theta,\Upsilon \vdash e : \varsigma \mapsto C}{\Gamma,\Theta,\Upsilon \vdash \&e : \varsigma * \mapsto C \cup \{\varsigma = \mathcal{E}(e)\}}$$

$$\textsc{Library-Call}\quad \frac{\Gamma,\Theta,\Upsilon \vdash f : \tau(\varsigma) \mapsto C_0 \qquad \Gamma,\Theta,\Upsilon \vdash e : \varsigma' \mapsto C_1}{\Gamma,\Theta,\Upsilon \vdash f(e) : \tau \mapsto C_0 \cup C_1 \cup \{\llbracket e \rrbracket \not\subseteq \llparenthesis\varsigma\rrparenthesis \iff \Theta(e) = \varsigma\}}$$

Fig. 2. Rules of proper-type inference.

TABLE I
THE DISTRIBUTION OF FIX PATTERNS FOR CVE BUGS.

Pattern	SC	ETC	DTC	ET	OTHER	**id**
Count	486	42	75	56	28	668

implementing semantics of library calls relevant to numerical operations such as `abs`, and 2) refining intervals with respect to path conditions and the use-def chain. Pointer analysis is based on Andersen's algorithm [27] and reaching definition analysis is based on classical data-flow analysis [28].

C. Repair Generation

A case study is conducted on 668 selected CVE identifiers as they contain sufficient information on buggy code, security impacts and upstream fixes. They come from 210 different applications ranging from 2001 to 2017. We identify recurring similar patches (i.e. fix patterns) and summarize them in Table I. The total occurrence of patterns is larger than the number of identifiers because some contain multiple patterns. There are totally four common patterns. Sanity check (SC, 72.8%) guards a critical operation against erroneous values. Explicit type casting (ETC, 6.3%) enforces the type of an expression. Declared type changing (DTC, 11.2%) adjusts the declared type for a variable. Expression transformation (ET, 8.4%) rewrites an expression as an equivalent form under integer arithmetic, such as transforming $x+y<8$ to $x<8-y$. Only 4.2% of fixes have no common patterns. From the perspective of repair mechanism, SC restricts the expression to have representable values while ETC and DTC elevate the precision of expression and variable, respectively. ET can be reduced to the combination of ETC and DTC since it is a workaround to prevent overflow without adjusting the precision. Hence, proper-types are utilized to generate SC, ETC and DTC fixes.

Let $\mathcal{I}$ be the derived solution of proper-type constraints and $\mathcal{F}$ be the set of candidate fixes. We have 1) $F_\varsigma^{\mathsf{DTC}}(x) \in \mathcal{F}$ iff $\mathcal{I}(\Gamma)(x) = \varsigma \neq \mathcal{D}(x)$; 2) $F_\varsigma^{\mathsf{SC}}(e) \in \mathcal{F}$ for $(e,\varsigma) \in \mathcal{I}(\Theta)$; 3) $F_\varsigma^{\mathsf{ETC}}(e) \in \mathcal{F}$ iff $\mathcal{I}(\varsigma_e) = \varsigma \neq \mathcal{E}(e)$. The notation $F(e)$ is used to collectively denote a SC fix or an ETC fix on e. We define the dependency relation $\lhd$ over $\mathcal{F}$, such that 1) $F(e_1) \lhd F(e_2)$ iff e_1 is a descendant of e_2 in the AST; 2) $F_*^{\mathsf{ETC}}(e) \lhd F_*^{\mathsf{SC}}(e)$; 3) $F_*^{\mathsf{DTC}}(x) \lhd F_*^{\mathsf{ETC}}(x)$. $F_1 \lhd F_2$ implies that the application result of F_2 depends on that of F_1.

$$\begin{array}{rcl}
M(F_\varsigma^{\mathsf{ETC}}(e_1 \Diamond e_2)) & = & M(F_\varsigma^{\mathsf{ETC}}(e_1)) \Diamond M(F_\varsigma^{\mathsf{ETC}}(e_2)) \\
M(F_\varsigma^{\mathsf{ETC}}((\varsigma')e)) & = & (\varsigma)e \\
M(F_\varsigma^{\mathsf{ETC}}(e)) & = & (\varsigma)e \\
M(F_\varsigma^{\mathsf{SC}}(e_1 \Diamond e_2)) & = & \mathsf{check}_\varsigma^{\Diamond}(M(F_\varsigma^{\mathsf{SC}}(e_1), M(F_\varsigma^{\mathsf{SC}}(e_2))) \\
M(F_\varsigma^{\mathsf{SC}}(e)) & = & \begin{cases} \mathsf{check}_\varsigma(e) & \varsigma \preceq \Sigma(e) \wedge \varsigma \neq \Sigma(e) \\ e & \text{otherwise} \end{cases}
\end{array}$$

Fig. 3. The expression transformation function M.

To apply the candidate fixes in $\mathcal{F}$, we iteratively choose a fix F that depends on no other fixes in $\mathcal{F}$ and remove it from $\mathcal{F}$. If F is a DTC fix, we change the certain declared type; otherwise, F is applied by expression transformation function M shown in Fig. 3 which lists the expression transformation schemes ordered by the priority of application. Note that SC fixes can introduce two kinds of check functions: 1) conversion check $\mathsf{check}_\varsigma(n)$ that returns n iff it is representable in ς, 2) arithmetic check $\mathsf{check}_\varsigma^{\Diamond}(n_1, n_2)$ that returns the result of $n_1 \Diamond n_2$ in mathematical arithmetic iff it is representable in ς.

D. Multi-entry Analysis

The scalability of our approach is restricted by static analysis and the capability of solver on complicated constraints. To make our approach scale to large code bases, we adopt multi-entry analysis to divide-and-conquer the whole-program reasoning. Firstly, the code base is decomposed into multiple call graph components, each of which has an *entry function* from which other functions in the component are reachable by function calls. Next, we perform proper-type inference on each component starting from its entry and then combine their results. To guarantee the compatibility of sub-results, we enforce variables spanning multiple scopes (e.g. global variables) to keep their original types. Furthermore, to reduce the precision loss brought by compositional analysis, we perform a function-wise pre-analysis to derive 1) function call context, 2) function and loop summary and 3) loop invariant. They are all based on interval abstract domain.

III. Implementation

The architecture of IntPTI is shown in Fig. 4. Given a C project, IntPTI firstly preprocesses it and constructs its control-flows for analysis. Next, IntPTI approximates values of expressions and collects proper-type constraints by running bounded CPA algorithm with several analyses. Candidate

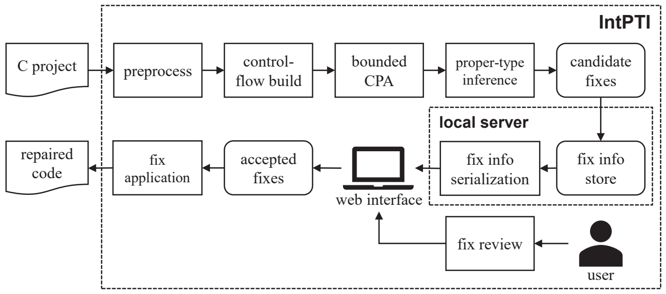
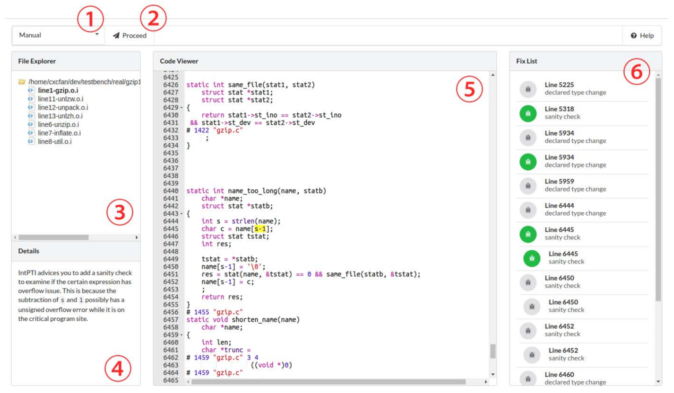


Fig. 4. The architecture of IntPTI.

```
{"UUID":"536335ff-f0a9-4b6b-aca0-358606f1fe9e","mode":"CAST","type":
      "long int","startLine":3060,"endLine":3060,"startOffset":22,"
      endOffset":23,"_defect":"overflow","_ary":2,"_op1":"i","_op2":"
      o","_optr":"+","_sign":1,"children":[]}
```

Fig. 5. An example of a serialized fix.



/home/cxcfan/dev/testbench/real/gzip1
- line1-gzip.o.i
- line11-unlzw.o.i
- line12-unpack.o.i
- line13-unlzh.o.i
- line6-unzip.o.i
- line7-inflate.o.i
- line8-util.o.i

IntPTI advices you to add a sanity check to examine if the certain expression has overflow issue. This is because the subtraction of s and 1 possibly has a unsigned overflow error while it is on the critical program site.

```

static int same_file(stat1, stat2)
    struct stat *stat1;
    struct stat *stat2;
{
    return stat1->st_ino == stat2->st_ino
 && stat1->st_dev == stat2->st_dev
# 1422 "gzip.c"
      ;
}




static int name_too_long(name, statb)
    char *name;
    struct stat *statb;
{
    int s = strlen(name);
    char c = name[s-1];
    struct stat tstat;
    int res;

    tstat = *statb;
    name[s-1] = '\0';
    res = stat(name, &tstat) == 0 && same_file(statb, &tstat);
    name[s-1] = c;
    ;
    return res;
}
# 1455 "gzip.c"
static void shorten_name(name)
    char *name;
{
    int len;
    char *trunc =
# 1459 "gzip.c" 3 4
                     ((void *)0)
# 1459 "gzip.c"

```

Line 5225	declared type change
Line 5318	sanity check
Line 5934	declared type change
Line 5934	declared type change
Line 5959	declared type change
Line 6444	declared type change
Line 6445	sanity check
Line 6445	sanity check
Line 6450	sanity check
Line 6450	sanity check
Line 6452	sanity check
Line 6452	sanity check
Line 6460	declared type change

Fig. 6. The web interface for fix review.

fixes are derived by solving the proper-type constraints using Z3 [25]. Then, IntPTI starts a web interface for users to review candidate fixes. Finally, IntPTI transforms the code to apply accepted fixes.

Preprocessing. First, the compilation commands in `Makefile` are captured by invocation order. Next, source files are preprocessed by, in the case that the compiler in use is gcc, adding the `-E` flag to make the compiler stop after preprocessing and replacing the output `*.o` files by the corresponding `*.i` files which are self-contained source files with macros expanded and necessary declarations included. The `*.i` files compiled into an executable or a library are organized as a *task*.

Bounded CPA algorithm. The static analysis algorithm used in IntPTI is extended from CPA algorithm [29] for multi-entry analysis. When the analysis reaches the boundary of current call graph component or maximum loop iteration or maximum loop nesting level, we summarize the upcoming function or loop. The algorithm runs multiple analysis components including but not limited to the three main analyses. Each analysis is implemented under the CPA framework [29].

Syntactic elements of the standard C language need to be handled carefully. Bitwise shift operations are approximated carefully for possible intentional wraparounds. For example, a left shift of an unsigned integer is always representable in its type as this kind of wraparound is usually served for modulo operation. The possible targets of a function pointer are derived by combining pointer analysis with type matching. If no targets can be found, certain function call is treated as a library call without implemented semantics. Type enforcements are imposed on critical program sites including function arguments, array indexes, return values and the operands of condition expressions. Furthermore, memory regions allocated by `*alloc` family are treated as arrays whose identifiers have the [**alloc*]-[*line number*] format.

The heuristic for call graph decomposition in multi-entry analysis is configurable by specifying the maximum levels of call stack, loop unrolling and loop nesting in the configuration.

Proper-type inference. Proper-type constraints are encoded under an extended SMT-LIB 2.0 format [30] over quantifier-free equality logic with uninterpreted functions [31]. More specifically, we use the `declare-datatype` command to define an enumeration `I` of integer types plus a dummy type `OVERLONG` which can represent every value in $\mathbb{Z}$, `declare-fun` commands to define a binary function over `I` to model $\uparrow$, two binary predicates `P`, `Q` over `I` where `P` models the containment relation over $(\!|T|\!)$ and `Q` models $\preceq$, and variables (also 0-ary functions) over `I` encoding type variables such as ς, $\Gamma(x)$ and $\Theta(e)$. $[\![e]\!]$ is encoded as the corresponding value of ς in `I` where ς is the type with the shortest byte length such that $[\![e]\!] \in (\!|\varsigma|\!)$, or `OVERLONG` if such ς does not exist. $(\!|\varsigma|\!)$ is encoded as the corresponding value of ς in `I`. We assert additional constraints such that every type variable is prohibited to be `OVERLONG`. To add a constraint with violation penalty w, we use the command `(assert-soft [formula] :weight` w`)` and for others (without violation penalty), the `assert` command is used. Penalty values w_p and w_v can be configured by users and their default values are 1 and 100, respectively, based on pilot experiments.

Fix info serialization. The candidate fixes are serialized as JSON data to be interchanged across a local server and web interface. Fig. 5 shows a serialized fix. The JSON object for a fix stores its UUID, its mode (`"SPECIFIER"` for DTC, `"CAST"` for ETC, `"CHECK_ARITH"` for arithmetic check and `"CHECK_CONV"` for conversion check), its specified type, the location of target code characterized by line number and offset, and the fixes depending on the current fix (as the value of `"children"`). Keys starting with "_" are mode-specific fields providing more details on this fix. For example, the reason for the fix shown in Fig. 5 is as follows: there is an overflow issue on the signed addition of `i` and `o`.

Web interface. End users review candidate fixes via the web interface shown in Fig. 6. The file explorer (③) lists source files of the current project. When a user selects a file, the code viewer (⑤) displays its contents while the relevant candidate fixes are loaded in the fix list (⑥). Indentions are used to visualize the dependency relations over fixes. The user can accept or reject a fix by toggling its button. For two fixes F_1 and F_2 such that $F_1 \lhd F_2$, the interface enforces that F_2 is toggled on only if F_1 is toggled on. By selecting a fix, the code viewer highlights the target code and scrolls to the certain line, while the detail panel (④) shows the description of the proposed fix and the reason in natural language. For now two working modes are supported and can be selected by the dropdown list at ①. In the global mode, all the candidate fixes are accepted and their statuses are locked. In the manual mode, however, all the candidate fixes are rejected by default and users can manually accept some of them. Finally, by clicking the confirmation button at ②, accepted fixes are submitted to IntPTI for code transformation.

Code transformation. Arithmetic checks and conversion checks are implemented as a library libTsmartIntFix. To apply a SC fix, we insert the corresponding check function with its declaration introduced. It is necessary to link this library to compile the repaired code.

IV. Experimental Evaluation

The effectiveness of IntPTI can be assessed in the terms of the following aspects:

1) What is the accuracy of IntPTI in repairing integer errors?
2) What is the runtime efficiency of IntPTI?

IntPTI is evaluated on 7 open-source projects: gzip 1.2.4, Vim 7.4, grep 2.10, PostgreSQL 9.0.15, JasPer 1.900.5, libarchive 3.1.2 and OpenSSL 1.0.1r. They are chosen for evaluation because 1) they are written in C and contain known integer vulnerabilities in the CVE database, 2) they covers various application domains including encoding and decoding, text processing, database system, graphics and cryptography. Repair accuracy should be measured by recall and precision. The former is evaluated by checking how many vulnerability are successfully repaired after running IntPTI in the global mode. The latter cannot be evaluated directly because we have no complete knowledge on integer errors in realistic code bases. Thus, we compare the numbers of critical program sites (which are typical locations for fixes) applied with fixes by various tools.

All experiments are conducted on a workstation under 64-bit Ubuntu 16.04, using Intel Core i7-6820HQ@2.70GHz CPU and 32GB memory. The maximum levels of the call stack and loop nesting are set to 3 while loops are not unrolled (thus loop bodies are always reasoned by loop invariants). Constraint penalties are set as default values. The data model employed is configured as ILP32 only if the target defects can only be triggered under ILP32, or LP64 otherwise. The experimental results of IntPTI are shown in Table II.

Columns 1-4 show general information about the evaluation programs. For PostgreSQL and OpenSSL, IntPTI directly analyzes all source files under the specified location (such as crypto/bn) because the target defects are not included in any preprocessed task. *KLOC* and *KLOC-P* refer to the lines of code before or after preprocessing, respectively. *KLOC* excludes all header files.

Columns 5-11 assess the recall. Column *CVE-ID* lists CVE identifiers for the defects. Some identifiers (such as CVE-2014-2669) contain multiple buggy code fragments with similar defect patterns. Column *DM* reports under which data model (32 for ILP32 and 64 for LP64) certain defect can be triggered. Column *Op* lists buggy operations (e.g. $\times_u$ denotes unsigned multiplication while "u2s" denotes the conversion from unsigned to signed). The following three columns count the different kinds of fixes. Column *F* reports whether certain defect is correctly repaired. The results show that 23 out of 25 defects are repaired correctly. Two missed bugs are due to flow-insensitivity of proper-type inference. For example, the bug belonging to CVE-2017-5499 involves an signed multiplication yielding an overflowed 64-bit integer which is then passed to a critical site, and the proper-type inference cannot capture the sequence of the overflowed multiplication and the critical site.

Columns 12-17 report runtime efficiency results. Columns *CF*, *SU*, *VA*, *TI* and *FA* report the time costs of control-flow build, summary computation, static value analysis, type inference and fix application, respectively. Static value analysis costs the majority of total time (78.6% on average). It is remarkable that type inference costs only 4.8% of total time and this result can be achieved because multi-entry analysis decomposes the complicated whole-program proper-type constraints into fragments which can be solved efficiently. Furthermore, IntPTI spends no more than 11 minutes on Vim, the largest project (over 244 KLOC) used in the evaluation.

The last 3 columns evaluate the precision. The *Baseline* column lists the total number of critical sites of integer type. We compare IntPTI's precision with CIntFix [23] and Coker's [22] because 1) they work on source code only without requiring additional inputs, 2) they support various kinds of integer errors. CIntFix changes every critical site (as the Column *Baseline* shows) because it needs to make sure whether a multi-precision integer value fits the target built-in type on every critical site. Since we are unable to obtain the artifact of Coker's work after our request to authors, we count the number of critical sites applied with fixes as if the add-integer-cast transformation is applied to all local variables, array subscripts and field access expressions, and the replace-arithmetic-operator transformation is applied to all arithmetic expressions and assignments, as the experimental settings in the paper. The results show that Coker's and ours make changes on 34.8% and 2.3% of all critical sites, respectively. In fact, by considering user's feedbacks, our false-positive rate can be further reduced. Coker's and CIntFix focus on transforming the internal integer model to a safer model, thus massive false-positives could be generated.

CIntFix is also evaluated on the chosen projects. The results show that CIntFix succeeds in repairing all 25 target defects while the total time cost is 392.268s. It is insufficient to conclude that CIntFix has better real-world applicability for its superior performance on the recall and runtime efficiency because precision is crucial to user experience as the criteria for static code analyzers. In fact, the recall and runtime efficiency of our tool could be improved by changing all integer types to 64-bit unsigned/signed integers and inserting sanity checks on every arithmetic operations, which would make it much more difficult for users to review the fixes and find possible integer defects in programs.

V. Conclusion

In this paper, we present IntPTI, an automatic repair tool for integer errors in C programs. It integrates type inference synergically with static value analysis to compose constraints

TABLE II
EVALUATION RESULTS OF IntPTI.

General info				Accuracy							Performance (s)						# Critical sites		
Project	**Task**	**KLOC**	**KLOC-P**	**CVE-ID**	**DM**	**Op**	**ETC**	**SC**	**DTC**	**F**	**CF**	**SU**	**VA**	**TI**	**FA**	Σ	**Baseline**	**Coker's**	IntPTI
gzip	gzip	5.22	17.09	2010-0001	64	$-_u$	2	0	1	✓	1.459	3.352	33.224	0.522	0.471	39.028	1166	342	29
Vim	vim	244.037	1195.814	2017-6350	32/64	$\times_u$	0	1	0	✓	28.325	68.612	521.505	27.416	10.102	655.960	56367	19225	1150
				2017-6349	32/64	$\times_u$	0	1	0	✓									
				2017-5953	32	$\times_u$	2	0	0	✓	29.362	71.050	467.811	29.723	9.920	607.866		19202	1236
grep	grep	40.577	132.482	2012-5667	32/64	$+_s$	3	1	1	✓	4.276	7.250	14.384	1.329	1.165	28.404	4271	2004	78
				2012-5667	32/64	u2s	0	0	1	✓									
PostgreSQL	contrib/hstore	2.94	43.222	2014-2669	32	$\times_u$	2	1	0	✓	3.325	1.279	7.154	2.110	1.173	15.041	1979	909	114
				2014-2669	32	$\times_u$	2	1	0	✓									
				2014-2669	32	$\times_u$	2	1	0	✓									
				2014-2669	32	$\times_u$	2	1	0	✓									
JasPer	jasper	26.932	138.348	2017-5499	32/64	$\times_s$	0	0	0	✗	5.716	13.658	57.792	7.435	2.204	86.805	11514	4037	351
				2017-5500	32/64	$\ll_s$	0	1	0	✓									
				2017-5502	32/64	$\ll_s$	0	3	0	✓									
				2016-9387	32/64	$\times_s$	2	1	0	✓									
				2016-9262	32/64	$+_s$	0	0	1	✓									
	imgcmp	26.973	138.4	2017-5501	32/64	$+_s$	3	1	0	✓	5.489	10.626	12.582	7.088	2.193	37.978	11601	4045	355
libarchive	bsdtar	77.424	398.783	2016-6250	32/64	$+_s$	3	0	3	✓	9.817	24.167	489.197	16.740	5.385	545.306	24184	8949	645
				2016-5844	32/64	$\times_s$	0	2	0	✓									
				2016-5844	32/64	$\times_s$	0	2	0	✓									
				2016-4300	32/64	$+_u$	0	0	0	✗									
OpenSSL	crypto/mdc2	0.305	4.456	2016-6303	32/64	$+_u$	0	1	0	✓	0.761	0.279	0.207	0.066	0.352	1.665	59	19	2
	crypto/evp	11.305	258.01	2016-2106	32/64	$+_s$	2	0	0	✓	7.057	3.053	2.465	5.433	3.593	21.601	3750	1230	68
				2016-2105	32/64	$+_s$	2	0	0	✓									
	crypto/bn	13.777	78.673	2016-0797	32/64	$\times_s$	0	0	1	✓	3.968	2.170	22.935	2.585	1.579	33.237	4745	1311	53
				2016-0797	32/64	$\times_s$	0	0	1	✓									

on the types of variables and expressions for preventing representation issues. It utilizes the solution of constraints with common fix patterns ETC, SC and DTC codified from the successful human-written patches. We have designed a web interface which fills the gap between end users and the tool to (1) help users better understand integer bugs and 2) reduce unnecessary fixes with user feedback. IntPTI is evaluated on 7 real-world projects and the results show that it is effective to repair various kinds of integer errors with high runtime efficiency while avoiding unnecessary changes on critical program sites. The source code and experimental data are publicly available at https://git.io/v99Jw.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments. This research was supported in part by NSFC Program (No. 61527812), National Science and Technology Major Project (No. 2016ZX01038101), MIIT IT funds (Research and application of TCN key technologies) of China, and The National Key Technology R&D Program (No. 2015BAG14B01-02).

REFERENCES

[1] 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

[2] CVE, "CVE-2016-9793," 2016. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9793

[3] ——, "CVE-2016-9754," 2016. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9754

[4] 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/

[5] C. Price, *MIPS IV Instruction Set*. MIPS Technologies, 1995.

[6] 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.

[7] 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.

[8] 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 *ASPLOS*, 2015, pp. 473–486.

[9] 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.

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

[11] W. Dietz, P. Li, J. Regehr, and V. S. Adve, "Understanding integer overflow in C/C++," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, pp. 2:1–2:29, 2015.

[12] 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.

[13] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72, 2012.

[14] 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.

[15] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *ICSE*, 2014, pp. 254–265.

[16] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE*, 2013, pp. 802–811.

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

[18] ——, "Automatic patch generation by learning correct code," in *POPL*, 2016, pp. 298–312.

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

[20] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "Minthint: automated synthesis of repair hints," in *ICSE*, 2014, pp. 266–276.

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

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

[23] X. Cheng, M. Zhou, X. Song, M. Gu, and J. Sun, "Automatic fix for C integer errors by precision improvement," in *COMPSAC*, 2016, pp. 2–11.

[24] C. Ansótegui, M. L. Bonet, and J. Levy, "A new algorithm for weighted partial maxsat," in *AAAI*, 2010.

[25] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, 2008, pp. 337–340.

[26] T. J. Hickey, Q. Ju, and M. H. van Emden, "Interval arithmetic: From principles to implementation," *J. ACM*, vol. 48, no. 5, pp. 1038–1068, 2001.

[27] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, University of Cophenhagen, 1994.

[28] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 1999.

[29] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis," in *CAV*, 2007, pp. 504–518.

[30] C. Barrett, A. Stump, and C. Tinelli, "The smt-lib standard version 2.0," SMT-LIB Initiative, Tech. Rep., 2012.

[31] D. Kroening and O. Strichman, "Equality logic and uninterpreted functions," in *Decision Procedures: An Algorithmic Point of View*. Springer Berlin Heidelberg, 2008, pp. 59–80.