INTREPAIR: Informed Repairing of Integer Overflows

Paul Muntean, Martin Monperrus, Hao Sun, Jens Grossklags, and Claudia Eckert

Abstract—Integer overflows have threatened software applications for decades. Thus, in this paper, we propose a novel technique to provide automatic repairs of integer overflows in C source code. Our technique, based on static symbolic execution, fuses *detection*, *repair generation* and *validation*. This technique is implemented in a prototype named INTREPAIR. We applied INTREPAIR to 2,052 C programs (approx. 1 million lines of code) contained in SAMATE's Juliet test suite and 50 synthesized programs that range up to 20 KLOC. Our experimental results show that INTREPAIR is able to effectively detect integer overflows and successfully repair them, while only increasing the source code (LOC) and binary (Kb) size by around 1 percent, respectively. Further, we present the results of a user study with 30 participants which shows that INTREPAIR repairs are more than 10x efficient as compared to manually generated code repairs.

---

1 INTRODUCTION

INTEGER overflows are a well-known cause for memory corruptions (e.g., C++ object type confusion [1]) which could lead to buffer overflows (IO2BO) and potentially to control-flow hijacking attacks [2]. Integer overflows are a widely known type of vulnerability [3] that has threatened programs for decades. It now even has a revival, with the detection of integer overflows in Ethereum's Solidity smart contracts [4]. To address this challenge, it has been, for example, proposed to apply the concept of automatic repair [5] (see our Definition 1) to fix integer overflows. This idea consists of employing a system that produces code repairs in order to fix integer overflows. The most recent systems are dynamic: for instance, SoupInt [6] is a runtime-based binary repair generator. On the contrary, we aim at having a fully static approach, with no need for defining or generating inputs that trigger the integer overflow.

In this paper, we tackle these two problems in a novel system, INTREPAIR,[1] usable for automatic repair of integer overflows. Our key idea is to use symbolic execution [3], [7], [8], [9], [10] to reason about the repair. More specifically, our key contribution compared to previous work is that fault detection[2] works hand in hand with program repair.[3] In particular, we fuse the fault localization and repair generation phases into a single algorithm via SMT solving. We present a novel integer overflow repair technique that realizes this idea and the corresponding prototype tool, INTREPAIR. To the best of our knowledge, INTREPAIR is the first approach for automated repair of integer overflows that does not require test cases.

Given a faulty program, INTREPAIR generates an SMT system that captures integer overflow manifestation and allows for valid repair (see our Definition 2) and correct repair (see our Definition 3) generation. This is achieved by interweaving the fault detection SMT constraints with newly synthesized constraints used for correct fault generation. Further, the integer overflow detection component on which INTREPAIR relies does not generate false negatives (i.e., if there is a genuine integer overflow then this will be detected). Further, we do not distinguish between intended or unintended integer overflows, yet false positives may happen due to certain implementation limitations, such as loop unrolling or recursion depth. Our goal is to catch all faults. Further, as far as we are aware of, there is no tool currently capable to distinguish between intended and unintended integer overflow faults. Note that for this reason we leave the final repair insertion decision to the programmer as only he knows the

- *P. Muntean, J. Grossklags, and C. Eckert are with the Department of Computer Science, Technical University of Munich, München 80333, Germany. E-mail: {paul.muntean, claudia.eckert}@sec.in.tum.de, jens.grossklags@in.tum.de.*
- *M. Monperrus is with the Department of Computer Science, the KTH Royal Institute of Technology, Stockholm 114 28, Sweden. E-mail: martin.monperrus@csc.kth.se.*
- *H. Sun is with the State Key Laboratory of Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing 210008, China. E-mail: shqking@gmail.com.*

1. Source code: https://github.com/TeamVault/IntRepair
2. Fault detection demo: https://goo.gl/uNvdRp
3. Fault repair demo: https://goo.gl/912Jux

exact intended program behavior. Lastly, in this way he may opt to insert or not the (otherwise) fully automated synthesized repair.

INTREPAIR's implementation is based upon the Codan static symbolic execution engine [11], [12], [13], which uses the Z3 SMT solver [14] to solve the extracted constraints. Note that our repair generation technique is general enough to be implemented on top of other symbolic execution engines as well. We evaluated INTREPAIR on 2,052 C programs contained in the SAMATE's C/C++ benchmark suite [15], totaling more than one million lines of code (LOC). The evaluated programs contain all possible program control flows that may lead to an integer overflow. Further, we use a synthesized benchmark containing 50 large C programs (up to 20 KLOC) seeded with integer overflows. Our experimental results show that INTREPAIR is able to: (1) effectively detect all integer overflows, and (2) successfully repair the programs at the source code level. In addition, we present the results of a user study with 30 participants showing that INTREPAIR is more effective compared to traditional debugging based program repair. To the best of our knowledge, there are no other open source tools with which we can compare against. In particular, DirectFix [16], Angelix [17], or SemFix [18] are not suitable for comparison purposes as these tools use test cases to locate the fault: on the contrary, we do not assume the presence of test cases.

In summary, we make the following contributions:

- *Repair Technique.* We design a novel source code repair generation technique for integer overflows in C programs.
- *Repair Tool.* We implement INTREPAIR, as a prototype of our novel integer overflow repairing technique, for C source code programs. It can be automatically used to repair integer overflows for multiple integer precisions.
- *Quantitative Evaluation.* We evaluate INTREPAIR thoroughly with 2,052 C programs contained in SAMATEs Juliet standard test suite for C/C++ source code. We also evaluate it with 50 synthesized programs which range up to 20 KLOC. We show that INTREPAIR's code repairs are effective and induce low runtime overhead.
- *Controlled Experiment.* We evaluate INTREPAIR within a controlled experiment with 30 participants and show that INTREPAIR is more time-effective than manually repairing the same programs.

Outline. The paper is organized as follows. In Section 2, we present the required background knowledge needed in order to understand the rest of this paper. In Section 3, we present the overview of our technique. In Section 4, we highlight design and implementation details of INTREPAIR. In Section 5, we evaluate INTREPAIR, and in Section 6, we discuss limitations of INTREPAIR. In Section 7, we review related work, and in Section 8, we offer concluding remarks.

2 BACKGROUND

In this section, we present background information.

2.1 Integer Overflows

Integer overflow is a known cause of memory corruption and a widely known type of vulnerability as mentioned by

```
1 int64_t data = 0LL;
2 //Flaw: use random value
3 data = (int64_t)RAND64();
4 if(data > 0){
5 //if (data*2)>LLONG_MAX,
6 //this will overflow
7 int64_t result = data * 2;
8 printLongLongLine(result);}
```

Fig. 1. Integer overflow shaded gray at line seven.

Wang et al. [3]. It often leads to stack or heap overflow and thus is usually exploited indirectly, as opposed to buffer overflows which are exploited directly, see Brumley et al. [19]. More specifically, integer overflows occur at runtime, when the result of an integer expression exceeds the maximum allowed value (i.e., $2^{32} - 1$).

Fig. 1 depicts an integer overflow memory corruption at line number 7, which could manifest because there is no proper check in place for verifying the range of admissible values for data. This integer overflow (and potential underflow) fault can be avoided by checking the value of data to see if it is less than or equal to $\frac{LLONG\_MAX\_VAL}{2} = 4.611.$ $686.018.427.387.903 \approx 4^{18}$ and greater than or equal to $\frac{-LLONG\_MAX\_VAL}{2}$.

2.1.1 Characteristics of Integer Overflows

Integer overflows can be classified as malicious or benign. Essentially, an integer overflow manifests itself, when the program receives user-supplied input and subsequently the input value is used in an arithmetic operation to trigger an integer overflow. Thus, a smaller than expected value is supplied to the memory allocation function and as a result a smaller than expected memory will be allocated. Deciding between the types of integer overflow related problems is rather difficult and a lot of research has been devoted in the last years to this type of classification [20]. The general desire in the research community is to categorize different hard to find integer overflows w.r.t. how exploit-prone these are in contrast to just finding and repairing them.

2.1.2 Problems Due to Integer Overflows

In this section, we review the integer overflow related problems that we address within INTREPAIR and briefly describe other related types. More precisely, in this paper, we address only integer overflow, as described in document CWE-190 [21], and integer underflow [22], which is the result of multiplying two values with each other and the result is less than the minimum admissible integer value. This is due to the fact that the product subtracts one value from another.

Further, other integer related problems are: CWE-192, integer coercion error [23], manifests during bad type casting and extension or truncation of primitive data types. CWE-193, off-by-one error [24], manifests during product calculation/usage; an incorrect maximum/minimum value is used which is one more, or one less, than the correct value. CWE-194, unexpected sign extension [25], appears when an operation performed on a number can cause it to be sign-extended when it is transformed into a larger data type. CWE-195, signed to unsigned conversion error [26],

manifests when a signed primitive that is used inside a cast to an unsigned primitive can produce an unexpected result if the value of the signed primitive cannot be represented using an unsigned primitive. CWE-196, unsigned to signed conversion error [27], manifests when an unsigned is used inside a cast to a signed primitive, which can produce an unexpected value if the result of the unsigned primitive cannot be represented using a signed primitive. CWE-197, numeric truncation error [28], manifests when a primitive is cast to a primitive of a smaller size and data is lost in the conversion. CWE-680, integer overflow to buffer overflow [29], appears when an integer overflow occurs that causes less memory to be allocated than expected, which can lead to a buffer overflow.

2.2 Repairing Integer Overflows

There are several approaches to repair integer overflow based memory corruptions. Most of these techniques are based on input validation. Of all these techniques, manual fixing is the most tedious and provides fewer guarantees than other techniques. Compiler-based approaches add repairs in all fault prone locations and scale well to large code bases. As such, we next present symbolic execution based input validation and compare it against other techniques. As compared to other approaches, symbolic execution based techniques can be used to systematically check program paths for integer faults and to propose repairs. As a consequence, the repairs are cheap to construct and to insert. On the one hand, symbolic execution-based techniques can achieve more guarantees than other repair generation techniques. On the other hand, these techniques are based on computationally intensive analysis strategies which, if not applied in a suitable manner, may not scale well (or at all) with large programs. For this reason, we believe that repair tools should be used by programmers early during development since the level of software complexity is low and it increases as the number of code lines becomes larger. Lastly, we believe that in general: (1) manually written source code repairs, the industry standard, should be avoided and only used in *easy* to address situations [30], (2) compilers should not be used for repairing integer overflows since the number of guarantees which these can offer is low, and (3) specialized tools which provide more guarantees should be used for repair generation.

3 OVERVIEW

In this section, we present the overview of our approach.

3.1 Repair Definitions

In this section, we provide the definitions of an automatic software repair as presented by Monperrus [5], and then of INTREPAIR's valid repair and correct repair.

Definition 1 (Software Repair). *Automatic software repair consists of automatically finding a repair r to a software fault identified by an oracle o, without human intervention, and which applied modifies one or more lines of source code.*

Our work is based on a workflow with two characterizations of repair: valid repairs (initial) and correct repairs (stronger) which we now define.

Definition 2 (Valid Repair). *Let r be a software repair for program P and a fault f, v is a validation oracle of r, I is the set of inputs on which P behaves as intended and which does not generate f, and B is the set of valid behaviors of P. Then r is said valid under v iff:*

- *r enforces v.*
- *r does not change B for I.*
- *r changes B for non-standard I.*

Definition 3 (Correct Repair). *Based on the same terminology, let c be a correctness oracle for software repair r of program P for a fault f. Let Q be the set of all reachable program paths of P and a program path $p \in Q$. Then r is correct under c iff:*

- *r enforces c.*
- *r removes f on all $p \in Q$ that reach f.*
- *r does not introduce other behavior than B, i.e., r removes f without introducing new faults.*

These definitions underlay INTREPAIR's automatically generated repairs w.r.t. to their fundamental properties. In this paper, we use symbolic execution to identify valid and correct repairs.

3.2 Symbolic Execution Engine

In this work, we use a symbolic execution based approach for fault detection and repair generation. A symbolic execution engine [11], [31] constructs a control flow graph (CFG) for each analyzed program and extracts execution paths. Constraints along the execution path are encoded into SMT equations, using the SMT-LIB format [32]. The translation of CFG nodes into SMT is performed by a translator algorithm, which extends the program's abstract syntax tree (AST) visitor class, according to the visitor pattern [33]. This is usually based on a *bottom-up* traversal of each program statement located on the currently analyzed program execution path.

Further, we use the Codan static symbolic execution engine [11], [12], [13]. In Codan, single static assignment (SSA) variables are created for C expressions, which are associated with no variables in the analyzed program. Before creating a new variable, the interpreter checks whether there is already a symbolic variable. Further, for all SMT formulas created for a particular program statement, one symbolic variable is created for each of the variables contained in the original program statement. Next, a single path is extracted from the previously computed CFG and traversed. In Codan: (1) loops can be traversed a configurable number of times, (2) the analysis can be customized to look, for example, for the first N faults located on the currently analyzed program path, and (3) Codan performs path-caching and backtracking traversal, which avoids traversing the whole program path from the beginning and collecting all constraints again. Codan uses the Z3 [14] solver as its backend in order to solve SMT constraints.

3.3 Main Fault Localization Engine Features

Unrestricted Context Depth. The symbolic execution engine performs an inter-procedural path sensitive analysis with a call string approach. For each function a CFG is built.

Loop Unrolling Trade-offs. Each program loop can be unrolled up to a certain depth by setting a specific threshold value. Currently, we unroll each loop up to 10 times.

Note that we experimented with unrolling each loop also 100 and 1000 times and for the tested programs this made no difference. Further, this value is configurable by the user.

Library Calls. Our approach can handle library calls (e.g., `memset`, `memcpy`, etc.) by providing upfront for each of these functions a stub SMT function, which models the various function behaviors.

Finding Program Paths. We use a fixed deterministic thread scheduling algorithm for running the symbolic execution. The symbolic execution is run with approximate path coverage which uses Depth-First Search (DFS). During DFS, program states are backtracked and branch decisions are changed.

Automatic Slicing. We perform automatic slicing over the data flow in order to verify conditions on a program path and over the control flow to separate the analysis of different program paths.

Context Sharing for Different Checkers. Our engine allows that multiple checkers (e.g., integer overflow, buffer overflow, race condition, etc.) run in parallel during symbolic execution. Checkers are allowed to share the contexts because there is separation from the symbolic path interpretation.

Logic Representation. We use the SMT-LIB sub-logic of arrays, uninterpreted functions and non-linear integer and real arithmetic (AUFNIRA) since this approach can be automatically decided. Pointers, for example, are handled as symbolic pointers by the engine interpreter with a target and a symbolic integer as offset formula and output as logical formulas when dereferenced.

Path Validation. The program execution path validation is triggered at branch nodes and uses the same interface as the checkers. For all path decisions up to the current branch, the path validator queries the equation SMT-LIB linear system slice. The path validator throws a path unsatisfiable exception if the solver answer is unsatisfiable; then the symbolic execution proceeds with the next path.

Satisfiability Modulo Theories Solving. IntRepair uses SMT solving for three main purposes. First, for checking satisfiable and not satisfiable program execution paths. Second, for checking the presence of an integer overflow, additional SMT constraints are added to the SMT linear constraint system which was used to check path satisfiability in order to check for the presence of an integer overflow. Third, for checking if a repair removes an integer overflow, additional SMT constraints are added to the SMT linear constraint system which was used to check path satisfiability.

Path-Sensitive Tracing of Shared Variables. It is possible to use shared variables between threads in a context-sensitive way. This is accomplished by first marking all global shared variables, and then the shared property is inferred over data flow constructs such as references, assignments, function call parameters and return values.

Deep Nested CStructs. The C language program statement to SMT translator engine component is designed to recursively traverse deep nested C/C++ structures in order to determine the real type of a field inside the struct. Accordingly, our tool does not lose precision when dealing with C struct data types.

3.4 Overflow and Underflow Checks

In the following, we present the three types of overflow and underflow checks that are supported by INTREPAIR. If one of the preconditions evaluates to true, then an integer overflow has been found. We selected these three operations since these solve a large proportion of integer overflows [34], [35]. We plan to address the detection of integer overflows with more sophisticated detection schemes in next iterations of our tool. Next, we present the checks used by INTREPAIR, which are modeled as the following preconditions.

First Precondition. The addition of two integers, in which one is a variable and the other is a positive constant, will lead to an integer overflow if the following expression evaluates to true: $((s_1 > 0) \land (s_1 > (\text{INT\_MAX} - s_2)))$. Note that s_2 is the positive constant.

Second Precondition. The multiplication of two integers, in which one is a variable and the other is negative, will lead to an integer overflow or underflow if the following expression evaluates to true: $((s_1 > 0) \land (s_1 > (\text{INT\_MIN}/s_2))) \lor ((s_1 < 0) \land (s_1 < (\text{INT\_MAX}/s_2)))$. Note that s_2 is the negative constant.

Third Precondition. The multiplication of two equal integers will lead to an integer overflow or underflow if the following expression evaluates to true: $((s_1 > 0) \land (s_1 > (\text{sqrt} (\text{INT\_MAX})))) \lor ((s_1 < 0) \land (s_1 < (-\text{sqrt}(\text{INT\_MAX}))))$.

The above preconditions are used over multiple types of inputs for s_1 or s_2 (e.g., `RAND32()`, this is the wrapper for the C random function, `fscanf()` etc.). The preconditions can be applied over multiple integer precisions, meaning that INT_MAX can take different values depending on the currently used integer precision in the analyzed program. We call this value the maximum admissible upper bound value of an integer. This value is determined automatically during program analysis. Further, the variables s_1 or s_2 can take different types: `char`, `int64_t`, `int`, `short`, `unsigned int`. In contrast to IOC's preconditions, which aim to avoid an unconfirmed integer overflow during program runtime, our preconditions help to guide repairs at the right code location, where the integer overflow was detected and confirmed. This results in avoiding the fault during runtime. Note that we always use symbolic execution again as a last step to confirm that the fault was completely fixed after a repair was inserted.

Other operations, which may lead to integer overflows, are: truncation, bit shifts and subtraction operations (see Fig. 6 in Pereira et al. [36] for more details). Currently, INTREPAIR does not support these operations, but we plan to support them in an updated version of INTREPAIR.

4 DESIGN AND IMPLEMENTATION

4.1 Workflow of INTREPAIR

Fig. 2 provides an overview of INTREPAIR showing its main components. Initially, the ❶ *source code* is passed into the ❷ *symbolic execution* engine, which first constructs a CFG and then extracts program execution paths from this CFG.[4] Program execution constraints are collected along each path ❸ and the *integer overflow detection* component identifies whether

---

4. A certain maximum recursion depth for each analyzed path is specified as configuration parameter.
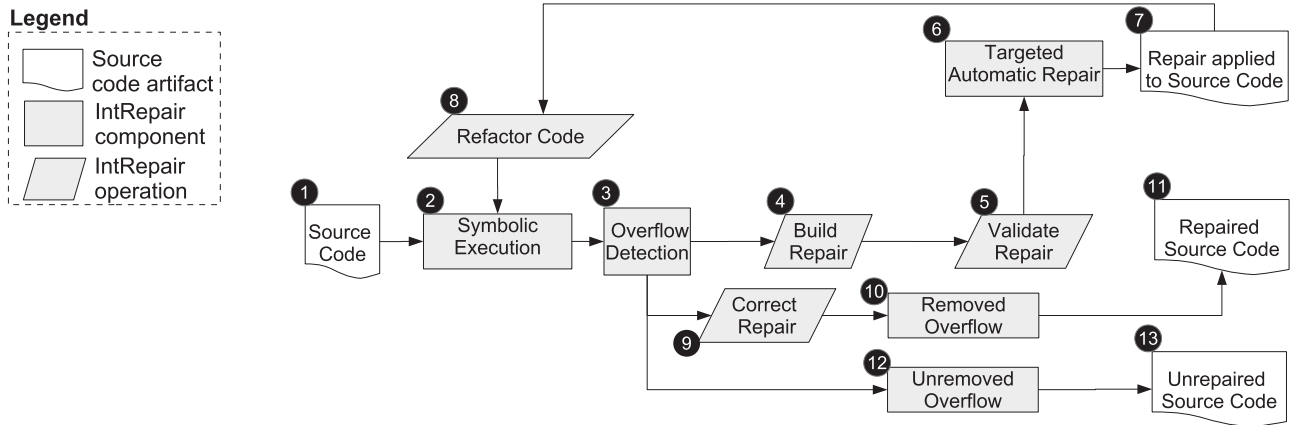
Fig. 2. Repair generation overview. The developer provides the faulty program as source code (1) which is next analyzed with the help of the symbolic execution engine (2) and depending on the result of integer overflow detection (3), next a repair is constructed (4) which represents a valid repair (5), next the steps (6) - (8) are followed in order to apply the generated repair and refactor the source code after applying the repair. Lastly, depending on step (3), either steps: correct repair (9), removed overflow confirmation (10), and repaired source code (11), or step (12) unremoved overflow, and step (13) unrepaired source code are executed.

an integer overflow might occur for one of the given execution paths. In ❹ a repair pattern is selected in order to build a repair. The ❺ validate repair (see Definition 2 for valid repair) step is performed. During this step it is checked whether the negated SMT constraints invalidate the previously detected integer overflow fault. If this is the case, in ❻ a *targeted automatic repair* is generated that potentially removes the previously detected fault. After applying the repair, we obtain the ❼ *repaired source code*, and next in ❽ the *refactored code* which is again validated through ❷. Further, in case ❸ *overflow detection* does not find an overflow, INTREPAIR produces a correct repair, ❾ and it confirms the ❿ *removed overflow*, indicating that the fault was successfully removed and ⓫ *repaired source code* has been synthesized. Further, in case the integer overflow was not removed, then INTREPAIR produces ⓬ an *unremoved overflow* report, and the result is ⓭ *unrepaired source code*. Further, note that we only ensure that the previously detected integer overflow was repaired. The program may still be exposed to other types of faults which were potentially hidden/masked by the fault which was just repaired.

Lastly, note that for verifying a repair after it was applied, INTREPAIR does not require any test case since the integer overflow component can decide if the fault was removed. However, a test case *can* be used in conjunction with the fault detection component in order to confirm that the fault was removed from the source code location where it was first detected before repairing.

4.2 Overflow Detection

In order to generate code repairs, INTREPAIR initially needs to detect the precise location where the integer overflow resides in the program. This is the goal of INTREPAIR's repair location search algorithm, which we now present. First, INTREPAIR constructs the CFG of the analyzed program. Next, the following steps are performed consecutively in order to detect a fault.

1) Each program execution path is extracted from the previously generated program CFG.
2) The extracted path is traversed and path satisfiability checks are performed at branch nodes.

3) When INTREPAIR encounters an integer fault prone code location (e.g., assignment statement) on the analyzed path, an integer overflow check is performed by notifying the interpreter.
4) The notification is delegated to the appropriate checker (e.g., integer overflow checker) by the interpreter.
5) The slice of SMT equations of the symbolic variable, which potentially may overflow, together with corresponding integer overflow satisfiability checks is queried by the integer overflow checker.
6) The check verifies if the symbolic variable, which potentially caused the integer overflow, can be greater (if this is true, then an integer overflow is reported by providing its fault identification information which contains the key information, in particular the line number of the fault) than the currently used integer upper bound value (i.e., INT_MAX). These upper bound values are extracted from the C standard library contained in the limits.h file. The lower bound is obtained by negating the currently used upper bound value and and adding 1 to the result.
7) In case the SMT solver replies SAT (satisfiable, integer overflow fault present) to the previously submitted SMT query, then a problem report with problem ID (unique system string, the ID refers to which checker detected the fault), the file name where the fault was detected, and the line number in the file where the fault is located will be created and stored.

Optimization. INTREPAIR searches for integer overflow locations on each analyzed path which are most fault prone to integer overflow such as assignment or multiplication statements and discards for now other places. Further, due to INTREPAIR s backtracking capabilities, see Section 2.2 paths are not always re-analyzed from the start. Essentially, during path analysis our engine, in case of a not satisfiable path, backtracks up to the next upper branch node. Next INTREPAIR starts to analyze an alternative path reusing part of the SMT constraints which it has gathered up to the path branch node. This avoids recomputing all path constraints

Fig. 3. Decision tree used by INTREPAIR for storing and retrieving repair patterns.

for that analyzed path from the beginning of the path. Lastly, this optimization helps to save a lot of runtime overhead (approx. 50 percent according to our experience).

4.3 Repair Patterns

In this section, we present the repair patterns available in INTREPAIR. We explain how these patterns can help to repair integer overflows and how the patterns can produce correct repairs. Repair patterns are stored in a decision tree. This decision tree is, before the fault repair analysis is started (before using INTREPAIR), manually constructed by INTREPAIR's maintainer and stored in a tree data structure for later usage. The tree construction algorithm is based on several types of C language statements. More specifically, the components of these statements, are decomposed into AST trees and mapped onto the tree nodes. As such, we have from the root node to the leave nodes the operator used in the faulty statement, the type of operations, the type of the result variable, and other information such as if the operands are constant variables and if these, for example, have side-effects. The main rationale behind the decision tree is to help INTREPAIR perform a time-efficient search for each detected fault and to pick a core repair that fits best.

Fig. 3 depicts a decision tree used to show repair patterns available to INTREPAIR. The series of dots indicate other operations (e.g., subtraction, bit shift, etc.) or arrows not depicted in this figure due to page constraints. Note that other repair patterns for `int64_t`, `int`, `short`, and `unsigned int` types are not depicted in order to not clutter Fig. 3. This tree contains, for example, $C1$ up to $C24$ which are the criteria needed by INTREPAIR for choosing a repair pattern. A repair is selected when at least two criteria are met. As such, multiple repair patterns may apply for a single repair location in case these satisfy the criteria of multiple patterns.

For example, the six arrows on the left bottom pointing up depict six repair patterns determined between $C1$ and one of the criteria from $C7 - C12$. Depending on whether

one of the components of the analyzed statement `type x = y operator z;` is a constant, a variable or a variable with side effects (e.g., `int z = i++;` or `short y = foo();`) a different repair pattern will be proposed.

More specifically, from top to bottom, the decision tree contains an addition (left node) and a multiplication (right node) operation that are used to determine the kind of operation, which is performed in the faulty statement. At the next level in the tree (top to bottom), the type of result variable is determined. Further, at the next level in the tree, it is determined if the parameters of the analyzed statement are constants, variables or have side effects. Fig. 3 depicts in total 360 repair patterns (360 possibilities to combine two criteria from the range $C1$ to $C12$ with each other, e.g., $C1$ and $C8$, see corresponding arrow in Fig. 3) multiplied by five (data types) and the result is then multiplied by two, two main tree branches, e.g., `add` and `multiply`) for the C statement `type x = y operand z;`.

Table 1 depicts four notable repair patterns in more detail. We omit the contents of the *if* and *else* branches in order to provide simple to understand repairs. The *if* branch contains the code to log the occurrence of an integer overflow or the code which terminates the execution of the program.[5] The *else* branch contains a statement where the integer overflow was detected.

Consider the C statement 544. $a = b * b;$ where an integer overflow was detected at line number 544 in a program's source code file. In this case, the pattern depicted in Table 1 on row one ($C15$ & $C19$) will be used to avoid the fault. This pattern will be selected based on the fact that two equal variables are multiplied. As a result, the statement will be surrounded with the code of the pattern in which parts are replaced based on the type of statement, 543. $if((b > 0 \ \&\& \ b >= sqrt(INT\_MAX)) \ || \ (b < 0 \ \&\& \ b < -sqrt(INT\_MAX)))$

---

5. Note that terminating the program is an option that is not safe in critical program executions, e.g., in case monetary transactions depend on the program execution or when the program is driving a train, etc.

TABLE 1
Four Repair Patterns of INTREPAIR

Criteria	Statement	Repair Pattern Format	Description
C15 & C19	`char a=y*y;`	```1 if(y>sqrt(INT_MAX)|| 2 y < -sqrt(INT_MAX)){ 3 log_or_die();} 4 else{...}```	multiply two equal variables
C15 & C17	`char a=y*3;`	```1 if(y > INT_MAX/3 || 2 y < INT_MIN/3){ 3 log_or_die();} 4 else{...}```	multiply a variable with a constant
C3 & C7	`char a=y+z;`	```1 if(y > INT_MAX - z || 2 y < INT_MIN - z){ 3 log_or_die();} 4 else{...}```	add two variables
C3 & C11	`char a=y+4;`	```1 if(y > INT_MAX - 4 || 2 y < INT_MIN - 4){ 3 log_or_die();} 4 else{...}```	add a variable with a constant

{545. *log_or_die*(); }546. *else*{ $a = b * b$; }. Note that the numbers (543 up to 546) used in this example represent the source code line numbers of the repaired program, respectively.

How to construct new repair patterns? The repair patterns are constructed programmatically in Java. More precisely, each repair pattern is contained in a Java method which implements a template where the template meta-variables are meant to be filled with C code (if and else statements and program variables) which are identified during fault detection. The programmer interleaves the fixed, generic part of the code with the parts which are dependent on each specific detected overflow.

How were the repair patterns designed? We first manually analyzed the Juliet test-suite for identifying the relevant repair patterns. Essentially, each code repair pattern is constructed by first analyzing the C source code statement in which an overflow could appear. Then, we identify a program branch condition which repairs the overflow if placed in such a way that it encloses the faulty statement. The goal of the if based condition is to avoid the specific computation which could lead to an overflow in the potentially faulty statement.

How does a repair help to fix a fault? A generated repair helps to fix a fault by providing to the programmer a finished repair that contains complex mathematical constraints, which are not always obvious.

Why are the produced repairs correct? A produced repair is correct by construction w.r.t. to the repair correctness Definition 3. On the implementation side, after repair insertion, the whole program together with the repair is recompiled; this discard potential faults introduced by the repair.

Are the repair patterns complete? To our experience, the union of all repair patterns covers all relevant cases we have listed. However, we make no claim about formal completeness, and there may exist corner cases that are not covered by the patterns.

Lastly, note that building a repair is not straightforward, as the construction relies on complex mathematical constraints that need to be inferred based on the currently detected fault. Next, these constrains need to be carefully plugged into each repair pattern in order to provide the final repair.

4.4 Build Repair

The automated repair generation algorithm of INTREPAIR consists of the following steps. Determine the integer upper bound value (Step 1: Section 4.4.1); Generate an SMT constraint system (Step 2: Section 4.4.2); Select constraint values (Step 3: Section 4.4.3); Recompute the bound checking constraints (Step 4: Section 4.4.4); Determine the fault type (Step 5: Section 4.4.5); Select the repair pattern (Step 6: Section 4.4.6); Determine the new constraint SMT system (Step 7: Section 4.4.7); Generate code repair (Step 8: Section 4.4.8).

This algorithm corresponds to the box ❹ depicted in Fig. 2. The generated repairs are valid and correct according to Definitions 2 and 3. These eight steps are performed in consecutive order as highlighted next.

4.4.1 Step 1. Determine Integer Upper Bound Value

In order to determine the currently used integer upper bound value in the analyzed program, INTREPAIR performs the following procedure, which allows INTREPAIR to be a multi-precision tool in the sense that it automatically determines the integer precision needed for each analyzed program. First, INTREPAIR retrieves the hardware overflow limits[6] for each integer type. Second, during the symbolic execution-based traversal of the analyzed program path, INTREPAIR searches for previously defined and used integer upper bound program variables. This search is realized by comparing each declared or used variable name (e.g., `data` in the code snippet depicted in Fig. 1) contained in the currently analyzed program execution path with one of the supported integer upper bound values (e.g., `CHAR_MAX`, `INT_MAX`, `LLONG_MAX`, `SHORT_MAX`, and `UINT_MAX`). Third, in case such an upper bound value is found, it will be set to be the currently used integer upper bound value. INTREPAIR can automatically detect for each environment (system), in which it runs, the currently used integer overflow upper bound limit (i.e., `INT_MAX`) value. This way, the integer precision is determined for each analyzed program individually. Next, this upper bound value will be used to validate the generated candidate code repairs and also to search for integer overflows.

4.4.2 Step 2. Generate an SMT Constraint System

Next, the symbolic variables and the constraints used inside the integer overflow checker are grouped together and stored for later processing. In particular, INTREPAIR stores: (1) the statement where an integer overflow is detected, (2) the SMT formula used to detect the fault in the first place, (3) the fault ID of the integer overflow checker which was used to detect the fault, (4) the symbolic variable which was used to detect the integer overflow, and (5) other symbolic variables on which the integer overflow triggering variable may depend.

For instance, let us consider a C language assignment statement `int result = varA + varB;` and its SMT counterpart `(assert (= resSymbolic ( + varAsymbolic varBsymbolic)))`. The C assignment statement depicts the addition of two variables and the result is stored in a third variable, with a potential integer overflow fault. For this statement, the information collected by INTREPAIR is: (1) `int result = varA + varB;`, (2) `... (assert (= resSymbolic ( + varAsymbolic varBsymbolic)));`

---

6. e.g., by parsing `/usr/include/limits.h` on Linux OS.

`... checksat`, note that this SMT constraint represents a part of the SMT system used to detect the fault, (3) `ID-Integer_Overflow_Fault`, (4) `resSymbolic`, and (5) the `varAsymbolic` variable.

4.4.3   Step 3. Select Constraint Values

Next, INTREPAIR selects the relevant SMT constraint variables based on the type of C language statement where the integer overflow is detected. For example, consider assignment `int result = varA + varB;` where the variable `result` and potentially the variables `varA` and `varB` need to be taken into consideration, because these influence the occurrence of the integer overflow directly.

Further, the symbolic variable, `resSymbolic`, was selected by INTREPAIR to be further constrained. This is done in order to check if the code repair that will be generated could remove the previously detected integer overflow fault. For fault removal checking, the whole SMT constraint system slice of this symbolic variable will be used. Note that this constraint system was previously (before running the fault removal analysis) determined during fault localization. More specifically, this variable will overflow in case of using too large values that cannot properly be stored in the result variable. Note that depending on the complexity of the analyzed statement, more or fewer variables can be taken into consideration in order to determine if the previously detected integer overflow would further manifest after re-constraining. This depends on how the selected symbolic variables were re-constrained. The intuition is that the integer overflow can be detected before a certain variable will overflow. This type of behavior is useful when generating repairs that are constraining more than one program variable. Finally, the selected variables will be used in the next step when re-constraining the bounds during the checking of SMT constraints of the SMT system. Note that this SMT system was used upfront to detect the integer overflow.

4.4.4   Step 4. Recompute the Bound Checking Constraints

In order to solve the problem of determining a suitable variable range for avoiding an overflow, INTREPAIR applies a specific technique for re-constraining the symbolic variable which has overflowed and which was selected in the previous step. The variable can be re-constrained after collecting the program execution path constraints for a single program execution path, which were used to check for the presence of an integer overflow. The presence of an integer overflow fault is indicated if for the selected SMT system the Z3 solver reports SAT (satisfiable, integer overflow fault present).

INTREPAIR re-constraints (for example, through integer upper bound negation) the variable(s) selected from the previous step: the original SMT equation, `(assert (> resSymbolic 2147483647 ))` and the negated SMT equation, `(assert (<= resSymbolic 2147483647`, in order to determine a potentially safe interval which will not lead to a second integer overflow of the symbolic variable. Note that other iterative techniques are possible (e.g., iterating backwards through a vector (from large to small values) of consecutive large values and checking by selecting each value

once as integer upper bound if the check conditions will hold). The goal is to determine if there is a second integer overflow if INTREPAIR re-constraints the selected variables as mentioned above. For this purpose, INTREPAIR will check in the next step if for the new SMT constraint system it gets an UNSAT (unsatisfiable, no integer overflow fault present) solver reply. The new constrained SMT will be composed of the old SMT constraint system, which was used to detect the integer overflow, complemented with the re-constrained SMT equations. If INTREPAIR gets an UNSAT solver reply, then it determines that there will be no integer overflow if it re-constraints the selected variable(s) with the new constraints (e.g., variable range negation, etc.) and as such the integer overflow can be avoided. Finally, the information collected at this step will be used in later steps.

4.4.5   Step 5. Determine the Fault Type

For a unique detected overflow, INTREPAIR attributes a report containing a unique fault identifier. Based on the generated fault identifier (ID), INTREPAIR can determine which fault type it currently deals with. This information is extracted from data stored during step two. With this ID, INTREPAIR checks in the list of currently supported checkers to which checker this stored information belongs to and determines which repair patterns can be used to repair the previously detected fault. In this way, repair patterns usable for other types of faults will not be suggested when repairing integer overflows.

4.4.6   Step 6. Selecting Repair Pattern

Based on the previously determined fault identifier (ID) in the previous step, INTREPAIR selects the suitable repairs for this integer overflow from the pool of repair patterns using the decision tree (see Fig. 3 for more details), as discussed next.

Pattern. The repair patterns of INTREPAIR need to address the following challenges. Each pattern needs to incorporate complex conditions with multiple branches depending on the type of the potentially overflowing code location. An INTREPAIR pattern needs to be pre-classifiable for typical code locations (e.g., statements) where it would best apply depending on the AST of the faulty statement(s). Further, each pattern needs to have at least two branches: (1) in the case the check succeeds, and (2) in case the check does not succeed w.r.t. fault logging. The pattern needs to have several customizable components which can be altered during static analysis with: context dependent values, mathematical functions, or C functions extracted from the faulty statement. As such, INTREPAIR's repair patterns need to be nontrivial fragments of incomplete code, which have to be versatile and applicable to different integer overflow repair scenarios.

Repairing. The repair patterns impose the following challenges on INTREPAIR's repairing process. Our technique is fully automated and in case INTREPAIR generates more than one possible repair for a single fault, then the user gets a window shown in which he can decide which repair to apply. This step is optional and can be turned on/off from the configuration file of INTREPAIR. In case this step is turned off, then INTREPAIR, in case of multiple repairs generated,

```
1 if(leftHandSide.equals(
2 rightHandSide)&&
3 operator.equals("*")) {
4 ...
5 return "if(sqrt("+leftHandSide+") <= sqrt("+value4+") &&
6         -sqrt("+ rightHandSide+") >= -sqrt("+value5 +"))
7        "+"{"+"\n"+"\t"+"\t"+"\t
8           +faultyStm10 + "\n }else{ \n"
9           +"FILE *fp=fopen (\"IO_"
10          +"fault_log.txt\", + \"w+\");"
11          +"\n fprintf(fp, \"IO_ID:%s
12          +FileName:%s LineNumber:%d",
13          +FileName+","+IO_fault+","
14          +LineNumber+");"
15          +"\n fclose(fp);" + " }\n";
16 }
```

Fig. 4. Repair pattern example.

chooses the first generated repair and inserts it in the program. Before repair insertion the repair pattern is selected by INTREPAIR with the help of the decision tree. Further, the automated repair generation mechanism of INTREPAIR, before insertion, needs to be able (1) to extract the components of the faulty statement such as variables, functions (e.g., `malloc`, etc.) and reuse them for augmenting the selected pattern; to achieve this, the repairing mechanism needs to be able to match these extracted components with the parts of the pattern where these fit, (2) next, the repairing mechanism needs to be able to precisely incorporate the faulty statement(s) inside the newly selected pattern, (3) the repairing mechanism needs to be able to precisely delete the previous statement(s) where the fault was detected and to rewrite the existing source file with the newly generated repair pattern at the correct source code location. As a consequence, the challenges imposed by the repair patterns and INTREPAIR's repairing process are nontrivial and require high precision during source code file rewriting.

Repair Pattern Description. INTREPAIR's repair patterns consist of C code skeletons where different repair parts will be replaced with: (1) concrete values after their values have been computed, (2) mathematical operations (e.g., division by a value, etc.), and (3) standard C library functions. Depending on the context, placeholder variables will be replaced with corresponding mathematical functions such as the square root function `sqrt` or other functions. In the situation in which a mathematical function is used, INTREPAIR does not need to compute the value of the function upfront (during static analysis) but rather leave this to be computed later during symbolic execution analysis (repair validation) or program runtime. This offers the advantage that INTREPAIR does not need to be able to compute any possible mathematical function.

Further, the code repair patterns used by INTREPAIR are based on preconditions similar to IOC's [34] checks (see Section 3.4 for more details). These preconditions can cover different types of mathematical operations such as multiplication of numbers and addition of variables. At the same time, the repair patterns are highly configurable and versatile. For example, the programmer can easily change, if needed, the fault handling function inside the repair pattern or can extend the precondition such that it captures more complex fault avoiding preconditions. This can be achieved by modifying a few lines of code inside an existing pattern or by creating a new pattern and defining the conditions (e.g., depending on the

structure of the AST of the statement where the fault was detected) when such a pattern fits best.

Selecting Repair Patterns. Next, we describe the steps used for selecting a code repair based on a C statement having three components, `leftHandSide`, `rightHandSide`, and `operator`. This translated into the previous example are: `int result = varA + varB`. However, INTREPAIR can deal with more complex statements as well, see Fig. 3 for more details.

INTREPAIR follows the following steps to select a suitable repair pattern. First, the code statement where the integer overflow fault was detected is divided into its components based on its AST. For example the AST components of a C statement such as `int result = varA + varB;` are `leftHandSide = varA`, `operator=+` and `rightHandSide = varB`. Second, a series of rules are checked against the AST of the previous C statement as follows: (1) is `leftHandSide` equal or is different than `rightHandSide`, (2) what type of operator do we have in the statement, and (3) how many components does the statement have after the = sign, and so on. Third, based on these rules the repair pattern which satisfies the highest number of constraints determined by the previously mentioned rules will be selected. Note that each repair pattern has a list of properties attached to it (e.g., use when `rightHandSide = leftHandSide` and `operator = +`, etc.) that are checked against the above stated rules. This list of properties is statically defined when the repair patterns were manually added to the pool of available repairs inside INTREPAIR. Further, in case there are more repair patterns that fulfill the same number of rules, INTREPAIR selects the first repair pattern occurring in the list. In case INTREPAIR cannot determine the pattern selection criteria due to, for example, a complex C language statement, then no repair will be proposed.

Lastly, note that, if needed, this approach can be updated such that all legitimate repair patterns will be proposed and for each a repair can be generated, and selected with a human-in-the-loop approach.

Repair Pattern Example. In the following, we present a repair pattern used by INTREPAIR.

Fig. 4 represents a code repair pattern used by INTREPAIR during integer overflow fault repairing. This code repair pattern contains C code compatible snippets (shaded in red color) interleaved with several stub variables which will be replaced with concrete variables names, values or mathematical functions depending on the type (depending on its AST structure) of code statement containing the fault. For example, the repair pattern depicted in Fig. 4 will be used by INTREPAIR when the `leftHandSide = varA` is equal (e.g., string wise comparison) to the `rightHandSide = varB` and the `operator` is equal (e.g., string wise comparison) to the product operator *. After these checks have been performed, the repair will be assembled as presented in the following steps.

First, `value4`, `value5` and `faultyStm10` located at lines 5, 6, and 8 in Fig. 4, are replaced with: (1) the squared root of the currently selected integer upper bound value `value4 ← sqrt(2147483647)`, (2) the negated integer upper bound value `value5 ← -sqrt(2147483647)`, and (3) the program code statement that contains the previously

detected integer overflow fault `faultyStm10 ← int result = data * data;`. Second, the variables `FileName`, `IO_fault` and `LineNumber` located at lines 15-16 in Fig. 4 are replaced with concrete values obtained during fault detection. Finally, note that: (1) other code repair patterns can be selected and used based on the format of the AST of the program faulty statement, (2) our technique can be easily generalized to more complex C code statements than the ones mentioned herein, and (3) each generated repair can easily be customized to fit to different types of integer overflow mitigation scenarios, e.g., fault logging, etc.

4.4.7   Step 7. Determine New Constraint System

In this step, INTREPAIR assembles the new SMT constraint system which is used to determine if the previously detected integer overflow is still present. During this step, INTREPAIR takes the constraints determined at the previous step (see Section 4.4.6 for more details) and inserts them in the SMT constraint system which was used to detect the integer overflow. Before inserting the new constraints in the SMT system, INTREPAIR needs to remove the original SMT constraints which were used to detect the presence of the integer overflow as follows. `(assert (> resSymbolic 2147483647 ))` is replaced with: `(assert (< resSymbolic 2147483647 ))`. The decision of removing these SMT constraints that are used to detect the integer overflow is made based on logical inter-dependencies of these SMT statements. More precisely, since the SMT constraints used to check for an integer overflow are added in the integer overflow checker in an incremental manner, INTREPAIR can precisely determine which constraints can be safely removed and replaced with new ones determined at the previous step. Next, this SMT system will be fed into the Z3 solver. In case the solver replies `SAT`, then the constraints added represent valid constraints which can be used to avoid the previously detected integer overflow. This new SMT constraint system serves as ground truth w.r.t. the fact that the integer overflow can be removed if certain symbolic variables are re-constrained in an appropriate way. Finally, note that the above-described symbolic variables have concrete counterparts values, which will be inserted inside the code repair when assembling it.

4.4.8   Step 8. Generate Code Repair

This step consists of putting together the final code repair(s) and saving them into a list in case multiple repairs are suggested. After the repair components have been inserted into the previously selected code repair pattern, INTREPAIR generates a C source code repair which is syntactically correct, can be compiled and can be further on edited after insertion (if desired).

4.5   Implementation

We have implemented the approach presented in Section 4 in a prototype named INTREPAIR. It is an Eclipse [37] plugin based on Codan [38]. INTREPAIR consists of approx. 10 KLOC and is developed as an Eclipse plug-in mainly because: (1) the Eclipse CDT API can be easily reused, (2) a GUI is easily obtainable, and (3) the obtained tool can be used in both online (during code typing) and off-line (after finishing code typing) modes. Note that Codan [38] is used by INTREPAIR to construct the program CFG, analyze the program AST statements, and perform bottom-up traversals by using a C program statement visitor in order to construct SMT constraints. Three additional auxiliary tools are implemented as well to work with INTREPAIR.

The assembled repair generated in Section 4.4.8 is sent to the Eclipse language tool kit (LTK) API [39] based component of our engine, which will assemble the repaired code. The LTK component adds information on how to position the repair in the faulty program such that the integer overflow will not occur after the repair was inserted. In case multiple repairs have been generated, INTREPAIR saves these repairs in a list of repair candidates for the previously detected integer overflow. Lastly, the repair which was selected by the programmer will be passed to the INTREPAIR repair insertion component, which will create two differential views, with the repair inserted in the file containing the fault and without.

Next, we briefly describe the three additional tools which we implemented. First, we have a generator of Eclipse CDT projects which is used for generating Eclipse CDT compatible projects from Juliet cases. Second, we have a source code refactoring tool based on the Eclipse LTK [39], JFace [40], and Eclipse CDT [37], for inserting repairs at the source code level. Third, we have a specific test case generator to assess the correctness of IntRepair. This tool works as follows. First, the assessed Juliet test suite program is manually annotated with strings that localize the fault (with an annotation one code line before the code line which contains the fault). Second, our test case generator tool parses the source code of the faulty programs (our test resources) and searches for this annotation. Third, it generates test cases based on code skeletons which contain, for example, the line number where the true positive fault was detected as well as the file name and function name in which the fault was previously detected. Finally, when a generated test case is executed, it is checked whether a reported fault is located at the line number, in the function and in the file which were previously annotated.

4.6   Graphical User Interface in an IDE

Fig. 5 shows the INTREPAIR Graphics User Interface (GUI). First, the integer overflow triggers a fault marker depicted in the black bordered box with a yellow fault icon. It means that on the C statement an integer overflow fault was detected. Second, by right clicking on this fault marker the user can start the code re-factoring wizard. The code re-factoring wizard is composed of two windows. The first window is used to make repair type decisions (currently only in-place repairs are available). The second window depicted in Fig. 5 (in the background) contains a differential files view visualizing the differences between the original file containing the fault and the modified file with the selected repair inserted. This second window helps the developer to analyze the repair before it is applied. Finally, it is possible to navigate between these two windows. In case the programmer decides to insert the generated repair (see Section 4.4), then this can be achieved by left-clicking on the `Finish` button.
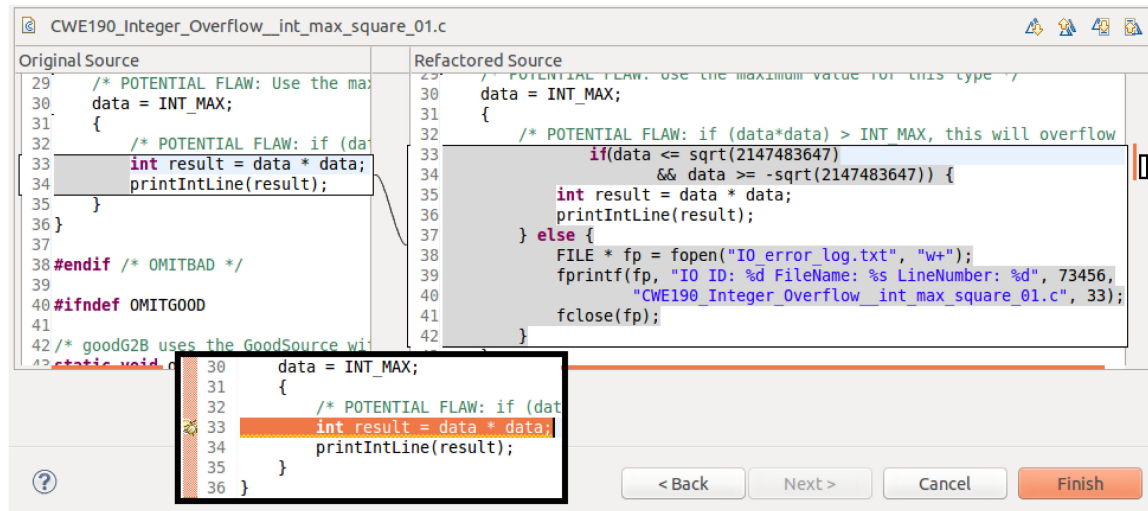
Fig. 5. INTREPAIR's view of the localized fault (left hand side) and the program repair (right hand side).

5 EVALUATION

In this section, we present evaluation results and address the following research questions (RQs):

- *RQ1:* How effective is INTREPAIR in repairing integer overflows? To what extent does INTREPAIR produce false positives? To which degree does INTREPAIR ensure that a repair completely removes the previously detected integer overflow?
- *RQ2:* What is the performance impact of INTREPAIR?
- *RQ3:* How do the fault repairs of INTREPAIR preserve program correctness?
- *RQ4:* How effective is INTREPAIR compared to manual repair of integer overflows?

5.1 Evaluation Setup

Table 2 depicts a summary of the programs used to evaluate INTREPAIR. Next, we give details about the used programs.

Juliet Benchmark. First, we selected 2,052 C programs contained in SAMATE's Juliet test suite [15], which is the largest open source test suite for C/C++ code to the best of our knowledge. From the set of all integer overflow test cases, we kept all 2052 C programs, but we excluded the 198 C++ programs which are out of scope of our work. In particular, we consider the CWE-190 category. Each program contains on average 476 LOC with a maximum of 638 LOC. Each program has integer overflows: exactly one true positive and several false positives (see characteristics of the Juliet test suite [15] for more details). We used these programs as they contain all possible program control flows which may lead to an integer overflow. Further, these help to assess previously known faults in a systematic way.

Synthesized Code. In addition, we build a benchmark with 50 synthesized programs with complex control flows and with a high number of code lines. We synthesized the programs by first manually selecting five representative seed programs contained in the Juliet test suite. For these original programs, we added additional code in order to increase the number of LOC in order to reach up to 20 KLOC. The added LOC represent chains of function calls and loops which cause the program to loop for numerous times through the

code as the program runs. The objective is to trigger the same overflow but in a convoluted manner.

In each synthesized program, the exact number of seeded integer overflows is known. The benchmark allows us to demonstrate that INTREPAIR can scale efficiently to large, and complex programs. The synthesis of programs is parameterized as follows: (1) the total number of function calls, (2) the number of loop iterations, and (3) each generated program contains exactly one true positive and a variable number of false positives, and the true positive should be located deep inside the program, e.g., several thousands of branches nested inside the program execution tree. Using these limits, we generate 50 programs ranging from 6 to 20 KLOC. The synthesis algorithm iterates through several loops in which it adds functions in the program which are calling each other. These functions contain a variable number of branches which are also counted until a certain depth is reached.

Statistics. Table 3 depicts several characteristics of the selected 2,052 analyzed programs contained in SAMATE's Juliet test suite and the 50 programs of our benchmark. The number of satisfiable program paths refers to paths for which the symbolic analysis component of INTREPAIR was able to reach the target fault. An unsatisfiable path is obtained when the symbolic analysis of INTREPAIR gets an unsatisfiable reply from the SMT solver.[7] Further, Table 3 depicts the number of analyzed program paths which were extracted from the program control flow graph. Next, in Table 3, we report the total number of branch nodes as well as program branches (C branch statements) along all program paths which INTREPAIR has analyzed. Table 3 shows that INTREPAIR can scale to large programs.

The programs contained in the CWE-190 contained on average: 56 satisfiable paths, 584 unsatisfiable program execution paths, and 5,800 program branches, which were counted during program analysis. In contrast, the programs contained in our own benchmark contained on average: 900

---

7. The number of paths for the 50 synthesized programs of the benchmark is smaller compared to the 2,052 C programs because we focused on adding loops and not on adding branches. For this reason, it makes no sense to compare those two numbers.

TABLE 2
Overview of the Subject Programs

Subject Prog.	LOC	# Programs	Description
CWE-190	up to 638	2,052	Open source testsuite
Benchmark	up to 20 K	50	We introduce a benchmark of synthetic programs

TABLE 3
Descriptive Statistics of our Subject Programs

Subject Prog.	Satisfiable Paths	Unsatisfiable Paths	Program Branch Nodes	Program Branches
CWE-190	115 K.	1.2 Mil.	5.73 Mil.	12 Mil.
Benchmark	45 K.	0.7 Mil.	4.4 Mil.	8.8 Mil.

satisfiable paths, 14,000 unsatisfiable program execution paths, and 28,000 program branches, which were counted during the analysis of the programs.

Note. Unfortunately, we cannot compare INTREPAIR with any existing approach as there are no open implementations which can statically analyze source code, detect the integer overflow and provide a repair at the source code level. Also, we are not aware of any static C source code analysis tool which can be used to run on the same programs as INTREPAIR.

Test Case Generation. The test cases were generated automatically by a tool which was specifically created for this purpose, described in Section 4.5.

User Study. We also performed a controlled experiment with 30 participants in which we assessed the efficiency of INTREPAIR during integer overflow repair. The used protocol is described in Section 5.5.

Experiment Setup. Experiments were conducted on a Dell desktop with an Intel CPU Q9550 @ 2.83 GHz, 64-bit, 12 GB RAM, Eclipse IDE Kepler and OpenSuse 13.1 OS.

5.2 Effectiveness (RQ1)

5.2.1 Fault Detection

In this section, we show the effectiveness of INTREPAIR w.r.t. fault detection. As our main test-bed, we select 2,052 C programs contained in SAMATE's Juliet test suite [15] and 50 programs from our customized benchmark.

Methodology. We prepare the 2,052 C programs in Juliet in order to be handled by INTREPAIR. Further, we generate 2,052 test cases which are used to dynamically assess if the detected integer overflow faults are detected at the correct source code location. For the automated generation of these test cases we used the automated test case generation tool presented in Section 4.5. We conduct the same pre-processing with the 50 programs from our own benchmark.

(1) We assess the effectiveness of integer overflow detection by running the generated test cases for all programs (including the 2,052 C programs in the Juliet test suite and 50 C programs contained in our benchmark) and checking the generated reports. More precisely, we checked whether each report describes an integer overflow reported at the expected location. (2) For those reports which are confirmed as true positives, INTREPAIR inserts a suitable repair into the overflow site. (3) Then, INTREPAIR was run again in order to check if this was a true positive and if it was removed.

We point out that, for the assessed programs, INTREPAIR did not detect any false positives. Further, INTREPAIR is able to detect and repair all previously detected true positives present in the analyzed programs without repairing any false positives. Finally, note that the ground truth is that each of the 2,052 C programs contains one genuine integer overflow (if detected this generates a true positive) and multiple not genuine overflows (if detected these generate false positives) and the same is true for the programs contained in our benchmark.

> *Results.* According to our experiments, INTREPAIR is able to detect all integer overflows (2,052 overflows contained in the Juliet test suite and 50 in our benchmark) at the expected locations in the analyzed programs. After the repairs conducted by INTREPAIR, the integer overflow reports are no longer generated. This means that INTREPAIR is able to perfectly repair all integer overflows for the analyzed programs.

5.2.2 Fault Removal

In the section, we explain how the integer overflows are completely removed by INTREPAIR. Recall that INTREPAIR checks if a fault was completely removed after a repair was inserted in two operation modes.

Methodology. In this experiment, we use INTREPAIR's *Manual Mode.* In this mode, the user can again analyze the repaired program by manually restarting the static analysis. In case the symbolic analysis detects no other (or the previously fixed) integer overflow, then no fault report will be generated. This helps the programmer to conclude that the fault was completely removed and no new fault was inserted into the program. Indeed, we used INTREPAIR to re-analyze each repaired program. For each of the repaired programs, we checked if after the program was repaired any new fault report was filed.

> *Results.* After repairing and subsequently reanalyzing each program, no new overflow reports are generated, meaning that the faults are completely removed without inserting new ones. For all programs, the faults are successfully removed by inserting the repair at the correct location.

5.3 Performance (RQ2)

In this section, we evaluate the performance of INTREPAIR in three ways (1) the time INTREPAIR takes; (2) binary size blow-up, and (3) the runtime overhead caused by the overflow repair. Finally, we show that INTREPAIR has the potential to scale to large programs as well.

5.3.1 Static Analysis Time

The time that INTREPAIR spends in static analysis, i.e., static symbolic execution and overflow repair processes, is an important criterion to assess the usability of our tool.

> *Results.* Table 4 depicts the average execution time over 10 runs for each of the benchmark programs grouped in three main categories based on their LOC. INTREPAIR handles all programs under 10 Min. (567 sec.) on average. Consequently, we consider INTREPAIR to be time-effective and usable in a development environment.

TABLE 4
Average Repair Generation Time in Seconds

6 KLOC Programs	46 sec. ($<$ 1 Min.)
11 KLOC Programs	151 sec. ($<$ 3 Min.)
20 KLOC Programs	567 sec. ($<$ 10 Min.)

5.3.2 Binary Size Blow-up

We assessed the source code and binary blow-up by counting the increase in source code lines and in bytes for the programs before and after applying the repairs.

First, we compared the total number of lines contained in the source code against the number of lines of code which were added after inserting all the repairs into the 2,052 vulnerable programs. As already mentioned, the initial number of lines was 977.7 KLOC. After applying all repairs, we added in total around 10,045 LOC.

> *Results*. INTREPAIR yields a binary blow-up of less than 0.56 percent in LOC. In our opinion, this represents an acceptable source code lines increase.

5.3.3 Execution Overhead

We evaluated the runtime overhead introduced by INTREPAIR by comparing the execution time of un-repaired programs against the one of the repaired programs. This is made possible since each analyzed Juliet program has its own input workload coming from the official Juliet test suite. We use this workload as is in our experiment. For the 50 synthesized programs, we use the input workload of the corresponding seed Juliet program (recall that we use five seed programs from which we generated our synthetic programs.)

> *Results*. In our experiments, we recorded an average runtime overhead of 1.57 percent. Thus, the inserted repairs do not considerably influence the runtime overhead of the repaired programs.

5.4 Correctness (RQ3)

We verify the correctness of our repairs by checking that our symbolic execution component generates semantically different code. In other words, we want to find out if the repairs influence the program behavior and whether the repair correctly removes the fault (see repair correctness Definition 3).

To achieve this goal, we compare the generated SMT model before and after the repair. For this, INTREPAIR's symbolic execution component stores the SMT system which was used to trigger the fault report. Next, this constraint system is compared with the new SMT system which was generated after repair validation. This results in the following analysis process:

1) For all programs contained in our benchmark and the Juliet test suite, we store the SMT system for each program and the node, containing source code line number and file name, where the fault is located.
2) We apply the repair to the previously detected integer overflow by automatically rewriting the program.

3) We re-run the symbolic analysis on the repaired program and store the new SMT system.
4) We semantically checked each SMT system stored at step (2) and each SMT system collected at step (4) based on string comparison of the node with another node, where a node is a complete statement, e.g., x = p + 5. This enables INTREPAIR to detect the differences between these two systems.
5) We compare the SMT system differences and proceed as follows: (1) if the new SMT system only reflects the semantics of the inserted repair: we report no program correctness violation, (2) otherwise, we report that the program correctness will potentially be affected by inserting this repair.

We automatically compared in total 225K SMT models and all repaired programs. These differ by up to five AST nodes and by less than three SMT constraints on average, respectively. Further, all checked SMT models differ only with respect to the repair semantics and no additional program semantics were introduced in the program due to the repairs.

Additionally, we performed the following evaluation process in order to increase the level of confidence w.r.t. repair correctness.

Recheck Repaired Program. We re-run the symbolic analysis, after applying the repair, on each of the programs in order to see if there is a potential new integer overflow fault or the old one is still present in the repaired program.

Check Syntactical Program Correctness. We recompile (we use the GCC compiler) each of the programs to investigate if the repaired program is compilable and thus syntactically correct. The compiler reported no syntactical errors.

> *Results*. The normal behavior of the repaired programs does not change after applying the repairs.

5.5 User Study (RQ4)

In order to evaluate INTREPAIR's effectiveness w.r.t. integer overflow repairing compared to manual repairing, we performed a user study with 30 graduate students (18 males and 12 females). We asked all potential participants about their programming experience, and we selected only the ones which had a C/C++ programming experience. Participants had on average between 1-3 years of industry programming experience.

Methodology. The experiment was conducted with each participant individually placed at a single computer with an additional person in the room who overlooked the whole experiment. In this way we avoided bias towards the fact that if several participants were in the same room at the same time then they could influence each other. The computer used in our experiment was equipped with two versions of Eclipse CDT. One computer had an Eclipse version with INTREPAIR installed and the other computer had an Eclipse version without INTREPAIR installed. In order to avoid gender bias, we split the 30 participants evenly in two groups in such a way that the number of females and males was split evenly between the two groups. We *randomly* selected in total two programs from our benchmark, and one program from the Juliet test suite. Next, we told each participant, of both groups, to search for a single integer overflow in each of the

three programs and to repair it. We instructed each participant that the usage of Eclipse IDE is recommended.

The participants were allowed to read and execute the program. Further, all participants had similar experience with the task at hand, particularly fault detection, as they had 1-3 years of experience as developers in different software companies. Lastly, in the case of manual debugging no test cases were prepared for simplicity reasons. The experiment supervisor assessed each fault detected by the participants by manual inspection after it was reported. In order to avoid bias, the participant was not given any feedback on the reported bug.

Group 1. Each participant had access to the latest Eclipse CDT IDE and to the GCC (v. 4.9.3) compiler [41] through terminal access. The Eclipse CDT IDE in this group were installed without INTREPAIR.

Group 2. Before the experiment, each participant from this group watched a short one minute demo movie showing how to use INTREPAIR. Each participant could search for faults and repair them with the help of INTREPAIR. In this way, we avoided bias related to how fast or slow a participant adapts to the new tool to use during the fault detection and repair task.

We measured the time needed for each participant to locate the fault and repair it and the success rate for each analyzed program after the participant decided that he/she was finished with all three programs at the end of their analysis. After the experiment, we asked each participant if (1) *he/she would reuse* INTREPAIR *in his/her daily routine* and if (2) *he/she would recommend it to other peers*. Each question could be answered with *yes* or *no*.

First, in total, the participants of *Group 1* needed 4,020 seconds (64 Min.) to repair the faults in the experiment, whereas participants from *Group 2* only needed 348 seconds (5.8 Min.) to repair them. That means, programmers using INTREPAIR in our experiment were over 11 times ($4,020/348$) faster compared to relying on debugging and manual repairing.

Second, we inspected the repairs inserted manually by *Group 1* and those inserted by INTREPAIR. 38 percent of the manual repairs were correct, i.e., the integer overflow faults were removed and no further vulnerabilities were introduced. By construction, all the repairs with the help of INTREPAIR were correct (see *RQ2* for more details). Hence we conclude that INTREPAIR is more effective w.r.t. overflow repair than manual analysis.

Third, out of the total participants of *Group 2*, 80 percent (12 participants) found INTREPAIR to be very useful and 93 percent (14 participants) would further recommend INTREPAIR to their peers. We informally asked these participants about their experiences with INTREPAIR after the experiment. First, the participants provided positive feedback about the usability of the user-interface. Second, having used advanced software engineering tools and/or Eclipse IDE in the past provided an immediate feeling of familiarity with INTREPAIR's concepts and likely also contributed to better task performance.

Lastly, a significant amount of time was needed by the participants in *Group 1* to determine the correct form of the repair. This comprised writing and ordering the conditions, and determining which variables should be best used to properly repair the overflow. Further, the participants reported that they spent a lot of time (around 35 percent) on evaluating if the repair is correct and if it really removes the fault.

> *Results.* This user study shows that INTREPAIR is usable and improves the repair of integer overflows compared to traditional program repairing which relies mostly on debuggers.

6 DISCUSSION

In this section, we discuss the threats to validity of our experiments and the limitations of INTREPAIR.

6.1 Threats to Validity

Internal validity depends on the correctness of our prototype implementation. It may also be affected by the evaluation setting and the execution of the conducted experiments. To mitigate the concern about runtime performance validity, we carefully repeated the experiments ten times and took the average performance values. Further, to validate the correctness of our automatically generated repairs, we inspected once each program repair manually.

External validity threats may derive from the selection of programs which we used in our evaluation. We validated our repairs on 2,052 programs and also on seeded programs from our benchmark. This benchmark has been especially designed for mitigating this threat.

Construct validity threats may appear from the fact that our repairs may introduce overhead or unwanted program behaviors. We point out that our repairs are suggested only for code locations where an integer overflow was previously confirmed. Further, our experiments show that our repairs do not harm performance in a significant way, do not change an observable program behavior.

6.2 Limitations

Restricted C Language Semantics. IntRepair covers all program constructs contained in the selected Juliet programs. Yet, the used static symbolic analysis framework (Codan) only supports a subset of the C/C++ programming language semantics, which means that full-fledged C code cannot be handled. Nevertheless, we do not consider this to be a major scientific limitation, adding all C/C++ statement types and function headers can eliminate this with sufficient engineering manpower. INTREPAIR currently cannot handle all C language semantics and as such we cannot run it on C programs. To our knowledge, no other open source detection and repair tool exists that can handle the full semantics of the C programming language. For this reason, it is also quite difficult to compare our tool with others. Further, in this paper we propose an original idea with a proof-of-concept implementation. Future efforts towards commercial usage of the technique will make it work for arbitrary programs.

Complex C Language Constructs. Complex C program expressions cannot be handled due to limited support in the currently available open source state-of-the-art SMT solvers for non-linear computations. Thus, we want to address this potential limitation by carefully providing SMT constraints which closely capture the original program constraints and by experimenting with different SMT solvers.

Loops and Recursion. INTREPAIR's implementation depends on program loop and recursion depth unrolling which incurs well-known precision penalties for all symbolic execution-based techniques, and these operations may, as a result, introduce false positives. Note that INTREPAIR does not produce false positives by design as each detected fault is a genuine one. Our static analysis is time-consuming. Its accuracy and performance will affect INTREPAIR's results. INTREPAIR does not have access to runtime information and a real environment is also not available. In order to deal with this limitation, INTREPAIR uses function stubs which can help to simulate the environment (interaction with third-party libraries). Note that loop unrolling and recursion depth have to be bounded when using symbolic execution-based approaches in general since we want to avoid the path explosion problem and thus infinite analysis time. We want to address this potential insufficiency by a prior analysis of program loops in order to derive loop invariants. Further, we plan to extend the list of currently supported stub functions for the main C most used system libraries.

Path Explosion Issues. INTREPAIR relies on static symbolic analysis of source code. It has several limitations which are common to all static analysis techniques when comparing to dynamic analysis techniques such as the well-known path explosion problem. Note that static searching through the program paths is different from dynamic path analysis where program path coverage is driven by the provided program input. We addressed this issue in INTREPAIR by implementing different path exploration techniques in order to analyze preferentially more fault-prone paths.

Test Programs. INTREPAIR's evaluation is based on the currently largest open source test suite for C/C++ programs, i.e., SAMATE's Juliet test suite, which contains complex control flows and a large number of situations where integer overflow can occur. The programs contained in our benchmark are comparable with large and complex programs. As a result, the findings of our evaluation do not necessarily reflect the behavior of INTREPAIR when applied to larger programs. However, we do not think that this limits the applicability of INTREPAIR, since our tool is highly scalable due to its configurable analysis. Further, we want to evaluate INTREPAIR on even larger programs as well.

False Positives/Negatives. While comparable with real-world programs, the test programs for INTREPAIR are relatively compact programs. Within this context, we have not identified any false positives or false negatives. However, due to the complexity of the task at hand, we expect that in even more complex and large real-world programs, INTREPAIR would report false positives.

User Study Results. Lastly, we tested INTREPAIR in a controlled experiment with a restricted number of participants. For this reason our findings may differ from industrial settings where real development conditions are available. One limiting aspect is that not all of the participants may have used Eclipse in the past for developing in industry. Nonetheless, we expect that our tool can help to drastically cut down the time needed for fault detection and repair due to its usability and low intrusiveness. In future work, after implementing all C language specific semantics needed for real-world programs, we plan to evaluate INTREPAIR also in industry settings as well.

6.3 Repair Pattern Generalization

The repair patterns used by INTREPAIR generalize beyond the programs used in the benchmarks for the following main reasons. First, the decision tree used to select a repair is general enough and its nodes can be extended with other operators and variable types such that for other types of C language statements suited repairs can be inferred. Second, the repair patters can be manually extended and customized very easily towards other needs or specific programs. Our repair patterns are designed such that the programmer can tailor the repairs towards his particular needs by updating the repair selection tree and the list of available repairs. Lastly, this makes our repair patterns flexible, customizable and usable for different programs.

7 RELATED WORK

There is a large body of research work focusing on integer overflow *detection*: Senx [42], ARCHER [43], UQBTng [44], PREfast [45], Rich [19], SAGE [46], CBMC [47], IntScope[3], Brick [48], IntFinder [49], SmartFuzz [50], PREfix [51], IntPatch [52], IOC [34], IntFlow [53], SoupInt [6], SIFT [54], TAP [55], Diode [56], Indio [57], Zhang et al. [58], and IntEQ [20].

In contrast, only few approaches focus explicitly on integer overflow *repairs*: CIntFix [59], IntPTI [60], SoupInt [54], CodePhage [61], TAP [55], and SIFT [54]. Out of these approaches, only TAP [55] (technical report) and SIFT [54] first explicitly detect the integer overflow and then repair it. The other tools—which do not first confirm fault existence—mostly blindly change the code in all fault prone locations in the hope to avoid integer related problems. For example: (1) CIntFix [59] utilizes integers of infinite size with two's complement encoding in place of original bounded integers, and (2) AIC/CIT/RAO [62] relies on three code transformations: add integer cast (AIC), replace arithmetic operator (RAO), and change integer type (CIT) to change the program in order to avoid integer overflows. However, these tools have a high runtime overhead.

SIFT [54] first detects the integer overflow (fault detection is based on Diode [56]) and then generates an input filter to eliminate the fault at the binary level. SIFT is a static input filter generation tool, which inserts input filters in the program binary for which the source code was previously manually annotated with source code annotations. The main drawback of SIFT is that it relies on tedious user source code annotations.

In the above-mentioned approaches, fault localization is performed before repair generation. In contrast, we combine fault localization and repair generation, and as a result, we obtain the capability of generating precise repairs which remove the fault for various program inputs (fault detection is not program input independent) and at the exact location where the fault was detected upfront. For this purpose, we use SMT solving for fault localization. However, unlike INTREPAIR, SMT solving is not used for repair synthesis by TAP or SIFT. As such, their fault removal process is bound to a limited number of test inputs to confirm that the fault was removed after the repair was inserted, which does not guarantee that the fault was really removed for all possible program inputs.

8 CONCLUSION

We presented INTREPAIR, a novel framework which provides comprehensive integer overflow detection, correct repair generation, and validation. To the best of our knowledge, we provide the first static symbolic execution-based technique that combines detection, generation and validation of source code repairs for C programs. We applied INTREPAIR to 2,052 C programs (approx. 1 million LOC) and to 50 programs contained in our seeded benchmark, which includes programs that have up to 20 KLOC. Our experimental results show that INTREPAIR was able to effectively detect integer overflows and successfully repair them with source code repairs. Lastly, our controlled experiment shows that INTREPAIR is over 10 times more time-effective and has a higher repair success rate than manual repairs.

ACKNOWLEDGMENTS

REFERENCES

[1] P. Muntean, S. Wuerl, J. Grossklags, and C. Eckert, "CastSan: Efficient detection of polymorphic C++ object type confusions with LLVM," in *Proc. Eur. Symp. Res. Comput. Security*, 2018.

[2] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert, "tauCFI: Type-assisted control flow integrity for x86–64 binaries," in *Proc. Symp. Res. Attacks, Intrusions, Defenses*, 2018.

[3] T. Wang, T. Wei, Z. Lin, and W. Zou, "IntScope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2009.

[4] C. F. Torres, J. Schuette, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proc. Annu. Comput. Security Appl. Conf.*, 2018, pp. 416–425.

[5] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, 2018, Art. no. 17.

[6] T. Wang, C. Song, and W. Lee, "Diagnosis and emergency patch generation for integer overflow exploits," in *Prc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2014, pp. 255–275.

[7] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2008, pp. 209–224.

[8] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[9] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 1083–1094.

[10] P. Marinescu and C. Cadar, "Make test-zesti: A symbolic execution solution for improving regression testing," in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 716–726.

[11] P. Muntean, V. Kommanapali, A. Ibing, and C. Eckert, "Automated generation of buffer overflow quick fixes using symbolic execution and SMT," in *Proc. Int. Conf. Comput. Safety, Rel., Security*, 2015, pp. 441–456.

[12] A. Ibing and A. Mai, "A fixed-point algorithm for automated static detection of infinite loops," in *Proc. Int. Symp. High Assurance Syst. Eng.*, 2015, pp. 44–51.

[13] A. Ibing, "Architecture description language based retargetable symbolic execution," in *Proc. Des., Automat. Test Eur. Conf. Exhibition*, 2015, pp. 241–246.

[14] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.

[15] U.S. National Institute of Standards and Technology (NIST), Juliet Test Suite v1.2 for C/C++. [Online]. Available: https://samate.nist.gov/SRD/testsuite.php

[16] S. Mechtaev, J. Yi, and A. Roychoudhury, "DirectFix: Looking for simple program repairs," in *Proc. 37th Int. Conf. Softw. Eng. - Vol. 1*, 2015, pp. 448–458.

[17] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proc. Int. Conf. Softw. Eng.*, 2016, pp. 691–701.

[18] H. Nguyen, A. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 772–781.

[19] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song, "RICH: Automatically protecting against integer-based vulnerabilities," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2007.

[20] H. Sun, X. Zhang, Y. Zheng, and Q. Zeng, "IntEQ: Recognizing benign integer overflows via equivalence checking across multiple precisions," in *Proc. Int. Conf. Softw. Eng.*, 2016, pp. 1051–1062.

[21] M. Corporation, "Integer overflow or wraparound." [Online]. Available: https://cwe.mitre.org/data/definitions/190.html

[22] Mitre, "Integer underflow (wrap or wraparound)." [Online]. Available: https://cwe.mitre.org/data/definitions/191.html

[23] Mitre, "Integer coercion error." [Online]. Available: https://cwe.mitre.org/data/definitions/192.html. Accessed: May 27, 2018.

[24] Mitre, "Off-by-one error." [Online]. Available: http://cwe.mitre.org/data/definitions/193.html. Accessed: May 27, 2018.

[25] Mitre, "Unexpected sign extension." [Online]. Available: https://cwe.mitre.org/data/definitions/194.html. Accessed: May 27, 2018.

[26] Mitre, "Signed to unsigned conversion error." [Online]. Available: https://cwe.mitre.org/data/definitions/195.html. Accessed: May 27, 2018.

[27] Mitre, "Unsigned to signed conversion error." [Online]. Available: https://cwe.mitre.org/data/definitions/196.html. Accessed: May 27, 2018.

[28] Mitre, "Numeric truncation error." [Online]. Available: https://cwe.mitre.org/data/definitions/197.html. Accessed: May 27, 2018.

[29] Mitre, "IO2BO: Integer overflow to buffer overflow." [Online]. Available: https://cwe.mitre.org/data/slices/680.html. Accessed: May 27, 2018.

[30] K. Yalin, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2015, pp. 295–306.

[31] P. Muntean, M. Rahman, A. Ibing, and C. Eckert, "SMT-constrained symbolic execution engine for integer overflow detection in C code," in *Proc. Inf. Security South Africa*, 2015, pp. 1–8.

[32] C. Barrett, A. Stump, and C. Tinelli, The SMT-LIB Standard Version 2.0, 2010. [Online]. Available: http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r10.12.21.pdf

[33] E. Gamma, J. Vlissides, R. Johnson, and R. Helm, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1994.

[34] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in C/C++," in *Proc. Int. Conf. Softw. Eng., IEEE/ACM*, 2012.

[35] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in C/C++," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 760–770.

[36] F. Pereira, R. Rodrigues, and V. Sperle Campos, "A fast and low-overhead technique to secure programs against integer overflows," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2013, pp. 1–11.

[37] Eclipse, "Eclipse CDT." [Online]. Available: https://eclipse.org/cdt/

[38] A. Laskavaia, "Codan-C/C++ static analysis framework for CDT," *EclipseCon*, 2011.

[39] Eclipse, "The language toolkit: An API for automated refactorings in eclipse-based IDEs. [Online]. Available: https://eclipse.org/articles/Article-LTK/ltk.html

[40] Eclipse, "Eclipse JFace." [Online]. Available: https://wiki.eclipse.org/JFace

[41] GCC, "The GNU compiler collection." [Online]. Available: https://gcc.gnu.org//

[42] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *Proc. IEEE Symp. Security Privacy*, 2019, pp. 1174–1189.

[43] R. Chinchani, A. Iyer, B. Jayaraman, and S. Upadhyaya, "ARCHERR: Runtime environment driven program safety," in *Proc. Eur. Symp. Res. Comput. Security*, 2004, pp. 385–406.

[44] R. Wojtczuk, "UQBTng: A tool capable of automatically finding integer overflows in Win32 binaries," *Chaos Commun. Congr. (22C3)*, 2005.

[45] Microsoft, "PREfast analysis tool." [Online]. Available: https://msdn.microsoft.com/en-us/library/ms933794.aspx

[46] P. Godefroid, M. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2008.

[47] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2004, pp. 168–176.

[48] P. Chen, Y. Wang, and Z. Xin, "Brick: A binary tool for run-time detecting and locating integer-based vulnerability," in *Proc. Int. Conf. Availability, Rel. Security*, 2009, pp. 208–215.

[49] P. Chen, H. Han, Y. Wang, X. Shen, X. Yin, B. Mao, and L. Xie, "IntFinder: Automatically detecting integer bugs in x86 binary program," in *Proc. Int. Conf. Inf. Commun. Security*, 2009, pp. 336–345.

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

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

[52] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, "IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time," in *Proc. Eur. Symp. Res. Comput. Security*, 2010, pp. 71–86.

[53] M. Pomonis, T. Petsios, K. Jee, M. Polychronakis, and A. D. Keromytis, "IntFlow: Improving the accuracy of arithmetic error detection using information flow tracking," in *Proc. Annu. Comput. Security Appl. Conf.*, 2014, pp. 416–425.

[54] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard, "Sound input filter generation for integer overflow errors," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 439–452, 2014.

[55] S. Sidiroglou-Douskos, E. Lahtinen, and M. Rinard, "Automatic discovery and patching of buffer and integer overflow errors," Massachusetts Inst. Technol., Cambridge, MA, USA, Tech. Rep. MIT-CSAIL-TR-2015–018, 2015.

[56] S. Sidiroglou-Douskos, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. Rinard, "Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 473–486, 2015.

[57] Y. Zhang, X. Sun, Y. Deng, L. Cheng, S. Zeng, Y. Fu, and D. Feng, "Improving accuracy of static integer overflow detection in binary," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*, 2015, pp. 247–269.

[58] B. Zhang, C. Feng, B. Wu, and C. Tang, "Detecting integer overflow in windows binary executables based on symbolic execution," in *Proc. 17th IEEE/ACIS Int. Conf. Softw. Eng., Artif. Intell., Netw. Parallel/Distrib. Comput.*, 2016, pp. 385–390.

[59] X. Cheng, M. Zhou, X. Song, M. Gu, and J. Sun, "Automatic fix for C integer errors by precision improvement," in *Proc. Comput. Softw. Appl. Conf.*, 2016, vol. 1, pp. 2–11.

[60] X. Cheng, M. Zhou, X. Song, M. Gu, and J. Sun, "IntPTI: Automatic integer error repair with proper-type inference," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 996–1001.

[61] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, P. Piselli, and M. Rinard, "Automatic error elimination by multi-application code transfer," MIT-CSAIL-TR-2014–024, Tech. Rep., 2014.

[62] Z. Coker and M. Hafiz, "Program transformations to fix C integers," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 792–801.

Paul Muntean is senior cyber security engineer with Novartis Institute for Biological Research (NIBR), Basel, Switzerland. In the past, he has been visiting researcher with EPFL, Lausanne, Switzerland, and The Ohio State University. He was a PhD student at the Technical University of Munich, Germany. His research focuses on the development of techniques and tools for program analysis in order to detect security vulnerabilities and generate program repairs automatically.

Martin Monperrus received the master's degree from the Compiègne University of Technology, France, and the PhD degree from the University of Rennes, France. He is a professor of software technology with KTH Royal Institute of Technology, Sweden. He was previously an associate professor with the University of Lille and an adjunct researcher with Inria, France. His research lies in the field of software engineering with a current focus on automatic program repair, program hardening and chaos engineering.
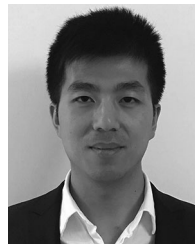
Hao Sun received the BSc and PhD degrees in computer science from Nanjing University, China. He was a visiting scholar with Purdue Uninversity, and worked as software engineer with Alibaba Group in Hangzhou, China. His main research interests include program analysis, software security and software testing. He has successfully published research results in venues such as ICSE, ASE, and AsiaCCS.

Jens Grossklags received the PhD degree from the School of Information, UC Berkeley. He is an associate professor with the Technical University of Munich. Previously, he served as a post-doctoral research associate with Princeton University, and as Haile Family early career professor with the Pennsylvania State University. In his interdisciplinary research agenda, he is investigating security and privacy challenges from the theoretical and practical perspectives.

Claudia Eckert is a full professor with the Technical University of Munich, Germany. She is the director of the Fraunhofer Institute for Applied and Integrated Security (AISEC). She advises government departments and the public sector at national and international levels in the development of research strategies and the implementation of security concepts. Her research and teaching activities are centered around the topics of information security, operating systems, middleware, and communication networks.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.