Program Transformations to Fix C Integers

Zack Coker and Munawar Hafiz
Auburn University
Email: {zfc0001@tigermail.auburn, munawar@auburn}.edu

***Abstract*—C makes it easy to misuse integer types; even mature programs harbor many badly-written integer code. Traditional approaches at best detect these problems; they cannot guide developers to write correct code. We describe three program transformations that fix integer problems—one explicitly introduces casts to disambiguate type mismatch, another adds runtime checks to arithmetic operations, and the third one changes the type of a wrongly-declared integer. Together, these transformations fixed all variants of integer problems featured in 7,147 programs of NIST's SAMATE reference dataset, making the changes automatically on over 15 million lines of code. We also applied the transformations automatically on 5 open source software. The transformations made hundreds of changes on over 700,000 lines of code, but did not break the programs. Being integrated with source code and development process, these program transformations can fix integer problems, along with developers' misconceptions about integer usage.**

Keywords: Program Transformation, Integer Problem.

I. Introduction

Ever since attackers shifted their attention from traditional buffer overflows to other types of attacks, integer vulnerabilities have been on the rise, even featuring as the second most common vulnerability behind buffer overflows [1]. The programs with integer vulnerabilities have not been written recently; they, as well as other mature programs, contain many integer problems, some of which result in vulnerabilities. The integer model of C is complex, unintuitive and partly undefined, making it easy to write code with integer problems.

Most of the recent works on integer problems focus on integer vulnerabilities, especially on how to detect them. Many sophisticated program analysis tools are available for detecting integer vulnerabilities in source code [2] [3] or binaries [4] [5] [6] [7] [8]. There are compiler based detection tools, such as GCC with `-ftrapv` option, which forces GCC compiler to insert additional calls (e.g., `_addvsi3`) before signed addition operations to catch overflow errors.

Existing approaches have three problems. First, the scopes of these tools are limited: They apply to some aspects of integer problems, most commonly overflows in integer arithmetic. But integer problems also involve signedness bugs and widthness bugs. A few tools target integer problems other than overflows, but they have problems with performance [8] and accuracy [7] [4]. Second, these tools, even the compiler extensions, are not used because runtime approaches have performance overhead (as high as 50X slowdown for BRICK [8]). However, the most important problem is that these approaches do not help developers produce better code. It is very easy to write C code with integer problems. Dietz and colleagues [9] reported that unintended integer overflows are very common in real programs. Existing approaches at best detect integer overflows, but they can not guide developers to understand and write code with safe integer operations.

We describe a program transformation-based approach that fixes all types of integer problems in C programs. We introduce three transformations—an ADD INTEGER CAST (AIC) transformation explicitly introduces casts to disambiguate integer usage and fix signedness and widthness problems, a REPLACE ARITHMETIC OPERATOR (RAO) transformation replaces arithmetic operations with safe functions to detect overflows and underflows at runtime, and a CHANGE INTEGER TYPE (CIT) transformation changes types to fix signedness and widthness problems. They are similar to refactorings [10], but they do not intend to preserve behavior. They are instead security-oriented program transformations [11] [12], that improve the security of systems by preserving expected behavior but removing integer problems. They transform the integer model of C programs towards a safe integer model, following CERT [13] and MISRA C [14] guidelines.

A transformation-based approach has several advantages.

- Program transformations help developers make small, frequent changes in source code to fix integer problems. People are bad at repetitive tasks—computers are better.
- A program transformation-based solution is more likely to be adopted, especially if the transformations make small changes [15]. Developers can use the transformations similar to refactorings during development, or use them to produce security patches during maintenance.
- Most importantly, the source-level program transformations would help people better understand and be aware of the subtleties of integer problems in C code.

Automatically transforming C programs to fix integer problems at source-level is challenging. C standard [16] has relaxed integer semantics, such as undefined behavior for signed integer overflows and underflows. There are many guidelines available, perhaps too many, e.g., CERT's [13] secure C coding guideline has over twenty rules and recommendations, MISRA C guideline includes more than ten rules just for arithmetic operations, etc. Another challenge is in the fact that integer overflows are subtle. For example, C allows truncating data, or wrapping around, or using signed values in unsigned context or vice versa—all handled implicitly. It is a challenge to make these implicit semantics visible to developers. Another engineering challenge is the fact that existing C IDEs, unlike Java IDEs, do not have a sophisticated infrastructure for developing program transformations.

We have implemented the program transformations as

Eclipse plugins; they target C programs but the idea can be used to build transformations for C++. The program transformations are developed using OpenRefactory/C [17], our infrastructure for developing program transformations for C. OpenRefactory/C features sophisticated analyses needed to support complex program transformations (Section III).

We validated the program transformations on benchmark programs and open source software with reported integer problems. We demonstrated that the three program transformations are sufficient to fix all possible C integer problems by automatically applying them to remove integer problems from all 7,147 benchmark programs of NIST's SAMATE reference dataset [18] (Section IV-A1). The program transformations preprocessed the programs before executing; they ran on more than 15 million lines of preprocessed programs. The SAMATE programs have a function showing normal behavior and another function showing problem behavior. In all cases, our program transformations preserved normal behavior, and modified behavior resulting from integer problems.

To demonstrate that our program transformations do not break existing code, we automatically applied the three transformations on all potential targets in 5 open source programs: libpng, Ziproxy, rdesktop, OpenSSL, and SWFTools (Section IV-B). AIC was applied on all local variables, parameter declarations, array access expressions, and structure element access expressions, RAO on all arithmetic expressions, and CIT on all local variables. In libpng, AIC was applied on 1857 target variables; it modified 751 expressions that contained unsafe variables. RAO modified 1,452 expressions, while CIT modified 152 declarations. Even after these large changes, the modified programs behaved similar to the original program and showed minimal performance overhead (Section IV-E).

While running the transformations automatically on test programs, we collected information about integer usage. Our empirical study reveals that on average 30% of integer variables are wrongly declared in the test programs. When declaring variables, signedness mismatch is more common that widthness mismatch. When using integers, misusing sign or rank are more common than misusing both sign and rank.

This paper makes the following contributions.

- It shows that integer problems in C programs can be prevented with a small number of source-to-source program transformations (Section II). These program transformations are similar to refactorings, but they modify the integer model of a C program towards a safe model. It also shows how the program transformations can explicitly reveal mistakes in integer operations (Section IV-D).
- It demonstrates that complex yet accurate source-level C program transformations can be implemented as part of the refactoring catalog of popular IDEs (Section III).
- It shows that the three program transformations can prevent all types of integer problems (Section IV-A). It also evaluates the accuracy of the program transformations by applying them automatically to make many small modifications to open source programs, yet producing programs that maintain functionality (Section IV-B) and have minimal overhead (Section IV-E).
- It presents an empirical study—based on well-known open source programs—about the types and patterns of integer problems in source code (Section IV-C).

II. Fixing Integer Problems with Program Transformations

When integer operations in C mix the type of operands, the result may be unexpected. There are four possible problems [19] [7]: 1) a *signedness bug* occurs when an unsigned type is interpreted as signed, or vice versa; 2) an *arithmetic overflow* occurs when integer operations such as addition or multiplication produce a result that overflows the allocated storage; 3) an *arithmetic underflow* occurs when integer operations such as subtraction and multiplication produce a result that is smaller than what can be stored; 4) a *widthness bug* or a *truncation bug* is the loss of information when a larger integer type is assigned to a smaller type, e.g., int to short.

This paper describes three program transformations that prevent the four types of integer problems. The root cause for signedness problem is that C allows signed variables to be used in unsigned contexts and vice versa; similar misuse occurs in case of truncation bugs. We present the Add Integer Cast (referred to as AIC in the paper) transformation that explicitly introduces casts to disambiguate the context of integer variables. Arithmetic overflow (and underflow) problems originate from the fact that such cases are handled silently. We present the Replace Arithmetic Operator (RAO) transformation that replaces arithmetic operations such as addition and multiplication with safe library functions. Finally, the Change Integer Type (CIT) transformation checks whether the way an integer variable is used differs from its declaration and modifies the type based on the analysis.

Consider this recent widthness/overflow vulnerability [20] in Ziproxy v.3.0.0 in lines 979-981 of image.c file.

```
979 raw_size = bmp->width * bmp->height * bmp->bpp;
```

In line 979, the result of multiplying three signed integers is stored in `raw_size`, a long long integer variable. The lvalue's type is supposed to ensure that it can store a large value resulting from multiplying large integers. However, C first performs the integer multiplication in an integer context, meaning that multiplying any values that produce results outside of the signed integer range would first wrap around; then the wrapped around value would be cast to long long int.

Traditional approaches to detect integer overflows would add runtime checks on the multiplication; our RAO transformation would similarly replace the multiplication operations with safe functions that detect overflow. However, this solution is not perfect, since the developer obviously wanted to use the full range of the long long integer variable to store the solution. So the context of the multiplication should be lifted so that a multiplication with long long integer values can be allowed. A developer can first detect the problem with RAO, but then apply AIC on `raw_size` to fix the context.

```
979 raw_size = (long long int) bmp->width *   \
980               (long long int) bmp->height *   \
981                 (long long int) bmp->bpp;
```

Additionally, a developer can apply RAO on the result, to remove possibilities of overflow. This adds defense in depth.

```
979 raw_size = mulsll((long long int)bmp->width, \
980              mulsll((long long int)bmp->height, \
981                (long long int) bmp->bpp));
```

Our program transformations make structural changes to source code using program analysis. They modify source code so that it conforms to a safe C integer model [13] [14]. However, programmers may intentionally use unsafe integer behavior [9]. It is impossible for an automated tool to unambiguously understand the intent of a programmer. This is where a program transformation-based solution has its merits: Developers decide whether a transformation is allowed or not. If a program intentionally uses unsafe integer behavior, then a programmer can disallow the modifications.

The root cause of all integer problems is that the mistakes in using integers are handled silently. Our program transformations make the mistakes explicit, either by introducing a cast (AIC), by suggesting a type change (CIT), or by raising a runtime exception (RAO). This not only fixes the problems, but also can make developers aware of them.

We describe the program transformations next.

A. Add Integer Cast (AIC)

You have a program in which integer operations have operands with different types; the end result may contain an unexpected value.

Add explicit casts for all type mismatches so that they are visible and are properly handled.

1) Motivation: The weak typing system of C may produce unexpected results, e.g., when an unsigned value is compared to a signed value in a boolean expression, the context of the comparison may vary and produce unexpected results. Also, because of arithmetic conversion and integer promotion rules [16], integer types are modified implicitly in a program.

Signedness and widthness mismatches are very common (Section IV-C). If the types were explicitly mentioned in a program (e.g., MISRA rules 10.1-10.6 [14]), a compiler can unambiguously enforce the types.

2) Precondition: A developer selects an integer variable and invokes the AIC transformation. The following preconditions are checked:

- The variable is of integer type including pointers to integer types and typedefs of integers.
- A reference of the variable is used in an unsafe operation, e.g., in an unsafe function such as `memcpy`, in a control statement, in an array access expression, or in a return statement.

The preconditions of AIC do not guarantee that the transformation will produce a change. It only means that the situation needs more in depth investigation.

3) Mechanism: AIC determines the declared type of a variable, checks all the references of the variable, determines if the variable is used in an unsafe context, and explicitly adds casts. The casts are added to make the types explicit; some of these casts remove a problem, others illustrate the way types are (mis)used. For example, a cast to a function parameter does not fix any problem; it notifies the programmer about a possible problem in using the parameter. On the other hand, a cast in a comparison expression disambiguates the way the comparison is done, and may prevent a signedness problem.

4) Example: Consider this program from NIST's SAMATE reference dataset [18]. It is an example of an unexpected sign extension problem (CWE 194), in particular showing a negative value passed to a `memcpy` function.

```
short data;
...
data = -1;
char dest[100] = "";
if (data < 100) {
   ...
   memcpy(dest, src, data);
```

A negative number for the variable `data` wrongly bypasses the conditional test. `memcpy` expects an unsigned integer as its third parameter, so the value of `data` will be converted to a very large unsigned integer value. This will overflow the buffer `dest`. AIC fixes this with explicit casts.

```
short data;
...
data = -1;
char dest[100] = "";
if ((unsigned int) data < 100) {
   ...
   memcpy(dest, src, (unsigned int)data);
```

Among the two casts introduced, the one in `memcpy` function is to make developers aware of the casts performed automatically by the weak-typed C system.

B. Replace Arithmetic Operator (RAO)

You have a C program that has a potential integer overflow (or underflow) problem originating from an arithmetic operation.

Replace arithmetic operations with a safe function call that detects an overflow (or underflow) and explicitly handles them.

1) Motivation: Integer overflows (and underflows) in C are silently handled based on defined and undefined semantics for integer operations. Intentional use of these behaviors are allowed, but the code may not be portable or may break in future compilers [9]. Unintentional uses of integer overflow (and underflow) are bugs.

Most approaches for detecting integer overflows analyze programs and add runtime checks. Similarly, t would be advantageous if arithmetic operations in C can be replaced with safe library functions, that explicitly notify an integer overflow and allow developers to write handlers for dealing with the situation.

2) Precondition: A developer selects an expression containing an arithmetic operation and invokes the transformation. The following preconditions are checked:

- The arithmetic expression contains integers or typedefs of integers. Pointer arithmetic is excluded.
- The selected expression or the lvalue assigned as the result of the expression is in an unsafe context, e.g., in an unsafe function such as `memcpy`, in a control statement, in an array access expression, or in a return statement.

3) Mechanism: The transformation determines if the integer operation is unsafe and replaces it with functions from a safe library. Our implementation uses CERT's IntegerLib library [13]. The safe functions in the library introduce callbacks that are explicitly invoked when an overflow occurs. The correct function is determined using operand types, and integer promotion and arithmetic conversion rules (e.g., CERT's rule INT02-C and INT-32C [13]).

Binary expressions ($+,-,*,/$), prefix and postfix expressions ($++,--$), and assignment expressions ($+=,-=,*=,/=$) are all handled differently. Binary expressions and stand-alone prefix and postfix expressions are straightforward to replace. When prefix, postfix, and arithmetic assignment expressions are part of other expressions, they are replaced following patterns that preserve the semantics of the original expressions. For example, assignment expressions in C have semantics that the lvalue is evaluated only once; the modified expression preserve the semantics.

4) Example: rdesktop is a popular remote desktop program that allows Linux/Unix systems to remotely control a Windows computer. An integer overflow vulnerability [21] was reported in rdesktop version 1.5.0. The problem is integer overflow in a call to `tcp_recv` function in line 101 inside `iso_recv_msg` function in iso.c file.

```
101 s = tcp_recv(s, length - 4);
```

If the length of the message is less than 4, the result of subtraction will be negative. Since `tcp_recv` expects an unsigned integer, the negative value will be converted to a large positive integer. This overflows buffer `s`. RAO transforms the program using safe functions from IntegerLib library [22]:

```
101 s = tcp_recv(s, ui2us(subui(length , 4)));
```

The variable `length` is unsigned short. Following arithmetic conversion rules, the subtraction function subtracts between two unsigned integers, and then a function downcasts to unsigned short. RAO also adds a header file (IntegerLib.h) to the transformed program. With these safe functions, if `length` is less than 4, the function will report an error and move to a customized handler.

C. Change Integer Type (CIT)

You have a program that has signedness and widthness problems from using variable types in incorrect contexts. These errors derive from incorrectly declared variables.

Change the declared type of variables so that the uses of the variable are not conflicting with the declaration.

1) Motivation: We found that it is very common for a C integer variable to have a declared type that differs from the underlying types (empirical study in Section IV-C). This can lead to both signedness and widthness problems.

AIC transformation fixes this problem by explicitly introducing casting. If a variable is used with a wrong type many times, AIC will introduce many cast expressions. A better solution is to change the type of the variable to match the underlying type.

2) Precondition: A developer selects an integer variable and invokes the CIT transformation. The following preconditions are checked:

- The variable is of integer type including pointers to integer types and typedefs of integers.
- The variable is not a global variable or declared as a function parameter.
- A reference of the variable is used in an unsafe operation, e.g., in an unsafe function such as `memcpy`, in a control statement, in an array access expression, or a return statement.

Similar to AIC, the preconditions of CIT does not guarantee that the transformation will produce a change. It only means that the situation needs more in depth investigation.

3) Mechanism: The program transformation determines the type of the variable, checks if it is used as another type in important contexts (e.g., when a variable is an lvalue or when a variable is used in a function call, etc.), and matches the declared type with the underlying type.

Justifying that there is a need for change and determining the correct type is the most critical step in CIT. The transformation checks for all contexts that a variable is used and filters out less important contexts. When a variable is used in an assignment expression, the lvalue is considered more important than an rvalue. So if a variables' underlying type is different in an lvalue position, it is considered an important context that justifies change. When a variable is used as an actual parameter in a function call, a change is only considered if the function is a sensitive one. A sensitive function is determined by analyzing the static call graph (details in Section III).

For each variable, the important contexts are collected along with the declared types and underlying types in each of them. A conservative CIT may only allow a variable to change if all the important contexts have a different underlying type. From empirical evidence, we found that a 75% mismatch is a valid reason for change.

After the change made by CIT, some of the casts already in the program will be unnecessary, while some casts may need to be introduced based on the newly-declared type. AIC should follow CIT to clean up the code.

4) Example: Consider this recent vulnerability in libpng v1.4.9 [23], in lines 267-290 in pngrutil.c file. The variable `copy` is declared as a signed integer and then used in `png_memcpy` as an unsigned integer argument.

```
267 int ret, avail;
268 ...
276 avail = png_ptr->zbuf_size-png_ptr->zstream.avail_out;
277 ...
283   if (output != 0 && output_size > count)
284     {
285     int copy = output_size - count;
286     if (avail < copy) copy = avail;
287     png_memcpy(output + count, png_ptr->zbuf, copy);
288     }
289
290     count += avail;
291 ...
```

CIT is first applied to `avail`, which is used as an unsigned integer in all important contexts (e.g., lvalue in line 276). Its type is modified to an unsigned int (line 267-268).

```
267  int ret;
268  unsigned int avail;
269  ...
```

Then, CIT is applied to `copy`. It is declared an integer (line 285), but used as an unsigned int in all four references of the variable—1) in line 285, it is an lvalue in an assignment expression, 2) in line 286, it is used in a comparison with an unsigned type, 3) in line 286, it is used in another assignment statement as an lvalue, and 4) in line 287, it is used as an actual parameter in a sensitive function, that expects an unsigned integer. An AIC transformation may follow CIT to remove casts, or introduce new casts (nothing happens in this case). Incidentally, an RAO transformation may also be applied to the subtraction expression in line 285 to prevent wraparound.

```
283 if (output != 0 && output_size > count)
284   {
285   unsigned int copy = subui (output_size, count);
286   if (avail < copy) copy = avail;
287   png_memcpy(output + count, png_ptr->zbuf, copy);
288   }
289
290   count += avail;
291       ...
```

III. Implementing the Transformations

We implemented the transformations as Eclipse plugins. We used OpenRefactory/C [17], our framework for building transformations for C programs. OpenRefactory/C's refactoring infrastructure is based on the design of Photran [24], a refactoring tool for Fortran which has been refined over several years and currently contains 39 refactorings. Like Photran, OpenRefactory/C's internal program representation is a rewritable abstract syntax tree generated by Ludwig [25] and augmented with preprocessor information. Semantic checks are based on a differential precondition checking infrastructure [26]. OpenRefactory/C supports name binding analysis, type analysis, control flow analysis, and static call graph analysis. Most of the underlying infrastructure of OpenRefactory/C is platform-independent. Thus our transformations can be ported to other IDEs such as Visual Studio, or Vim.

When implementing the transformations, we had to make a few design decisions. For example, RAO applies to arithmetic expressions, but nesting of expressions might create special situations. Consider, `A[++p]+= b`. The ++ operator in the lvalue has to be evaluated only once. RAO first transforms the expression into an unnamed block that preserves the semantics.

```
{ int *A_ptr = &A[++p];
  *A_ptr = *A_ptr + b;}
```

RAO names the newly-introduced variable using an approach similar to Lisp's `gensym`. It checks the symbol table of the current context to ensure that a name clash does not occur.

Another important design decision was to determine when to change a variable's type in CIT. The underlying type of a variable may be different from its declared type in many contexts, but not all contexts are of equal importance. From empirical evidence based on benchmark programs and real programs, we identified that a misuse of type in an lvalue position of an assignment expression, a parameter to a function call, or a comparison expression are more important contexts to justify a change in type.

Misuse of a type in a parameter to a function call requires more analysis. We identified that not all function calls are equal, and implemented an analysis to determine sensitive functions. The analysis is based on static call graph analysis and control flow analysis of OpenRefactory/C. We started with a seed of sensitive functions that had an unsafe parameter and identified all functions that reach those sensitive functions through the unsafe parameter. For example, `memcpy` is a sensitive function and its third parameter specifying length (suppose `len`) is unsafe. All functions that contain `memcpy` are considered sensitive; also sensitive are functions that are predecessors in the call graph and contain the parameter `len` as a formal parameter. If a type misuse happens in a sensitive function, then it is a better justification to change the type.

IV. Evaluation

To evaluate the usefulness of our program transformations and understand integer problems in general, we answer the following research questions:

A. Are the program transformations effective in securing systems? More precisely, do they fix different types of integer vulnerabilities?
B. Does a program transformation-based technique work? Do the program transformations scale to large programs? Do they break the original program?
C. How can we understand the integer problems better? What types of integer problems are more common?
D. Are the program transformations useful for developers? Can the suggested modifications from the transformations guide developers in writing better code?
E. How is performance affected by the transformations?

Although the three transformations are supposed to be used similar to refactorings, we automated the process of applying the transformations in order to answer the five questions better. On the test programs, the targets of each program transformation was different. AIC was applied on all local variables, all formal parameters, all array access expressions, and all structure element access expressions; it introduced casts. RAO was applied on all binary expressions, prefix

expressions, postfix expressions, and arithmetic assignment expressions (e.g., +=); it replaced some arithmetic operations with safe function calls. CIT was applied on all local variables; it changed the type if needed.

Our test corpus included benchmark programs and real software. To answer the first question, we automatically applied the program transformations on benchmark programs of NIST's SAMATE reference dataset [18]. These programs represent all variants of integer vulnerabilities. We also applied the program transformations on 5 open source software with reported integer vulnerabilities. We applied them to demonstrate that the suggested changes would remove the vulnerability. We then applied the transformations automatically on all potential targets in these 5 programs. This answers the second and third questions. We studied the actual changes made in the programs to understand the trends in integer problems (question 4). In order to answer the fifth question, we ran the original program and the 3 modified programs, each resulting from applying AIC, ITC and RAO transformations on all targets. This was done for 3 out of 5 open source programs.

A. Are the transformations effective in securing systems?

1) Securing Benchmark Programs: SAMATE [18] is a joint project of the National Institute of Standards and Technology (NIST) and the Department of Homeland Security (DHS). It has a suite of test bench programs in C, C++, and Java to demonstrate common security problems.

TABLE I
CWEs THAT DESCRIBE INTEGER VULNERABILITIES

Program Transformation	CWE	Total Programs	KLOC	PP KLOC
Add Integer Cast (AIC)	CWE 194: Unexpected Sign Extension	1296	160.8	2371.1
	CWE 195: Signed to Unsinged Coversion Error	1296	158.8	2371.1
Remove Arithmetic Operator (RAO)	CWE 190: Integer Overflow or Wraparound	2430	375.4	8528.6
	CWE 191: Integer Underflow	810	124.6	966.9
	CWE 680: Integer Overflow to Buffer Overflow	324	39.3	160.9
Change Integer Type (CIT)	CWE 196: Unsigned to Signed Coversion Error	19	2.9	2.4
	CWE 197: Numeric Truncation Error	972	124.6	1334.7
		7147	967.0	15735.7

The SAMATE benchmark presents security errors in design, source code, binaries, etc. For C/C++ code, SAMATE's Juliet test suite version 1.0 provides 45,324 test programs for 116 different Common Weakness Enumerations (CWE). A new version 1.1 has been released on July 2012. The tests described in this paper were done on Juliet test suite version 1.0.

SAMATE is the most comprehensive benchmark available for integer vulnerabilities in C and C++. Table I lists 7 CWEs that describe integer vulnerabilities the benchmark programs. In total, there were 7,147 C programs with 967 KLOC. We applied AIC, CIT and RAO transformations to verify that they prevent different types of integer vulnerabilities. The program transformations preprocessed test programs and ran automatically on the preprocessed versions in order to collect all definitions. The last column in Table I shows that the transformations ran on more than 15 million lines of C code.

CWE 194 and 195 are represented by 2,592 programs with signedness problems (an example program is in the description of AIC in Section II-A). The signedness problem allows an attacker to bypass a check. AIC explicitly introduced casts in control statements so that the checks cannot be bypassed.

CWE 190 (and 191) are integer overflow (and underflow) vulnerabilities presented in 2,430 (810) programs. These programs contain integer arithmetic operations that are replaced with safe function calls by RAO. The functions introduce runtime checks. The modified programs still had the overflows (underflows), but they were caught by the safe functions and were reported in error messages by the handler functions. The handler functions can be customized; a custom handler can even terminate execution when an overflow occurs.

CWE 680 is represented by 324 programs that demonstrated integer overflow to buffer overflow (IO2BO) problems. For example, a vulnerable program allocated memory using `malloc` function: `malloc(data * sizeof(int))`. The multiplication may wrap around and allocate a small buffer. A latter part of the program overflows the small buffer. RAO replaced the multiplication operation in `malloc` with a safe function that reports error messages when an overflow occurs.

CWE 196 is represented by 19 programs with signedness bugs, where an unsigned value is assigned to a signed value without bounds checking. AIC does not fix this: A cast only shows that a type mismatch is happening in an assignment expression, but it does not fix the mismatch. CIT is applied to modify the type of the signed variable to an unsigned variable.

CWE 197 represents widthness bugs (972 programs) for which CIT lifts the type from a lower rank to a higher rank.

Programs in SAMATE have a good function and a bad function. The good function demonstrates normal behavior, and the bad function demonstrates a vulnerability. AIC and CIT fixed the problems in bad functions; RAO reported them during runtime. All transformations modified some parts of good functions, but the changes had no effect on the behavior.

2) Securing Real Programs: We tested the transformations on 5 open source programs with recently reported integer vulnerabilities: libpng 1.4.9, Ziproxy 3.0.0, rdesktop 1.5.0, OpenSSL 0.9.8, and SWFTools 0.9.1. The programs are all widely used C programs. We applied the appropriate program transformations to remove the root cause of the reported vulnerability. In four cases, the reports contained descriptions about the root cause. SWFTools only described the program that had the vulnerability; we manually identified the root cause. We matched our transformed programs with the suggested patches to test that we removed the vulnerabilities.

SWFTools 0.9.1 had multiple reported vulnerabilities [27] in lib/png.c and lib/jpeg.c files. In line 464 of lib/png.c file, the variable `len` was declared an integer and was passed to malloc. CIT changed the declared type to unsigned integer. A similar signedness problem was in lib/jpeg.c file, line 314: `int width = *_width = cinfo.output_width;`.

TABLE II
RUNNING AIC ON TEST PROGRAMS

Software	Tokens [C1]	Unsafe Tokens					Instances Checked [C7]	Tokens w/ Changes [C8]	Instances Changed [C9]	% of Unsafe Tokens [C6/C1]	% of Instances Changed [C9/C1]
		Local Variables [C2]	Parameters [C3]	Array [C4]	Structure [C5]	Total [C6]					
libpng 1.4.9	1847	490	162	79	531	1262	6978	358	751	68.33	10.76
Ziproxy 3.0.0	581	299	105	32	45	481	2321	141	275	82.79	11.85
rdesktop 1.5.0	1811	605	341	79	301	1326	11284	412	1154	73.22	10.23
OpenSSL 0.9.8	3899	771	892	75	1047	2785	11300	690	1278	71.43	11.31
SWFTools 0.9.1	7608	2721	640	444	1095	4900	17408	1310	3785	64.41	10.96
	15746	4886	2140	709	3019	10754	49291	2911	7243	(avg) 72.04	(avg) 11.02

CIT modified the type to unsigned integer.

OpenSSL 0.9.8 had a widthness vulnerability [28] in crypto/asn1/a_d2i_fp.c file. The variable `want` in asn1_d2i_read_bio function is declared as int, but it is used as unsigned long int. CIT performs the transformations.

The signedness problem [23] in libpng 1.4.9 is fixed with a combination of CIT and RAO transformations (Section II-C). The overflow vulnerability in rdesktop 1.5.0 [21] is fixed with RAO transformation (Section II-B). A vulnerability [20] in Ziproxy is fixed by a combination of AIC and RAO transformations (Section II).

B. Does a transformation-based technique work?

Our program transformations modify program behavior to fix a problem, but should not break normal behavior. We automatically applied the transformation on all appropriate targets of 5 open source programs. The transformations were each applied to more than 700,000 lines of preprocessed code containing 4,493 functions in 222 files (Table III).

TABLE III
TEST PROGRAMS

Programs	# of C Files	# of Functions	KLOC	PP KLOC
libpng 1.4.9	16	337	23.2	49.4
Ziproxy 3.0.0	20	196	11.6	23.1
rdesktop 1.5.0	28	450	21.8	80.1
OpenSSL 0.9.8	64	1410	51.3	220.6
SWFTools 0.9.1	94	2100	161.8	336.3
	222	4493	269.7	709.5

AIC was applied on all local variables, parameters, array access expressions, and structure element access expressions—1,847 total in libpng (Table II). Of which, 1,262 were considered unsafe—490 local variables, 162 parameters, 79 array access expressions, and 531 structure element access expressions. These passed the preconditions of AIC (1262/1847, i.e., 68.33%). When the transformation was applied, 6,978 references of the 1,262 variables were analyzed. A total of 358 tokens had changes—751 references were modified by adding or removing or updating a cast. On average, each unsafe token was changed about twice (751/358=2.1). The final column in Table II shows the percent of instances changed out of the unsafe instances that were examined. Only about 10.76% (751/6978) of the references of unsafe tokens were changed.

AIC was ran on 11,879 total variables. On average, 72% of the variables passed the preconditions. 66,404 instances were checked; of which, 7,243 instances had a cast modified (on average 10.9%). Most of these casts would have been introduced by the compiler. The results suggest that, on average, a C compiler adds a cast to about 1 out of every 9 variable references.

TABLE IV
RUNNING RAO ON TEST PROGRAMS

Software	Expressions [C1]	Unsafe Expressions					% Changed [C6/C1]
		Binary [C2]	Prefix [C3]	Postfix [C4]	Assignment [C5]	Total [C6]	
libpng 1.4.9	11849	1310	2	62	78	1452	12.25
Ziproxy 3.0.0	4764	194	1	185	53	433	9.09
rdesktop 1.5.0	15414	1520	6	698	87	2311	14.99
OpenSSL 0.9.8	23260	723	3	115	149	990	4.26
SWFTools 0.9.1	47387	8685	34	674	442	9835	20.75
	102674	11433	46	1734	809	15021	(avg) 12.27

RAO was applied on all arithmetic expressions (Table IV), 11,849 in total in libpng. This contained binary expressions, prefix expressions, postfix expressions, and arithmetic assignment expressions. 12.25% of all arithmetic expressions were considered unsafe in libpng—1,310 binary expressions, 2 prefix expressions, 62 postfix expressions, and 78 arithmetic assignment expressions.

TABLE V
RUNNING CIT ON TEST PROGRAMS

Programs	Tokens Checked [C1]	Unsafe Tokens [C2]	Instances Checked [C3]	Declarations Changed [C4]	% Changed [C4/C1]
libpng 1.4.9	531	453	2404	152	28.63
Ziproxy 3.0.0	330	321	1697	132	40.00
rdesktop 1.5.0	717	605	5560	152	21.20
OpenSSL 0.9.8	821	782	4416	224	27.28
SWFTools 0.9.1	3233	2940	34382	992	30.68
	5632	5101	48459	1652	(avg) 29.56

CIT was applied on all local variables (Table V). For libpng, CIT was applied on 531 local integer variables; 453 passed the preconditions. There were 2,404 references to these unsafe tokens that were checked. In the end, CIT modified the declaration of 152 local variables in libpng, i.e., 28.63% (152/531) of the variables that were checked.

We ran the original programs and the modified programs and found that they had the same observable behaviors. For libpng and OpenSSL, we ran the test suite that came with the code (`make test`). We ran SWFTools to convert files to SWF format. rdesktop was used to connect to a remote Windows machine. Ziproxy was used as a proxy for our web browser. In all cases, the modified programs from applying AIC and CIT had the same visible behavior as the original program. The programs produced from applying RAO showed the same behavior, except in some cases, there were overflows and underflows at runtime, which the default handler caught and printed error messages. OpenSSL had many of these error messages. This is expected, because the cryptographic libraries of OpenSSL uses overflow and underflow behavior as a feature. The test cases for OpenSSL however passed for both the original and the modified program.

While these transformations are designed to be applied selectively by the developer, and not automatically to all possible cases as done here, the large number of changes show that these transformations can be applied throughout a program without breaking it. It also shows that the preconditions limit the places where the transformations can be applied to a measurable degree. Nevertheless, programs contain many places that could benefit from the transformations.

C. How can we understand the integer problems better?

We collected additional information about integer usage in the 5 test programs while we were automatically running AIC, CIT, and RAO transformations on them. For space issues, we focus on a few key results from the empirical study.

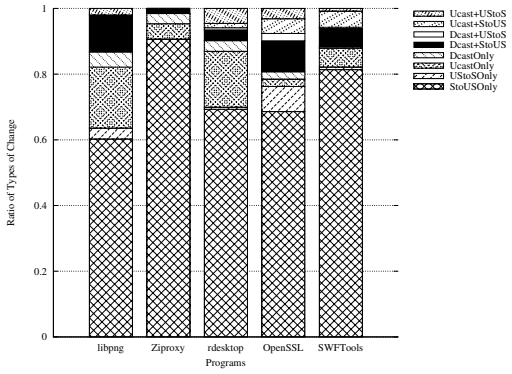

Fig. 1. Ratio of Types of Changes Made by CIT

The most surprising aspect is the number of integer problems in matured software. For example, Table V shows that on average about 30% of the variables in our test programs are declared incorrectly. This large number implies that, in many cases, developers do not understand the contexts where the variables will be used when declaring the variable or that they do not update the declared types.

CIT removes these problems. We investigated all the changes made by CIT to understand what type of developer mistakes are more common. Figure 1 shows the distributions. It shows that signedness problems of the same rank are the most common mistakes made by developers, more common than widthness problems. The most common problem is declaring a variable as signed but using it as unsigned. This is perhaps because developers do not use additional type specifiers for signedness when they declare a variable (`int` is used in place of `unsigned int`).

Similarly, several casts are introduced by AIC (Table II). Figure 2 shows the distributions of these casts. Unlike Figure 1, there is no clear pattern here. This is because AIC introduces casts for misuses in all contexts, but CIT determines a type change based on some contexts that are more important than the others. When all contexts are considered, type mismatches vary more.

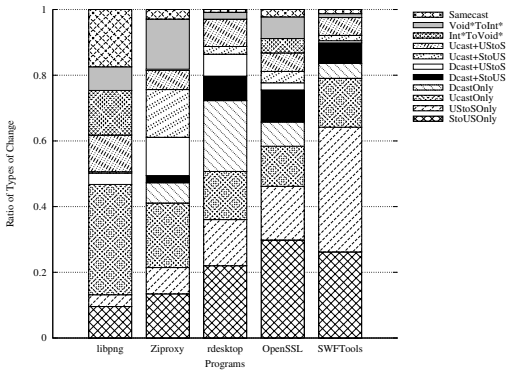

Fig. 2. Ratio of Types of Changes Made by AIC

Single misuse in rank or signedness is more common than misuses in both rank and sign. Signed to unsigned cast is more common following the observation about type declaration (Figure 1). However, SWFTools has many unsigned to signed casts: This is because there was one program (lib/gocr/ocr0n.c) that had several function calls that were performing unsigned integer operations in the parameter, but the parameter was expecting a signed integer.

AIC removes unnecessary casts introduced by programmers. Interestingly, libpng had many of these. This shows that, in some cases, the programmers are too cautious about explicit casting, and may introduce casts even if they are unnecessary.

D. Are the transformations useful for developers?

A specified goal of the transformations is to make implicit integer problems visible, so that developers can be aware of the problems and fix them. Each program transformation modifies code in many places to notify developers about integer problems; not all of them fix an integer vulnerability.

For example, AIC modified 751 instances in libpng (Table II), either by adding casts or removing unnecessary casts. Out of these instances, 139 (18.5%) changes were in an assignment expression, 212 (28.2%) were in the parameter type or the return type of a function call expression, 269 (35.8%) were in a binary expression, and 131 (17.1%) were affecting another cast expression. Changes in assignment expressions and changes in function call expressions are to make developers aware of integer problems. On the other hand, changes in binary

expressions affect binary comparison expressions only; thus they guide developers, and also fix potential signedness and widthness vulnerabilities. Changes to cast expressions remove or replace an existing cast. It is not very common, but libpng had many of these (see Section IV-C). Other test programs show similar trends.

Another way the transformations are useful is if they do not make many unnecessary changes. In fact, AIC only made changes to about 10% of all instances (Table II), and RAO made changes to about 12% of all expressions (Table IV). CIT had the highest change rate (Table V). While this is surprising, the changes were all valid since they did not break program behavior (Section IV-B).

The test results were collected by brute-forcing the changes. In practice, the transformations are supposed to be applied similar to refactorings. This allows developers to selectively apply the transformations only in places that they deem unsafe, unlike other mechanisms that add checks in all places. At the same time, they could use the transformations to explore potential integer problems in source code.

E. How is performance affected by the transformations?

Our program transformations fix integer problems, but they may have performance overhead. We measured overhead using the programs in our corpus: We ran programs modified by AIC, CIT, and RAO transformations against the original one. The programs were tested on a MacBook Pro machine with 2.3 GHz Intel Core i7 processor and 8GB 1333 MHz DDR3 memory running Mac OS X Lion 10.7.4. We used llvm-gcc 4.2.1 to compile the programs; each program was compiled with no optimization, and then enabling optimization flags (`-O1`, `-O2` and `-O3`). We tested 3 out of 5 programs that had test codes. Ziproxy and rdesktop did not have test codes.

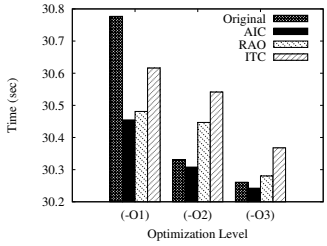

Fig. 3. OpenSSL Performance

For space issues, we only show the performance numbers of OpenSSL (Figure 3). Others follow similar trends. In all cases, AIC results were faster than the original, RAO performed slightly worse, and CIT the worst. This is because adding casts explicitly allows a compiler to optimize because the modified program does not need integer promotions, etc. Surprisingly, RAO sometimes performed better than the original, perhaps because of function inlining. CIT performed the worst. A type change was justified by considering some contexts that are more important, but perhaps the changed type warrants more casting in the contexts that were ignored. We mentioned that an CIT transformation is typically followed by an AIC transformation. That may reduce the overhead of CIT.

Figure 3 does not show results from no optimization, which follows the same trend. It was not shown because it skews the graph, which makes the subtleties of `-O1`, `-O2` and `-O3` performance harder to follow.

V. Related Works

Research on fixing integer problems primarily focuses on integer vulnerabilities, either statically or dynamically. The static approaches create an analysis framework and add runtime checks to program points; the analysis can be done on both source code and binary code. IntScope [6] modifies binaries to an intermediate representation, and then checks for integer overflow combining symbolic execution and taint analysis. UQBTng [5] also decompiles and then applies model checking with CBMC [29]. On the other hand, some approaches examine source code, e.g., Microsoft's PREfast [30], ARCHERR [31], etc. Ashcraft and colleagues' [3] approach uses bounds checking and taint analysis to see if an untrusted value is used in trusted sinks; bounds checking is also used by Sarkar and colleagues [32]. However, both approaches are not applicable to detect all types of integer problems. Ceesay and colleagues [2] added type qualifiers to detect integer overflow problems. Their approach requires user annotation and only detects integer overflow.

Among the dynamic approaches, several tools can detect all types of integer problems, e.g., RICH [7], BRICK [8], and SmartFuzz [4]. RICH [7] instruments programs to detect safe and unsafe operations based on well-known subtyping theory. BRICK [8] detects integer overflows in compiled executables using a modified Valgrind [33]. Its accuracy and efficiency depend on the test input used to exercise the instrumentation. It is either slow (50X slowdown) or has many false positives. SmartFuzz [4] is also based on Valgrind, but it uses dynamic test generation techniques to generate inputs, leading to good test coverage. SAGE [34] also uses dynamic test generation, but it targets fewer integer problems. IOC [9] is an integer overflow detection tool integrated with the Clang compiler. It only targets integer overflow and underflow problems.

AIC, CIT and RAO are types of security-oriented program transformations [11]. They are similar to refactorings [10], but they do not preserve the original behavior. Our program transformations modify towards a safe integer model [13] [14]; they apply the safe integer model on the applied type of integer variables and suggest changes. The work has similarities to subtyping rules used in RICH [7] and the type analysis approach used in IntPatch [35]. Other approaches of modeling safe integer operations create a constraint graph that describes ordering relationship between integer expressions [32], or create a decision tree using a finite state grammar describing arithmetic instructions [36]. These approaches are not applicable to all types of integer problems.

Our RAO implementation uses IntegerLib library [22], because the library can be applied to C programs. For C++ programs, SafeInt [37] is a well-known option, as well as

arbitrary-precision arithmetic packages such as CLN [38] and GMP [39]. Another approach is to define an integer model that has well-defined semantics for most of C/C++ integer-related undefined behaviors, as done in the As-if Infinitely Ranged integer model [40]. It produces well-defined results for integer operations or else traps.

Only two prior works explore facts about integer problems. Brumley and colleagues [7] surveyed CVE database [41] and identified four types of integer problems—overflow, underflow, signedness, and truncation errors. Dietz and colleagues [9] distinguished between intentional and unintentional uses of defined and undefined C semantics to understand integer overflows. They found that the intentional use of undefined behavior is common. The focus of our empirical study is broader: It looks into all types of integer problems and studies the actual misuses that are in source code.

VI. Future Works and Conclusion

We have made several design decisions while implementing the program transformations. We plan to validate these decisions with a usability study. We will explore how the preconditions and outcomes of a program transformation can be tuned to make them more user-friendly. At the same time, we want to collect integer usage data from real world applications. Our empirical study generated interesting results and it seems to merit a more encompassing investigation. For example, we want to explore how to improve the justification of a type change in CIT transformation, either through more test samples or user input.

It is very easy to write a C program with misused integers. Our program transformations intend to make integer problems visible. At the same time, they successfully fix all variants of integer problems that result in vulnerabilities. Ultimately, these power tools could be a simple and effective method of guiding towards correct integer code.

Details about the program transformations and results are available at: http://munawarhafiz.com/research/intproblem.

References

[1] MITRE Corporation, "Vulnerability type distribution in CVE," 2007.

[2] E. N. Ceesay, J. Zhou, M. Gertz, K. N. Levitt, and M. Bishop, "Using type qualifiers to analyze untrusted integers and detecting security flaws in C programs," in *DIMVA*, ser. Lecture Notes in Computer Science, vol. 4064. Springer, 2006, pp. 1–16.

[3] K. Ashcraft and D. Engler, "Using programmer-written compiler extensions to catch security holes," in *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002.

[4] X. L. David Molnar and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary Linux programs," in *Proceedings of the 18th USENIX Security Symposium*, 2009.

[5] R. Wojtczuk, "UQBTng: A tool capable of automatically finding integer overflows in Win32 binaries," in *Chaos Communication Congress*, 2005.

[6] T. Wang, T. Wei, Z. Lin, and W. Zou, "Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution," in *NDSS*, 2009.

[7] D. Brumley, D. X. Song, T. cker Chiueh, R. Johnson, and H. Lin, "RICH: Automatically protecting against integer-based vulnerabilities," in *NDSS*. The Internet Society, 2007.

[8] P. Chen, Y. Wang, Z. Xin, B. Mao, and L. Xie, "Brick: A binary tool for run-time detecting and locating integer-based vulnerability," in *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, march 2009, pp. 208 –215.

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

[10] M. Fowler, *Refactoring: Improving The Design of Existing Code*. Addison-Wesley, Jun 1999.

[11] M. Hafiz, "Security on demand," Ph.D. dissertation, University of Illinois Urbana-Champaign, 2010.

[12] M. Hafiz, P. Adamczyk, and R. Johnson, "Systematically eradicating data injection attacks using security-oriented program transformations," in *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS-09)*, Feb 2009.

[13] R. Seacord, *CERT C Secure Coding Standard*. Addison-Wesley, 2008.

[14] *MISRA-C: 2004 — Guidelines for the use of the C language in critical systems*, MISRA Consortium, 2004.

[15] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *ICSE*. IEEE, 2012, pp. 233–243.

[16] International Organization for Standardization, *ISO/IEC 9899:1999: Programming Languages — C*, Dec 1999.

[17] M. Hafiz and J. Overbey, "OpenRefactory/C: An infrastructure for developing program transformations for C programs," in *OOPSLA '12: Companion to the 27th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2012.

[18] National Institute of Standards and Technology (NIST), "SAMATE - Software Assurance Metrics and Tool Evaluation," 2012.

[19] blexim, "Basic integer overflows," *Phrack*, vol. 60, 2002.

[20] National Vulnerability Database, "CVE-2010-1513," 2010.

[21] ——, "CVE-2008-1801," 2008.

[22] CERT, "Integerlib library."

[23] National Vulnerability Database, "CVE-2011-3026," 2012.

[24] "Photran - An Integrated Development Environment and Refactoring Tool for Fortran," http://www.eclipse.org/photran/.

[25] J. Overbey and R. Johnson, "Generating rewritable abstract syntax trees," in *Software Language Engineering: First International Conference, SLE 2008. Revised Selected Papers*. Springer-Verlag, 2009.

[26] J. L. Overbey and R. E. Johnson, "Differential precondition checking: A lightweight, reusable analysis for refactoring tools," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011*. IEEE, 2011, pp. 303–312.

[27] National Vulnerability Database, "CVE-2010-1516," 2010.

[28] ——, "CVE-2012-2110," 2012.

[29] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[30] Microsoft Corporation, "PREfast analysis tool."

[31] Chinchani, Iyer, Jayaraman, and Upadhyaya, "ARCHERR: Runtime environment driven program safety," in *ESORICS: European Symposium on Research in Computer Security*. LNCS, Springer-Verlag, 2004.

[32] D. Sarkar, M. Jagannathan, J. Thiagarajan, and R. Venkatapathy, "Flow-insensitive static analysis for detecting integer anomalies in programs," in *SE'07: Proceedings of the 25th conference on IASTED International Multi-Conference*. Anaheim, CA, USA: ACTA Press, 2007.

[33] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, J. Ferrante and K. S. McKinley, Eds. ACM, 2007, pp. 89–100.

[34] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS*. The Internet Society, 2008.

[35] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, "Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time," in *15th European Symposium on Research in Computer Security ESORICS 2010*. Springer, 2010.

[36] S. Chen, Z. Kalbarczyk, J. Xu, and R. K. Iyer, "A data-driven finite state machine model for analyzing security vulnerabilities," in *Proceedings 2003 International Conference on Dependable Systems and Networks (DSN 2003)*. IEEE Computer Society, Jun. 2003, pp. 605–614.

[37] D. LeBlanc, "Integer handling with the C++ SafeInt class," 2004.

[38] "CLN: Class Library for Numbers," http://www.ginac.de/CLN/.

[39] "GMP: Gnu Multiple Precision Arithmetic Library," http://gmplib.org/.

[40] R. Dannenberg, W. Dormann, D. Keaton, R. Seacord, D. Svoboda, A. Volkovitsky, T. Wilson, and T. Plum, "As-If Infinitely Ranged integer model," in *2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, nov. 2010, pp. 91 –100.

[41] MITRE Corporation, "Common vulnerabilities and exposures."