Program Transformations to Fix C Integers

Zack Coker
Auburn University
zfc0001@tigermail.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 programs. 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.

Our work has been accepted at ICSE 2013 [1]. More information is available at the project page: `http://www.munawarhafiz.com/research/intproblem/`. One can try the three program transformations on the web: `http://www.openrefactory.org/demo.html`.

1. Problem and Motivation

Ever since attackers shifted their attention from traditional buffer overflows to other types of attacks, attacks on integer vulnerabilities have been on the rise, even featuring as the second most common vulnerability behind buffer overflows [2]. Typical programs with reported 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. Our results show that even mature programs contain these problems.

There are four types of integer problems [3] [4]: 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.

Most of the recent works on integer problems focus on detecting integer vulnerabilities. These approaches are insufficient in 3 ways. First, they only apply to integer overflows. A few tools target integer problems other than overflows, but they have problems with performance [5] and accuracy [4] [6]. Second, these tools are not used because runtime approaches have performance overhead (as high as 50X slowdown for BRICK [5]). 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 [7] reported that unintended integer overflows are very common in real programs. At the moment, these 3 factors are a part of why these tools are not more widely adopted.

2. Background and Related Work

Research on integer problems primarily focuses on detecting integer overflow vulnerabilities either statically or dynamically. The static analysis approaches target either source code or binary code. IntScope [8] modifies binaries to an intermediate representation, and then checks for integer overflow combining symbolic execution and taint analysis. UQBTng [9] also decompiles and then applies model checking with CBMC [10]. On the other hand, some approaches examine source code, e.g., Microsoft's PREfast [11], ARCHERR [12], etc. Ashcraft and colleagues' [13] 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 [14]. However, both approaches are not applicable to detect all types of integer problems. Ceesay and colleagues [15] added type qualifiers to detect integer overflow problems. Their approach requires user annotation and only detects integer overflow.

Some dynamic analysis tools can detect all types of integer problems, e.g., RICH [4], BRICK [5], and SmartFuzz [6]. RICH [4] instruments programs to detect safe and unsafe operations based on well-known subtyping theory. BRICK [5] uses a modified version of Valgrind [16]. It is

either slow (50X slowdown) or has many false positives. SmartFuzz [6] is also based on Valgrind, but it uses dynamic test generation techniques to generate inputs, leading to good test coverage. SAGE [17] and IOC [7] are other dynamic approaches, but they target fewer integer problems.

3. Approach and Uniqueness

3.1 General Approach

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 that explicitly introduces casts to disambiguate integer usage and fix signedness and widthness problems, a REPLACE ARITHMETIC OPERATOR (RAO) transformation that replaces arithmetic operations with safe functions to detect overflows and underflows at runtime, and a CHANGE INTEGER TYPE (CIT) transformation that changes types to fix signedness and widthness problems. They are similar to refactorings [18], but they do not intend to preserve behavior. They are instead security-oriented program transformations [19] [20], 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 [21] and MISRA C [22] guidelines.

A transformation-based approach has several advantages. First, program transformations remove a frequent and repetitive task from developers. People are bad at repetitive tasks—computers are better. Second, program transformations making small changes are more likely to be adopted [23]. Most importantly, the source-level program transformations would help developers better understand and be aware of the subtleties of integer problems.

We have implemented the program transformations as Eclipse plugins. The program transformations are developed using OpenRefactory/C [24], our infrastructure for developing program transformations for C. OpenRefactory/C features sophisticated analyses needed to support complex program transformations.

3.2 Contributions

Our approach 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 3.4). These program transformations modify the integer model of a C program towards a safe model. They guide developers by explicitly revealing mistakes in integer operations (Section 4.4).
- It demonstrates that complex yet accurate source-level C program transformations can be implemented as part of the refactoring catalog of popular IDEs.
- It shows that the three program transformations can prevent all types of integer problems (Section 4.1). It also evaluates the accuracy of the program transformations by applying them automatically to make many small modifications to open source programs, in a way that produces programs that maintain functionality (Section 4.2) and have minimal overhead (Section 4.5).
- It presents an empirical study—based on well-known open source programs—about the types and patterns of integer problems in source code (Section 4.3).

3.3 Program Transformations in Action

Consider this recent widthness/overflow vulnerability [25] in Ziproxy v.3.0.0 in line 979 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. 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; the wrapped around value would then be cast to long long int. During the wrap around, information will be lost.

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. 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));
```

3.4 Description of the Three Transformations

3.4.1 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 properly handled.

Motivation: Signedness and widthness mismatches are very common (Section 4.3) and may lead to unexpected results in a program's behavior. If the types were explicitly

mentioned (e.g., MISRA rules 10.1-10.6 [22]), a compiler can unambiguously enforce the types.

Precondition: A developer selects an integer variable and invokes the AIC transformation. The variable's references are checked to make sure that it is used in an unsafe operation (e.g., parameter to `memcpy` function). The precondition does not guarantee a change. It only means that the situation needs more in depth investigation.

Mechanism: AIC determines the declared type of a variable, checks all references, 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.

Example: Consider this program from NIST's SAMATE reference dataset [26].

```
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 the `memcpy` function call makes developers aware of the casts performed automatically by the weak-typed C system.

3.4.2 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.

Motivation: Integer overflows and underflows are silently handled in C. It would be advantageous if arithmetic operations in C could be replaced with safe functions, that explicitly notify during an integer overflow and allow developers to write handlers for dealing with the situation.

Precondition: A developer selects an expression containing an arithmetic operation: a binary operation ($+$,$-$,$*$,$/$), a unary prefix or postfix operation ($++$,$--$), and an arithmetic assignment operation ($+=$,$-=$,$*=$,$/=$). The transformation checks if the selected expression or the lvalue storing the result is used in an unsafe context.

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 [21]. 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 [21]).

Example: An integer overflow vulnerability [27] was reported in rdesktop version 1.5.0. The problem is an 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 [28]:

```
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.

3.4.3 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.

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 4.3). This can lead to both signedness and widthness problems. To prevent these problems, it would be advantageous to change the type of the variable to its underlying type.

Precondition: A developer selects an integer variable and invokes the CIT transformation. It checks that a reference to the variable is used in an unsafe context.

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.

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. AIC should follow CIT to clean up the code.

Example: Consider this recent vulnerability in libpng v1.4.9 [29], 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 ...

285   int copy = output_size - count;
286   if (avail < copy) copy = avail;
287   png_memcpy(output + count, png_ptr->zbuf, copy);
288 ...
```

CIT is applied to `avail` and then `copy`, which are used as an unsigned integer in all important contexts. Their types are modified to an unsigned int with `avail` shown below.

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

An AIC transformation may follow CIT to remove casts, or introduce new casts.

4. Results and Contributions

We ask the following research questions:

A. Are the program transformations effective in securing systems from the different type of integer vulnerabilities?

B. Does a program transformation-based technique work?

C. How can we understand integer problems better? What types of integer problems are more common?

D. Are the program transformations useful for developers and will they help developers write better code?

E. How is performance affected by the transformations?

4.1 Can the transformations secure systems?

Table 1. 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

Securing Benchmark Programs. SAMATE is the most comprehensive benchmark available for integer vulnerabilities in C and C++. Table 1 lists 7 CWEs that describe integer vulnerabilities in 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 1 shows that the transformations ran on more than 15 MLOC.

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. More details are in our paper [1].

Table 3. 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

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. We applied the appropriate program transformations to remove the root cause of the reported vulnerability.

SWFTools 0.9.1 had multiple reported vulnerabilities [30] 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;`. CIT modified the type to unsigned integer.

OpenSSL 0.9.8 had a widthness vulnerability [31] in crypto/asn1/a_d2i_fp.c file. The variable `want` in a function (asn1_d2i_read_bio) is declared as int, but it is used as unsigned long int. CIT performs the transformations.

The signedness problem [29] in libpng 1.4.9 is fixed with a combination of CIT and RAO transformations (Section 3.4.3). The overflow vulnerability in rdesktop 1.5.0 [27] is fixed with RAO transformation (Section 3.4.2). A vulnerability [25] in Ziproxy is fixed by a combination of AIC and RAO transformations (Section 3.3).

4.2 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 contained in 4,493 functions in 222 files (Table 3) and our results are shown below.

AIC was applied on all local variables, parameters, array access expressions, and structure element access expressions—

Table 2. 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/C7]
		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 %	21.74 %
	15746	4886	2140	709	3019	10754	49291	2911	7243	(avg) 72.04 %	(avg) 13.18 %

1,847 total in libpng (Table 2). Of these, 1,262 were considered unsafe. 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 (10.76% of all references). RAO results are in our paper [1].

Table 4. 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 4). 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.

While these transformations were not designed to be applied automatically in all cases as done here, it shows that a large number of changes still allow the programs to run correctly, which we verified.

4.3 Can we understand the integer problems better?

For space issues, we focus on certain key results from the empirical study. The most surprising aspect is the number of integer problems in matured software. For example, Table 4 shows that 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 integer type mistakes are more common. Figure 1 shows the distributions. It shows that 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`).

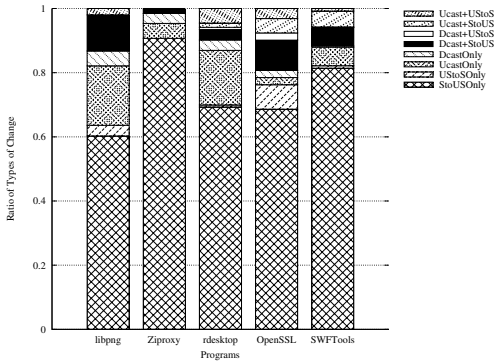

Figure 1. Ratio of Types of Changes Made by CIT

4.4 Are the transformations useful for developers?

A specified goal of the transformations is to make implicit integer problems visible so 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 2). 139 (18.5%) of these 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 on 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; which guide developers and fix potential vulnerabilities.

4.5 What is the performance overhead?

Our test results show that AIC actually lowered the runtime speed of the programs while RAO and CIT both produced a slight overhead. The ICSE paper [1] has more.

5. Conclusion

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.

Acknowledgments

I would like to thank my advisor Dr. Munawar Hafiz for all the help he gave me throughout the research project. I would also like to thank Paul Adamczyk, Farhana Ashraf, Chucky Ellison, Jeffrey Overbey, and other anonymous reviewers. This work was supported by the NSF grant CCF-1217271.

References

[1] Z. Coker and M. Hafiz, "Program transformations to fix C integers," in *Proceedings of the International Conference on Software Engineering, ICSE'13*, 2013.

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

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

[4] 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.

[5] 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.

[6] 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.

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

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

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

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

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

[20] 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.

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

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

[23] 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.

[24] 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.

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

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

[27] National Vulnerability Database, "CVE-2008-1801," 2008.

[28] CERT, "Integerlib library."

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

[30] ——, "CVE-2010-1516," 2010.

[31] ——, "CVE-2012-2110," 2012.