Integers In C: An Open Invitation To Security Attacks?

Zack Coker
zfc0001@auburn.edu

Samir Hasan
szh0064@auburn.edu

Jeffrey Overbey
joverbey@auburn.edu

Munawar Hafiz
munawar@auburn.edu

Christian Kästner†

Auburn University
Auburn, AL, USA

†Carnegie Mellon University
Pittsburgh, PA, USA

ABSTRACT

We performed an empirical study to explore how closely well-known, open source C programs follow the safe C standards for integer behavior, with the goal of understanding how difficult it is to migrate legacy code to these stricter standards. We performed an automated analysis on fifty-two releases of seven C programs (6 million lines of preprocessed C code), as well as releases of Busybox and Linux (nearly one billion lines of partially-preprocessed C code). We found that integer issues, that are allowed by the C standard but not by the safer C standards, are ubiquitous—one out of four integers were inconsistently declared, and one out of eight integers were inconsistently used. Integer issues did not improve over time as the programs evolved. Also, detecting the issues is complicated by a large number of integers whose types vary under different preprocessor configurations. Most of these issues are benign, but the chance of finding fatal errors and exploitable vulnerabilities among these many issues remains significant. A preprocessor-aware, tool-assisted approach may be the most viable way to migrate legacy C code to comply with the standards for secure programming.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.m [**Programming Languages**]: Miscellaneous

General Terms

Languages, Security

Keywords

C, C Preprocessor, Integer Issues, Safe C standard.

1. INTRODUCTION

The C programming language has a plethora of integer types, which vary in both width and sign (e.g., `int` vs. `unsigned long`). When integer operations are applied to operands of different types, the results may be unexpected. There are four possible *integer issues* [4, 5]: (1) a *signedness issue* occurs when a value of an unsigned type is interpreted as a signed value, or vice versa; (2) an *integer overflow* occurs when integer operations such as addition or multiplication produce a result that exceeds the maximum value for a type; (3) an *integer underflow* occurs when integer operations such as subtraction or multiplication produce a result that is less than the minimum value for a type; and (4) a *widthness issue* is the loss of information when a value of a larger integer type is assigned to a location with a smaller type, e.g., assigning an `int` value to a `short` variable.

Secure coding standards, such as MISRA's *Guidelines for the use of the C language in critical systems* [37] and CERT's *Secure Coding in C and C++* [44], identify these issues that are otherwise allowed (and sometimes partially defined) in the C standard and impose restrictions on the use of integers. This is because most of the integer issues are benign. But, there are two cases where the issues may cause serious problems. One is in safety-critical sections, where untrapped integer issues can lead to unexpected runtime behavior. Another is in security-critical sections, where integer issues can lead to security vulnerabilities (*integer vulnerabilities*).

Integer issues and corresponding vulnerabilities can be very subtle. Consider this recent widthness/overflow vulnerability [40] in Ziproxy v.3.0.0 in line 979 of image.c file.

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

In the program, the result of multiplying three `int` variables is stored in `raw_size`, a `long long int` variable. The developer perhaps thought the `long long int` type on the left side of the assignment expression ensured that the multiplication would be able to store a value outside the range of `int`. However, C first performs the integer multiplication in `int` 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`. Even though `raw_size` can hold a large number, the multiplication result will be limited to `int` because of the wrap around behavior during integer arithmetic.

Currently, C handles these integer issues silently. For example, the overflow above is never reported to a user (unless specific compiler reporting options are turned on); instead, the compiler silently truncates the result. Hence, developers may remain unaware of the consequences when they declare or use integer variables in an unsafe manner. Therefore, even mature C programs, that have evolved through many versions and have been reviewed by many programmers, may contain numerous instances of these integer issues. Because the issues are subtle, they may remain dormant for a long time. For example, two integer overflow vulnerabilities in Pac-Man's assembly code were discovered nearly two decades after it was written [27].

To date, there has been very little work studying integer usage in C empirically. There has been one recent empirical study on integer vulnerabilities, but it focuses only on integer overflows and underflows: Dietz and colleagues [17] distinguished between intentional and unintentional uses of

defined and undefined C semantics to understand overflows and underflows. Our previous work introduced a program transformation-based approach to fix integer issues [13]; we additionally performed a small-scale empirical study on what are the most common mistakes in declaring C integers.

The scope of this study is much broader. In this paper, we seek to understand how prevalent integer issues are (prevalence), what kinds of mistakes are more commonly made, if the integer issues change over time (evolution), and what effects varying preprocessor configurations have on integer issues (variation). We asked the following questions:

(1) How common are the four different types of integer issues? Are there certain patterns of inconsistencies for each kind of integer issue? (§3)

(2) As an application evolves over time, how do integer declarations and uses change? Do developers fix the integers whose declared types do not match the used types? Does the number of integer problems decrease over time? (§4)

(3) When C programs are configurable via the preprocessor, integer variables may have different types in different configurations. How frequently does this occur, and do integer issues tend to occur across all configurations or only in particular configurations? (§5)

To answer the first question, we detected all four integer issues occurring in 7 well-known, mature C programs, automatically analyzing over 900,000 lines of preprocessed code in the process. For the second question, we studied a total of 52 versions of these 7 programs (over 6 million lines of preprocessed code) released over a span of over 9 years on average. We tracked the integers across multiple versions and collected information about how integer declarations and uses evolved. To answer the third question, we analyzed recent versions of BusyBox and the Linux kernel to understand whether the configuration options complicate the detection of integer issues. This analysis was done on nearly one billion lines of partially-preprocessed code.

Key results include the following.

(1) Developers typically write C code that do not follow the stricter integer standards. The most common issue is a variable's declared type not matching the contexts in which it is used. One out of every four declarations are inconsistent with how the integers are used, and one out of every eight uses of integers are inconsistent with the declarations. Inconsistencies between signed and unsigned integer values are more frequent than inconsistencies in integer width. (§3)

(2) Over time, while the developers fix some inconsistent declarations, new variables (with new inconsistent declarations) are introduced, so there is no net improvement in the number of integers inconsistently declared. (§4)

(3) A significant number of integer variables have types that may vary according to the preprocessor configuration, and many integer problems are only present under very specific sets of configuration options. (§5)

(4) Changing code that was developed without using a strict integer standard to a safer integer standard will have many complex difficulties.

This is the first study that explore integer issues in C at a large scale, including how integer issues change as software evolves and the impact configurability (via the C preprocessor) has on integer issues. Our results benefit both the programmers and the toolsmiths. Programmers can benefit from knowing what pitfalls are the most common, as well as how integer usage tends to change over time, so that mitigation strategies can be incorporated into the development process when appropriate. Tool builders will benefit particularly from our results regarding the C preprocessor, which indicate that detecting integer issues within a single configuration is insufficient and masks a significant (and complex) source of possible errors. Finally, this study documents some of the complexities of bringing code developed without using a strict integer standard to meet those guidelines.

More details about the study and results are available at: `http://munawarhafiz.com/research/intstudy/`.

2. OVERVIEW OF THE STUDY

2.1 Identifying Integer Issues

There are standards describing a safe C integer model, including the CERT C Secure Coding Standard [44] and the MISRA C guidelines [37]. In our study, we identify each deviation from the safe integer model as an integer issue. We do not distinguish intentional use of unsafe integer behavior, which can also happen [17]. An automated analysis cannot unambiguously understand the intent of a programmer.

Signedness and widthness issues are described by CERT rule INT31-C and MISRA rules 10.1, 10.3, 10.4, 10.6, and 10.7. These specify that integer conversions may result in lost or misinterpreted data. Also, the weak typing system of C may produce unexpected results. For example, 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 (for example, `-1 < 80u` is false, since the comparison is done in an unsigned context).

Integer overflow and underflow are caused by wraparound operations as described by CERT rules INT02-C, INT-30C, and INT-32C, and MISRA rules 10.1–10.6. These specify the problems that can be caused by integer promotion and integer conversion rank in C [28] and the consequences of signed and unsigned overflow behavior.

2.2 Research Study

The following three sections describe the research questions posed and the answers discovered. Section 3 quantifies the frequency with which integer issues occur, and it discusses what types of issues are encountered, as well as whether these issues are localized to particular parts of a program. Section 4 considers how integer variable declarations change over time. By analyzing multiple versions of the same programs, it quantifies how often integer variables change and how this impacts the number of integer issues in the program. Finally, Section 5 discusses the impact of the C preprocessor on integer variable declarations. Using the C preprocessor can cause a variable to be declared or used differently, depending on configuration options set at compile time. In this section, we quantify the number of variables whose types are determined by preprocessor settings, as well as whether integer problems are limited to single configurations or common among all configurations. Data for Sections 3 and 4 were collected using OpenRefactory/C [26], which was used to collect similar data in prior work [13]; data for Section 5 were collected using TypeChef [29,30,34], which supports parsing and analyzing C programs under

multiple preprocessor configurations.

3. MISUSE OF C INTEGERS

3.1 Research Questions

Programmers can allow inconsistencies in their code by not following more secure C coding parctices while they are declaring integer variables, using the declared integer variables, and specifically while they are performing arithmetic operations on integers. This section characterizes the mistakes made in these three contexts (RQ1, RQ2, and RQ3). We also explore if integer issues are concentrated in some parts of the code (perhaps made by a few developers), or spread throughout the code, making it harder to bring to a safer standard (RQ4). We asked four questions:

RQ1. Are integer variables declared consistently in C programs? What kind of mistakes happen in declarations?

RQ2. What kind of signedness and widthness issues occur when integers are used?

RQ3. What kind of overflow and underflow issues occur when integers are used in arithmetic expressions?

RQ4. Are integer issues concentrated in a few places, or spread throughout the programs?

3.2 Test Corpus

We studied 7 well-known open source software. Some had reported integer vulnerabilities (e.g., OpenSSL's widthness vulnerability [41]); others are programs that are matured (been around for 13 years on average) and widely used (e.g., zlib, released in 1995, is a de facto standard and has been used by thousands of applications for compression). We analyzed 381 files and 2,673 functions in these programs (Table 1). The analysis was run after preprocessing the programs; we analyzed nearly one million lines of preprocessed code.

Table 1: Test Programs

Programs	# of C Files	# of Functions	KLOC	PP KLOC	Maturity (years)
libpng - 1.6.2	16	421	32.2	53.83	18
Ziproxy - 3.3.0	18	122	5.7	15.72	8
SWFTools - 0.9.2	19	264	12.9	49.42	9
rdesktop - 1.7.1	28	414	19.7	80.24	10
zlib - 1.2.6	15	117	8.8	18.55	16
GMP - 4.2.3	93	95	9.7	79.11	17
OpenSSL - 1.0.1e	192	1,240	9.9	684.99	14
	381	2,673	98.9	981.86	(avg) 13.14

KLOC: Lines of code / 1000; PP KLOC: Preprocessed KLOC; Maturity: Years till the version was released

3.3 Data Collection Method

In our previous work, we described three program transformations that can be applied to fix all possible types of integer issues in C programs [13]. We updated and reused the analysis that accompanied these transformations. We used the name binding, type analysis, and control flow analysis analysis features of OpenRefactory/C.

We analyzed the declared types of all local variables and formal parameters. We determined the *declared type* of a variable, checked if it is used as another type (*underlying type*) in important contexts, and reported inconsistencies. We considered the following three use contexts as *important contexts*: (1) when a variable is used on the left hand side of an assignment expression, (2) when the variable is used as an actual parameter in a predefined set of critical function calls (e.g., `memcpy`), (3) when the variable is used as the index of an array access. These are the known contexts in which integer issues commonly occur as identified by prior research [13] and safe C standards [37, 44].

For each variable, the important contexts are collected along with the declared types and underlying types in each of them. We considered a type declaration inconsistent if we find 75% of the underlying types in important contexts as a single type differing from the declared type [13]. Otherwise we assume the variable declaration is consistent.

We also analyzed the context each integer is used by analyzing all the uses that include local variables, formal parameters, array access expressions, and structure element access expressions. We determined the context an integer is used, whether the declared type differs from underlying types, and reported the inconsistencies.

Finally, we analyzed all binary expressions ($+,-,*,/$), prefix and postfix expressions ($++,--$), and arithmetic assignment expressions ($+=,-=,*=,/=$). We determined if the integer arithmetic happens in an important context, or the integer arithmetic assigns a value to an integer that flows to an integer context. We considered all arithmetic operations in these contexts to potentially overflow.

We ran the analysis on preprocessed files. We only counted the integer variables that are locally declared and used inside functions in the C files; so all global variables and the variables coming from header files are excluded.

Table 2: Variables Declared Inconsistently

Programs	# of Variables [C1]	References [C2]	Inconsistent Decl.[C3]	% Inconsistency [C3/C1]
libpng	1,107	5,223	263	23.76
Ziproxy	350	1,443	114	32.57
SWFTools	628	2,184	117	28.18
rdesktop	1,291	7,295	227	17.58
zlib	383	2,256	94	24.54
GMP	250	868	60	24.00
OpenSSL	3,103	11,897	553	17.82
	7,112	31,166	1,428	(avg) 24.06

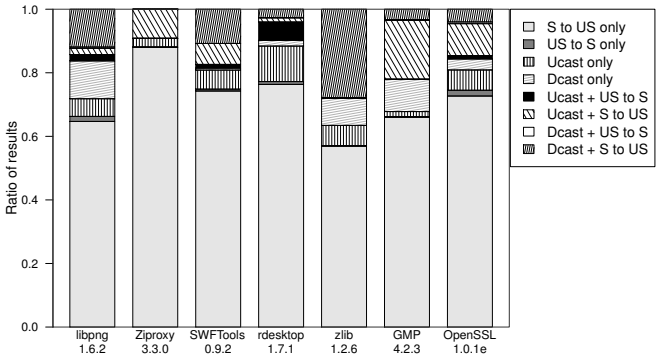
3.4 Results

RQ1. Are integer variables declared consistently in C programs? What kind of inconsistencies appear in declarations?

Key results: Every one out of four integers are inconsistently declared. Declaration inconsistencies with signedness of the same rank are more common than those with widthness. For widthness issues, upcasting are more common than downcasts.

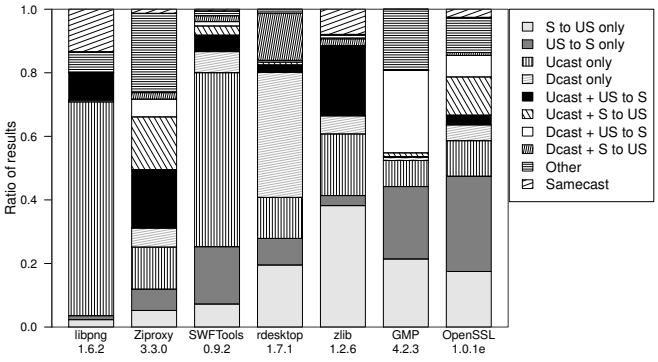
Table 2 shows the number of inconsistencies in variables declarations in our corpus. For libpng, we analyzed 1,107 integer variables and their 5,223 references (uses). We identified 263 inconsistent declarations. Surprisingly, a lot of variables have inconsistencies with their type declaration—23.76% (263/1,107) in libpng, on average 24.06% overall, i.e., about one issue with declaration in every four variables.

Figure 1 shows the distribution of the kinds of inconsistencies in variable declaration. We categorize a declaration as 'signed to unsigned only' if it is declared as a signed variable but used as an unsigned variable of the same rank in important contexts (Section 3.3). It shows that signedness



S: Signed; US: Unsigned; Ucast: Upcast; Dcast: Downcast

Figure 1: Kinds of Issues in Declaration



S: Signed; US: Unsigned; Ucast: Upcast; Dcast: Downcast

Figure 2: Kinds of Issues in Variable Use

Table 3: Variables Used Inconsistently In Programs

Programs	# of Variables [C1]	Var. Used Inconsistently [C2]	References Checked[C3]	Inconsistent Use [C4]	% Variables Changed [C2 / C1]	% Instances Changed [C4 / C3]
libpng	7,601	2,610	81,793	12,532	34.34	15.32
Ziproxy	995	258	3,897	537	25.93	13.78
SWFTools	4,782	1,396	50,697	10,443	29.19	20.60
rdesktop	8,199	4,565	782,831	71,319	55.68	9.11
zlib	4,344	1,750	123,823	14,582	40.29	11.78
GMP	734	221	9,308	500	30.11	5.37
OpenSSL	20,838	5,801	168,663	27,329	48.46	16.20
	47,493	16,601	1,221,012	137,242	(avg) 37.71	(avg) 13.17

issues of the same width are the most common issues, more common than widthness issues (shown as upcast and downcast and combinations with signedness). The most common inconsistency is declaring a variable as signed but using it as unsigned. This is perhaps because developers are unaware of the potential problems that might arise from inconsistent declarations and do not use additional type specifiers for signedness when they declare a variable unless they deem it absolutely necessary (e.g., **`int`** used in place of **`unsigned int`**).

Among the widthness mismatches, upcasting issues are more common, i.e., a variable is declared as a lower ranked type but used as a higher ranked type (e.g., declared as **`char`** used as **`int`**). Consider the variable `c` in parser.yy.c file in libpng.

```
1176    char c,c1=yytext[0];
...
1179    c=input();
...
1182    printf("%c", c);
```

The variable `c` is declared as **`char`** but receives the **`int`** type returned by `input` function. It is safer to declare it as **`int`** to prevent widthness issues. The opposite kinds of mismatches do happen, but they may not have consequences other than inefficient memory use. In these cases, developers have proactively declared integers to be large even though they contain a small range of values.

RQ2. What kind of signedness and widthness issues occur when integers are used?

Key results: Every one out of eight uses are in a context different from the declaration. Inconsistencies with signedness are more common than those with widthness. Developers use type casts to specify contexts, but sometimes they introduce casts not needed.

Table 3 shows the number of variables used inconsistently in our test programs. We analyzed 7,601 variables in libpng and 81,793 references (uses) of these variables. We identified that 12,532 references were in contexts different from the declarations. About 15.32% (12,532/81,793) instances in libpng were using the variables inconsistently, on average 13.17% overall, i.e., one signedness/widthness issue in every ten instances of variable used.

Figure 2 shows the distributions of these inconsistencies. We categorized them according to what the declared type is to what the underlying type is. For example, a variable declared as an **`int`** but used as an **`unsigned int`** in a context is reported as 'signed to unsigned only', a variable declared as an **`int`** but used as a **`long`** in a context is reported as 'upcast only', etc. Unlike Figure 1, there is no clear pattern here, denoting that all sorts of inconsistencies happen.

Inconsistencies in signedness are more common than inconsistencies in widthness following the observation about type declaration (Figure 1). However, signed to unsigned mismatches happen a little bit more than unsigned to signed mismatches for the majority of the software. Only SWFTools has many unsigned variables used in signed contexts. This is because there is one file (lib/gocr/ocr0n.c) that has several function calls that perform unsigned integer operations in the parameter, but the parameters expect signed integers.

There are many void pointers used as integer pointers, and vice versa ('other' category in Figure 2). Using void pointers without explicitly casting them to the type they are used as seems to be a common practice among C programmers.

Sometimes programmers explicitly cast contexts that do not require a cast ('samecast' category in Figure 2). One example is the variable `curr` in the file deflate.c of zlib-1.2.6.

```
1502  ulg curr = s->strstart + (ulg)(s->lookahead);
...
1520  init = (ulg)curr + WIN_INIT - s->high_water;
```

The declared type, `ulg`, is a *typedef* for **`unsigned long int`**. The cast in line 1520 is not needed since the addition will happen in a correct context even without the cast. We tracked this variable back; it was added in version 1.2.4 of zlib along with the redundant cast. This shows that, in a few cases, either developers are too cautious about a variable's underlying type or feel the need to explicitly document the type there for easy understanding later.

RQ3. What kind of overflow and underflow issues occur when integers are used in arithmetic expressions?

Key results: One of every seven integer arithmetic is used in an important context; overflow and underflow problems have severe consequences in these cases.

Table 4 shows the arithmetic expressions with issues. libpng had the most number of integer expressions in important contexts (27.56%), on average 15.86% overall, i.e., one potential overflow/underflow risk in every seven arithmetic operations.

Table 4: Arithmetic in Important Context

Programs	Expressions Analyzed [C1]	Arithmetic in imp. context [C2]	% Arithmetic in imp. context [C2 / C1]
libpng	24,480	6,746	27.56
Ziproxy	2,805	323	11.52
SWFTools	6,407	686	10.71
rdesktop	15,212	2,515	16.53
zlib	5,868	1,197	20.40
GMP	1,630	285	17.48
OpenSSL	31,217	2,131	6.83
	87,619	13,883	(avg) 15.86

Most arithmetic operations are between **`int`** types and **`long`** types. This is because the declared type of integer variables determine the type level of arithmetic operations and these two types are the most common. GMP had a much higher percentage of variables declared as **`unsigned long int`** (actually `size_t`, which was **`unsigned long int`** in our platform) and a lot of arithmetic operations at that type level.

RQ4. Are integer issues concentrated in a few places, or spread throughout the programs?

Key results: Integer issues are spread throughout the software affecting most of the files.

Figure 3 shows the percentage of files and functions on our seven test programs that have at least one of the four kinds of integer issues. These issues are present in most of the files. Only in GMP, the integer issues are concentrated in a few files, but these files contain most of the code.

Such prevalence of integer issues complicates the task for developers wanting to migrate to a safe integer standard. It also hints at the general unawareness (or apathy) of developers about the integer issues that they introduce.

4. INTEGERS IN MULTIPLE VERSIONS

4.1 Research Questions

Section 3.4 shows that matured C programs contain a lot of variables declared inconsistently. Most of the inconsistencies are about not using the appropriate signedness specifier.

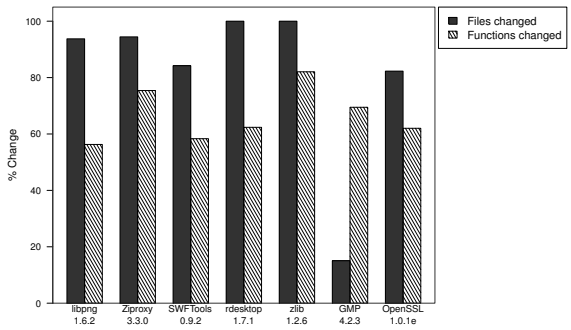


Figure 3: Files and Functions with Integer Issues

One reason may be that developers do not know how an integer variable will be used when they are declaring it; hence they declare the variable as the simplest type possible (**`int`** being easier to declare than **`unsigned int`**). However, it may also happen that developers initially declare variables that are consistent with how the variables are used, but the declarations become inconsistent as the software evolves and the uses change. If developers actively fix the declared types that become inconsistent, the state of integers in software should become better over time.

To gain insight on integer declaration issues, we need to understand how integer variables are declared and used over time. We explore the following research questions:

RQ5. How often does the type and usage of integer variables change? What kind of changes are more common?

RQ6. When only the uses of an integer variable changes, does the variable's declared type become inconsistent?

RQ7. How frequently is the declared type of an integer modified? Does the change in a declared type follow a change in the way the variable is used?

RQ8. Do integer declarations become more consistent with their use as software evolve?

4.2 Test Corpus

We studied 52 release versions of the 7 programs that were used in the first part. The release versions cover software evolution for over 9 years on average and analyzed over 6 million lines of preprocessed C code (Table 5).

Table 5: Test Programs for Multi-Version Study

Programs	Versions	# of C Files	# of Functions	KLOC	PP KLOC	Years from first release to last
libpng	0.9.5, 1.0.40, 1.2.50, 1.4.0, 1.4.9, 1.5.8, 1.5.17, 1.6.2	133	2,859	190.00	376.83	8
Ziproxy	1.4.0, 1.9.0, 2.0.0, 2.2.0, 2.4.0, 2.7.2, 3.0.0, 3.3.0	88	545	24.8	86.79	7
SWFTools	0.7.0, 0.8.0, 0.8.1, 0.9.0, 0.9.2	73	1,157	53.6	188.04	6
rdesktop	1.2.0, 1.3.1, 1.4.1, 1.5.0, 1.6.0, 1.7.0, 1.7.1	162	2,461	102.61	409.08	9
zlib	1.1.3, 1.1.4, 1.2.1, 1.2.2, 1.2.4, 1.2.5, 1.2.7, 1.2.8	126	1,016	66.26	142.91	12
GMP	4.0, 4.1, 4.2.3, 4.3.0, 4.3.2, 5.0.0, 5.0.4, 5.1.2	734	632	74.19	728.36	11
OpenSSL	0.9.6b, 0.9.7, 0.9.7l, 0.9.8a, 0.9.8u, 1.0.0, 1.0.0k, 1.0.1e	1,410	--	610.13	4,490.66	11
	52 versions	2,726	8,670	1,121.59	6,422.67	(avg) 9.14

KLOC: Lines of code / 1000; PP KLOC: Preprocessed KLOC

4.3 Data Collection Method

We analyzed the inconsistencies in declared types and underlying types for all release versions similar to the approach

in Section 3.3. Then we tracked the variables across multiple releases. The variables were tracked using their names, along with the functions and the files that contain them.

Our approach to keep track of variables in multiple releases was simple (e.g., did not detect rename refactorings [18, 31, 35]), but it worked well in practice. On average, we were able to track variables for about 4 versions.

Figure 5 shows the total number of variables that we analyzed in each version. It also shows the number of variables that we were able to keep track of from the starting version. In 3 out of 7 software (OpenSSL, rdesktop, SWFTools), we were able to keep track of over 50% of the variables that we started with throughout all of the versions. Only in Ziproxy, we lost track of over 75% of the variables (21.21% of the initial variables were tracked to the final version).

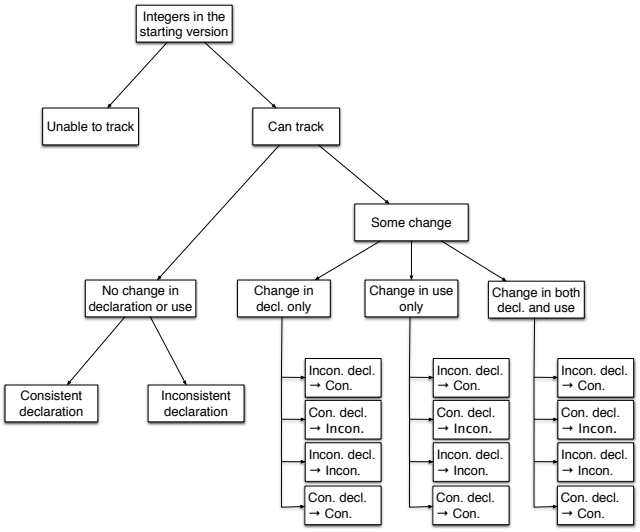


Figure 4: Categories of Changes
Incon.: Inconsistent; Con.: Consistent

We marked a variable as 'Unable to track' if we cannot track it in at least two versions. For the tracked variables, we distinguished three independent types of changes—change in a variable's declaration, change in a variable's uses, and change in both the declaration and the uses. We kept track of whether the changes impact the consistency of the declarations. The change categories are shown in Figure 4.

4.4 Results

RQ5: How often does the integer variables evolve? What kind of changes are more common?

Key results: The number of integers, and thus the complexity of maintaining safe integer code, almost always increase in subsequent releases. Most of the time, only the uses of a variable change. Declaration changes do not happen when a usage change occurs.

Figure 5 shows information about the 52 release versions of the seven software in our study (Section 3.2). It shows the number of variables that we analyzed per release version and the number of variables that can be tracked from the first version. It also shows when these variables from the first version had changes: only in uses, only in declaration, and both declaration and uses.

Integers evolves with software. So we lost track of a few integers. zlib was the most stable of the programs studied, only increasing by 23 variables from the first to the last version under study. Ziproxy, on the other hand, had the greatest percent increase, with the final integer variable count being over 7 times its original integer variable count.

Among all the variables tracked from the starting version, 32.88% had one or more changes. The most common change (89%) is a usage change (Figure 5). Surprisingly, declaration changes do not happen when a usage change occurs; it happened in only 3% of the cases. The remaining 8% were changes in variable declarations only; developers sometimes actively fixed the inconsistencies in declarations.

We investigated if the changes had some patterns, e.g., more that 50% of the changes follow a trend. No specific trends were discovered for all software, except for some project-specific patterns. For example, libpng had changes in which the resulting variables had reduced width which was due to commonly used typedefs being changed between releases (it also causes more declaration changes than use changes, the only such incident, in libpng version 1.4.0, Figure 5), OpenSSL had changes in which the variables mostly convert from signed integers to unsigned integers, etc.

RQ6: Do use changes make declarations inconsistent?

Key results: Usage changes did not have a big effect on the consistency of variable declarations.

We counted the total number of variables that we can track in at least two versions starting from any version (unlike tracking them from the starting version as in RQ5 and Figure 5). We analyzed a total of 6,583 integer variables. In 1,476 variables, only the uses of the variables changed. Figure 6 shows whether the use changes have any impact on the declared type, such as making the consistent declaration inconsistent, etc. *In most of the cases, the declaration had no impact on type correctness.* That means variables declared consistently remained the same even after the uses changed (1,004, $1{,}004/1{,}476 \approx 68.02\%$). Similarly, variables declared inconsistently remained inconsistent (324, $324/1{,}476 \approx 21.95\%$).

Only in 10% of the cases, the changes in uses affected the originally declared types. This includes variables originally declared consistently but now becoming inconsistent (67, $67/1{,}476 \approx 4.54\%$), and vice versa (81, $81/1{,}476 \approx 5.49\%$). An example of a usage change making the declaration inconsistent is in the variable `filter` in the file pngrutil.c of libpng. In version 1.4.8, the variable is used as:

```
2975 void /* PRIVATE */
2976 png_read_filter_row(png_structp png_ptr, png_row_infop
     row_info,
2977 png_bytep row,
2978    png_bytep prev_row, int filter)
...
2980 png_debug2(2, "row=%lu,filter=%d",
2981     (unsigned long)png_ptr->row_number, filter);
2982  switch (filter)
```

The only uses of the variable is in the `png_debug2` function and a switch statement. With these use contexts we were unable to determine the appropriate type of the variable so we considered that the originally declared type is correct. In the next version that we checked, version 1.5.8, the function has changed considerably but not the variable's declared type:

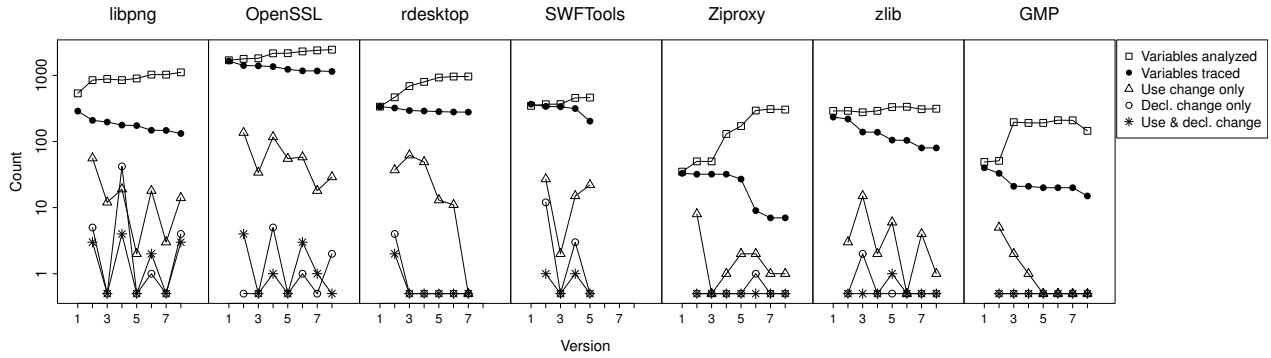


Figure 5: Changes In Variable Declaration And Use Across Versions

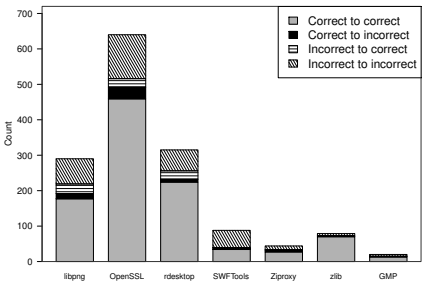


Figure 6: Usage Only Changes

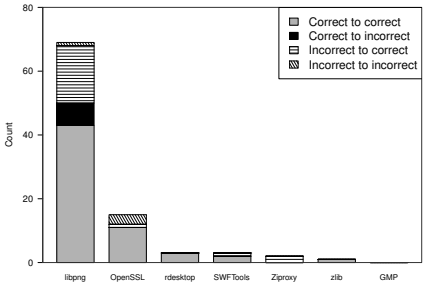


Figure 7: Declaration and Usage

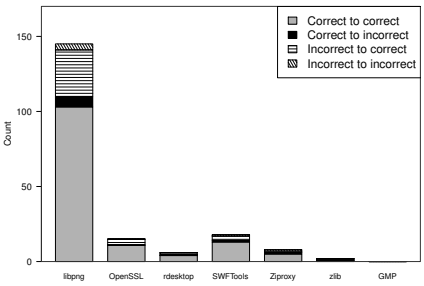


Figure 8: Declaration Only

```
3744 void /* PRIVATE */
3745 png_read_filter_row(png_structp pp, png_row_infop
     row_info,
3746 png_bytep row,
3747    png_const_bytep prev_row, int filter)
...
3750   if (filter > PNG_FILTER_VALUE_NONE && filter <
3751           PNG_FILTER_VALUE_LAST)
3752      pp->read_filter[filter-1](row_info, row, prev_row);
```

Our heuristic determined that the types of the variable `filter` in lines 3750 and 3752 are **`unsigned int`** according to the context. The variable's type should be changed to **`unsigned int`**. In this case, the developers changed the code but did not spend the time to change the type of the variable. Declaration changes are discussed next (RQ7).

RQ7: How frequently do declarations change? Do declaration changes occur with use changes?

Key results: Changes to variable declarations happen relatively infrequently. In most of the cases, developers actively change the declarations of variables. But declaration changes along with uses changes seldom happen. When developers change declarations, they are consistent most of the time.

Developers seldom change declarations when usage change occurs. We analyzed the variables that can be tracked starting from any version (similar to RQ6). We counted 93 times in which declaration changes along with use changes. Interestingly, declaration only changes are two times more common; we counted 194 declaration only changes.

Figure 7 categorizes the changes in both declaration and use. Most of the changes modified the declared types so that they remain correct even after the changes in uses (60, 60/93 $\approx$ 64.52%). The wrongly-declared types are also corrected in some cases (22, 22/93 $\approx$ 23.66%).

Occasionally, such changes result in a wrongly-declared variable (11.83%). 7 of the declaration changes with usage changes went from a consistent type to an inconsistent type (7/93 $\approx$ 7.53%), while 4 of the declaration changes with usage changes went from an inconsistent type to a different inconsistent type (4/93 $\approx$ 4.30%).

Therefore, it is much more common for developers to update a variable's type in a separate version than it is to make a type change when a usage change occurs.

Changes only in variable declaration happens relatively infrequently; we found 194 cases. Figure 8 shows the effect of these changes. A lot of these are consistent declarations changed to other consistent declarations (137, 137/194 $\approx$ 70.62%). Consider the example from version 1.2.50 of libpng, file pngutil.c, function `png_crc_finish`:

```
152  int /* PRIVATE */
153  png_crc_finish(png_structp png_ptr, png_uint_32 skip)
     ...
158  for (i = (png_size_t)skip; i > istop; i -= istop)
```

The variable `skip` is declared as `png_uint_32` (defined as a *typedef* in pngconf.h) which is defined as an **`unsigned long int`** in v1.2.50 and **`unsigned int`** in v1.4.0. The developers have a comment in pngconf.h file in v1.2.50 that they are thinking about changing the type of the typedef to **`unsigned int`** to insure a widthness of 32 bits in 64 bit compilers.

10 of the 194 variables went from a consistent declared type to an inconsistent type (5.15%), 41 went from an inconsistent type to a consistent type (21.13%), and 6 went from an inconsistent type to another inconsistent type (3.09%).

Therefore, developers sometimes made effort to change the variable declaration for consistency.

RQ8: Do integer declarations get better over time?

Key results: Although developers correct some past mistakes, each version introduces a lot of new, and possibly inconsistently-declared, variables, which offsets the improvement in accuracy of the integer declarations.

Integer declarations did not improve in the 52 releases. On average, 26.17% of the integers were declared inconsistently.

We managed to track nearly 90% of all variables in all the versions of test programs (5,851 out of 6,583). When a declaration changed, the majority of the changes made the declarations consistent. When a declaration had a consistency change, 78% of the time it was an improvement; going from inconsistent to consistent. However, a lot of new and inconsistently declared integer variables were added every version (Figure 5). Although developers corrected some past mistakes, the effort did not offset the large number of inconsistently declared integers added to the code, and thus the consistency did not significantly improve.

5. INCONSISTENCIES IN C INTEGERS IN THE PRESENCE OF CPP

Almost all C projects are configurable through the C preprocessor [33]. Typically, macros defined by the user at compile time are used to decide which lines of code are included for compilation through conditional-compilation directives, such as #ifdef, #if and so forth. Even if the project itself does not contain any #ifdef directives, the C header files that are included before compilation typically do. Essentially all recent analysis tools to detect integer problems [5, 16, 49] either work on binaries or follow dynamic analysis; they do not consider the variability introduced by the configurations. The previous sections, considering a single default configuration on preprocessed C code, demonstrate that integer issues are very common. But only considering a single configuration is a bad choice for a tool builder since large parts of the code are potentially unchecked and security claims may not be true for other configurations.

5.1 Research Questions

Do configurations complicate the detection of integer issues? In this section, we will investigate how frequently variables and functions change integer types depending on configuration options. Furthermore, we will approximate how integer issues are distributed in the configuration space and whether analyses on a single configuration or of simple sampling strategies will be effective to detect them.

We ask the following research questions:

RQ9. How many integer types in a program are defined differently depending on build-time options (e.g., as `long` in some configurations and as `unsigned int` in others)?

RQ10. How are integer issues distributed in configuration space? Do they occur in all configurations or only when specific combinations of options are activated?

5.2 Test Corpus

We investigate two highly-configurable software systems to answer these questions: Busybox and the Linux kernel.

Busybox is a reimplementation of command-line utils in a single binary for embedded systems. We analyzed Busybox version 1.21.1 with 294 KLOC code (before preprocessing), 844 configuration options, and 536 files.

The Linux kernel is a highly configurable operating system kernel with over 10,000 build-time configuration options implemented in over 6 million lines of C code [29,45]. When compiling the kernel, users can select from a large number of configuration options, such as different memory models, different file systems, and different drivers. We analyzed the x86 architecture of release 2.6.33.3 including 32 and 64 bit versions with 7691 files, nearly 899 million lines of C code.

5.3 Data Collection Method

Analyzing all configurations separately is not feasible—we do not even know any scalable method to compute the number of configurations in such large configuration spaces, but we can (safely) assume that both systems have more configurations than there are atoms in the universe (about 2^{320}). Instead, we rely on the *TypeChef* infrastructure [29, 30, 34] to parse and analyze C code without preprocessing it first (preprocessing would make a selection for each #ifdef and loose variability). TypeChef parses C code and represents variability in the input (through #ifdefs) as local variations in the abstract syntax tree. For example, if a function is declared with two alternative return types, TypeChef would produce an AST with two alternative subtrees representing the different return types; each alternative is connected to a constraint over the configuration parameters (a propositional formula). During parsing, TypeChef explores all configuration options and then applies a brute-force approach to represent all possible configurations of the source code without relying on heuristics. The technical details are not important for this paper and described elsewhere [29,30,34].

On top of the TypeChef parser, we have previously implemented a type system for GNU C that can handle variations in types [30,34]. For type checking, TypeChef creates a symbol table of previously declared symbols and their types. At times, a symbol will only be defined in certain configurations or have different types in different configurations. For each symbol, we represent the possible types as a choice between types. For instance, `i -> int` describes that symbol `i` has the same type `int` in all configurations, whereas function

```
inw_p -> Choice(CONFIG_SLUB,
    Choice(CONFIG_X86_32,
      (unsigned int) => struct kmem_cache**,
      (unsigned long) => struct kmem_cache**),
    UNDEFINED)
```

is only defined if macro `CONFIG_SLUB` is defined and has a different parameter type when the macro `CONFIG_X86_32` is defined.

For research question RQ9, we investigated variations in the symbol table. To capture local variables, we inspected the symbol table at each local scope inside a compound statement. We counted how many top-level declarations and local variables have configuration-dependent types, specifically when they were only different by integer types (including different integer types in function parameters, return types of functions, and pointer types). Types defined locally through the `typedef` specifier are resolved (that is, we compare only primitive C types). Configuration-dependent types that differ in other ways, e.g., in the number of function parameters or in float vs. double are not relevant for our analysis and excluded from the reported numbers. The results also exclude variability in structs (e.g., a field of a struct or union may have alternative integer types).

Since many top level declarations are actually included from header files and since similar header files are included

Table 6: Alternative Integer Types in Highly Configurable Systems

	Busybox		Linux kernel	
Alternative Integer Types	**1**	**> 1**	**1**	**>1**
Excluding headers (sum)				
Top-level declarations	23,113	35 (0.2%)	183,832	7,919 (4.1%)
Local variables	18,542	71 (0.38%)	638,028	10,135 (1.6%)
Including headers (avg ± std)				
Top-level declarations	2277 ± 247	5 ± 2	6283 ± 1761	301 ± 85
Typedefs	249 ± 29	1 ± 0.1	254 ± 76	29 ± 3

in many C files, we separately report results including and excluding header files. We additionally report variability in local types defined through the `typedef` specifier.

For RQ10, we pursue a different strategy. We implemented two simple analyses using the type system to detect integer issues in security-relevant locations (e.g., in pointer arithmetic and memory-allocation functions). These analyses roughly implemented the the safe integer standard previously described. Ranging over all configurations, the analysis reports a constraint for each warning, describing the configurations in which the potential overflow occurs, e.g., in all configurations with macro `CONFIG_SLUB` activated. For research question RQ10, we investigated how the reported warnings are distributed over multiple configurations.

5.4 Results

RQ9: How many integer types in a program are defined differently depending on compile-time options?

Key results: A significant number of variables has alternative integer types depending on the configuration.

Table 6 shows that a small percentage but overall a *significant number* of symbols has alternative integer types depending on the configuration in both software. In Linux the number of symbols with alternative integer types is higher, due to the explicit support for 32 and 64 bit versions of Linux. In Linux, several frequently used types, such as `size_t`, defined through *typedef* declarations can have different underlying integer types. The table shows that while a large amount of variability is found in header files, the bodies of .c files also contain several instances of differing integer types for the same variable in both projects.

The following simplified excerpt illustrates part of the variability in the inline function `dma_capable` which, due to a header file, is found in almost every C file in Linux. The used type `dma_addr_t` is either defined as **`unsigned int`** or **`unsigned long long int`** and similarly, `size_t` has different integer types for different configurations.

```
//arch/x86/include/asm/types.h:
#if defined(CONFIG_X86_64) || defined(CONFIG_HIGHMEM64G)
typedef unsigned long long dma_addr_t;
#else
typedef unsigned int dma_addr_t;
#endif

//arch/x86/include/asm/dma-mapping.h
static inline bool dma_capable(struct device *dev, dma_addr_t
     addr, size_t size) {...}
```

Table 7: Distribution of warnings and statements for integer issues in the configuration space.

I.D.	Busybox			Linux kernel		
	Over.	**Coer.**	**Stmt.**	**Over.**	**Coer.**	**Stmt.**
0	269	795	14,147	442	9,307	454,071
1	1,909	5,603	98,523	32,394	242,604	10,383,952
2	315	1,226	22,669	101,357	520,654	24,415,587
3	59	271	3,746	161,983	471,863	29,263,459
4	0	31	537	122,185	326,362	29,286,254
5	0	0	48	72,076	182,266	23,821,018
6	0	2	40	40,935	99,919	15,972,229
7	0	0	0	17,507	76,866	8,620,447
8	0	0	0	5,502	40,403	3,194,076
9	0	0	0	2,007	8,586	826,592
10	0	0	0	391	492	195,427
11	0	0	0	8	20	58,988
12	0	0	0	10	0	14,574
13	0	0	0	0	0	2,553
14	0	0	0	0	0	241
15	0	0	0	0	0	3

I.D.: interaction degree; Over.: reported potential Integer overflow warnings (INT30-C and INT32-C); Coer.: reported implicit coercion warning (INT31-C); Stmt.: statements

RQ10: How are integer issues distributed in the configuration space (the statements that vary in configurations)?

Key results: Integer issues are distributed throughout the configuration space. Analyzing a single configuration provides insights to that configuration, but does not generalize well to other configurations.

The metrics in Table 6 show that the (large number of) reported potential issues in a configurationare related to the number of statements in the configuration. In Busybox, due to the build system and #ifdef directives, most code fragments are only compiled if at least one corresponding configuration option is selected (or deselected), but there are other statements with potential vulnerabilities that require more specific combinations of selected configuration options. In Linux, we see similar results at a larger scale; in fact many statements are included only if at least 2–7 configuration options are selected or deselected, and we find a large number of issues occur in those code fragments. New issues can still be found in more complex configurations. Two reported issues required setting 11 configuration options. At such high interaction degrees, it is unlikely that conventional combinatorial sampling strategies are suitable for finding all the problems.

6. THREATS TO VALIDITY

There are several threats to validity of our study; here we discuss them following the four classic tests and discuss how they have been mitigated.

Construct Validity: There may be a concern about the heuristics we used and the interpretation of our data. We considered a deviation from safe C integer model to be an integer issue. Very few of these ($\ll 1\,\%$) may actually lead to

security vulnerabilities. However, the large number of issues may create more opportunities for an attacker. Also, safety critical systems should not have any of these issues. They require stricter standards to be be enforced upon programmers (MISRA-C for motor vehicle systems). In addition, while considering the deviations, there may have been other reasons to change the variable's type other than how the variable was used; there may also be deliberate uses of overflow or underflow in programs [17]. However, we can not determine the developers' intent from the static analysis. Our study did not consider these, for the fact that any guess as to what those reasons could be would be a pure speculation based on the information that we currently have.

External Validity: Generalizability of our interpretation may be an additional concern to our study. We studied 52 different versions of 7 different programs and also two large and highly configurable programs (Busybox and the Linux kernel) to collect our results. The test programs are all mature C programs that have evolved over a long period. Yet, all of them showed similar trends in the kinds of integer issues present in them. Moreover, in Linux we had to exclude 10 files from the analysis due to technical difficulties with our infrastructure. However, the sample size was big enough to make this exclusion negligible for our results.

Internal Validity: Since it is an exploratory study on real data, we do not have any concern on internal validity.

Reliability: There may be concern on whether we have collected the results from correct release versions in our multi-version study. Perhaps there are sweeping changes correcting integer problems that happen in some versions that were not studied by us. There can be two measures to counter this: (1) include the latest release version of the software in the study, and (2) do not study consecutive versions of software, instead skip some versions so that the study covers a long period. To account for this, we studied the latest versions of each software in June 2013. We also made sure that the versions covered a large portion of the software's history. On average, from the first to the last version of a software that we studied, we covered the last 9 years and 7 months of the software's life span.

7. RELATED WORK

Researchers have conducted many empirical studies on how developers use language features. For example, there have been several studies on Java applications about various language features, e.g., inheritance in Java [47], overriding [46], and other object-oriented features [25]. However, there have been very few empirical studies on how C and C++ features are used. A few studies explored the complexities introduced by C preprocessor directives without analyzing how they affect the underlying types [19,33]. Chidamber and Kemerer [9] studied commercial C++ programs about object-oriented metrics such as use of inheritance.

A few research works focused on empirical data of integer issues. Brumley and colleagues [5] surveyed CVE database [38] and identified four types of integer problems—overflow, underflow, signedness, and truncation errors; their categories are similar to the original categories distinguished by blexim [4]. Dietz and colleagues [16] studied integer overflow and underflow vulnerabilities. They 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. Coker and Hafiz [13] introduced three program transformations that can be applied to C programs to fix all possible types of integer issues. They applied the program transformations on all possible targets of various real software [13, 22] and collected information about the mismatch in how an integer variable is declared. The focus of this study is broader: It looks into all types of integer issues and studies various aspects of misuses that are in source code.

Several guidelines define a safe integer model [37, 44]. There are also safe integer libraries such as the IntegerLib library [7] for C programs, and SafeInt [32], CLN [12], and GMP [23] libraries for C++ programs. 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 [15]. The integer model produces well-defined results for integer operations or else traps. However, none of these are used widely.

Most of the research on integer issues focus on detecting integer overflow vulnerabilities either statically or dynamically. Static analysis approaches can be done on both source code [2, 6, 10, 36, 43] and binary code [8, 11, 49, 50]. Most of these approaches are only applicable to detect integer overflows. Among the dynamic approaches, several tools can detect all types of integer issues, e.g., RICH [5], BRICK [8], and SmartFuzz [39]. On the other hand, SAGE [24] and IOC [17] target fewer integer issues. There are compiler based detection tools, such as GCC with `-ftrapv` option, which forces the compiler to insert additional calls (e.g., `_addvsi3`) before signed addition operations to catch overflows. The dynamic approaches, even the compiler extensions, introduce overhead (as high as 50X slowdown [8]).

Our analysis of the influence of configuration options builds on our prior infrastructure for parsing and type checking unpreprocessed C code [29,30,34], which was inspired by work on analyzing configurable systems [3, 14, 20, 48]. Prior work on analyzing preprocessor usage relied on heuristics to approximate the actual usage [1, 19, 21, 33, 42]. Our infrastructure scaled these ideas to real-world C code enabling accurate type analysis in the entire configuration space.

8. CONCLUSION

Our study highlights the common issues surrounding integer usage in C. We hope it educates programmers about common inconsistencies, so that they are aware of possible issues in their next coding session. Automated tools to fix these inconsistencies would definitely help their task. Our study also outlines the complexities that such tools will need to address when migrating legacy code to stricter integer standards.

Even a small C program has a lot of integers, a significant fraction of which may have inconsistent use and declarations. These discrepancies increase even more when multiple preprocessor configurations allow an integer variable to be declared and used as different types in different cases. Although the standard C specification permits such inconsistencies, and most of the issues may not even be potential vulnerabilities, a more secure integer standard helps to avoid them entirely. Even a few of these silent issues may turn out to be fatal errors or serious security vulnerabilities. It is better to be safe.

9. REFERENCES

[1] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan. Can we refactor conditional compilation into aspects? In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 243–254, New York, 2009. ACM Press.

[2] 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*, Washington, DC, USA, 2002. IEEE Computer Society.

[3] L. Aversano, M. D. Penta, and I. D. Baxter. Handling preprocessor-conditioned declarations. In *Proc. Int'l Workshop Source Code Analysis and Manipulation (SCAM)*, pages 83–92, Los Alamitos, CA, 2002. IEEE Computer Society.

[4] blexim. Basic integer overflows. *Phrack*, 60, 2002.

[5] 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.

[6] 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*, volume 4064 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2006.

[7] CERT. Integerlib library.

[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*, pages 208 –215, march 2009.

[9] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

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

[11] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[12] CLN: Class Library for Numbers. `http://www.ginac.de/CLN/`.

[13] Z. Coker and M. Hafiz. Program transformations to fix C integers. In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 792–801. IEEE / ACM, 2013.

[14] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220, New York, 2006. ACM Press.

[15] 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)*, pages 91 –100, nov. 2010.

[16] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 760–770, Piscataway, NJ, USA, 2012. IEEE Press.

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

[18] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 404–428, Berlin, Heidelberg, 2006. Springer-Verlag.

[19] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Trans. Softw. Eng.*, 28(12):1146–1170, Dec. 2002.

[20] M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(1):6:1–6:27, 2011.

[21] A. Garrido and R. Johnson. Analyzing multiple configurations of a C program. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 379–388, Washington, DC, 2005. IEEE Computer Society.

[22] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov. Systematic testing of refactoring engines on real software projects. In G. Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 629–653. Springer Berlin Heidelberg, 2013.

[23] GMP: Gnu Multiple Precision Arithmetic Library. `http://gmplib.org/`.

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

[25] T. Gorschek, E. D. Tempero, and L. Angelis. A large-scale empirical study of practitioners' use of object-oriented concepts. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010), Cape Town, South Africa, May, 2010*, pages 115–124. ACM, 2010.

[26] 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.

[27] D. Hodges. Why do Pinky and Inky have different behaviors when Pac-Man is facing up? `http://donhodges.com/pacman_pinky_explanation.htm`, December 2008.

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

[29] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824, New York, Oct. 2011. ACM Press.

[30] C. Kästner, K. Ostermann, and S. Erdweg. A variability-aware module system. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, New York, 2012. ACM Press.

[31] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.

[32] D. LeBlanc. Integer handling with the C++ SafeInt class. 2004.

[33] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114, New York, 2010. ACM Press.

[34] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, New York, NY, 8 2013.

[35] G. Malpohl, J. Hunt, and W. Tichy. Renaming detection. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 73–80, 2000.

[36] Microsoft Corporation. PREfast analysis tool.

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

[38] MITRE Corporation. Common vulnerabilities and exposures.

[39] D. Molnar, X. Li, , 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.

[40] National Vulnerability Database. CVE-2010-1513, 2010.

[41] National Vulnerability Database. CVE-2012-2110, 2012.

[42] Y. Padioleau. Parsing C/C++ code without pre-processing. In *Proc. Int'l Conf. Compiler Construction (CC)*, pages 109–125, Berlin/Heidelberg, 2009. Springer-Verlag.

[43] 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, 2007. ACTA Press.

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

[45] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. The variability model of the Linux kernel. In *Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 45–51, Essen, 2010. University of Duisburg-Essen.

[46] E. D. Tempero, S. Counsell, and J. Noble. An empirical study of overriding in open source Java. In B. Mans and M. Reynolds, editors, *Proceedings of the Thirty-Third Australasian Computer Science Conference (ACSC 2010), Brisbane, Australia, January, 2010*, volume 102 of *CRPIT*, pages 3–12. Australian Computer Society, 2010.

[47] E. D. Tempero, H. Y. Yang, and J. Noble. What programmers do with inheritance in Java. In G. Castagna, editor, *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP 2013), Montpellier, France, July, 2013.*, volume 7920 of *Lecture Notes in Computer Science*, pages 577–601. Springer, 2013.

[48] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104, New York, 2007. ACM Press.

[49] T. Wang, T. Wei, Z. Lin, and W. Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2009.

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