A Format String Checker for Java

Konstantin Weitz
University of Washington, USA
weitzkon@cs.uw.edu

Siwakorn Srisakaokul
University of Washington, USA
ping128@cs.uw.edu

Gene Kim
University of Washington, USA
genelkim@cs.uw.edu

Michael D. Ernst
University of Washington, USA
mernst@cs.uw.edu

ABSTRACT

Java supports format strings, but their use is error prone because: Java's type system does not find any but the most trivial mistakes, Java's format methods fail silently, and format methods are often executed infrequently.

This paper presents the Format String Checker that is based on the format string type system presented in [3]. The Format String Checker guarantees that calls to Java's Formatter API will not throw exceptions.

We evaluate the Format String Checker on 6 large and well-maintained open-source projects. Format string bugs are common in practice (we found 104 bugs), and the annotation burden on the user of our type system is low (on average, for every bug found, only 1.0 annotations need to be written).

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures*

General Terms

Experimentation, Languages, Reliability, Verification

Keywords

Format string, printf, type system, static analysis

1. INTRODUCTION

Format strings provide a convenient and easy to internationalize way to communicate text to the user. Java provides format string functionality with format methods such as `System.out.printf` and `String.format`.

Unfortunately, incorrect usage of format methods are frequent and hard to detect because:

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
ACM 978-1-4503-2645-2/14/07
http://dx.doi.org/10.1145/2610384.2628056

```
public class Main {                                          1
  public static void main(String[] args) {                  2
    System.out.printf("%s", "Hello");                        3
    System.out.printf("%d", "World");                        4
    System.out.printf("%s");                                5
    System.out.printf("%s", "Hello", "World");               6
    System.out.printf("%y", 7);                             7
                                                             8
    if (args.length >= 1) {                                  9
      String f = args[0];                                    10
      if (f == "%d") {                                       11
        System.out.printf(f, 7);                             12
        System.out.printf(f, 7.2);                           13
      }                                                      14
    }                                                        15
  }                                                          16
}                                                            17
```

Listing 1: Example program *Main.java* that illustrates the use of the Format String Checker.

- Java's type system does not find any but the most trivial mistakes.
- Java's format methods fail silently, for example if too many arguments are passed.
- Format methods are often used to report error messages. Hence, they appear in code that is infrequently executed.

In a previous paper [3], we presented the format string type system that promises to detect and prevent the incorrect use of format methods. The paper evaluates an implementation of the type system (Section 7), and compares it to the related work (Section 8).

In this paper, we take a more hands-on look at the Format String Checker. The Format String Checker is an implementation of our format string type system for Java's Formatter API, which is provided by the `Formatter` class.

The Format String Checker is implemented as a pluggable type system in the Checker Framework [2]. A pluggable type system extends Java's type system in a backward compatible way, to provide more guarantees about the absence of certain errors. The Format String Checker guarantees the absence of format method related errors.

The Checker Framework is implemented as an annotation processor for the Java compiler (*javac*). To run the Format String Checker with a project's usual build, the programmer simply adds the `-processor` command-line option to the invocation of the `javac` command.

The Format String Checker is shipped along with the Checker Framework. Both are open-source. Installation instructions, information on how to integrate them with a build system such as *maven* or *ant*, and more can be found at:

http://checkerframework.org

The Format String Checker was submitted for Artifact Evaluation and met the expectations of the committee.

2. THE FORMAT STRING CHECKER

2.1 Terminology

Consider the following code

```
String.format("The %s is %d.","answer",42);
```

which yields `"The answer is 42."`. `String.format` is a format method, `"The %s is %d."` is a format string, `"%s"` and `"%d"` are the format specifiers, and `"answer"` and `42` are format arguments.

2.2 Format Method Invocation Errors

The Format String Checker detects and prevents the incorrect use of Java's format methods. Listing 1 shows an example program that contains a number of incorrect uses of format methods. Executing the following command runs the Format String Checker on the example:

```
javac -processor
org.checkers.formatter.FormatterChecker Main.java
```

The Java compiler generates its usual output, along with errors that indicate the invalid use of format methods.

The first error issued by the Java compiler is:

```
Main.java:4: incompatible types in argument.
  System.out.printf("%d", "World");
                    ^
  found   : java.lang.String
  required: INT conversion category (one of:
      byte, short, int, long, BigInteger)
```

The error indicates that the type of the format argument passed to the format method is wrong. This invalid invocation will lead to a runtime exception if `main` is run. The error can be fixed by either replacing `"%d"` with `"%s"`, or passing a number instead of the `"World"` argument.

The next errors issued by the Java compiler are:

```
Main.java:5: missing arguments;
             expected 1 but 0 given
  System.out.printf("%s");
            ^
Main.java:6: too many arguments;
             expected 1 but 2 given
  System.out.printf("%s", "Hello", "World");
                                   ^
```

These errors indicate that the wrong number of format arguments was passed to the format methods. The first one will surely lead to a runtime error, the second error does not lead to a runtime error, but the output does not contain the `"World"` argument.

If passing unused arguments is the intended behavior, the programmer can use the following annotation to suppress the error.

```
@SuppressWarnings("formatter")
```

The next error issued by the Java compiler is:

```
Main.java:7: invalid format string
             "Conversion = 'y'"
  System.out.printf("%y", 7);
                    ^
```

The error indicates that an invalid format string was passed to a format method. In this case, the format string is invalid because `%y` is not a valid format specifier. Using `%d` instead fixes the error.

The last error issued by the Java compiler is:

```
Main.java:13: incompatible types in argument.
  System.out.printf(f, 7.2);
                    ^
  found   : double
  required: INT conversion category (one of:
      byte, short, int, long, BigInteger)
```

This error shows that checking is not only done for literal format strings. In this example, the Format String Checker uses a data-flow analysis to infer that `f` can only be `"%d"`, and thus that `7.2` cannot be used as a format argument.

2.3 Type Qualifiers

This section investigates how the Format String Checker was able to detect the invalid format method invocations from the previous section's example.

Consider a format method invocation in standard Java. Even with the Format String Checker disabled, the Java type system already provides certain guarantees about the invocation of a format method.

For example, because the format string argument of a format method is of type `String`, Java's type system guarantees that the format string is a string, and cannot be a list or number. The `String` type thus restricts the possible values that can be passed as the format method's format string argument.

The Format String Checker extends the type system with *type annotations*. A type annotation is attached to a standard Java type and further restricts the possible values of that type. The type annotation `@Format` attached to a `String` (written as `@Format String`), for example, restricts the values to be valid format strings[1].

The fundamental guarantee of the Format String Checker is the following: every format method is called with a format string that is annotated with the correct `@Format` type annotation. Section 2.5 provides more details.

If the format string passed to a format method is not annotated with a type annotation, the Format String Checker issues the following error:

```
invalid format string
(is a @Format annotation missing?)
  System.out.printf(format, args);
                    ^
```

This guarantee is surprising, given that none of the examples from Listing 1 contain any type annotations. The examples work because the Format String Checker uses a data-flow analysis to automatically infer the correct annotations in many cases.

[1]The JDK documentation for the Formatter class explains the requirements for a valid format string [1].

```
// code without annotations
String.format("%d", 42);
String f = "%d %f";
String.format(f, 5, 7.2);

// inferred annotations for the code above
String.format( (@Format({INT}) String) "%d", 42);
@Format({INT,FLOAT}) String f =
  (@Format({INT,FLOAT}) String) "%d %f";
String.format(f, 5, 7.2);
```

Listing 2: `@Format` annotation inference.

Listing 2 shows examples of inferred type annotations. Note that inferred type annotations are not inserted into the code, they are only inserted in the compiler's internal source code representation.

The type annotations from Listing 2 contain more structure than discussed so far. Every `@Format` annotation is equipped with a list of *conversion categories*.

We already mentioned that the possible values of the qualified `@Format String` type are restricted to be valid format strings. The list of conversion categories further restricts the possible values of the type. `@Format({INT,FLOAT})String`, for example, restricts the values to valid format strings with two format specifiers that respectively require "float-like" and "integer-like" format arguments.

2.4 Conversion Categories

The following list of frequently used conversion categories makes the notion of "xxx-like" precise.

GENERAL imposes no restrictions on a format argument's type. Applicable for format specifiers `%b`, `%B`, `%h`, `%H`, `%s`, and `%S`.

CHAR requires that a format argument represents a Unicode character. Specifically, `char`, `Character`, `byte`, `Byte`, `short`, and `Short` are allowed. `int` or `Integer` are allowed if `Character.isValidCodePoint(value)` would return `true` for the format argument (the Format String Checker currently permits any value of type `int` or `Integer` without issuing a warning or error — see Section 2.5). Applicable for format specifiers `%c`, and `%C`.

INT requires that a format argument represents an integral type. Specifically, `byte`, `Byte`, `short`, `Short`, `int`, `Integer`, `long`, `Long`, and `BigInteger` are allowed. Applicable for format specifiers `%d`, `%o`, `%x`, and `%X`.

FLOAT requires that a format argument represents a float-like type. Specifically, `float`, `Float`, `double`, `Double`, and `BigDecimal` are allowed, but integral types are not. Applicable for format specifiers `%e`, `%E`, `%f`, `%g`, `%G`, `%a`, and `%A`.

TIME requires that a format argument represents a date or time. Specifically, `long`, `Long`, `Calendar`, and `Date` are allowed. Applicable for format specifiers ending in `t` and `T`.

2.5 Guarantees

The Format String Checker guarantees that format methods never throw an exception at runtime, with a few caveats. This section explains the guarantees precisely.

```
public final void log (
  @FormatFor("args") String format,
  Object... args)
{
  logfile.printf(format, args);
}
```

Listing 3: A `@FormatFor` type annotation on the format parameter of a format method wrapper function.

The Format String Checker guarantees that format methods will never be called with an *invalid format string*. Thus, a format method never throws any of the following exceptions:

- `IllegalFormatException`
- `DuplicateFormatFlagsException`
- `FormatFlagsConversionMismatchException`
- `IllegalFormatConversionException`
- `IllegalFormatFlagsException`
- `IllegalFormatPrecisionException`
- `IllegalFormatWidthException`
- `MissingFormatArgumentException`
- `MissingFormatWidthException`
- `UnknownFormatConversionException`
- `UnknownFormatFlagsException`

The Format String Checker also guarantees that a format method will never be called with *missing format arguments* or *format arguments of the wrong type*. Thus, a format method never throws `MissingFormatArgumentException` or `IllegalFormatConversionException`.

We now discuss erroneous format string invocations that are outside the scope of the Format String Checker. This means that a format method call may fail, despite the fact that the Format String Checker issues no warning.

- The only exception that is directly thrown by Java's format methods, other than those listed above, is the `IllegalFormatCodePointException` exception. It is thrown if a conversion category is `CHAR`, and the type of the respective format argument α is `int` or `Integer`, and if `Character.isValidCodePoint(`α`)` would return `false`.
- If the format string is `null`, then a format method will throw a `NullPointerException`. Checking for null values is orthogonal to restricting the values of non-null format strings. The Checker Framework already ships with a Nullness Checker [2] that a programmer can run, in conjunction with the Format String Checker, to eliminate `NullPointerException`s.
- A method implemented by one of the format arguments may throw an exception. This can happen with a format argument's `toString` method, or if the format argument implements the `Formattable` interface and throws an exception in the `formatTo` method.

2.6 Polymorphism for Format Methods

We have shown how to write a method that takes as an argument a format string of a specific type. However, some methods are polymorphic with respect to their format string parameter: the method's parameters' types depend on the value of the format string.

Table 1: Case study overview. Code size is computed without blank lines or comments. *False Positives* is the number of warnings that were suppressed with a `@SuppressWarnings` annotation.

Project	Java Lines of Code	Format Method Call Sites	Annotations		False Positives	Bugs	
			`@Format`	`@FormatFor`		Submitted	Fixed
Apache Hadoop	678K	332	20	6	22	3	2
Apache Hive	538K	213	0	1	7	1	0
Apache Lucene	664K	148	2	0	0	0	0
Apache HBase	569K	96	0	0	1	2	2
Daikon	205K	1583	0	30	7	95	95
Findbugs	122K	133	7	1	3	3	3
Total	2777K	2505	29	38	40	104	102

Consider for example Listing 3. If the `log` method is called with the format string `"%d"`, then `args` must be an array of one "integer-like" value. If the format string is `"%f %f"`, then `args` must be an array of two "float-like" values.

Our type system provides the `@FormatFor` type annotation to express this situation. The `@FormatFor("x")` annotation specifies that the variable (or parameter) `x` is an array of format arguments that matches the format string of the annotated variable.

We annotated all 14 format methods in the JDK with the `@FormatFor` annotation. The format methods provided by the JDK are all the `format` and `printf` methods in the `Formatter`, `String`, `PrintStream`, `PrintWriter`, and `Console` classes.

2.7 Run-time Checks

Strings obtained from an external source (such as stdin), have to be tested at runtime before they can be passed into a format method. The validity of a format string can be tested using the `hasFormat(String, ConversionCategory...)` method. The method returns true if the argument is a syntactically valid format string with format specifiers that match the passed conversion categories and false otherwise.

If the conversion categories are passed to `hasFormat` as literals, the Format String Checker flow-sensitively infers, at compile time, a `@Format` annotation for the string in the true branch of any test against the result (and, more generally, in all code that is reachable from the true branch but not reachable from the false branch).

Listing 4 illustrates the use of `hasFormat` to catch an invalid format string at the time when it is read, instead of when it is potentially used.

```
// Bad version, throws an exception if
// an invalid format string is used
Scanner s = new Scanner(System.in);
System.out.printf(s.next(), "hello", 42);

// Improved version, reports an error when
// an invalid format string is read
Scanner s = new Scanner(System.in);
String f = s.next()
if (!hasFormat(f, GENERAL, INT)) {
  // ... good error reporting here ...
  System.exit(2);
}
// f is now known to be of type:
// @Format({GENERAL, INT}) String
System.out.printf(f, "hello", 42);
```

Listing 4: The `hasFormat` method serves as a way to dynamically check whether a string is valid.

3. EVALUATION

We evaluated the Format String Checker on 6 large and well-maintained open-source projects — namely Apache Hadoop, Apache Hive, Apache Lucene, Apache HBase, Daikon, and FindBugs.

Running the Format String Checker revealed 104 previously unknown bugs, as summarized in Table 1. We reported all of these bugs. The developers fixed 102 bugs, won't fix 1 bug as it is part of deprecated code, and have not yet commented on 1 bug.

Table 1 also indicates the effort required to use the Format String Checker.

Except for bug fixes, the only code changes introduced by us were additional annotations (the sum of *Annotations* and *False Positives*).

The ratio of code changes to format method call sites is only 0.04. This is due to the fact that literals are annotated automatically, and the use of data-flow analysis to automatically infer the correct `@Format` annotations for other format strings.

The ratio of code changes to bugs is very favorable at 1.0. This means that for every annotation written, the programmer is on average rewarded by finding 1.0 new unknown bugs.

Daikon contained more bugs than any of the other projects. This is partially due to the fact that Daikon uses by far the most format methods of any of the other projects, making it much more likely that a format method invocation is faulty. Daikon is also the project that produces the largest and most diverse command line output. These two facts may be correlated. Finally, Daikon has the smallest number of users and developers — it may be the least mature and robust.

4. REFERENCES

[1] Java Formatter class documentation. `http://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html`.

[2] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 22–24, 2008.

[3] K. Weitz, G. Kim, S. Srisakaokul, and M. D. Ernst. A type system for format strings. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014.