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:

- 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.

public class Main {
    public static void main(String[] args) {
        System.out.printf("%s", "Hello");
        System.out.printf("%d", "World");
        System.out.printf("%s", "Hello", "World");
        System.out.printf("%y", 7);
        String f = args[0];
        if (f == "%d") {
            System.out.printf(f, 7);
            System.out.printf(f, 7.2);
        }
    }
}

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

```java
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");
     ^
```

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.

```java
@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.

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.

1The JDK documentation for the Formatter class explains the requirements for a valid format string [1].
Listing 2: @Format annotation inference.

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:

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

The Format String Checker also guarantees that 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 IllegalFormatException exception. It is thrown if a conversion category is CHAR and the type of the respective format argument is int or Integer and Character.isValidCodePoint(a) 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 use, in conjunction with the Format String Checker, to eliminate NullPointerExceptions.
- 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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Java Lines of Code</th>
<th>Format Method Call Sites</th>
<th>Annotations</th>
<th>False Positives</th>
<th>Submitted</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Hadoop</td>
<td>678K</td>
<td>332</td>
<td>20</td>
<td>6</td>
<td>22</td>
<td>3</td>
</tr>
<tr>
<td>Apache Hive</td>
<td>538K</td>
<td>213</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Apache Lucene</td>
<td>664K</td>
<td>148</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Apache HBase</td>
<td>569K</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Daikon</td>
<td>205K</td>
<td>1583</td>
<td>0</td>
<td>30</td>
<td>7</td>
<td>95</td>
</tr>
<tr>
<td>Findbugs</td>
<td>122K</td>
<td>133</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Total</td>
<td>2777K</td>
<td>2506</td>
<td>29</td>
<td>38</td>
<td>40</td>
<td>104</td>
</tr>
</tbody>
</table>

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 it is read, instead of when it is potentially used.

4. REFERENCES