A Type System for Format Strings

Konstantin Weitz Gene Kim Siwakorn Srisakaokul Michael D. Ernst
University of Washington, USA
{weitzkon,genelkim,ping128,mernst}@cs.uw.edu

ABSTRACT

Most programming languages support format strings, but their use is error-prone. Using the wrong format string syntax, or passing the wrong number or type of arguments, leads to unintelligible text output, program crashes, or security vulnerabilities.

This paper presents a type system that guarantees that calls to format string APIs will never fail. In Java, this means that the API will not throw exceptions. In C, this means that the API will not return negative values, corrupt memory, etc.

We instantiated this type system for Java's Formatter API, and evaluated it on 6 large and well-maintained open-source projects. Format string bugs are common in practice (our type system 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. Most programming languages therefore provide at least one format string API. For example, Java provides format routines such as `System.out.printf` and `String.format`.

A format routine's specification requires that:

- The format string's syntax is valid.
- The correct number of arguments is passed.
- Each argument has the appropriate type.

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
ACM 978-1-4503-2645-2/14/07
http://dx.doi.org/10.1145/2610384.2610417

```
// Untested code (Hadoop)
Resource r = ...
format("Insufficient memory %d", r);

// Unchecked input (FindBugs)
String urlRewriteFormat = read();
format(urlRewriteFormat, url);

// User unaware log is a format routine (Daikon)
log("Exception " + e);

// Invalid syntax for Formatter API (ping-gcal)
format("Unable to reach {0}", server);
```

Listing 1: Real-world code examples of common programmer mistakes that lead to format routine call failures. Section 7.2 explains these common programmer mistakes.

Format string APIs are often used incorrectly. Listing 1 shows some common programmer mistakes. Each kind of programmer mistake can violate multiple requirements of the format routine specification.

These violations of the format routine specification are often hard to detect, because:

- The programming language's type system does not find any but the most trivial mistakes. Most type systems detect if a number is used where a format string is expected, but fail to detect more complex mistakes, such as missing arguments.
- The format string API fails silently, for example if too many arguments are passed.
- Format string APIs are often used to report error messages. Hence, they appear in code that is infrequently executed.

The implications of using a format string API incorrectly range from unintelligible text output (because information is missing or scrambled), to program crashes, to security vulnerabilities (for example in *wu-ftpd* [7]).

Previous work addresses the problem of format string bugs by lexical analysis of source code [10], static tracking of tainted input [29], checks for literal format strings [15, 25], using a dependent type system [17], dynamically checking certain safety properties of format routine calls [6, 28, 34], or introducing alternative formatting APIs that can be checked in standard type systems [4, 9, 18, 19]. As discussed in Section 8, these approaches either cannot guarantee that a format routine call will never fail at run time, are intractable to understand or implement, or do not support internationalization.

Therefore, we have developed a type system that guarantees that format routine calls never fail at run time. Our type system exposed 104 bugs in 6 open-source projects.

A string-based DSL can be very readable and expressive, because its syntax is unconstrained by the host language and can be tuned to the domain. And, it requires no host language changes. The same lack of constraints makes use of the DSL error-prone, because the host language gives no support to help programmers use the DSL correctly. Therefore, some people discourage use of string-based DSLs. Our work shows that you can have your cake and eat it too: uses of a string-based DSL for format routines can be statically verified in a mainstream programming language. We hope this success will inspire verification of other DSLs and a change in attitudes about their use.

Our type system could be useful not only for string-based DSLs, but also for other APIs such as C's `syscall` function. The syscall function's first argument is a "tag value" that chooses the syscall to make and decides which arguments are valid, such as:

```
tid = syscall(SYS_gettid);
tid = syscall(SYS_tgkill, getpid(), tid, SIGHUP);
```

1.1 Format String Type System Overview

We propose a qualifier-based type system that guarantees that format routine calls never fail at run time. Our format string type system introduces two main qualifiers: `Format` and `InvalidFormat`.

The `Format` qualifier, attached to a `String` type, represents syntactically *valid* format strings, such as `"%s %d"` in C's printf API, `"%[1]d"` in Go's fmt API, and `"{0}"` in C#'s String.Format API.

The `InvalidFormat` qualifier, attached to a `String` type, represents syntactically *invalid* format strings.

The `Format` qualifier is parameterized over the expected number and type of arguments passed into a format routine along with the format string. For example, in the Java Formatter API, the format string `"%d %c"` requires two arguments. The first one needs to be "integer-like" and the second one needs to be "character-like". Consider the following example:

```
@Format({INT,CHAR}) String fmt = "%d %c";
System.out.printf(fmt, 5, 'c'); // Ok
// Compile-time error: invalid format string:
System.out.printf("%y", 5, 'c');
// Compile-time error: too few arguments:
System.out.printf(fmt, 5);
// Compile-time error: argument of wrong type:
System.out.printf(fmt, 5, "hello");
```

1.2 Contributions

This paper makes the following contributions:

- A qualifier-based type system that guarantees that format routine calls never fail at run time.
- A publicly-available instantiation and implementation of the type system for Java's Formatter API, called the Format String Checker[1], available at `http://checkerframework.org`.
- An evaluation on 6 open-source projects. The evaluation found 104 bugs. It also shows that the overhead of using the Format String Checker is low (on average, for every bug found, only 1.0 annotations need to be written).

The rest of this paper is organized as follows. Section 2 provides background and terminology about format string APIs. Section 3 presents the format string type system that guarantees that format string calls never fail at run time. Section 4 instantiates the format string type system for Java's Formatter API. Section 5 presents an implementation of this instantiation, called the Format String Checker. Section 6 instantiates the format string type system for Java's i18n format string API. Section 7 presents the results of applying the Format String Checker to 6 open-source projects. Section 8 reviews related work, and Section 9 concludes.

[1]The Format String Checker met the expectations of the ISSTA Artifact Evaluation Committee.

2. BACKGROUND

Many format string APIs share common concepts. This chapter introduces the common concepts of most format string APIs, including the APIs provided by C, Java, C#, and Go.

Consider the following format string usage in C:

```
printf("%s %d", "str", 42);
```

`printf` is the procedure for string formatting that is provided by C's standard library. `printf`-like functions are called *format routines*. A format routine takes, as an argument, a *format string* and a list of *format arguments*. In this example, `"%s %d"` is the format string, and `"str"` and `42` are the format arguments.

The format string contains *format specifiers*. In C, these are introduced with the `%` character. In this example, `%s` and `%d` are the format specifiers.

The call to the format routine produces a new string. The produced string is the format string where each format specifier has been replaced by the corresponding format argument. In our example, the result is thus `"str 42"`. Depending on the format routine, this string is either returned directly or forwarded to an output stream.

The format specifier not only determines how the output is formatted, but also determines the legal types for its corresponding format argument. In C, `%s` requires that the corresponding format argument be a pointer to a null-terminated `char` array.

In some format string APIs, the format specifier can select a specific format argument. In Java, by default, the format arguments are consumed left to right. But if the programmer inserts `n$` into a format specifier, the positive integer `n` is used as a one-based index into the format argument list.

In the following Java example, the `2$` component of the first format specifier specifies that the second argument is used (instead of the first). The result is thus `"42 str"`.

```
String.format("%2$d %1$s", "str", 42);
```

C# supports the same feature, with a different syntax:

```
String.Format("{1} {0}", "str", 42);
```

In this case, `{1}` and `{0}` are the format specifiers, and `1` and `0` select the format argument.

Format string APIs differ in the syntax used for format strings and in the available format specifiers.

3. TYPE SYSTEM

This section presents the format string type system. It guarantees that format routine calls never fail at run time.

The type system can be instantiated for a specific format string API by providing three parameters: conversion categories (Section 3.3), a subset relation among them (Section 3.4), and type introduction rules for literals (Section 3.5). Section 4 instantiates the format string type system for Java's Formatter API, and Section 6 instantiates the format string type system for Java's i18n API.

For the sake of clarity, the type system is introduced with concrete examples from its instantiation for Java's Formatter API.

3.1 Qualifier-Based Type Systems

The format string type system is a qualifier-based type system [13]. In a qualifier-based type system, a type qualifier is attached to every occurrence of a type in the language. If a type is interpreted as

a collection of values, the type qualifier is a restriction that removes certain values from the collection.

Integrating a qualifier-based type system into an existing language requires three changes to the language's type system.

Firstly, a *type qualifier* must be attached to every occurrence of a type in the language definition.

Secondly, the language's *subsumption* rule must be extended with a new premise that checks that the qualifiers are in a subtype relationship $\leq_q$. The language's existing subtyping rules $\leq$ stay unchanged.

$$\frac{\Gamma \vdash t : Q'T' \qquad Q' \leq_q Q \qquad T' \leq T}{\Gamma \vdash t : QT}$$

Finally, existing *introduction* rules for literals must be extended to infer the correct qualifier, using a function *qualifier* that maps literals to type qualifiers.

$$\frac{Q = qualifier(l)}{\Gamma \vdash l : QT}$$

3.2 Type Qualifiers

Our type system provides four type qualifiers:

- The `Format` qualifier, attached to a `String` type, stands for the collection of format strings that are syntactically *valid*. To allow verifying that the format arguments match the format specifiers of the format string, the `Format` qualifier is polymorphic, and must be parameterized with a list of conversion categories. Conversion categories are discussed in Section 3.3.
- The `InvalidFormat` qualifier, attached to a `String` type, stands for the collection of format strings that are not syntactically valid.
- The `Unknown` qualifier imposes no restriction on the type. Its values are the union of `Format` and `InvalidFormat` values.
- The `FormatBottom` qualifier imposes the restrictions of both `Format` and `InvalidFormat`. In Java, its only value is `null`.

For simplicity, this paper does not discuss how the qualifiers are applied to non-string types. The full type system and the implementation do handle those cases.

3.3 Conversion Categories

Section 3.2 introduced the `Format` qualifier. It restricts the values of the attached String type to syntactically valid format strings.

But syntactic validity of the format string is not enough to guarantee that a format routine call never fails. To prevent that the wrong number or type of arguments is passed, the format specifiers of the format string must also match the format arguments of the call.

The `Format` qualifier is polymorphic — it is parameterized over the expected number and type of arguments passed into a format routine along with the format string.

Listing 2 shows how the `Format` qualifier could be used on code found in FindBugs. The `Format(INT,INT,GENERAL)` qualifier requires that the first two format arguments are "integer-like", and that the first two format specifiers can deal with any "integer-like" arguments without failure (there are no restrictions on the last argument).

Conversion categories make the notion of "integer-like" precise. A conversion category is a set of permissible format argument types.

The conversion categories differ substantially between format string APIs. The type system is parameterized over conversion categories. Therefore, each instantiation of the type system for a specific format string API must define its own conversion categories.

```
void printBoard(PrintWriter w,
  @Format({INT,INT,GENERAL}) String format)
{
  int pos =    // ...
  int num =    // ...
  String key = // ...

  w.printf(format, pos, num, key);
}

printBoard(w, "%d %d %s"); // Ok
printBoard(w, "%d %d");    // Bad
```

Listing 2: A method definition from FindBugs (paraphrased for brevity). We have added a `Format` qualifier to explicitly state the contract of method `printBoard`: the String parameter must be a format string that can be used in a format routine call, where the first two format arguments are "integer-like" and the last argument has no restrictions.

```
@Format({FLOAT, INT}) String f;

f = "%f %d";     // Ok
f = "%s %d";     // Ok, %s is weaker than %f
f = "%f";        // Ok, last argument is ignored
f = "%f %d %s"; // Error, too many arguments
f = "%c %d";     // Error, %c not weaker than %f

// Ok, because f's type is
// consistent with 0.8 and 42
String.format(f, 0.8, 42);
```

Listing 3: Examples showcasing the subtyping rules.

The conversion category UNUSED is required if a format argument is not used as the replacement for any format specifier. For example, in Java `"%2$s"` ignores the first format argument. UNUSED has to be provided by all instantiations of our type system.

3.4 Subtyping

Subtyping among the type qualifiers is required by the subsumption rule of Section 3.1. The subtyping rules among type qualifiers are expressed in Figures 1 and 2, and the examples in Listing 3 show the subtyping rules in action.

The fifth subtyping rule reflects the fact that:

- If a format routine call succeeds with a certain format string, it will also succeed if one of the format specifiers in the format string has been replaced by a format specifier with weaker restrictions. In the format routine call `format("%d", 5)`, `%d` can be replaced with `%s` and the call will still succeed.
- Format string APIs allow the programmer to pass more arguments to a format routine than are actually required by the format string (e. g. `format("%d",1,1)` is legal).

The last two subtyping rules combined capture the fact that if the last conversion category is `UNUSED`, it is the same as omitting that conversion category.

Note that the subtyping rules require that a subset relation is defined among the conversion categories. The type system is parameterized over this subset relation. Therefore, each instantiation of the type system must define its own subset relation.

$$\boxed{S \leq_q T}$$

$$\frac{}{\text{InvalidFormat} \leq_q \text{Unknown}}$$

$$\frac{}{\text{Format}(s_0,\ldots,s_n) \leq_q \text{Unknown}}$$

$$\frac{}{\text{FormatBottom} \leq_q \text{Format}(s_0,\ldots,s_n)}$$

$$\frac{}{\text{FormatBottom} \leq_q \text{InvalidFormat}}$$

$$\frac{\forall i \in 0..n,\ t_i \subseteq s_i}{\text{Format}(s_0,\ldots,s_n) \leq_q \text{Format}(t_0,\ldots,t_n,\ldots,t_{n+k})}$$

$$\frac{}{\text{Format}(s_0,\ldots,s_n,\text{UNUSED}) \leq_q \text{Format}(s_0,\ldots,s_n)}$$

Figure 1: Subtyping rules among type qualifiers.

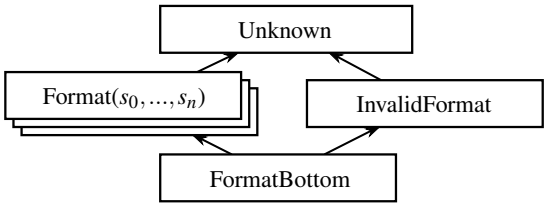

Figure 2: Part of the format string type system's qualifier hierarchy (Figure 1), depicted pictorially.

3.5 Qualifier Introduction Rules

Format string APIs differ in the syntax used for format strings. An instantiation of the type system must therefore provide an implementation of the *qualifier* function of Section 3.1, which infers the correct qualifier for literals.

3.6 Polymorphism

We have shown how to write a routine that takes as an argument a format string of a specific type. However, some routines are polymorphic with respect to their format string parameter: the routine's parameter types depend on the value of the format string. This can be viewed as type polymorphism or as a dependent type.

Consider for example Listing 4. 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 qualifier to express this situation. The `FormatFor(x)` qualifier specifies that the variable or parameter `x` is an array of format arguments that matches the format string of the qualified variable.

The `FormatFor` qualifier is useful for expressing the types of format routines not only in the standard library, but also for programs that define their own *format routine wrappers*. Listing 4 is an example of a format routine wrapper found in Daikon.

3.7 Security

Misuse of format string APIs can cause security vulnerabilities. In C, this was first noticed with the exploit of a format string bug in *wu-ftpd* [7].

The most severe attacks on C's format string API take advantage of the `%n` format specifier. It writes the length of the string produced by the format routine so far, to the location pointed to by the corresponding format argument. Note how this is in contrast to all other format specifiers, which simply read the format argument. Assume that an attacker has control over the format string. By simply changing the length of the format string, the attacker can control the value that is written to `%n`'s format argument.

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

log("%d", 42);            // Ok
log("%f %f", 1.2, 3.4);   // Ok
log("%d", "str");// Compile-time error: parameter
                 // and argument are incompatible
```

Listing 4: A `FormatFor` type qualifier on the format parameter of a format routine wrapper. The routine is taken from the Daikon project.

If `%n`'s format argument points to the function's return address, the attacker is able to make the program jump to code at an arbitrary location, often a shell.

For this attack to succeed in practice, the attacker must not only be able to provide the format string to a format routine, but must also be able to provide an incorrect number of format specifiers (to "search" the heap for interesting values to overwrite), and format specifiers of the wrong type (because most format routine calls are not intended to use `%n`). Tsai and Singh provide a more detailed description of the attack [34].

Our type system restricts the format strings that can be passed into format routines to valid format strings. This has two advantages:

- Even if the attacker can provide the format string, the string must have the correct number and type of format specifiers, which makes our type system effective in stopping format string attacks [6].
- All user-provided strings can be invalid format strings. The format string type system therefore prevents a programmer from passing unverified user-provided strings as the format string to a format routine like `syslog`. Section 5.5 discusses how a programmer can validate user-provided input before passing it to a format routine.

4. FORMATTER API INSTANTIATION

This section instantiates the format string type system for Java's Formatter format string API, which is provided by the `Formatter` class [21].

The instantiation provides the three parameters of the format string type system: conversion categories (Section 4.1), a subset relation among conversion categories (Section 4.2). and the *qualifier* function (Section 4.3).

4.1 Conversion Categories

The instantiation for Java's Formatter API provides the following conversion categories:

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.is-`

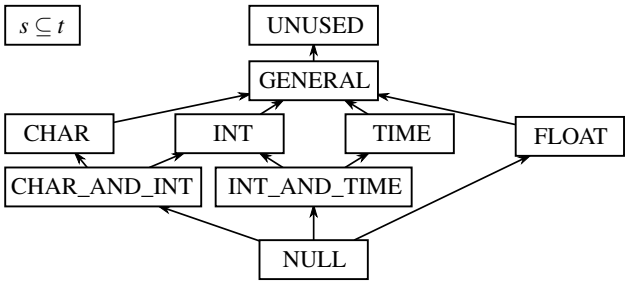

Figure 3: The subset relation among the conversion categories, for the instantiation of the format string type system for Java's Formatter API.

`ValidCodePoint(value)` would return `true` for the format argument. 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.

UNUSED imposes no restrictions on a format argument.

In the Formatter API, the same format argument may serve as a replacement for multiple format specifiers. In this case, the argument is restricted by the *intersection* of the associated conversion categories. Therefore, every intersection of conversion categories must be added as a new conversion category.

An example format string that requires conversion category intersection is `"%1$c %1$d"`. The first argument is restricted by both CHAR and INT.

Only three additional conversion categories are required to represent all possible intersections of the conversion categories that were previously mentioned:

CHAR_AND_INT = CHAR ∩ INT is used if a format argument is restricted by a `CHAR` and an `INT` conversion category. Specifically, `byte`, `Byte`, `short`, `Short` are allowed. `int` and `Integer` are allowed with restrictions.

INT_AND_TIME = INT ∩ TIME is used if a format argument is restricted by an `INT` and a `TIME` conversion category. Specifically, `long` and `Long` are allowed.

NULL is used if no value of any type, except `null`, can be passed as an argument. As `null` is a member of all non-primitive types in Java, it is allowed by all conversion categories.

All other intersections lead to already-existing conversion categories. For example GENERAL ∩ CHAR = CHAR and CHAR_AND_INT ∩ INT_AND_TIME = NULL.

4.2 Subtyping

The instantiation for Java's Formatter API provides the subset relation among all conversion categories. Figure 3 shows the subset relation.

4.3 Qualifier Introduction Rules

The instantiation for Java's Formatter API provides the *qualifier* function for literals. Given a literal, the *qualifier* function returns a type annotation. If the literal is:

- a valid format string, the function returns a `@Format` annotation with the appropriate conversion categories, for example *qualifier*(`"%s %d"`) =`@Format({GENERAL,INT})`.
- an invalid format string, the *qualifier* function returns an `@InvalidFormat` type annotation.
- the `null` literal, the function returns `@FormatBottom`.

In Java, `null` is an element of every non-primitive type, and it must also be an element of every type qualifier. Consider the following code:

```
<T> T f() {
  return null;
}
```

This type checks in Java, because no matter how the caller of `f` instantiates `T`, `null` is an element of `T`.

4.4 Implicit Casts

The following call to Java's format routine throws an exception at run-time:

```
printf("%f", 2);
```

This is surprising because in most contexts the Java compiler implicitly casts int to float or double when necessary. For example,

```
Math.sqrt(2);
```

can be compiled and executed, even though `sqrt` is only defined for doubles. In the case of `printf`, the compiler cannot insert an implicit cast from float to int, because the compiler lacks information about the expected type of the format arguments.

Exploiting the information from our type system, the compiler could infer the expected type of format arguments, and insert appropriate implicit casts. We have not yet implemented this compiler extension.

4.5 Guarantees

The instantiation for Java's Formatter API guarantees that a format routine never throws an exception at run time, with a few caveats. This section explains the guarantees precisely.

The instantiation guarantees that a format routine will never be called with an *invalid format string*. Thus, a format routine never throws any of the following exceptions:

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

The instantiation also guarantees that a format routine will never be called with *missing format arguments*, and thus never throws `MissingFormatArgumentException`.

The instantiation also guarantees that a format routine will never be called with *format arguments of the wrong type*, and thus never throws `IllegalFormatConversionException`.

We now discuss erroneous format routine calls in Java that are outside the scope of the type system. This means that a well-typed format routine call may throw an exception.

```
// IllegalFormatCodePointException
String.format("%c", (int)-1);

// NullPointerException
String.format(null);

// Callback error
class A {
  public String toString() {
    throw new Error();
  }
}
String.format("%s", new A());
```

Listing 5: Errors in format routine usage that are outside the scope of our format string type system. Our implementation issues no errors or warnings for these invocations of format routines.

- The only other exception directly thrown by Java's format routines, other than those listed above, is the `IllegalFormatCodePointException` exception. It is thrown if a conversion category is `CHAR_AND_INT` or `CHAR`, and the type of the respective format argument *a* is `int` or `Integer`, and `Character.isValidCodePoint(a)` is `false`.[2] The type system does not track the potential values of integers, but it could make use of an external analysis that does so [5, 24, 33].
- If the format string is `null`, then a format routine will throw a `NullPointerException`. Checking for null values is orthogonal to restricting the values of non-null format strings. Again, the type system could make use of an external nullness analysis [24, 27, 31] to eliminate `NullPointerException`s.
- A callback 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.

Listing 5 illustrates these cases.

5. FORMATTER API IMPLEMENTATION

5.1 Checker Framework

The Format String Checker is the implementation of the instantiation of the format string type system for Java's Formatter API. The Format String Checker is a pluggable type system built using the Checker Framework [27]. 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-routine-related errors.

Java 8 has native support for type qualifiers, in the form of type annotations. 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 has no effect on run-time behavior: javac produces the same bytecodes whether or not javac is running the Format String Checker annotation processor.

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`.

[2] `isValidCodePoint` returns `false` if the parameter is not in the range of Unicode characters, namely [0x0..0x10ffff].

5.2 Optional Checks

The Format String Checker implements optional checks that issue a warning when format routines are used in a suspicious way. These code smells often indicate bugs, but the code never throws an exception at run time. With these optional checks enabled, the Format String Checker exhibits lint-like behavior. Two optional checks are implemented.

Unused Format Arguments.

With most format string APIs, it is legal to provide more arguments to the format routine than are required by the format string.

In C, the reason is that the `printf` implementation has no way to know how many arguments were passed to the function, because no special terminator appears at the end of the array. Thus, `printf` has no way to check that the number of arguments is correct at run time.

In Java, the format string API implementation could check whether too many arguments were passed, but it chooses not to do so. We speculate that this is in part to make it more similar to the C implementation, and in part to enable a variety of polymorphism, allowing certain format strings to ignore arguments. However, we found that unused format arguments often lead to silent failures, so the Format String Checker provides the option to issue a warning.

Format Arguments That Can Only Be null.

It is possible to write a format string for which the only possible argument value is `null`. An example is `"%1$d %1$f"`.

If this is the intention, it is better style to replace the format specifiers with `"null"`, as in `"null null"`.

5.3 Library Annotations

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

These annotations are stored in a separate file that the Format String Checker reads. This allows a user of the Format String Checker to use their own unmodified JDK, and to easily add annotations to other libraries that cannot be modified.

5.4 Implicit Annotations

The Format String Checker infers the type annotation of literals using the *qualifier* function, and it performs flow-sensitive intraprocedural type inference to propagate annotations within each method body.

Overall, a programmer mainly needs to write `@Format` and `@FormatFor` annotations on types of fields and method signatures.

5.5 Run-time Checks

If a program obtains a string from an external source (such as a configuration file), the program must test the string at run time before passing it into a format routine. Otherwise, there is no guarantee that the string is of the correct form, and the call may fail.

The Format String Checker provides the `hasFormat(String, ConversionCategory...)` method for this purpose. `hasFormat` returns true if the argument is a syntactically valid format string with format specifiers that match the passed conversion categories.

After a `hasFormat` test, the tested string is given a `@Format` type in all code that is reachable from the true branch but not reachable from the false branch.

```
// Bad version, throws an exception if
// an invalid format string is used.
Scanner s = new Scanner(System.in);
String f = s.next()
System.out.printf(f, "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 6: The `hasFormat` method dynamically checks whether a string is valid. The Format String Checker issues a warning about the first call to `printf`, but no warning about the second call.

Listing 6 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.

6. I18N API INSTANTIATION

This section instantiates the format string type system for Java's i18n (internationalization) format string API, which is provided by the `MessageFormat` class [22].

The instantiation provides the three parameters of the format string type system.

Conversion Categories.

The instantiation for Java's i18n API provides the following conversion categories:

GENERAL imposes no restrictions on a format argument's type. Applicable for the *implicit* format specifier, e. g. `{0}`.

DATE requires that a format argument represents a date or time. Specifically, instances of `Number` and `Date` are allowed. Applicable for format specifiers *date* and *time*.

NUMBER requires that a format argument represents a number. Specifically, instances of `Number` are allowed. Applicable for format specifiers *number* and *choice*.

UNUSED imposes no restrictions on a format argument.

The same format argument may serve as a replacement for multiple format specifiers. No additional conversion categories are required to represent the possible intersections of the conversion categories.

Subtyping.

The instantiation for Java's i18n API provides the *subset relation among all conversion categories*. Figure 4 shows the subset relation.

Qualifier Introduction Rules.

The *qualifier* function of the instantiation for Java's i18n API is identical to the *qualifier* function defined in Section 4.3, except that the `@Format` and `@InvalidFormat` annotations are inferred according to the i18n API's format string syntax. For example, *qualifier*(`"{0,date} {1}"`) = `@Format({DATE,GENERAL})`.

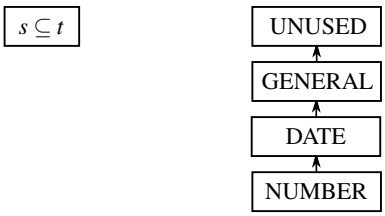

Figure 4: The subset relation among the conversion categories required by the instantiation of our type system for Java's i18n API.

Implementation.

We have not yet implemented the instantiation for Java's i18n API, but we plan to extend the Format String Checker to support the API, and automatically analyze translation files.

7. EVALUATION

7.1 Methodology

We evaluated the Format String Checker, an implementation of the instantiation of our type system for Java's Formatter API, on 6 open-source projects. *Apache Hadoop* [2] is a framework for distributed storage and processing of large-scale data. *Apache Hive* is a data warehouse infrastructure built on top of Hadoop that facilitates data summarization, querying, and analysis. *Apache Lucene* [3] is a library that provides indexing and search capabilities. *Apache HBase* is a non-relational distributed database modeled after Google's Bigtable. *Daikon* [8, 11] dynamically detects likely program invariants. *FindBugs* [12] uses static analysis to look for bugs in Java code.

For each project, we modified the build file to run the Format String Checker. We then executed the Format String Checker.

For every warning, we performed one of the following three actions. If the warning indicated a real bug, we reported it to the maintainers. If the warning indicated a missing annotation, we wrote `@Format` or `@FormatFor` in the source code. If the warning was a false positive, we wrote a `@SuppressWarnings("formatter")` annotation.

7.2 Bugs Revealed

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 because it is in deprecated code, and have not yet commented on 1 bug.

We categorized the root causes of the bugs according to the classification of Listing 1. When multiple root causes could be applicable we used our judgment to choose the best category. For example, in Hadoop `SwiftUtils.debug(LOG, filename)` might have been caused because the programmer was unaware of the format wrapper, or because he forgot to check inputs.

Untested Code.

Bugs classified as untested code would have been found had the format routine call ever been executed.

In one Hadoop bug, a format argument of the wrong type was passed into a format routine with a literal format string. No matter what values the arguments are, an execution of this format routine call always throws an exception. Despite this fact, the bug went unnoticed. The format routine call was inside an error handling block, that was apparently not tested.

This category contained 1 bug.

Table 1: Case study overview. Code size is computed without blank lines or comments. *Library Call Sites* is the number of call sites to library format routines (e.g. `String.format`). *Wrapper Call Sites* is the number of call sites to non-library methods annotated with `@FormatFor`. *Literal Call Sites* is the number of call sites of format routines with a literal format string. *False Positives* is the number of warnings that we suppressed with a `@SuppressWarnings` annotation. Table 2 categorizes false positives.

Project	Java Lines of Code	Format Routine Call Sites				Annotations		False Positives	Bugs	
		Total	Library	Wrapper	Literal	`@Format`	`@FormatFor`		Submitted	Fixed
Apache Hadoop	678K	332	283	49	251	20	6	22	3	2
Apache Hive	538K	213	213	0	197	0	1	7	1	0
Apache Lucene	664K	148	148	0	141	2	0	0	0	0
Apache HBase	569K	96	96	0	92	0	0	1	2	2
Daikon	205K	1583	930	653	1558	0	30	7	95	95
FindBugs	122K	133	130	3	119	7	1	3	3	3
Total	2777K	2505	1800	705	2358	29	38	40	104	102

Unchecked Inputs.

Passing an unchecked user-provided format string into a format routine can always fail, because the string may be syntactically invalid or may contain format specifiers that are not expected by the format routine. Format strings can be checked using the `hasFormat` method described in Section 5.5.

For example, a format string was read from a configuration file and passed unchecked into a format routine in FindBugs. If this format string was invalid, the format routine failed, terminating the tool's initialization thread. This in turn disabled some features of the graphical user interface, such as closing the window. There was no obvious connection between the non-working features and the configuration file.

This category contained 15 bugs.

Unawareness of Format Wrappers.

Sometimes, programmers are not aware that they are calling a format routine. While this is unlikely for well-known format routines like `printf`, it is quite common for routines that do more than just formatting, such as format routine wrappers and C's `syslog`.

Suppose that an arbitrary string is passed to a format routine, as in `printf(s)`. This invocation works, *unless* the string `s` contains (legal or illegal) format specifiers. More specifically, if the string contains `%` characters, then the invocation may fail at run time. The correct way to print one string is to call `printf("%s", s)`.

The Daikon program contained an example of this bug:

```
log(String.format("a=%s, b=%s, c=%s", a, b, c));
```

The programmer must not have been aware that Daikon's `log` method is a format routine wrapper, otherwise the programmer would have written:

```
log("a=%s, b=%s, c=%s", a, b, c);
```

This category contained 88 bugs.

Invalid Literal Syntax.

We found no invalid literal format strings in the six programs of our main evaluation. But, as a preliminary experiment, we ran a search on Ohloh [26] for Java's format routines. We downloaded the first 10,000 resulting source files and found 21 invalid literal format strings.

Bugs in Daikon.

Daikon contained more bugs than the other projects. This may be related to the fact that Daikon uses by far the most format routines of any of the projects (*Format Routine Call Sites* columns of Table 1), giving more opportunities for error. Daikon is also the project that produces the largest and most diverse textual output. These two facts may be correlated.

Table 2: False positive warnings in our case studies. The categories are described in Section 7.4.

Project	Constant Propagation	Dynamic Width		Exception Handled	Misc.
		any value	non-neg.		
Apache Hadoop	10	4	2	0	6
Apache Hive	3	0	2	1	1
Apache Lucene	0	0	0	0	0
Apache HBase	0	0	0	0	1
Daikon	0	0	6	0	1
FindBugs	0	0	0	3	0
Total	13	4	10	4	9

7.3 Usage Effort

The only code changes introduced by us were additional annotations (the sum of *Annotations* and *False Positives* in Table 1).

The ratio of annotations to bugs is very favorable at 1.0 (7.8 without Daikon). This means that for every annotation written, the programmer is on average rewarded by finding 1.0 new unknown bugs (0.13 without Daikon). The ratio of annotations to format routine call sites is only 0.04 (0.08 without Daikon).

Our methodology required us to run the Format String Checker, identify the right annotations to write in the cases where they were not already inferred automatically, and understand the false positives. In our evaluation, this was less time-consuming than learning how to build the evaluated projects.

7.4 False Positives

Like every conservative static analysis, the Format String Checker issues false positive warnings. In these cases, we manually inspected the code and verified that it cannot fail at run time.

Table 2 categorizes the false positives generated by our type system. We have designed analyses that would eliminate 31 of the 40 false positives. Once these are implemented, the Format String Checker's precision (bugs/(bugs+false positives)) would increase from 72% to 92%, the lines of code per false positive would increase from 70K to 308K, and the format routine call sites per false positive would increase from 63 to 278.

Compile-Time Constants.

This category contains all false positives in which the format string's value is a compile-time constant. This includes format strings built by concatenating literal strings. For example:

```
format("%"+"d", n);
```

These false positives could be eliminated by using constant propagation.

```
<T> void print(String format, Iterator<T> iter) {
  while (iter.hasNext()) {
    System.out.format(format, iter.next());
  }
}

List<Byte> l = ...
print("%X", l.listIterator());
```

Listing 7: False positive in Hadoop. The restrictions of the format string depend not on the main type qualifier on the function argument, but on the type qualifier on the type argument of the function argument.

Dynamic Width.

This category contains all false positives that are due to a format string being computed by concatenating compile-time constant strings with the printed representations of non-constant integers. This enables a format specifier's width or precision to be controlled at run time, as in the following example found in Hadoop:

```
format("%"+width+"f", n);
```

In cases like the one above, the computed string is a valid format string for all run-time values of the integer expression. These false positives could be eliminated by a special case in the Format String Checker.

In other cases, the computed string is a valid format string only if the integer expression has a non-negative value, because the format string already contains a `-` flag (indicating left-justification). Here is an example found in Hive:

```
format("%-"+width+"f", n);
```

These false positives could be eliminated by adding a type system that tracks the potential values of integers [5, 24, 33].

Exception Handled.

This category contains false positives where the Format String Checker correctly warns that a format routine call may throw an exception, but the exception is caught by a try-catch block and thus does not cause a user-visible failure.

The try-catch block exists for the purpose of error handling, like that illustrated for the `hasFormat` method in Section 5.5. In all the cases that we found in our evaluation, a user-provided format string was the reason why an exception may have been raised.

These false positives could be eliminated if the Format String Checker suppressed warnings in the bodies of try-catch constructs that catch and do not re-throw `IllegalFormatException`.

Miscellaneous.

This category contains all other false positives. This category primarily consists of complex format string computations. It also includes a case in Hadoop where failure of a format method call was *intended* to terminate the program with a stack trace, as a diagnostic for a sophisticated user such as a programmer.

Another example is a polymorphic format routine wrapper in Hadoop, illustrated by Listing 7. The restrictions on the format string depend on the polymorphic type of the function's argument. A conservative approximation, using the GENERAL conversion category, yields warnings, and in this case they were false positives.

7.5 Optional Checks

The optional checks of Section 5.2 issued no warnings for the six programs of our main evaluation. We did find 76 bugs in other programs from Ohloh that would have been revealed by the optional warnings. An example is provided in Listing 1.

8. RELATED WORK

FORTRAN was the first language to standardize a format string API, in 1966 [1]. C standardized `printf` in 1989 [20]. Modern languages continue to provide format string APIs. Java provides the `Formatter` and `MessageFormat` classes, PHP the `prinft` function, OCaml the `Printf` module, Haskell the `Printf` package, C# the `String.Format` method, Go the `fmt` package, and Rust the `format!` macro.

Format string mistakes are an important source of bugs. As a result, others have addressed the problem before us.

Lexical Analysis.

The PScan [10] tool runs a lexical analysis on C files and finds, among others, code lines that look as if they are calling the `printf` function without a literal format string.

The main advantage of such techniques is that they can be run on projects without the need to become familiar with the project's build process or library dependencies. One disadvantage is the lack of guarantees about the absence of errors. The lack of sophisticated reasoning also leads to many false positives.

Static Taint Tracking.

A value that was provided by an untrusted source (such as user input) is regarded as tainted. The solution by Shankar et al. [29] disallows tainted strings to be used as the format strings in format routines and format routine wrappers.

Because the approach uses only static analysis, there is no overhead at run time. The downside of this approach is that it only checks that the format string is not tainted. A format routine call may still fail with invalid syntax or the wrong number or type of arguments. This technique also hinders internationalization, as format strings that are read from internationalization files are tainted.

Analysis of Literal Format Strings.

Some compilers and languages, such as GCC [15] and OCaml [25], can check at compile time whether the number and type of arguments passed to a format routine are correct. GCC supports a function attribute [16] to extend these checks to user-defined format routines. The analysis is not sophisticated enough to check these properties for anything but literal format strings.

Dependent Type Systems.

Gronski et al. show how to implement a simple `printf` function in the dependently typed language SAGE [17]. The function's type requires a proof at every call site that the right type and number of arguments is passed.

SAGE can use static analysis to automatically discharge some of these proof obligations, in particular for short literal format strings. No experiments have been done to evaluate how well SAGE's static analysis performs on programs that call their implementation of `printf`. If the analysis run by SAGE fails, it generates a run-time check for the proof obligation.

Our type system goes beyond Gronski et al.'s `printf` by supporting format routines that handle invalid format strings, format specifiers with argument selection, and format specifiers that allow multiple types. The type of Gronski et al.'s `printf` shows similarities to our *qualifier* function. Our notion of `InvalidFormat` and conversion categories could potentially be used to extend their `printf` implementation.

One can interpret our type system as a concrete instance of a dependent type system. On the one hand, it is limited to format string APIs, but it is also simpler, easier to use (no background in type theory is required), and more tractable to implement. Dependent types are not supported by any mainstream language. Type qualifiers, on the other hand, are a powerful but relatively unintrusive way of extending a language's type system. For this reason even Java, known for its conservatism when adapting new language features, has included type annotations with Java 8 [23].

Dynamic Checking.

Safer implementations of C's `printf` have been proposed. These make `printf` act more like Java's format routines, throwing immediate run-time errors rather than suffering silent or delayed failures.

FormatGuard [6] uses GCC-specific properties of macros to count the arguments passed to `printf`, and ensures at run time that they match the arguments required by the format string.

Libsafe 2.0's [34] implementation of `printf` terminates the program if certain format arguments point to suspicious locations on the stack, such as a function's return address.

The approach by Ringenburg et al. [28] uses a combination of static analysis and dynamic checks to ensure that a format routine only modifies a memory location x if the programmer explicitly passed a pointer to x.

This kind of solution is easily deployed, often without even recompiling the target application, and it protects against format string exploits in unsafe languages.

On the other hand, dynamic analysis gives no guarantees about the absence of errors, and the program may fail or crash at run time. Furthermore, existing dynamic analysis do not guard against all kinds of format routine errors.

Alternative Formatting APIs.

An alternative to making existing format string APIs safer, is to replace them with a different API.

C++ introduced iostreams [19], a type-safe alternative to `printf`. Continuation-based approaches can be used for languages with ML-style type systems [4, 9, 18].

While these replacements are easier to type-check, format string routines are more readable and support internationalization. Consider the internationalization examples from Listing 8, that compare format string APIs with alternative APIs. The alternatives make translation hard because variables split up the sentences that have to be translated. In the first example, a translator must inspect the context of the string to understand that `"bugs"` has to be translated to `"Bugs entdeckt"`. Yet worse, in the second example, the variables have to be reordered in the translation — a task unachievable without recompilation.

Static Analysis of Domain Specific Languages.

A format string can be thought of as a program written in a DSL, embedded inside a string in a host language.

Previous work investigated static analysis for embedded DSLs like SQL queries [14, 32] and regular expressions [30].

Summary.

In contrast to previous work, our format string type system statically guarantees that calls to format routines never fail, is tractable to understand and implement, supports internationalization, has a low annotation burden, and suffers few false positives.

```
// 1a. Java i18n API
format("We detected {0,number} bugs", n);
format("Wir haben {0,number} Bugs entdeckt", n);

// 1b. ML continuation passing style
// Parts of sentences cannot be translated
fmt (lit "We detected " oo int oo lit " bugs") n;
fmt (lit "Wir haben " oo int oo
     lit " Bugs entdeckt") n;

// 2a. Java Formatter API
printf("It's a %s %s", adj, noun);
printf("Es un %2$s %1$s", adj, noun);

// 2b. C++ iostreams
// Recompilation required
cout << "It's a " << adj << " " << noun;
cout << "Es un " << noun << " " << adj;
```

Listing 8: Comparison of format string APIs and alternative APIs for internationalization. 1) The verbs `"detected"` and `"entdeckt"` are in different positions relative to the variable. Thus, sentences have to be translated as a whole; their parts cannot be translated independently. 2) The variables `adj` and `noun` have to be reordered. Thus, the code cannot be translated without recompilation.

9. CONCLUSION

We created a qualifier-based type system that guarantees that format routine calls never fail at run time. The type system is applicable to many format string APIs, and could for example be instantiated for C, C#, Java, or Go.

We instantiated this type system for Java's Formatter format string API. An implementation is available for download at `http://checkerframework.org`. Our tool generates few false positives, has a low annotation burden, and found 104 bugs in 6 large and well-maintained, open-source projects.

10. ACKNOWLEDGMENTS

We thank Werner Dietl for his help with many aspects of this project, and the reviewers for their insightful comments. This material is based upon work supported by the United States Air Force under Contract No. FA8750-12-C-0174.

11. REFERENCES

[1] ANSI x3.9-1966. FORTRAN, 1966.

[2] Apache Hadoop. `http://hadoop.apache.org/`.

[3] Apache Lucene. `http://lucene.apache.org`.

[4] K. Asai. On typing delimited continuations: three new solutions to the printf problem. *Higher-Order and Symbolic Computation*, 22(3):275–291, 2009.

[5] G. Chen and M. Kandemir. Verifiable annotations for embedded Java environments. In *CASES*, 2005.

[6] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium*, 2001.

[7] CVE wu-ftpd bug. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0573`.

[8] Daikon. `http://plse.cs.washington.edu/daikon`.

[9] O. Danvy. Functional unparsing. *Journal of Functional Programming*, 8:621–625, 1998.

[10] A. DeKok. PScan: A limited problem scanner for C source files. http://deployingradius.com/pscan/.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE*, 1999.

[12] FindBugs. http://findbugs.sourceforge.net.

[13] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *SIGPLAN*, 1999.

[14] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting SQL injection vulnerabilities. In *COMPSAC*, 2007.

[15] GCC -Wformat warning option. http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html.

[16] GCC format attribute. http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Function-Attributes.html.

[17] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, 2006.

[18] R. Hinze. Formatting: a class act. *Journal of Functional Programming*, 13:935–944, 2003.

[19] ISO/IEC 14882:2011. Information technology — Programming languages — C++, 2011.

[20] ISO/IEC 9899:1990. Programming languages — C, 1990.

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

[22] Java tutorials on internationalization. http://docs.oracle.com/javase/tutorial/i18n/format/messageintro.html.

[23] JSR 308: Annotations on Java types. https://jcp.org/en/jsr/detail?id=308.

[24] K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Technical report, Compaq Systems Research Center, 2000.

[25] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 4.01. http://caml.inria.fr/pub/docs/manual-ocaml/.

[26] Ohloh. http://ohloh.net.

[27] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, 2008.

[28] M. F. Ringenburg and D. Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *Computer and Communications Security*, 2005.

[29] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, 2001.

[30] E. Spishak, W. Dietl, and M. D. Ernst. A type system for regular expressions. In *FTfJP*, 2012.

[31] F. Spoto. Nullness analysis in Boolean form. In *SEFM*, 2008.

[32] Z. Tatlock, C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Deep typechecking and refactoring. In *OOPSLA*, 2008.

[33] A. Tomb, G. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *ISSTA*, 2007.

[34] T. Tsai and N. Singh. Libsafe 2.0: Detection of format string vulnerability exploits. *Avaya Labs*, 2001.