Proceedings of the 10th USENIX Security Symposium

Washington, D.C., USA
August 13–17, 2001

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Detecting Format String Vulnerabilities with Type Qualifiers[*]

Umesh Shankar Kunal Talwar Jeffrey S. Foster David Wagner
{ushankar,kunal,jfoster,daw}@cs.berkeley.edu
University of California at Berkeley

May 11, 2001

Abstract

We present a new system for automatically detecting format string security vulnerabilities in C programs using a constraint-based type-inference engine. We describe new techniques for presenting the results of such an analysis to the user in a form that makes bugs easier to find and to fix. The system has been implemented and tested on several real-world software packages. Our tests show that the system is very effective, detecting several bugs previously unknown to the authors and exhibiting a low rate of false positives in almost all cases. Many of our techniques are applicable to additional classes of security vulnerabilities, as well as other type- and constraint-based systems.

1 Introduction

Securing systems that interact with malicious parties can be a tremendous challenge. Indeed, systems written in C are especially difficult to secure, given C's tendency to sacrifice safety for efficiency. One of the more subtle pitfalls facing implementors is the so-called format string vulnerability. Since the discovery of this failure mode in the past year, security experts have identified format string vulnerabilities in dozens of widely-deployed security-critical systems [2, 4, 5, 8, 9, 10, 11, 22, 23, 24, 25, 27, 30, 35, 43], and attackers have begun exploiting these security holes on a large scale [10, 27], gaining root access on vulnerable systems. It seems likely that many legacy applications still contain undiscovered format string vulnerabilities.

Format string bugs arise from design misfeatures in the C standard library combined with a problematic implementation of variable-argument functions. Consider a typical usage of format strings:

```
printf("%s", buf);
```
(correct)

The first argument to `printf()` is a format string that specifies the number and types of the other arguments. No checking is done, either at run-time or compile-time, to verify that `printf()` was indeed called with the correct number and types of arguments. Thus the following innocuous-looking simplification of the above call can be dangerous:

```
printf(buf);
```
(*may be incorrect!*)

If `buf` contains a format specifier (e.g., "`%s`"), `printf()` will naively attempt to read nonexistent arguments off the stack, most likely causing the program to crash. The C standard library contains a number of other, similar primitives that put the programmer at risk for format string bugs. Other examples include the message-logging `syslog()` function, as well as `setproctitle()`, which sets the X window name associated with the current process.

A perhaps unexpected consequence of format string bugs is that they can be devastating to security. When a knowledgeable adversary has control of the value of the format string *s* involved in a format string bug, they can use *s* to write to arbitrary memory locations. For example, including the "`%n`" specifier in a format string causes `printf`-like functions to store the number of characters printed so far into a location pointed to by the associated argument. When combined with other tricks, this often leads to a complete compromise of security. Techniques for exploiting format string bugs have been described elsewhere [30]; for the purposes of this paper, the details are unimportant.

The main contribution of this paper is to describe a system for automatically detecting format string bugs at compile-time. Our system applies static, type-theoretic analysis techniques from the programming languages literature to the task of detecting potential security holes. We have implemented our system as a tool built on top of an extensible *type qualifier* framework [19]. We have tested our tool on a number of real-world software systems, in the process independently re-discovering several format string bugs that were unknown to the authors at the time.

[*]This research was supported in part by the National Science Foundation CAREER Award No. CCR-0093337, NSF CCR-9457812, NASA Contract No. NAG2-1210, NSF CCR-0085949, a UC Regents Fellowship, and an equipment donation from Intel.

```
while (fgets(buf, sizeof buf, f)) {
   lreply(200, buf);
   ⋮
}

void lreply(int n, char *fmt, ...)  {
   ⋮
   vsnprintf(buf, sizeof buf, fmt, ap);
   ⋮
}
```

Figure 1: A format string vulnerability found in `wuftpd` 2.6.0, paraphrased for brevity.

Before describing the ideas behind our tool in more detail, we discuss some of the alternatives to static analysis; more are discussed in Section 6.

One natural alternative to static analysis is testing. The main weakness of testing is coverage—it is extremely difficult to construct a test suite that exercises all possible paths through a program. Unfortunately, a security auditor is most interested in exactly the paths that are never followed in ordinary operation. For example, a major source of format string bugs comes from error reporting code (e.g., calls to `syslog()`). Such code is triggered only on rare, exceptional paths, and it is easy to overlook such paths—and hence, such bugs—with run-time testing. With static analysis, on the other hand, vulnerabilities can be proactively identified and fixed before the code is ever run.

Another alternative to automated static analysis is manual code review. Unfortunately, humans are not especially good at finding format string bugs by inspection. Figure 1 shows a representative example, excerpted from a recent version of `wuftpd` [2, 43]. The code in Figure 1 reads a line of text from the network and passes it to `lreply()`, where it will later be used as a format string specifier to `vsnprintf()`. The correct syntax would have been `lreply(200, "%s", buf)`, but the programmer omitted the `"%s"`. As before, this introduces a serious security vulnerability.

In real code, the omission of a format string is often located far away from the place where the requirement for a trusted format string specifier becomes apparent. In the case of our `wuftpd` example, the offending call to `lreply()` was not even in the same file as the eventual use of `vsnprintf()`. Figure 1 also shows why naive static analysis—e.g., searching for all occurrences of `printf(s)` and replacing them with `printf("%s", s)`—does not work in practice. Very often format string bugs occur within wrapper functions to `printf()`, and these non-localized bugs require more sophisticated analysis techniques.

A third alternative would be to re-implement the application in a safe language (such as Java). However, such an approach is likely to be too costly for most legacy applications.

1.1 Type Systems for Finding Format String Bugs

Format string vulnerabilities occur when untrustworthy data (i.e., data that could potentially be controlled by an attacker) is used as a format string argument. Therefore, in our analysis we treat all program inputs that could be controlled by the adversary as "tainted," and we track the propagation of tainted data through each of the program's operations. Any variable assigned a value derived from tainted data will itself be marked as tainted, and so on. If there is any execution path in which tainted data will be interpreted as a format string by some C library function, we raise an error.

Our approach is thus conceptually similar to Perl's successful taint mode [32, 42], but with an important difference. Rather than using run-time taint propagation (which is more easily implemented for interpreted languages, such as Perl, than for compiled languages like C), we apply a static taint analysis so that we can detect bugs before the program is ever run.

We model tainting by extending the existing C type system with extra *type qualifiers*. The standard C type system already contains qualifiers such as `const`; we add a new qualifier, `tainted`, to tag data that originated from an untrustworthy source. We label the types of all untrusted inputs as tainted, e.g.,

```
tainted int getchar();
int main(int argc,
         tainted char *argv[]);
```

The first annotation specifies that the return value from `getchar()` should be considered tainted. The second specifies that the command-line arguments to the program should be treated as a tainted value.

We construct typing rules so that taint information will be propagated appropriately. Given a small set of initial tainting annotations, we infer a typing for all program variables indicating whether each variable might be assigned a value derived from a tainted source. If any expression with a `tainted` type is used as a format string, we warn the user of the potential security hole. This use of type inference for automated detection of security vulnerabilities in legacy applications is, to our knowledge, novel, and we conjecture that it may find applications elsewhere as well.

We would like to emphasize that, although in this paper we present type qualifiers in the context of finding format string bugs in C programs, in fact our implementation is expressly designed to be extensible to other kinds of type qualifiers, and indeed the idea of a type qualifier system can be applied to most standard type systems.

A key advantage to using type qualifiers is that they extend the existing type system in a backwards-compatible way. Our tool comes with default type annotations for the standard C library functions, which allows us to analyze legacy code for format string vulnerabilities with little annotation effort from the code reviewer and no modification to application source code. At the same time, type qualifiers provide a way for developers to express more detailed assertions about trust relationships in the program, and therefore programmers who are willing to spend time adding application-specific annotations can reap the extra benefits of this additional information. In other words, type qualifiers have the beneficial property that the value one obtains from the tool is proportional to the effort invested.

Type systems have several advantages over other program analysis techniques:

1. Types are a familiar way to annotate programs. We want to make it convenient for programmers to add information to their programs about tainted inputs and must-not-be-tainted variables. Type-based methods meet this goal, because programmers are accustomed to expressing invariants using types.

2. Types are a familiar way to express the output of our analysis. To be useful, when errors are reported, our tool needs to explain why the erroneous code was rejected. Giving a typing on the relevant program variables is a way to express this output in a form that programmers can readily understand.

3. Type theory is well understood. There are many efficient algorithms known in the programming languages community for inferring and manipulating types.

4. Types provide a sound basis for formal verification. Once we have found and eliminated bugs from our code, it is useful to have tools to verify that there are no format string bugs left. Because it is well-known how to build a sound type system (i.e., one where all programs that typecheck will be guaranteed free of format string bugs), types provide a single foundation that can be applied both to bug-finding and to software verification.

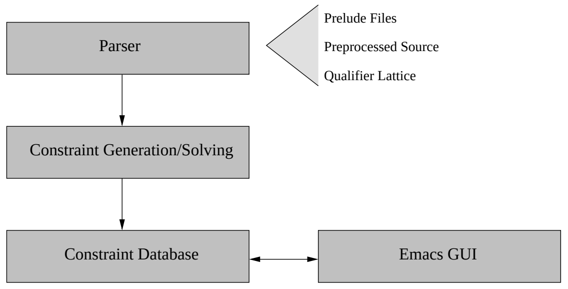


Figure 2: The architecture of the cqual system. The C source code and the configuration files are parsed, producing an annotated Abstract Syntax Tree (AST). cqual traverses the AST to generate a system (or database) of type constraints, which are solved on-line. Warnings are produced whenever an inconsistent constraint is generated. The analysis results are presented to the programmer in an emacs-based GUI, which interactively queries the constraint solver to help the user determine the cause of any error messages.

In summary, we focus our attention on type-based methods primarily because types provide a uniform, understandable interface to our tool.

Although our work relies heavily on theoretical techniques from the programming languages community, we emphasize that our efforts are aimed at providing a practical tool. Thus, we set out to build a tool that is easy to use, efficient on common hardware, effective at finding typical format string bugs, and unlikely to generate many false alarms.

2 Background

Our tool is built on top of `cqual`, a C implementation of an extensible type qualifier framework [19]. In this section we describe the underlying theory and design of `cqual`, which has broad applicability as an extension of the C type system.

2.1 System Architecture

Figure 2 shows the structure of the `cqual` tool. The main input to the tool is the preprocessed C code the user wishes to analyze. The user also provides two types of configuration files to customize `cqual` to the particular checking task. The lattice file describes the type qualifiers the user is interested in (Sections 2.2 and 2.3). The prelude files contain annotated function declarations that override the declarations in the source being analyzed.

Given preprocessed C code and configuration files, `cqual` performs type inference on the program (Section 2.4). Finally, the results of the type inference phase are presented to the user interactively using *Program Analysis Mode* (PAM) for emacs (Section 3).

The configuration files make `cqual` usable "out-of-the-box," i.e., without making any changes to the source except preprocessing. We were able to analyze all of our benchmark programs with the same standard prelude file and, in virtually all cases, no direct changes to the application source code. Typically, a few application-specific entries were added to a special local prelude file, to improve accuracy in the presence of wrappers around library functions (though the GUI indicates which ones to add). This goes a long way toward making `cqual` an easily usable tool.

2.2 Type Qualifiers and Subtyping

To find format string bugs, we use a type qualifier system with two qualifiers, `tainted` and `untainted`. We mark the types of values that can be controlled by an untrusted adversary with `tainted`. All other values are given types marked `untainted`. This is similar to the concept of tainting in Perl [32, 42].

Intuitively, `cqual` extends the type system of C to work over *qualified types*, which are the combination of some number of type qualifiers with a standard C type. We allow type qualifiers to appear on every level of a type. Examples of qualified types are `int`, `tainted int`, `untainted char` $*$ (a pointer to an untainted character), and `char` $*$ `untainted` (an untainted pointer to a character).

The key idea behind our framework is that type qualifiers naturally induce a *subtyping* relationship on qualified types. The notion of subtyping most commonly appears in object-oriented programming. In Java, for example, if B is a subclass of A (which we will write $B < A$), then an object of class B can be used wherever an object of class A is expected.

Consider the following example program:

```
(1)  void f(tainted int);
     untainted int a;
     f(a);
```

In program (1), `f`, which expects tainted data, is passed untainted data. In our system, this program typechecks. Intuitively, if a function can accept tainted data (presumably by doing more checks on its input), then it can certainly accept untainted data.

Now consider another program:

```
(2)  void g(untainted int);
     tainted int b;
     g(b);
```

In this case, `g` is declared to take an `untainted int` as input. Then `g` is called with a `tainted int` as a parameter. Our system should complain about this program: tainted data is being passed to a function that expects untainted data.

Putting these two examples together, we have the following subtyping relation:

$$\texttt{untainted int} < \texttt{tainted int}$$

As in object-oriented programming, if $T_1 \leq T_2$ (read T_1 is a subtype of T_2), then T_1 can be used wherever T_2 is expected, but not vice-versa. We write $T_1 < T_2$ if $T_1 \leq T_2$ and $T_1 \neq T_2$.

2.3 The Qualifier Lattice

The `cqual` tool needs to know not only how integer types with qualifiers relate but also how qualifiers affect pointer types, pointer-to-pointer types, function types, and so on. Fortunately, standard results on subtyping tell us how to extend the subtyping on integers to other data types [1, 28].

We supply `cqual` with a configuration file placing the qualifiers (in this case, `tainted` and `untainted`) in a lattice [14]. A lattice is a partial order where for each pair of elements x and y, the least upper bound and greatest lower bound of x and y both always exist. Using a lattice makes the implementation slightly easier. For finding format string bugs, we specify in the lattice configuration file that `untainted` $<$ `tainted`.

Given this configuration file, `cqual` extends the supplied lattice on qualifiers to a subtyping relation on qualified C types. We have already seen one of the subtyping rules:

$$\frac{Q_1 \leq Q_2}{Q_1\ \texttt{int} \leq Q_2\ \texttt{int}}$$

This is a natural-deduction style inference rule. In general, an inference rule says that if the statements above the line are true, then the statements below the line are also true. This particular inference rule is read as follows: If $Q_1 \leq Q_2$ in the lattice (Q_1 and Q_2 are qualifiers), then Q_1 `int` is a subtype of Q_2 `int` (note the overloading of $\leq$). For our example, it means that `untainted int` $\leq$ `tainted int`. The same kind of rule applies to any primitive type (`char`, `double`, etc.).

For pointer types, we need to be a little careful. Naively, we might expect to use the following rule for pointers:

$$\frac{Q_1 \leq Q_2 \qquad T_1 \leq T_2}{Q_1\ \mathit{ptr}(T_1) \leq Q_2\ \mathit{ptr}(T_2)}\ \text{(Wrong)}$$

Here the type $Q_1\ \mathit{ptr}(T_1)$ is a pointer to type T_1, and the pointer is qualified with Q_1. Note that T_1 represents an extended C type, and thus may itself be decorated with tainted/untainted qualifiers. In C, the type $Q_1\ \mathit{ptr}(T_1)$ might be written

```
typedef T1 *ptr_to_t1;
typedef Q1 ptr_to_t1 q1_ptr_to_t1;
```

The rule (Wrong) says that if $Q_1 \leq Q_2$ in the lattice and T_1 is a subtype of T_2, then we can conclude that $Q_1\ \mathit{ptr}(T_1)$ is a subtype of $Q_2\ \mathit{ptr}(T_2)$.

Unfortunately, this turns out to be unsound, as illustrated by the following code fragment:

```
  tainted char *t;
  untainted char *u;

  t = u;    /* Allowed by (Wrong) */
  *t = <tainted data>;
    /* Oops! This writes tainted data
       into untainted buffer *u */
```

According to (Wrong), the first assignment `t = u` typechecks, because *ptr*(`untainted char`) is a subtype of *ptr*(`tainted char`). Then `*t` becomes an alias of `*u`, yet they have different types. Therefore we can store `tainted` data into `*u` by going through `*t`, even though `*u` is supposed to be untainted.

This is a well-known problem, and the standard solution, which is followed by `cqual`, is to use the following rule:

$$\frac{Q_1 \leq Q_2 \qquad \tau_1 = \tau_2}{Q_1\ \mathit{ptr}(\tau_1) \leq Q_2\ \mathit{ptr}(\tau_2)}$$

The key restriction here is that $\tau_1 = \tau_2$. Intuitively, this restriction says that any two objects that may be aliased must be given exactly the same type.[1] In particular, if τ_1 and τ_2 are decorated with qualifiers, the qualifiers must themselves match exactly, too.

2.4 Type Inference

So far we have concentrated on the *type checking* problem: given a program fully annotated with type specifiers on all expressions, confirm that the types are consistent. Typechecking a program is straightforward. For example, the assignment `x = y` typechecks if and only if the type of `y` is a subtype of the type of `x`. The function call `f(x)` typechecks if and only if the type of `x` is a subtype of the type of the formal parameter of `f`. More detailed rules, and a proof of soundness, can be found in [19].

[1]Java uses the rule (Wrong) for arrays. In Java, if S is a subclass of T, then S[] is a subclass of T[], where X[] is an array of X's. Java gets away with this by inserting run-time checks at every assignment into an array to make sure the type system is not violated. Since we seek a purely static system, Java's approach is not available to us.

The type checking system described so far, however, is not useful in practice. The problem is that it requires all types to be annotated with qualifiers: for our running example, all types would need to be marked as either `tainted` or `untainted` at every level of each type. Clearly this is an undesirable property for two reasons. First, we are interested in finding bugs in legacy code that does not have any type qualifier annotations. Second, even if we are writing a program with type qualifiers in mind, adding and maintaining annotations on every type in the program would be prohibitively expensive for programmers.

The solution to this problem is *type inference*. In this model, the user introduces a small number of annotations at key places in the program, and `cqual` infers the types of the other expressions in the program. `cqual` generates fresh *qualifier variables* (variables which range over type qualifiers) at every position in a type, constrained by any annotations specified in the program. `cqual` analyses the program and generates *subtyping constraints*—i.e., inequalities of the form $T_1 \leq T_2$ for qualified types T_1 and T_2.

A *solution* to a set of subtyping constraints is a mapping from qualifier variables to qualifiers such that all of the constraints are valid according to our subtyping rules. Thus, in our system, we solve the constraints by assigning every qualifier variable to either `tainted` or `untainted`.

In our type inference algorithm, qualifier variables are introduced at every position in a type. We write qualifier variables in italics, and name them after the corresponding program variables. The i^{th} argument of function `f` has associated qualifier variable *f_argi*, and the return value of function `f` has qualifier variable *f_ret*.

Since qualifiers are implicitly introduced on all levels of a type by the type inference algorithm, to name them we modify the name of the outermost qualifier of a type. For example, given the declaration `char *x`, `cqual` generates two qualifier variables: the variable *x* qualifies the reference `x` itself, and the variable *x_p* qualifies the location `*x`. Moreover, the programmer may also explicitly introduce named qualifier variables into the pro-

`tainted char *getenv(const char *name);`	$\mathit{getenv\_ret\_p} = \texttt{tainted}$
`int printf(untainted const char *fmt, ...);`	$\mathit{printf\_arg0\_p} = \texttt{untainted}$
`char *s, *t;`	
`s = getenv("LD_LIBRARY_PATH");`	$\mathit{getenv\_ret} \leq \mathit{s}$
	$\mathit{getenv\_ret\_p} = \mathit{s\_p}$
`t = s;`	$\mathit{s} \leq \mathit{t}$
	$\mathit{s\_p} = \mathit{t\_p}$
`printf(t);`	$\mathit{t} \leq \mathit{printf\_arg0}$
	$\mathit{t\_p} \leq \mathit{printf\_arg0\_p}$

Figure 3: An example of constraint generation. The left column is a code fragment; the right column gives the inferred constraints on the qualifier variables.

gram; in this case, they begin with a dollar sign (“`$`”) in the source code to distinguish them lexically from other tokens.

For example, after the declaration `char *x;` we assign the qualified type $\mathit{x\_p}$ `char *` x to `x`. Similarly, a function declared with the prototype

```
tainted char *getenv(char *name);
```

is assigned the following fully qualified type:

$$\mathit{getenv\_ret\_p}\ \texttt{char *}\ \mathit{getenv\_ret}$$
$$\texttt{getenv(}\mathit{getenv\_arg0\_p}\ \texttt{char *}\ \mathit{getenv\_arg0}\ \texttt{name);}$$
$$\text{(where } \mathit{getenv\_ret\_p} = \texttt{tainted}\text{)}$$

If we then encounter an assignment `x = getenv(...)`, our type inference algorithm will conclude that the type of `getenv()`’s return value must be a subtype of the type of `x`, i.e.,

$$\mathit{getenv\_ret\_p}\ \texttt{char *}\ \mathit{getenv\_ret} \leq \mathit{x\_p}\ \texttt{char *}\ x\ .$$

As a consequence, we can infer (using the subtyping rules introduced in Section 2.2 and 2.3) that we must have the following constraints on the qualifier variables:

$$\mathit{getenv\_ret\_p} = \mathit{x\_p} = \texttt{tainted}, \quad \mathit{getenv\_ret} \leq x.$$

In essence, our declaration of `getenv()` has ensured that whatever it returns will be labeled as tainted. Note that this might be used to model, for instance, a scenario where environment variables are under the adversary’s control.

We give next a more detailed example. Figure 3 shows a fragment of code that manipulates tainted data in an unsafe way, along with the typing constraints generated by the type inference algorithm. The constraint $\mathit{getenv\_ret\_p} = \mathit{s\_p}$ encodes the conclusion that the return value of `getenv()` is treated as tainted (as discussed above). The prototype for `printf()` (typically found in the global prelude file) specifies that `printf()` must not be called with a tainted format string argument, by requiring that its first argument be a subtype of `untainted char *`.

The call `s = getenv("LD_LIBRARY_PATH")` generates the constraints

$$\mathit{getenv\_ret} \leq s$$
$$\mathit{getenv\_ret\_p} = \mathit{s\_p}$$

Notice the equality constraint, arising from our corrected rule for subtyping pointer types. The assignment `t = s` generates a similar constraint. Finally, the call `printf(t)` generates a subtyping constraint on the $\mathit{printf\_arg0\_p}$ because `printf`’s first argument is `const` (see Section 4.4).

Taking the transitive closure of these constraints, we have a chain of deductions

$$\texttt{tainted} = \mathit{getenv\_ret\_p} = \mathit{s\_p} = \mathit{t\_p}$$
$$\leq \mathit{printf\_arg0\_p} = \texttt{untainted},$$

implying that for this example to type check, we would need $\texttt{tainted} \leq \texttt{untainted}$. As explained in Section 2.2, this does not hold in our lattice, so this code fragment does not type check, indicating a possible format string bug. This demonstrates how our type inference algorithm can be used to identify unsafe manipulation of format strings.

In our implementation, the subtyping constraints are solved on-line as they are generated. If the constraint system ever becomes unsatisfiable, an error is flagged at the first illegal expression in the code. This allows us to pinpoint the location of unsafe operations on tainted data. The inference then continues after any errors, though in this case the quality of the remaining error messages can vary tremendously.

We observe that efficient algorithms for this type inference problem are known. Given a fixed-size qualifier lattice and n constraints of the form $l \leq q$, $q \leq l$, or

```
tainted char *getenv(const char *name);
int printf(untainted char *fmt, ...);

/* Point 1 */
char *f3(char *s) { return s; }

/* Point 2 */
char *f2(char *s) { return f3(s); }

/* Point 3 */
char *f1(char *s) { return f2(s); }

int main()
{
    char *s, *unclean;

    /* Point 0 */
    unclean = getenv("PATH");

    s = f1(unclean);  /* Point 4 */
    printf(s);        /* Point 5 */
}
```

Figure 4: An example of a taint flow path. The string `unclean` is tainted by the call to `getenv` at Point 0, and ultimately that data is passed to `printf` at Point 5.

$q_1 \leq q_2$, where l is a lattice element and q, q_1, and q_2 are qualifier variables, a solution to the constraints can be computed in $O(n)$ time using well-known algorithms [21]. The idea is to express these constraints as a directed graph with qualifier variables as vertices and subtyping constraints as directed edges: the constraint $v_1 \leq v_2$ induces an edge from v_1 to v_2. The constant qualifiers `tainted` and `untainted` are also vertices in this graph, and a directed path from `tainted` to `untainted` corresponds to a possible format string bug. We call this path a *taint flow path*. See Figure 4 for an example.

3 User Interface

Thus far, we have presented the theory underlying our tool. For a program analysis to be useful, however, one needs both a sound theoretical foundation and an intuitive, efficient interface for understanding the results.

In the folklore of type inference, it is well known that the more powerful a type inference system is, the harder it is to understand why a program contains a type error. For example, type errors from a C compiler, which performs little inference, are easy to localize. The compiler simply reports the line number where the type error occurred, and this is almost always enough to tell the programmer why the error occurred.

In our type qualifier system, however, type errors occur at the point where the type constraint system becomes unsatisfiable, and that point can be distant from the actual source of the problem. Again, consider Figure 4. In this example, the string `unclean` is tainted by the call to `getenv` at Point 0, and ultimately that data is passed to `printf` at Point 5. Given this input program, our system will warn the user of a potential format string bug at point 5. But program points 1–5 are all involved in the error, and to understand and fix the error a programmer may need to examine all five program points. In general these program points could be spread across multiple files.

Thus reporting line numbers with error messages is no longer enough. In this section, we describe the techniques we use to display the results of our tainting analysis to the user. We emphasize that without the GUI described in this section, performing the experiments described in Section 5 would have been extremely difficult.

3.1 Program Analysis Mode

Our tool `cqual` presents the results of the tainting analysis to the programmer using *Program Analysis Mode* (PAM) for Emacs [20], a GUI developed at Berkeley that is designed to add hyperlinks and color mark-ups to the preprocessed text of the program.

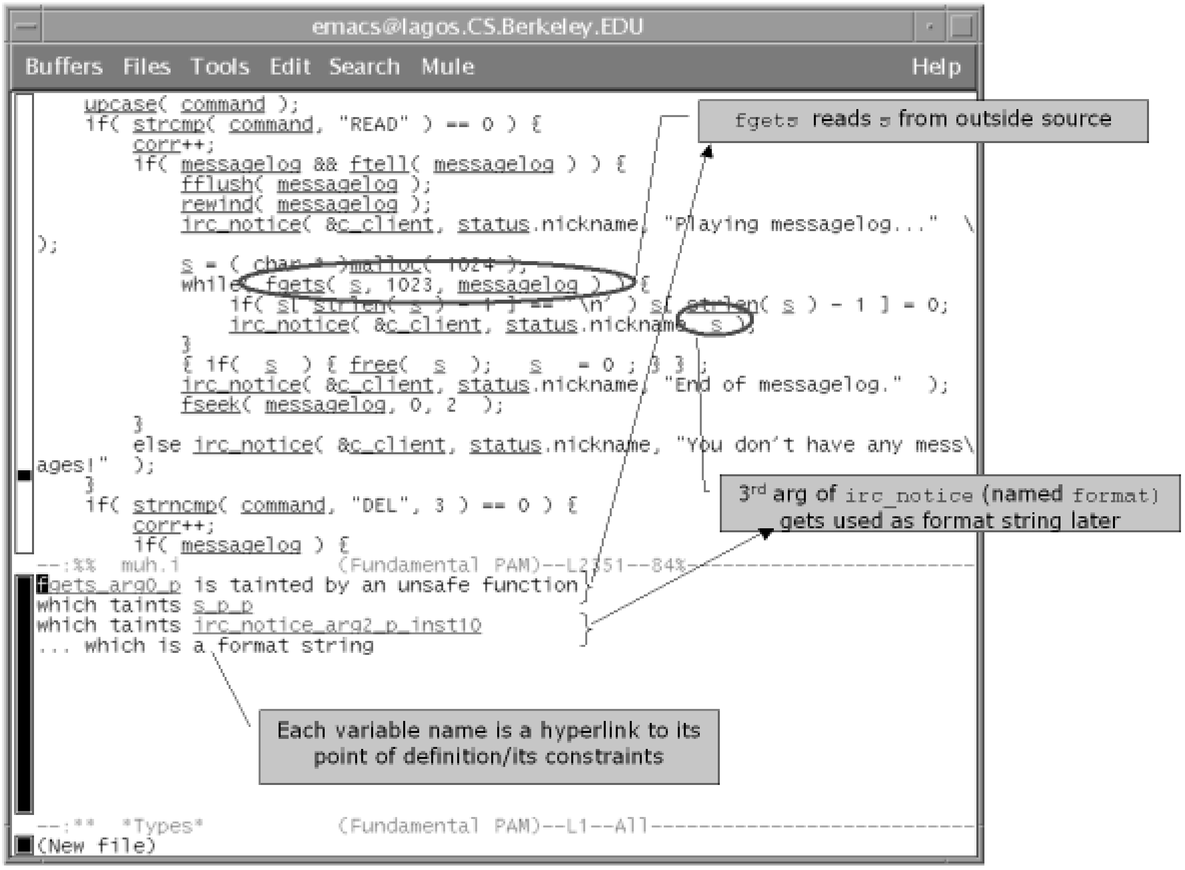
Figure 5 shows a screenshot of a run of `cqual` on `muh`, an IRC proxy application. `cqual` initially displays a list of all files analyzed and any errors that occurred. The user can click on a filename to jump to that file or click on an error message to jump to information about that error (see below).

Each identifier in a file is colored according to its inferred qualifiers. Tainted identifiers (those whose type contains a tainted qualifier somewhere) are colored red, untainted identifiers are colored green, and any identifiers that could be either tainted or untainted are not colored. Intuitively, this last set of qualifiers could all be marked untainted, but it is easier on the user to reduce the number of marked up identifiers.

The user can click on an identifier to display its fully qualified type, with each individual qualifier colored according to its taintedness.

3.2 Added Features

Beyond the basic coloring of qualifiers, we designed several extensions to make it easy to find and fix potential format string bugs. Many of these features are applicable to other kinds of qualifiers, and perhaps to


emacs@lagos.CS.Berkeley.EDU
Buffers Files Tools Edit Search Mule Help
fgets reads s from outside source
3rd arg of irc_notice (named format) gets used as format string later
Each variable name is a hyperlink to its point of definition/its constraints

```
    upcase( command );
    if( strcmp( command, "READ" ) == 0 ) {
        corr++;
        if( messagelog && ftell( messagelog ) ) {
            fflush( messagelog );
            rewind( messagelog );
            irc_notice( &c_client, status.nickname, "Playing messagelog..."  \
);

            s = ( char * )malloc( 1024 );
            while( fgets( s, 1023, messagelog ) ) {
                if( s[ strlen( s ) - 1 ] == '\n' ) s[ strlen( s ) - 1 ] = 0;
                irc_notice( &c_client, status.nickname, s );
            }
            { if(  s  ) { free(  s  );   s   = 0 ; } } ;
            irc_notice( &c_client, status.nickname, "End of messagelog."  );
            fseek( messagelog, 0, 2  );
        }
        else irc_notice( &c_client, status.nickname, "You don't have any mess\
ages!"  );
    }
    if( strncmp( command, "DEL", 3 ) == 0 ) {
        corr++;
        if( messagelog ) {
--:%%  muh.i            (Fundamental PAM)--L2351--84%------------------------
fgets_arg0_p is tainted by an unsafe function
which taints s_p_p
which taints irc_notice_arg2_p_inst10
... which is a format string

--:**  *Types*          (Fundamental PAM)--L1--All---------------------------
(New file)
```

Figure 5: Screenshot of a run of `cqual` on the `muh` application.

other kinds of type inference systems as well.

Taint Flow Paths. Recall from Section 2.4 that the subtyping constraints can be thought of as inducing a directed graph among qualifiers. A path in the constraint graph from `tainted` to `untainted` indicates a type error.

For each type error, we provide a hyperlink to a display of the particular path from `tainted` to `untainted` that caused that error. Since each path in the constraint graph typically corresponds to a flow of data through the program, this helps identify the unsafe sequence of operations that lead to a type error. However, since there are typically many such paths (and possibly even cycles) in the constraint graph, displaying all of them may overload the user. Therefore, to reduce the burden on the user, we display the shortest such path, as computed with a breadth-first search. In our experience, this heuristic is very important for usability.

Figure 5 shows one example. Each qualifier in the path is hyperlinked to the definition of the identifier with that qualifier, which makes it easy to navigate the source code to determine the cause of the error.

Unannotated Functions. Our standard prelude files contain annotated versions of most standard library functions. Programs, of course, can also use system- and application-dependent libraries. In order to have a sound inference, the user must provide annotated declarations of these libraries.

To make it easy for the user to find and annotate these functions, we generate a list of hyperlinks to declarations of functions that have neither been defined nor have been declared in a prelude file.

A common idiom in many programs is to write functions that simply massage their inputs and then call a library function. For example, a program might contain a function `log_error(fmt, ...)` that calls `fprintf(stderr, fmt, ...)`. As described in Section 4.3, for soundness and to improve the precision of the analysis the user should add annotations to such wrapper functions around potentially-vulnerable library calls. To aid in the annotation process we provide a hyperlinked list of unannotated variable argument functions to the user.

Hotspots. Although many of the features of the system are geared toward reducing false positives and, where there are real bugs, reducing the number of resulting warnings, occasionally the user will be faced with hundreds of warnings.

To help the user decide which warnings to investigate first, we attempt to determine "hotspots" in the code. For each error message, we compute the shortest taint flow path and increment a counter associated with each qualifier on the path. We then present the user with a hyperlinked list of the "hottest" qualifiers, i.e., those involved in the largest number of (shortest) taint flow paths. The idea—borne out by our experience—is that adding a single annotation at an important point can dramatically reduce the number of warnings.

One extension to this idea, which we have not yet implemented, is to find the hottest constraints rather than the hottest qualifiers. This may help point the user to a particular erroneous expression in the code, rather than to an identifier.

4 Finding Format String Bugs

In Section 2 we described the basic workings of the `cqual` tool. In this section we discuss extensions to make the basic tool sound in the presence of type casts and variable argument functions, and to decrease false positives by using the programmer's knowledge about the program being analyzed.

4.1 Leaf Polymorphism

Type inference is a powerful tool for computing qualifiers given a few annotations. However, when inferring types of functions, we need to introduce some new machinery to avoid getting a large number of false positives.

To understand the problem, consider the following simple example code:

```
char id(char x) { return x; }
...
tainted char t;
untainted char u;
char a, b;
a = id(t); /* 1 */
b = id(u); /* 2 */
```

Because of call 1, we infer that `x` is a `tainted char`, and therefore we also infer that `a` is `tainted`. Then call 2 typechecks (because `untainted char` $\leq$ `tainted char`), but we infer that `b` must also be `tainted`.

While this is a sound inference, it is clearly overly conservative. Even though this simple example looks unrealistic, we encounter this problem in practice, most notably with library functions such as `strcpy`. This leads to a large number of false positives.

The problem arises because we are summarizing multiple stack frames for distinct calls to `id` with a single function type—`x` has to either be untainted everywhere or tainted everywhere. The solution to this problem is to introduce *polymorphism*, which is a form of context-sensitivity.

A function is said to be *polymorphic* if it has more than one type. Notice that `id` behaves the same way no matter what qualifier is on its argument `x`: it always returns exactly `x`. Thus we can give `id` the signature α `char id(`α `char x)` for any qualifier α.

Operationally, when we call a polymorphic function, we *instantiate* its type—we make a copy of its type, replacing all the generic qualifier variables α with fresh qualifier variables. Intuitively, this corresponds exactly to inlining the function, except that instead of making a fresh copy of the function's code, we make a fresh copy of the function's type.

We need a way to write down polymorphic type signatures—for example, we should be able to express that if the `strcat()` function is passed a `tainted` second argument, then its first argument should also be `tainted`, but not vice versa. We can do this by writing a polymorphic type with side constraints on the qualifiers:

```
α char *
strcat(α char *dst, β const char *src);
```
(where $\alpha \geq \beta$)

More generally, we want to be able to specify a polymorphic function

α `f(`β `arg0,` δ `arg1,` $\ldots$ `);`

with some arbitrary inequality constraints on the qualifier variables α, β, δ, etc. We define a concise notation for expressing these inequality constraints by using the following theorem.

Theorem 4.1 *Let $(P, \leq)$ be any finite partial order. Let $(2^{\mathbb{N}}, \subseteq)$ be the lattice of subsets of $\mathbb{N}$ with the set inclusion ordering. Then $(P, \leq)$ can be embedded in $(2^{\mathbb{N}}, \subseteq)$, i.e., there exists a mapping $\phi : P \to 2^{\mathbb{N}}$, such that $a \leq b \iff \phi(a) \subseteq \phi(b)$ and $\phi(x)$ is a finite subset of $\mathbb{N}$ for all $x \in P$.*

The theorem is formally proved in the appendix, and may be viewed as a concrete example of the Dedekind-MacNeille Completion [14].

This theorem enables us to define the partial order implicitly by the naming of the qualifier variables on the function arguments and return type. We represent a qualifier a in the partial order P by $\phi(a)$, which we denote as a '_' separated string of the integers in the set. If $\phi(a) = \{1, 2\}$, then a is represented as $\$\_1\_2$. Hence, the polymorphic declaration of `strcat` can now be written as

```
$_1_2 char *
strcat($_1_2 char *, $_1 const char *)
```

which means that the qualifier on the return type is the same as the qualifier on the first argument, and that they are both supertypes of the second argument. In other words, since $\{1, 2\} \supseteq \{1\}$, the names of the qualifiers encode the implicit inequality constraint $\$\_1\_2 \geq \$\_1$. Hence for any instantiation of `strcat()`, we have

$$\begin{aligned} \texttt{strcat\_ret\_p} &= \texttt{strcat\_arg0\_p} \\ &\geq \texttt{strcat\_arg1\_p}. \end{aligned}$$

This avoids the need to write subtyping constraints on the side for each polymorphic function. Instead, the constraints are encoded implicitly in the annotations themselves, which provides a concise framework for expressing subtyping annotations.

To keep our implementation simple, we only support polymorphism for library functions, i.e., functions with no code. To be more precise, any function may be declared polymorphically, but the polymorphic prototype will not be typechecked against its implementation. This restriction is not fundamental; there are well-known efficient algorithms for more general polymorphism [19, 33]. Our standard prelude files contain appropriate polymorphic declarations for most of the standard library functions.

4.2 Explicit Type Casts

The treatment of type casts in the program's source code is very important to the correct operation of our tool. In most cases, a pointer cast is used to implement generic functions, to emulate object subtyping, or to otherwise bypass the limitations of the C type system. Since a pointer cast usually preserves the semantic meaning of the data being pointed to, we want to preserve the taintedness of data through ordinary C typecasts. Consider the following program fragment:

```
void *y;
char *x = (char *) y;
```

If `y` is tainted, then `x` should also be tainted, even though their types do not otherwise match.

Casts to `void *` are particularly problematic because one can cast any type to a `void *`. For example, a programmer might write

```
char **s, **t;
void *v = (void *) s;
t = (char **) v;
```

Here the type structure of `v` has two levels, while the type structure of `s` has three. Hence there is no direct correspondence between the qualifiers of the two types.

We solve this problem by "collapsing" qualifiers at a type cast. If we cast a type t to a type u, then we match up the qualifiers level-by-level between t and u as deeply as possible. For example, when casting `char *x` to a `void *`, we add the constraints $x \leq$ *cast* and $x\_p =$ *cast_p*, where *cast* is the name we use for the qualifiers on the `void *`. As soon the structures of types t and u diverge, we equate all the remaining qualifiers. For example, when casting a `char **x` to a `void *`, we add the constraints $x \leq$ *cast* and $x\_p = x\_p\_p = cast\_p$. Putting this together, in the above example if if either `*s` or `**s` is tainted, then `*v` becomes tainted. When `v` is cast to `char **t`, both `*t` and `**t` will become tainted.

We also allow the knowledgeable programmer to indicate that some program data has been validated and should consequently be considered untainted despite its origins. Such an annotation can be expressed in our system by writing an explicit cast to an `untainted` type. To enable this, we do not add any constraints in case of an explicit cast containing a qualifier. For example, in the following code

```
void *y;
char *x = (untainted char *) y;
```

the assignment does not taint `x`, regardless of the inferred taintedness of `y`.

This feature allows the security-aware developer to implement functions that parse an input string and filter out dangerous substrings without departing from our framework. We assume that the programmer will add such an annotation only after ensuring that the string is validated by some rigorous checking procedure. There is no way to verify this assumption automatically. However, our syntax is designed to make it easy to manually audit all such annotations, since they can typically be easily identified by simply `grep`ing the source code for the keyword `untainted`.

Collapsing the qualifiers at casts is conservative but sound for the most common casts in a program. There are two ways in which our implementation is currently unsound with respect to casts. First, we have found that if we collapse qualifiers on structure fields at type casts, the analysis generates too many false positives (too much becomes tainted). Thus in our implementation if one aggregate is cast to another, we ignore the cast and do not collapse type qualifiers.

Second, because we use a subtyping-based system, the qualifier-collapsing trick does not fully model casts from pointers to integers. Consider the following code:

```
char *x, *y;
int a, b;

a = (int) x;      (1)
b = a;            (2)
y = (char *) b;   (3)
```

For line (1), we generate the constraints $x\_p = x = a$. For line (2), we generate the constraint $a \leq b$. And for line (3), we generate the constraints $b = y\_p = y$. Notice that we have $x\_p \leq y\_p$ but we do not have $y\_p \leq x\_p$, so our deductions are unsound.

We leave as future work the solution to these problems. We believe that the best solution will be to combine techniques that attempt to recover the semantic behavior of casts with conservative alias analysis for ill-behaved casts [12, 36, 37].

4.3 Variable Argument Functions

C allows functions to have a variable number of arguments, through the *varargs* language feature. However, there is no obvious way of specifying constraints on the individual varargs: even their type is not fixed. For example, in the expression `sprintf(s, "%s", t)`, if `t` is tainted, then we would like our type inference algorithm to force `s` to be tainted as well.

We have extended the C grammar so that the varargs specifier "`...`" can be annotated with a type qualifier variable. In the `sprintf()` example, we would like the first argument of `sprintf()` to be tainted if any of its varargs is tainted, so we use the type declaration

```
int sprintf($_1_2 char *,
        untainted char *, $_2 ...);
```

Consequently, if any of `sprintf()`'s arguments (excluding the first two) are tainted, we will infer that the first argument must be tainted as well. More precisely, for each qualifier q on any level of a type passed to the `...` of `sprintf()`, we add the constraint $q \leq$ `$_1_2`.

The type inference system ignores parameters beyond the last named argument of an unannotated varargs function. Thus for soundness the user must annotate all potentially-vulnerable varargs functions; as mentioned in Section 3.2, we provide a list of unannotated varargs functions to the user to help with this task. Our implementation also does not model varargs function pointers fully. Both of these issues can be easily addressed, and we plan to do so in the future.

4.4 `const` Allows Deep Subtyping

As described in Section 2.3, we use a conservative rule for pointer subtyping. This rule can lead to non-intuitive *reverse taint flow*, which often causes false positives. For example, consider the following code:

```
f(const char *x);
char *unclean, *clean;
unclean = getenv("PATH");
f(unclean);
f(clean); /* 'clean' gets tainted */
```

Here the `getenv()` function call imposes the condition *unclean_p* = `tainted`. The first call to `f` adds the constraint *f_arg0_p* = *unclean_p*. The second function call generates the constraint *f_arg0_p* = *clean_p*, thereby marking `*clean` as tainted, which is counter-intuitive.

Observe, however, that `f`'s argument `x` is of type `const char *`, so `f` can not make `*x` tainted if it is not tainted in the first place. Consequently, we modify the constraints in Section 2 as follows: For an assignment

```
const char *s;
char *t;
...
s = t;
```

we add the constraints *t* ≤ *s* and *t_p* ≤ *s_p*, if `*s` has a `const` qualifier. This is to be compared with the constraint *s_p* = *t_p* which we would otherwise have imposed. In this way we can use "deep subtyping" to improve precision for formal parameters marked `const`.

This extra precision, which helps avoid many false positives (especially in library functions), is the main reason we work in a subtyping system. Note that we rely on the C compiler to warn the programmer about any casts which discard the `const` qualifier, i.e., we assume that a variable that is `const` is never cast to anything that is not `const`.

5 Real-World Tests

We tested the effectiveness of `cqual` on several popular C programs that are potentially vulnerable to format string attacks. Some of them had known vulnerabilities; others did not. In all cases, attackers from across the network have control over some string input to the program. If this input is used as a format string, a carefully chosen input can crash the program or give the attacker root access.

5.1 Metrics

The ideal bug detector would detect all extant bugs without flagging correct code as being incorrect. The initial output from `cqual` is a list of warnings that indicate a type error somewhere in the program. Some of these correspond to real bugs; others are *false positives* stemming from our conservative tainting approach (and lack of full polymorphism). *False negatives* are also of interest: we would like all vulnerabilities to show up as warnings. One complicating factor is that many warnings can result from the same bug—for example, if many functions reading network data call a single function that has a format string bug, then all the warnings may go away when that bug is fixed.

We chose the following metrics, measured per-program:

- How many known vulnerabilities were detected and how many went undetected?
- How many false positives were there?
- How easy was it to check whether a warning was a real bug?
- How long did the automatic analysis take, and what were its resource needs?
- How easy was it to prepare programs for analysis?

5.2 Test Setup

Testing was performed on a dual-processor 550MHz Pentium III Xeon machine running the Linux 2.2.16-3smp kernel. Only one processor was used in testing. The machine had 2GB of memory. Tools used in preparation and testing were gcc, version egcs-2.91.66; emacs, version 20.7.1; and PAM (Program Analysis Mode for emacs), version 3. Some programs were prepared (preprocessed) on an UltraSparc-based machine running Solaris 7 and gcc 2.95.2.

To test our system, we chose several widely-used daemons written in C that were likely to contain security vulnerabilities. We also included several programs with reported format string bugs in order to test the coverage (false negative rate) of our system. Two of these cases—mingetty [24] and mars_nwe [25]—are particularly interesting because hand audits had revealed potentially dangerous function calls, but owing to the difficulty of manual verification, no actual bugs had been reported. In some other cases, such as cfengine [35] and bftpd [4], we detected bugs that were unknown to us at the time of the experiment, but that we later discovered had already been known to others.

5.3 Results

Following is a brief description of the analysis results on some test samples:

cfengine: The first run gave many warnings; hotspot analysis led to a real format string vulnerability previously unknown to us. The vulnerability turned out to be known to others [35]. In addition, there were a few warnings unrelated to taint analysis.

muh: The first run generated many warnings. After looking at the hotspots and the list of unannotated functions, six library function wrappers were annotated with polymorphic types in the local prelude file. A subsequent run showed twelve warnings, one of which was a real vulnerability (known to others [22]).

bftpd: The hotspots from the first run guided us to mark one function with a polymorphic type. After this, there were two warnings, one of which was a bug of which we were not previously aware. We later found that this bug had already been discovered by others [4].

mars_nwe: In the first run, there were a few hundred warnings, but the hotspots suggested making two functions polymorphic. When this was done, there were no more warnings. Note that others had previously reported questionable function calls where the auditor was not able to determine whether the property could be exploited [25]; our tool gives strong evidence that they are not exploitable.

Name	Version	Description	Lines	Preproc.	Time	Warnings	Bugs
cfengine	1.5.4	System administration tool	24k	126k	28s	5	1
muh	2.05d	IRC proxy	3k	103k	5s	12	1
bftpd	1.0.11	FTP server	2k	34k	2s	2	1
mars_nwe	0.99	Novell Netware emulator	21k	73k	21s	0	0
mingetty	0.9.4	Remote terminal control utility	0.2k	2k	1s	0	0
apache	1.3.12	HTTP server	33k	136k	43s	0	0
sshd	2.3.0p1	OpenSSH ssh daemon	26k	221k	115s	0	0
imapd	4.7c	Univ. of Wash. IMAP4 server	43k	82k	268s	0	0
ipopd	4.7c	Univ. of Wash. POP3 server	40k	78k	373s	0	0
identd	1.0.0	Network identification service	0.2k	1.2k	3s	0	0

Figure 6: Results of our experimental evaluation of the tool. The size of the program is measured unpreprocessed and preprocessed, in thousands of lines of code, excluding comments. Time is the wall clock time for a run of `cqual`. Warnings counts the total number of warnings issued by `cqual` after the GUI's recommendations were followed, and Bugs is the number of real vulnerabilities found.

mingetty: No warnings issued. As with mars_nwe, an auditor had previously reported a suspicious function call of unknown exploitability [24]; `cqual` made it easy to verify that these calls were safe.

apache: In the first two runs, there were some warnings due to inconsistent declarations in the prelude and the source files. After these were set right, no warnings were issued.

sshd: The first run suggested annotation of twelve vararg functions. After these were made polymorphic, there were no more warnings.

imapd, ipopd, and **identd:** No warnings issued.

5.4 Evaluation

Our system reliably found all known bugs in the tested programs, including bugs we were not aware of when we applied our tool. Code without known bugs, and which was later examined by hand and found to be unlikely to contain bugs, yielded few false positives. Indeed, in our tests all false positives occurred in programs with actual bugs once varargs functions were annotated. The heuristics described in Section 3.2 were extremely useful in such cases. The annotation of varargs functions flagged by `cqual` was usually enough to remove most false positives. The hotspots pinpointed the actual bug in most cases. The GUI was invaluable in the analysis, making quick detection and correction of bugs possible. The source of most bugs was found within a few minutes of manual inspection of unfamiliar code. Thus, our experience shows that false positives—a common drawback of many tools based on static analysis—do not seem to be a problem in our application.

The automated analysis usually took less than a minute, and never more than ten minutes. The manual effort required for each program was usually within a few tens of minutes.

Preparation of the programs for analysis typically took between thirty and sixty minutes each. Note that we were not familiar with the layout and particular structure of the source code for any of the test programs. Preparation consisted of modifying the build process to output preprocessed, filtered source. In practice this could be more systematically added to the build process.

In summary, we evaluated our tool on a number of security-sensitive applications, demonstrating the ability of our tool to find security holes that we were not previously aware of. We feel that this validates the power of our approach.

6 Related Work

Lexical Techniques. `pscan` [15] is a simple tool for automatically scanning source code for format string vulnerabilities. `pscan` searches the input source code for lexical occurrences of function calls syntactically similar to, e.g., `sprintf(buffer, variable)`. Because `pscan` operates only on the lexical level, it cannot reason about the flow of values through the program and fails in the presence of wrappers around C libraries (see, e.g., Figure 1). `pscan` also cannot distinguish between safe calls when the format string is a variable and unsafe calls—it flags any call where a format string is non-constant.

Others have exploited lexical source code analysis to find security bugs [7, 38]. The main advantages of lexical analysis are that it is extremely fast, it can find bugs in non-preprocessed source files, and it is virtually language independent. However, because lexical tools have no knowledge of language semantics, many errors—such as those involving aliasing or non-local control paths—cannot be detected.

Taint Analysis. Our use of tainting, inspired by Perl's taint mode [32], bears some resemblance to a Biba integrity model [6] and thus is distantly related to previous work on enforcing information flow policies through typing [29, 39, 40]. However, because we do not have to deal with maliciously constructed code, we avoid the need to solve many of the most vexing challenges (e.g., covert channels) in enforcing information flow policies.

Type Qualifiers. The basic framework for type qualifiers, as presented in Section 2, is due to Foster et al. [19] and has been used to build Carillon, a tool for finding Y2K bugs in C programs [16]. As described in Section 4, we developed several refinements to make tainting analysis practical: improved handling of casts and variable-argument functions; the notation for polymorphic type signatures; and the improved user-interface. However, the one feature present in previous tools that is missing from our system is automated type inference of polymorphic types for all functions. We are planning to incorporate polymorphic recursion [33] in the future to remedy this.

Static Bug Detection. Many authors have noted that static analysis can be a useful tool for detecting bugs. For instance, LCLint [18] uses dataflow analysis to search for common errors in C programs; Engler et al.'s Meta-level Compilation [17] statically simulates the behavior of a user-defined finite state machine and has been successful at finding many new bugs; and the Extended Static Checking system (ESC) [26] uses theorem proving to verify the validity of annotated Java source code.

These systems have been very successful at detecting many common bugs. However, they are not well suited to detecting format string vulnerabilities, for two reasons. First, they focus primarily on local properties, whereas format string vulnerabilities often arise due to global mishandling of strings. Second, many of them (e.g., ESC and, to a lesser degree, LCLint) require extensive annotations from the user, which we would like to avoid. Our type-based techniques address these challenges directly.

Run-time Techniques. Another defense against format string vulnerabilities is to dynamically prevent exploits through appropriate modifications to the C runtime [3], compiler, or libraries. `libformat`, a library designed to halt execution of any program that might be susceptible to a format string bug, follows this approach: it intercepts calls to `printf`-like functions and aborts the application if the format string specifier contains `%n` and the format string is in a writable portion of the address space [34]. However, this approach is fragile, since the `libformat` mechanism must be kept in perfect synchronization with the `libc` implementation of all `printf`-like functions.

FormatGuard, a compiler modification, injects code to dynamically check and reject all `printf`-like function calls where the number of arguments does not match the number of "`%`" specifiers [13]. Of course, only applications that are re-compiled using FormatGuard will benefit from its protection. Also, one technical shortcoming of FormatGuard is that it does not protect user-defined wrapper functions (see, e.g., Figure 1).

Moreover, a common limitation of both `libformat` and FormatGuard is that programs with format string vulnerabilities remain vulnerable to denial of service attacks. Nonetheless, an important advantage of these runtime techniques is that they are cheap and require almost no human intervention. Thus, we feel that run-time and static measures are both useful and complement each other well.

7 Conclusions

We have described a tool for automated detection of format string vulnerabilities in legacy source code. We have shown that our tool has very low false positive and false negative rates and is useful in practice at detecting even security holes that were unknown to us. Therefore, we feel that our work represents a strong step toward a usable bug-detection system.

The key technique we exploit is type qualifier inference, applied to the problem of static taint analysis. This approach allowed us to scale to large programs with hundreds of thousands of lines of code and to present an intuitive user interface to the programmer. Consequently, we conjecture that these techniques may find use in future applications as well.

Acknowledgments

We would like to thank David Gay for developing the C front end we used, Chris Harrelson and Henning Niss for developing PAM, Martin Elsman for his work on Carillon, the precursor to `cqual`, and the anonymous referees for their comments.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Lamagra Argamal. "ftpd: the advisory version." `bugtraq` mailing list, 23 June 2000. `http://www.securityfocus.com/archive/1/66544`.

[3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. "Efficient Detection of All Pointer and Array Access Errors." In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994.

[4] Christophe Bailleux. "Asynchro," `bugtraq` mailing list, 8 December 2000. `http://www.securityfocus.com/archive/1/149977`.

[5] D.J. Bernstein, "Re: Logging question." `qmail` mailing list, 13 September 1996. `http://www.ornl.gov/its/archives/mailing-lists/qmail/1996/12/msg00314.html`.

[6] K. J. Biba. "Integrity considerations for secure computer systems." Technical Report ESD-TR-76-372, MTR-3153, The MITRE Corporation, USAF Electronic Systems Division, Bedford, MA, April 1977.

[7] M. Bishop and M. Dilger. "Checking for Race Conditions in File Accesses." *Computing Systems*, 9(2):131–152, Spring 1996.

[8] CERT Advisory CA-2000-13. "Two Input Validation Problems in FTPD." 7 July 2000.

[9] CERT Advisory CA-2000-17, "Input Validation Problem in rpc.statd." 18 August 2000.

[10] CERT Incident Note IN-2000-10, "Widespread Exploitation of rpc.statd and wu-ftpd Vulnerabilities." 15 September 2000.

[11] CERT Advisory CA-2000-22. "Input Validation Problems in LPRng." 12 December 2000.

[12] Satish Chandra and Thomas W. Reps. "Physical Type Checking for C." In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Toulouse, France, September 1999. , pages 66–75.

[13] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. "FormatGuard: Automatic Protection From printf Format String Vulnerabilities." This volume.

[14] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[15] Alan DeKok. "PScan: A limited problem scanner for C source files." Available at `http://www.striker.ottawa.on.ca/~aland/pscan`.

[16] Martin Elsman, Jeffrey S. Foster, and Alexander Aiken. "Carillon—a System to Find Y2K Problems in C Programs." Available at `http://www.cs.berkeley.edu/Research/Aiken/carillon/doc.ps.gz`.

[17] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions." In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[18] David Evans. "Static Detection of Dynamic Memory Errors." *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia, Pennsylvania, May 1996, pages 44–53.

[19] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. "A Theory of Type Qualifiers." In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*, Atlanta, Georgia, May 1999.

[20] Christopher Harrelson. "Program Analysis Mode." `http://www.cs.berkeley.edu/~chrishtr/pam`.

[21] Fritz Henglein and Jakob Rehof. "The Complexity of Subtype Entailment for Simple Types." In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*, Warsaw, Poland, July 1997, pages 352–361.

[22] Maxime Henrion. "muh IRC bouncer remote vulnerability." FreeBSD Security Advisory FreeBSD-SA-00:57. `http://www.securityfocus.com/advisories/2741`.

[23] Maxime Henrion. "format string bug in muh." `bugtraq` mailing list, 09 September 2000. `http://www.securityfocus.com/archive/1/81367`.

[24] Jarno Huuskonen. "Some possible format string errors." Linux Security Audit Project mailing list, 25 September 2000. `http://www2.merton.ox.ac.uk/~security/security-audit-200009/0118.html`.

[25] Jarno Huuskonen. "syslog(prio, buf) in mars_nwe." Linux Security Audit Project mailing list, 27 September 2000. `http://www2.merton.ox.ac.uk/~security/security-audit-200009/0136.html`.

[26] K. Rustan M. Leino and Greg Nelson. "An Extended Static Checker for Modula-3." In Kai Koskimies, editor, *Compiler Construction: 7th International Conference, CC'98*, volume 1383 of Lecture Notes in Computer Science, pages 302-305. Springer, April 1998.

[27] Robert Lemos. "Internet worm squirms into Linux servers." *Special to CNET News.com*, 17 January 2001. `http://news.cnet.com/news/0-1003-200-4508359.html`.

[28] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, July 1991.

[29] Andrew C. Myers and Barbara Liskov. “Protecting Privacy using the Decentralized Label Model.” *ACM Transactions on Software Engineering and Methodology*, 9(4), April 2001.

[30] Tim Newsham. “Format String Attacks.” Guardent, Inc. September 2000. `http://www.guardent.com/docs/FormatString.PDF`.

[31] Robert O’Callahan and Daniel Jackson. “Lackwit: Practical Program Understanding With Type Inference.” In *Proceedings of the 19th International Conference on Software Engineering*, pp. 338-348, Boston, Massachusetts, May 1997.

[32] Perl Security. `http://www.perl.com/pub/doc/manual/html/pod/perlsec.html`.

[33] Jakob Rehof and Manuel Fähndrich. “Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability.” In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, London, United Kingdom, January 2001.

[34] Tim J. Robbins. libformat. Available at `http://box3n.gumbynet.org/˜fyre/software`.

[35] Pekka Savola. “Very probable remote root vulnerability in cfengine.” `bugtraq` mailing list, 1 October 2000. `http://www.securityfocus.com/archive/1/136751`.

[36] Michael Siff, Satish Chandra, Thomas Ball, Thomas Reps, and Krishna Kunchithapadam. “Coping With Type Casts in C.” In *ACM Conference on Foundations of Software Engineering (FSE)*, September 1999.

[37] Bjarne Steensgaard. “Points-to Analysis in Almost Linear Time.” In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996, pages 32–41.

[38] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. “ITS4: A Static Vulnerability Scanner for C and C++ Code.” In *16th Annual Computer Security Applications Conference (ACSAC 2000)*, December 2000.

[39] D. Volpano, G. Smith, and C. Irvine. “A sound type system for secure flow analysis.” *Journal of Computer Security*, 4(3):1–21, 1996.

[40] D. Volpano and G. Smith. “A type-based approach to program security.” *Proceedings of TAPSOFT’97, Colloqium on Formal Approaches in Software Engineering*.

[41] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. “A First Step Toward Automated Detection of Buffer Overrun Vulnerabilities.” In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, California, February 2000.

[42] Larry Wall, Tom Christiansen and Jon Orwant. *Programming Perl*, 3rd Edition. July 2000. O’Reilly & Associates.

[43] “WuFTPD: Providing remote root since at least 1994,” `bugtraq` mailing list, June 23, 2000, `http://www.securityfocus.com/archive/1/66367`.

A Proof of Theorem 4.1

Theorem A.1 *Let $(P, \leq)$ be any finite partial order. Let $(2^{\mathbb{N}}, \subseteq)$ be the lattice of subsets of $\mathbb{N}$ with the set inclusion ordering. Then there exists a mapping $\phi : P \rightarrow 2^{\mathbb{N}}$, such that $\forall x, y \in P, x \leq y \iff \phi(x) \subseteq \phi(y)$ and $\phi(x)$ is a finite subset of $\mathbb{N}$ for all $x \in P$.*

Proof : We prove the theorem by induction on $|P|$.
Base Case : Let $|P|$ = 1. Then the claim trivially holds.
Induction Hypothesis : Let the claim hold for all P such that $|P| \leq k$.
Induction Step : $|P| = k + 1$.
Let $(P, \leq)$ be a partial order such that $|P| = k + 1$. Since P is finite, P has a minimal element, say a. Consider the partial order $(P \setminus \{a\}, \leq)$. Clearly this is a partial order and $|P \setminus \{a\}| = k$. Hence by induction hypothesis, there exists $\phi : P \setminus \{a\} \rightarrow 2^{\mathbb{N}}$, such that $\forall x, y \in P \setminus \{a\}, x \leq y \iff \phi(x) \subseteq \phi(y)$ and $\phi(x)$ is a finite subset of $\mathbb{N}$ for all $x \in P \setminus \{a\}$. Let $n = \max_i \{i \in \cup_{x \in P \setminus \{a\}} \phi(x)\}$. Define $\phi' : P \rightarrow 2^{\mathbb{N}}$ as follows.

$$\phi'(x) = \begin{cases} \{n+1\} & \text{if} x = a \\ \phi(x) \cup \{n+1\} & \text{if} x \neq a \text{ and } x \leq a \\ \phi(x) & \text{otherwise} \end{cases}$$

Since a was chosen to be a minimal element, the only relations involving a are of the form $a \leq x$, and for these, by definition, $\phi'(a) = \{n+1\} \subseteq \phi(x) \cup \{n+1\} = \phi'(x)$. For all x such that $a \not\leq x$, we have $\phi'(a) = \{n+1\} \not\subseteq \phi(x)$ by choice of n. For relations not involving a, the show below that the set containment relations are preserved. Let $\phi(x) \subseteq \phi(y)$. Since $\phi'(y) \supseteq \phi(y)$, the case when $\phi'(x) = \phi(x)$ is trivial. So assume $\phi(x) \subseteq \phi(y)$ and $\phi'(x) = \phi(x) \cup \{n+1\}$. This implies that $a \leq x$, and $x \leq y$, and therefore $a \leq y$. Thus $\phi'(y)$ would be defined as $\phi(y) \cup \{n+1\}$, and hence $\phi'(x) \subseteq \phi'(y)$.Thus the induction step holds. ■