Detecting Errors with Configurable Whole-program Dataflow Analysis

Samuel Z. Guyer
Dept. of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
sammy@cs.utexas.edu

Emery D. Berger
Dept. of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
emery@cs.utexas.edu

Calvin Lin
Dept. of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
lin@cs.utexas.edu

ABSTRACT

In this paper, we present an automatic compiler-based approach for detecting programming errors. Our system uses a configurable and scalable whole-program dataflow analysis engine driven by high-level programmer-written annotations. We show that our system can automatically detect a wide range of programmer errors in C programs, including improper use of libraries, information leaks, and security vulnerabilities. We show that the aggressive compiler analysis that our system performs yields precise results. Further, our system detects a wide range of errors with greater scalability than previous automatic approaches. For one important class of security vulnerabilities, our system automatically finds all known errors in five medium to large C programs without producing any false positives.

1. Introduction

As software has become more complex and more pervasive, the impact of programming errors has grown dramatically. Approaches to detecting and correcting these errors include code reviews [11], testing, and formal verification [9, 15]. Code reviews and testing are often effective at finding superficial errors but are neither reliable nor exhaustive. Formal verification guarantees program correctness but requires the creation of a complete specification of the program, a difficult and often impractical undertaking.

Recent work has focused on automatic systems designed to detect errors with minimal or no manual intervention. Such systems include lexical techniques [20], enhanced type systems [19, 22], and compiler-based approaches that use finite-state machines [10] or model checking [1]. All of these approaches have problems that limit their usefulness. Lexical approaches detect only superficial errors. The type-based approach requires manual intervention and generates numerous false positives, placing the burden on the programmer to determine which errors are real. Previous compiler-based approaches suffer from state explosion problems that limit their scope to single procedures or to relatively small programs.

The key contribution of this paper is an interprocedural compiler-based approach to detecting errors that improves on previous work by simultaneously providing both precision and scalability. Our system treats error detection as a dataflow analysis problem and solves this problem using a configurable whole-program dataflow analysis engine. Programmers or library writers write one set of *annotations* for each class of error, and our system then detects these errors in unmodified C programs with no manual intervention.

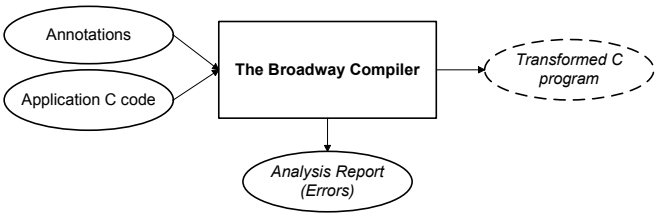


Figure 1: Overview of the Broadway system.

Our system detects errors that are generated at or propagate to procedure calls. One important error in this class that our system detects is format string vulnerabilities [3, 16]. In contrast to previous approaches, which require substantial manual program analysis and which produce false positives [19], we show that our system quickly detects all instances of format string vulnerabilities in five C programs and produces no false positives. These C programs range in size from one thousand lines to 45 thousand lines. Our system can also pinpoint the source lines and calling contexts leading to the improper uses of sockets, double locking, and privacy leaks.

We believe that the aggressive compiler analysis that our system employs is critical for effective error detection. To find errors that propagate through pointers, we use dependence and pointer analysis. To avoid excessive false positives and to capture more errors, we perform interprocedural, context and flow-sensitive analysis. Despite the apparent cost of this approach, we show that our system is more scalable than previous work. We also provide a simple method for throttling context-sensitivity that allows our system to trade precision for faster analysis.

The remainder of this paper is organized as follows. We provide an overview of the Broadway system in Section 2. We contrast our system with previous work in Section 3. In Section 4, we summarize our language for describing errors and explain how our system performs domain-specific analysis. In Section 5, we describe the wide range of errors that our system is capable of detecting and provide empirical results of our analysis. We discuss the scalability of our approach in Section 6. We address future directions in Section 7, and conclude in Section 8.

2. The Broadway System

The Broadway compiler takes C source files and a set of annota-

Submitted to PLDI 2002 Berlin, Germany

tions as input. Figure 1 shows the overall architecture of our system. Guided by the annotations, Broadway analyzes the C code, reports on its analysis and optionally generates a transformed C program as output. Broadway performs a whole program analysis on the code, referring to the annotations at relevant procedure call sites. The annotations allow properties to be associated with the inputs and outputs of procedures. The annotation writer can specify error conditions as well as code transformations.

We originally designed the Broadway compiler to support library-level optimization [12, 13], building on work in partial evaluation [2, 6] and abstract interpretation [7, 14] to support domain-specific analysis and optimization. In particular, our compiler focuses on supporting two types of domain-specific optimizations. First, it extends many of the traditional optimizations passes, such as dead-code elimination and constant propagation, to apply to library routines. These optimizations can have a significant impact on performance because they modify or eliminate entire library calls. Second, it provides the tools necessary to identify and exploit special-case library routines. Libraries often contain many routines that perform essentially the same computation, but with different assumptions about inputs and with different performance characteristics. The Broadway compiler can use such information captured by annotations to specialize application programs. In this work, we show how Broadway can use such information to statically detect errors.

3. Related Work

Compilers have always performed error checking of some sort. However, these checks have traditionally been limited to the semantics of the base programming language. Recent work extends error checking to high-level semantics that are not built into the programming language.

Two recent papers have focused on using type systems to express high-level programming constraints. Shankar et al. present a system for detecting format string vulnerabilities using type inference [19]. In this approach, two new type qualifiers, "tainted" and "untainted," are introduced to the C language and added to the signatures of the standard C library functions. These qualifiers denote whether associated variables have possibly received input from an untrusted source. Type inference is performed by an extensible type qualifier framework, which derives a consistent assignment of these type qualifiers to string variables. Security errors are reported as type mismatches.

The inherent limitations of static typing can cause this approach to produce large numbers of false positives. For example, the type of a variable cannot change, even though operations on it can change its taintedness. Similarly, a procedure can only have one type signature, even though the application may call it many times in different contexts that have different combinations of tainted and untainted arguments. Shankar et al. address this problem by adding a form of polymorphism to the type system. However, this solution can require understanding and annotating the application program itself in order to get acceptable precision. Our approach recognizes that taintedness is a property of the state of an object, not its type. Flow sensitivity and context sensitivity keep this information distinct at different points and call sites in the program.

The Vault system provides programming language support for explicitly expressing constraints on the use of domain-specific resources such as files and sockets [8]. In order to use Vault, the programmer first translates the input program into the Vault programming language, adding resource constraints where necessary. The language uses *type guards* to control when a particular operation is valid for a resource. Vault avoids some of the problems of type qualifiers by introducing *keys* that track flow-sensitive conditions. However, the system cannot reconcile conflicting conditions at control-flow merge points. Our approach avoids this problem by using lattices to represent these error conditions, which allows us to specify precisely how information should be combined at merge points.

The MC system checks for errors in operating system code using programmer-written checkers based on state machines [10]. A checker consists of a set of states and a set of syntax patterns that trigger transitions on the state machine. The compiler pushes the state machine down each path in the program and reports any error states that it encounters. While this approach has proven quite successful in finding errors, it has limitations. First, the scope of the analysis is limited to a single procedure at a time because the number of paths through a program can be extremely large. Second, since the analysis is syntax-driven, the compiler lacks deep information about the program semantics, such as dataflow dependences and pointer relationships. Our approach is more scalable and utilizes deeper program information.

The SLAM Toolkit approach is similar to MC but is more rigorous and more powerful [1]. SLAM includes a pointer analyzer and can check programs interprocedurally. Input to the toolkit consists of the C program to analyze, along with a separate specification that describes the safety properties to test. The toolkit first generates an abstraction of the program that represents its behavior only with respect to the properties of interest. It then uses a model checker to perform path-sensitive analysis on the abstracted program. Our system differs from SLAM in many ways. While Broadway does not currently describe all types of constraints that SLAM can, it also does not require any formal specifications. Further, because it is not path-sensitive, it is more scalable than SLAM, as we discuss in Section 6.

Our work differs from previous work in both expressiveness and scalability. While it is constrained to recognizing errors that propagate via dataflow, we show that this class includes many important errors, including privacy leaks and security vulnerabilities. Because we do not rely on static typing, we report few false positives. Our use of interprocedural analysis allows us to find errors that state-machine based approaches cannot because of scalability problems. Finally, while we cannot capture as extensive a range of safety properties, we avoid the inherent state space explosion problem of path-sensitive analyzers.

4. Errors as Dataflow Problems

In this section, we describe how to use the Broadway compiler to detect errors. We cast error detection as a dataflow analysis problem, which our compiler solves using a configurable dataflow analysis engine. Analysis problems are specified by defining flow values and transfer functions that track error-related properties through the target program. Our system avoids much of the difficulty of defining new dataflow analysis problems by providing a simple annotation language that can express a wide range of useful problems [13]. Other approaches for detecting errors support much more complex specifications, but our system is effective at finding errors because of its aggressive analysis framework.

4.1 Annotation Language

We have described our annotation language in detail elsewhere [13]. Here, we summarize its main features. Our annotation language focuses on describing the behavior of library routines and the domain-specific abstractions that they represent. This focus comes from the Broadway compiler's original goal of optimizing the use of software libraries. This focus is important for error checking because many interesting and difficult-to-detect programming errors arise from the unsafe or incorrect use of software libraries. Indeed, most

libraries encapsulate some sort of domain-specific semantics, such as files, sockets, and locks, and since these abstractions exist beyond the semantics of the base language, they typically receive no semantic checking support from conventional compilers.

Defining Analysis Problems

Our annotation language contains both basic dependence annotations and more advanced analysis annotations. The *basic* annotations provide a summary of each library routine's behavior with respect to pointer analysis and dependence information. To describe pointer relationships at the beginning and end of the procedure, Broadway provides `on_entry` and `on_exit` annotations. Both annotations use the `-->` operator to indicate that one object points to another object. The `access` and `modify` annotations simply list the uses and defs of the procedure. Figure 2 shows the basic annotations for the `strcpy()` string copying function. The annotations indicate that both parameters to `strcpy()` point to other objects, that these other objects are named `src_string` and `dest_string`, and that the routine reads the `src_string` object and modifies the `dest_string` object.

```
procedure strcpy(dest, src)
{
  on_entry { src --> src_string
             dest --> dest_string }
  access { src_string }
  modify { dest_string }
}
```

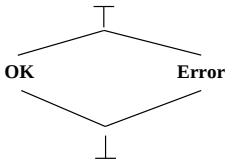
Figure 2: Basic annotations describe pointer and dependence behavior.

The *advanced* annotations describe the dataflow analysis problems to be solved by the framework. To define a new dataflow analysis, the library annotator specifies a simple lattice using the `property` annotation, along with a set of transfer functions, one for each library routine. The transfer functions are specified using the `analyze` annotation, which summarizes the effects of each library routine on the lattice values. Currently, we limit the structure of the lattices to a simple hierarchy of named categories (much like an enumerated type), from which the compiler automatically infers the meet function. In the annotations, We can compare lattice values using two operators: `is-exactly` tests for an exact match, and `is-atleast` tests the lattice greater-or-equal function.

The annotation shown in Figure 3 defines a minimal error detection lattice called `State` that has only two elements, `OK` and `Error`. We can then augment the annotations for `strcpy()` with an `analyze` annotation that tells the compiler that the function passes the error state from the source string to the destination string. This example also highlights an important benefit of including pointer information in the annotations: We can associate the error state with the actual string contents, not just with the surface variables. With this capability, we can follow the states of these objects as they are passed throughout the program, regardless of the pointers that refer to them.

Using Analysis Results

The annotations identify error conditions by testing the analysis result for flow values that represent incorrect states. The *report* annotation consists of a conditional expression and a message string. At each library routine callsite, the compiler emits the message for all the calling contexts in which the condition is true. The message string can contain special tokens that the compiler replaces with callsite-specific information. The special tokens include the complete call stack, the line number and source file, the names of the actual arguments, and the flow values of the objects.




```
property State : { OK, Error }

procedure strcpy(dest, src)
{
  on_entry { src --> src_string
             dest --> dest_string }
  access { src_string }
  modify { dest_string }

  analyze State {
    if (src_string is-exactly OK)
      dest_string <- OK
    if (src_string is-exactly Error)
      dest_string <- Error
  }
}
```

Figure 3: The `strcpy()` function passes error states from the source to the destination.

Figure 4 shows a report annotation for the `puts()` system call. It prints an error message whenever the input string is in the `Error` state. It also provides the location of the call in the source and the name of the argument.

```
procedure puts(s)
{
  on_entry { s --> the_string }
  access { the_string }

  report
    if (State : the_string is-exactly Error)
      "Error at " ++ @callsite
      ++ ": Argument " ++ [ s ]
      ++ " is in error state.\n";
}
```

Figure 4: The `strcpy()` function passes error states from the source to the destination.

4.2 Analysis Framework

For each property specified in the annotations, the dataflow analysis framework associates values from the lattice with objects in the application code. Objects in our representation are fine grained, including local and global variables, structures and structure fields, and heap allocated memory. Heap objects are distinguished by the full context path to their allocation site. At a library call, the compiler tests the current lattice values associated with the actual arguments, and then updates those states according to the transfer functions. The annotations can also direct the framework to analyze a property forwards or backwards in the target program. In Section 5 we give examples of forward and backward analyses and compare the kinds of problems they solve.

Our framework uses a flow-sensitive, context-sensitive, interprocedural dataflow analysis. The compiler first builds factored use-def chains for all objects in the target program. This pass relies on the basic annotations to precisely describe the dependence and

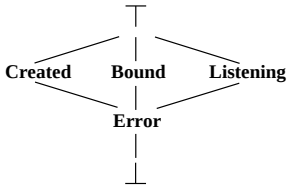
pointer behavior of each library routine. The compiler then performs all of the library-specific analyses in a single pass, using the use-def chains to efficiently propagate flow values. By analyzing all of the properties together, the transfer functions for one property can test the current state of other properties. Finally, the compiler visits each library routine call site and invokes any reports that are applicable.

5. Detecting Errors with Broadway

In this section, we describe a number of errors that Broadway can detect. These include the invalid uses of libraries and double locking of mutexes, as well as more complicated errors, including privacy leaks and security vulnerabilities. We provide examples of erroneous code and we describe the dataflow analyses that enable Broadway to detect such errors. We analyze a number of actual and synthetic applications and present results including the number of actual errors that our system detects, the number of false positives that it reports, and the accuracy of its error reporting. We ran our system under Linux on a 2GHz Pentium 4 system with 512 megabytes of RAM. The Broadway system was compiled with gcc version 2.95.2, at optimization level `-O4`.

5.1 Invalid Library Usage

Library interfaces often contain implicit constraints on the order in which their routines may be called. File access rules are one example of this kind of usage constraint. A program can only access a file in between the proper open and close calls, and the kind of access (reading or writing) must match the mode in the open call. A more sophisticated analysis is required to model the semantics of UNIX sockets. The proper sequence of UNIX calls to create a server socket is `socket()`, `bind()`, and `listen()`, followed by `accept()` to accept a connection. Figure 5 shows how we can model this constraint by marking each socket variable according to its progress through this sequence.




```
property SocketState :
 { Error { Created , Bound , Listening }}
```

Figure 5: Lattice for the steps in creating a socket.

In this model, if a socket has conflicting states on different paths, then it goes to the error state. Figure 6 shows the full annotations for the `listen()` function. When the input socket is bound to a name, the annotations transition it to the listening state. Any other call to `listen()` is invalid and our system will print an error message at the offending line.

5.2 Simple Deadlock Detection

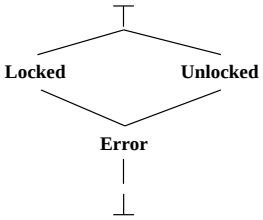
While deadlock detection is an NP-hard problem, we can use Broadway to detect a number of situations that lead to deadlocks, including double locking. Because the Broadway compiler performs interprocedural analysis, we can find instances of double locking that are widely separated in the program source. Note that we make no attempt to model the concurrent semantics of threads.

In the pthreads library, locking is controlled by a mutex variable of type `pthread_mutex_t`. We use the annotations to track

```
procedure listen(socket, backlog)
{
  analyze SocketState {
    if (socket is-exactly Bound)
      { socket <- Listening }
    default
      { socket <- Error }
  }
  report
    if ( ! SocketState : socket
                         is-exactly Bound)
      "Error at " ++ @context ++ "socket "
      ++ [ socket ] ++ " not bound.\n";
}
```

Figure 6: Annotations for `listen()` that ensure that `bind()` has been called.

whether a particular mutex is locked or unlocked. Figure 7 shows the lattice for this analysis.




```
property Mutex : { Error { Locked,
                           Unlocked }}
```

Figure 7: Lattice for lock states.

The library functions `pthread_mutex_lock()` and `pthread_mutex_unlock()` lock and unlock the given mutex object. Figure 8 shows how the locking function can be annotated to test for double locking.

```
procedure pthread_mutex_lock(mutex_ptr)
{
  on_entry { mutex_ptr --> mutex }
  analyze Mutex {
    if (mutex is-exactly Unlocked)
      { mutex <- Locked }
  }
  report if (Mutex : mutex is-exactly Locked)
    "Error at " ++ @context ++ ": Mutex " ++
    [ mutex_ptr ] ++ " is already locked.\n";
}
```

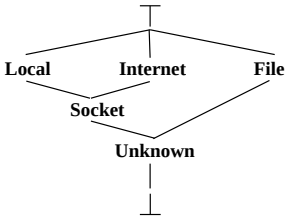
Figure 8: Annotations that check for double locking.

We applied the analysis we describe in Section 5.2 to a synthetic program that tests `timer_getoverrun()`, a POSIX threads function supported by the linuxthreads package for the GNU C library. The implementation of this function in the GNU C library version 2.1.2 contains a double locking error which, though simple, nonetheless escaped manual detection and testing. Our system finds this bug in .05 seconds, reporting no false positives, and issues the following error report:

```
Error at pthread_mutex_lock (timertest.c:16)
in timer_getoverrun (timertest.c:24 ):
Mutex timer_getoverrun::__TE2 is already locked.
```

5.3 Information Leaks

Our dataflow analysis engine allows us to track the state of objects throughout the program. For example, we can use such analysis to verify that a program does not reveal private information to the world. We consider information originating on a local disk to be private, and information coming from the network to be public. Using our system, we can ensure that a program does not transmit private information over the network[1].




```
property FDKind :
  { Unknown { Socket { Internet,
                       Local },
              File } }
```

Figure 9: The `property` keyword introduces a new lattice.

In Figure 9, we define a lattice that identifies the kind of device associated with a file descriptor. The `FDKind` property takes advantage of the lattice structure to minimize information loss at control-flow merge points. For example, if we have an if-else statement that creates an Internet socket on one branch and a Local socket on the other, the compiler can at least infer that the resulting file descriptor is some kind of socket.

We augment this model by marking the data as *public* or *private* depending on its source. We can then use the compiler to test information flow constraints. Figure 10 shows the annotations for the `write()` function that report this condition. Notice that we also report a warning when the type of socket cannot be determined.

```
procedure write(fd, buffer_ptr, size)
{
  on_entry { buffer_ptr --> buffer }

  report
    if ((FDKind : fd is-exactly Internet) &&
        (FDKind : buffer is-exactly File))
    "Error at " ++ @context ++
    ": File data is sent to the Internet.\n";

  report
    if ((FDKind : fd is-atleast Socket) &&
        (FDKind : buffer is-exactly File))
    "Warning at " ++ @context ++
    ": File data may be sent to the Internet.\n";
}
```

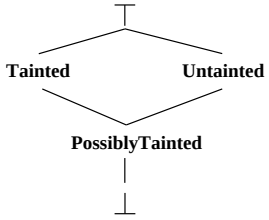
Figure 10: Annotation for `write()` to report when data from the file system is sent out on the network.

5.4 Security Vulnerabilities

We can expand on the information flow analysis described above to detect an entire class of errors that arise when data from an untrusted source (such as the network or user) reaches a vulnerable part of the program. Following the terminology introduced for the Perl programming language [21] and used by Wagner [19], we consider data to be *tainted* when it comes from an untrusted source. We then mark vulnerable functions as requiring their input to be *untainted*.

Using taintedness analysis, we can detect format string vulnerability errors [3, 16, 19], which arise when untrusted data is used as a format string argument for functions like `printf()`. Certain C functions take strings as arguments that define how to format output. For instance, `"(%s)"` is a format string that `printf()` can use to print a string argument surrounded by parentheses. A program contains a format string vulnerability when it reads string data from an untrusted source, such as the network, and that data ends up as a format string argument. By carefully manipulating input strings, a hacker can access the stack and execute arbitrary code. The format string vulnerability has been identified in several widely used software packages and has been the subject of numerous CERT advisories [4, 5].

We now describe in detail how we define annotations that provide taintedness analysis for detecting these vulnerabilities. Our formulation of the taintedness analysis starts with a definition of the lattice, shown in Figure 11.




```
property Taint :
  { PossiblyTainted { Tainted,
                      Untainted }}
```

Figure 11: The lattice for taintedness.

We include the `PossiblyTainted` value to capture conflicting states on different control-flow paths. For instance, it is possible to untaint data by setting all data in a string to zero with `memset()`.

Next, we annotate the standard C library functions that produce tainted data. These include such obvious sources of untrusted data as `scanf()` and `read()`, and less obvious ones such as `readdir()` and `getenv()`. Figure 12 shows the annotations for the `read()` system call. Notice that the annotations assign the `Tainted` property to the contents of the buffer rather than to the buffer pointer. This helps prevent false positives when a single pointer can point to both tainted and untainted data in the same program.

```
procedure read(fd, buffer_ptr, size)
{
  on_entry { buffer_ptr --> buffer }
  access { Disk }
  modify { buffer }
  analyze Taint { buffer <- Tainted }
}
```

Figure 12: The `read()` system call taints data in the buffer.

Taintedness is a property that can propagate across functions that manipulate the data. For example, if the program concatenates two

[1]Note that our notion of "information flow" is weaker than others that have been described in the literature [22], as we only track the flow of data in the program.

tainted strings, the result is also tainted. Therefore, we also annotate various string operations to indicate how they affect the taintedness of their arguments. Figure 13 shows the annotations for the `sprintf()` system call. Since this function takes a variable number of arguments, the compiler binds the third argument, `args`, to the rest of the actual arguments.

```
procedure sprintf(buffer_ptr, format_ptr, args)
{
  on_entry { buffer_ptr --> buffer
             format_ptr --> format
             args --> arg_contents }
  access { format, arg_contents }
  modify { buffer }
  analyze Taint { buffer <- arg_contents }
}
```

Figure 13: The `sprintf()` call can pass tainted data from the arguments to the buffer.

The annotations dereference the `args` argument and pass the taintedness state from those arguments to the buffer contents. Since `args` could represent multiple objects, the compiler meets together all of their lattice values. Thus, if any of them is tainted, the buffer contents become tainted.

Finally, we annotate all the standard C library functions that accept format strings (including `sprintf()`) to report when the format string is tainted. Figure 14 shows the annotations for the `syslog()` function, which is frequently the offender in format string vulnerabilities.

```
procedure syslog(priority, format_ptr, args)
{
  on_entry { format_ptr --> format
             args --> arg_contents }
  access { format, arg_contents }
  modify { Disk }

  report if (Taint : format is-exactly Tainted)
    "Error at " ++ @context
    ++ ": Argument " ++ [ format_ptr ]
    ++ " is tainted.\n";
}
```

Figure 14: The `syslog()` function reports a tainted format string.

The report annotation tests the contents of the format string for taintedness. If true, it generates an error message by replacing the various special tokens in the report with information about the actual call site. The compiler replaces the `@context` token with the full context path where the error occurred. It replaces the `[ format_ptr ]` with the name of the corresponding actual argument at the call site.

We applied taintedness analysis to five programs, using versions that are known to contain format string vulnerabilities. We found these programs, which are all daemons or systems applications, by combing through actual security advisories online. Table 1 summarizes the results of the experiments. For these results, we disabled all context-sensitivity except for the annotated library routines, which are always treated by our compiler in a context-sensitive manner. In all cases, the compiler properly identified the locations of the format string errors. In fact, for all cases where patches exist, the locations reported by our compiler exactly match the recommended patches. We found that the analysis time is roughly proportional to the square of the number of lines of code, except for lpd, which has an exceptionally bushy call graph.

Significantly, our system reported no false positives. By contrast, previous solutions based on type theory report false positives, even after significant manual program analysis and intervention; Shankar et al. applied their system to three of the same programs that we did, yielding 5 false positives for cfengine, 12 for muh, and 2 for bftpd [19].

5.5 Backwards Taintable Analysis

Using our approach, we can also define a backwards analogue to the taintedness analysis, which we call "taintable". In this analysis, we mark format strings as "Untaintable" and then track them backward to their sources. We report an error if data from an untrusted source can eventually be used as a format string. Figure 15 shows the output from the taintable analyses for the `muh` program. This analysis is useful in conjunction with the taintedness analysis to narrow down the exact cause of the vulnerability.

```
Argument muh_commands::s at fgets (muh.c:842)
    in muh_commands (muh.c:933)
    in read_client (muh.c:1057)
    in run (muh.c:1244)
 may end up as a format string.
```

Figure 15: Backwards analysis tells us the sources of tainted data.

Manual inspection of the programs yielded several interesting observations. First, taintedness is a property of the string contents, not the surface variables in the program. Without pointer analysis, we could not properly model this fact. Furthermore, taintedness can change during execution: at some points a buffer contains tainted data, at other times not. For example, if we have to associate taintedness with the surface variables, then the code in Figure 16 will result in a false positive on the second call to `syslog()`.

```
char * p;
p = malloc(100);
read(fd, p, 100);           // *p is tainted
syslog(5, p);               // Format string error
p = "Internal Message";     // *p not tainted
syslog(5, p);               // Okay
```

Figure 16: Flow-sensitivity is needed to avoid false positives.

Second, programmers often pass data throughout the program, including storing and retrieving it from data structures. For example, the `named` program in the BIND package reads host names and stores them in an array of request structures. Later, it traverses this data structure and extracts the names to service the requests. At this point, it can pass the hostname to `syslog()`, resulting in a format string vulnerability. Precise interprocedural pointer information properly tracks individual strings though such computations, preventing taintedness from spreading unnecessarily.

Finally, programs often define "wrapper" functions around standard system calls. These wrappers may have the same signature as the system call, but they perform some additional processing. Wrappers often serve as error handlers, and therefore they are frequently the culprits in format string vulnerabilities. Unfortunately, the application may call the wrapper in hundred or thousands of different places, making it difficult to discover the source of the problem. Using full context-sensitivity, we can often identify the exact call stack in which the error occurs.

Program	Package	Lines	Procedures	Time (mm:ss)	Known errors	Errors found	False positives
Format string vulnerabilities							
bftpd	bftpd 1.0.11	1,017	180	0:01	1	1	0
muh	muh 2.05c	5,002	228	0:06	1	1	0
named	BIND 4.9.4	25,820	444	1:11	1	1	0
lpd	LPRng 3.6.23	38,174	726	23:57	1	1	0
cfengine	cfengine 1.5.4	45,102	700	6:38	6	6	0

Table 1: A summary of our analysis results for format string vulnerabilities. Unlike previous approaches, our system finds all known errors with no false positives.

6. Scalability

Program analysis systems have to make a tradeoff between the scalability of their analysis and the quality of their results. For example, path-sensitive analysis provides extremely detailed results, but because the number of paths grows so fast, these techniques are limited to small programs or small parts of programs. The cost of context-sensitive interprocedural analysis also grows rapidly with size of the program, but provides a much larger scope of information. Our system attempts to strike a good balance between precision and scalability. We do this in several ways. First, we take a path-insensitive approach. Second, we provide a simple mechanism for controlling the degree of context-sensitivity in an analysis. Finally, because the annotations summarize information about library routines, we always treat library calls as context-sensitive without incurring the cost of analyzing the library source.

We control context-sensitivity with a threshold on the depth of the procedures in the program's call graph. The compiler treats procedures deeper in the call graph as context-sensitive, and it treats procedures shallower in the call graph as context-insensitive. We have explored more complex schemes, but this approach seems to capture the essential benefit of context-sensitivity. Calls to small, frequently reused procedures are treated independently, while large, high-level procedures are analyzed only once.

To detect errors in large programs, we start with a minimal level of context-sensitivity. Even in this mode, the compiler still finds all errors, but it does not report the exact call stack in which the errors occur, and it may report false positives, although these are rare. Once we know a program has bugs, we can rerun the analysis with a higher context-sensitivity threshold.

As precise as our analysis is, it is not as detailed as path-sensitive techniques, particularly the interprocedural path-sensitive analysis used by the SLAM Toolkit [1]. However, path-sensitive approaches suffer from a severe state explosion problem caused by the extremely large number of paths through a program. By comparison, context-sensitive analysis is relatively cheap. Consider the simple call graph depicted in Figure 17. The procedure `C` has two calling contexts, one through `A` and one through `B`. However, for each calling context, the number of *paths* to procedure `C` is the product of the number of paths through each intervening procedure. To see how these numbers compare in a real program, the format string bug in `muh` occurs six levels deep in the call graph, and has 19 calling contexts. However, there are nearly 140,000 possible paths through the program to that location. We believe that avoiding this state space explosion is vital in order to preserve scalability.

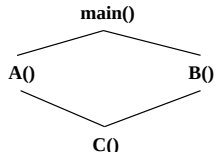


Figure 17: A commonly used procedure may have many calling contexts, but a huge number of paths.

7. Future Work

We are currently exploring several ways to help the library writer develop annotations. Exposing more information to Broadway about program objects (e.g., the size of array objects) would allow us to detect a broader class of errors without significantly complicating the job of the annotation writer. The compiler could assist in the generation and checking of the basic dependence annotations using the library source code. Other research has demonstrated techniques to automatically check a library routine summary against the implementation of the routine [17, 18]. The compiler could also help by checking the transfer function annotations. In our current implementation, it is possible to define analyses that do not converge if the transfer function is non-monotonic. Since our lattices are finite, and tend to be relatively small, the compiler could perform an offline check that exhaustively tests for non-monotonic behavior. In practice, however, none of our experiments have exhibited this problem because all of our lattices and analyses have been simple.

We also have not yet applied any performance tuning to Broadway. We plan to implement a number of optimizations to further improve Broadway's space and runtime performance.

8. Conclusions

In this paper, we have described Broadway, a compiler-based approach for performing automatic error detection. We show how Broadway can find a wide range of programming errors, and we apply it to find format string vulnerabilities in several C programs. In our solution, a user provides simple annotations that describe library routines, and then invokes our compiler to iteratively analyze an application with ever-increasing amounts of context-sensitivity.

Our results show that our aggressive analysis capabilities are necessary for precise error detection, but that they are scalable enough to be practical. Approaches that are more superficial cannot find errors as well, while more detailed approaches cannot handle large programs. We believe aggressive analysis is necessary for the following reasons. First, dependence and pointer analysis is required in order to find errors in the usage of libraries, which often use pointers and pointer-based data structures. Flow-sensitivity and context-sensitivity are important in order to avoid excessive false positives, because objects and error conditions change state during program execution, and library routines are called in many contexts with different implications for the existence of errors. Finally, interprocedural analysis is necessary because objects are typically passed throughout the program, often through multiple layers of the library or through library wrappers.

The cost of such analysis can be significant for large programs, but our procedure-oriented approach is more scalable than path-sensitive approaches. Our approach also provides a simple method

of throttling context-sensitivity that effectively trades precision for faster analysis. While our analysis remains expensive compared to traditional compiler analysis, taking tens of minutes for programs of up to 45,000 lines of code, we believe that the precision of our approach justifies its cost. This analysis time pales in comparison to the cost and effort of finding such errors manually, which can take days or weeks. Because it is so important to find errors and security violations, an automated approach is desirable, even if it needs to be run overnight.

In previous work, we have shown how to use Broadway to perform domain-specific optimizations. This paper shows that Broadway is also extremely effective at error detection. Together, this work illustrates the power of incorporating domain-specific information into the compilation process. We believe that this idea has broader implications for applying compiler technology to further improve software quality and programmer productivity.

9. References

[1] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *International SPIN Workshop on Model Checking of Software*, May 2001.

[2] A. Berlin and D. Weise. Compiling scientific programs using partial evaluation. *IEEE Computer*, 23(12):23–37, December 1990.

[3] James Bowman. Format String Attacks 101. http://rr.sans.org/malicious/format_string.php, October 2000.

[4] CERT Advisory CA-2000-22. LPRng can pass user-supplied input as a format string parameter to syslog() calls. December 2000.

[5] CERT Advisory CA-2001-02. ISC BIND 4 contains input validation error in nslookupComplain(). January 2001.

[6] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 1993 ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, 1993.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[8] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *SIGPLAN Conference on Programming Language Design and Implementation*, June 2001.

[9] E. Allen Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B — Formal Models and Semantics, pages 995–1072. North-Holland, 1990.

[10] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation*, October 2000.

[11] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, July 1976.

[12] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Second Conference on Domain Specific Languages*, pages 39–52, October 1999.

[13] Samuel Z. Guyer and Calvin Lin. Optimizing the use of high performance software libraries. In *Languages and Compilers for Parallel Computing*, pages 221–238, August 2000.

[14] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994. 527–629.

[15] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.

[16] Tim Newsham. Format String Attacks. http://www.guardent.com/docs/FormatString.pdf, September 2000.

[17] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *Proceedings of the International Conference on Compiler Construction*, April 2001.

[18] Radu Rugina and Martin Rinard. Design-driven compilation. In *Proceedings of the International Conference on Compiler Construction*, April 2001.

[19] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[20] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: a static vulnerability scanner for C and C++ code. In *16^{th} Annual Computer Security Applications Conference*, December 2000.

[21] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl, 3^{rd} Edition*. O'Reilly, July 2000.

[22] Steve Zdancewic and Andrew C. Myers. Secure infornation flow via linear continuations. In *Higher Order and Symbolic Computation*, 2001.