Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis

Walter Chang
walter@cs.utexas.edu

Brandon Streiff
bstreiff@mail.utexas.edu

Calvin Lin
lin@cs.utexas.edu

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

ABSTRACT

Current taint tracking systems suffer from high overhead and a lack of generality. In this paper, we solve both of these issues with an extensible system that is an order of magnitude more efficient than previous software taint tracking systems and is fully general to dynamic data flow tracking problems. Our system uses a compiler to transform untrusted programs into policy-enforcing programs, and our system can be easily reconfigured to support new analyses and policies without modifying the compiler or runtime system. Our system uses a sound and sophisticated static analysis that can dramatically reduce the amount of data that must be dynamically tracked.

For server programs, our system's average overhead is 0.65% for taint tracking, which is comparable to the best hardware-based solutions. For a set of compute-bound benchmarks, our system produces no runtime overhead because our compiler can prove the absence of vulnerabilities, eliminating the need to dynamically track taint. After modifying these benchmarks to contain format string vulnerabilities, our system's overhead is less than 13%, which is over 6× lower than the previous best solutions. We demonstrate the flexibility and power of our system by applying it to file disclosure vulnerabilities, a problem that taint tracking cannot handle. To prevent such vulnerabilities, our system introduces an average runtime overhead of 0.25% for three open source server programs.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Information Flow Controls*

General Terms

Reliability, Security, Verification

Keywords

Dynamic Data Flow Analysis, Security Enforcement, Static Analysis

CCS'08, October 27–31, 2008, Alexandria, Virginia, USA.

Traditional Tainted Data Attacks
Format String Attacks
SQL Injection
Command Injection
Cross-Site Scripting
Privilege Escalation
Other Security Problems
File Disclosure Vulnerabilities
Labeled Security Enforcement
Role-Based Access Control
Mandatory Access Control
Accountable Information Flow

Table 1: A sampling of the kinds of problems that our system can handle. Taint tracking can only handle the top set of problems.

1. INTRODUCTION

Many security attacks rely on the ability to pass carefully crafted hostile data to vulnerable portions of a target program. One method of preventing such attacks is to perform *dynamic taint analysis* [46, 37, 44, 16, 38, 10, 39, 48, 31]. Dynamic taint analysis marks untrusted data as tainted, tracks the flow of tainted data through the system, and checks that tainted data is not misused. Existing taint analysis research has largely followed one of two directions: (1) improving runtime overhead and (2) extending the generality of taint tracking.

Significant progress has been made in the first direction. Recent work [39] has reduced the extreme overheads of early systems [37] by performing taint-specific optimizations, but performance still remains a challenge, with average overheads of 260% for compute-bound applications [39]. Alternative techniques [48] can reduce the average overhead to 75% for compute-bound applications, but this result requires processor and OS-specific assumptions that are not generally applicable.

The second direction, which has received less attention, extends the generality of taint analysis, recognizing that taint tracking is a special case of data flow tracking. Whereas taint analysis tracks one bit of information, data flow tracking can track multiple bits of information and can combine the information in more flexible ways than taint analysis. More concretely, taint-based systems are limited to the top set of exploits listed in Table 1, while general data flow tracking, which is based on the more general notion of data flow analysis [29], can handle all of them. Such generality will become particularly important as developers move to memory-safe languages such as Java and C#, where the use of taint tracking to enforce secure control flow is not needed.

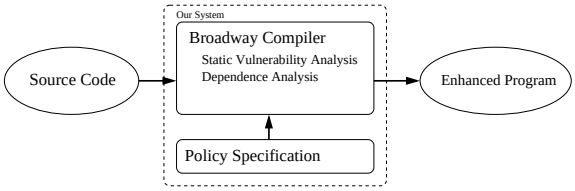

Figure 1: The overall structure of our system. The compiler takes a source program and a security policy and produces an enhanced version of the program that enforces the policy by performing dynamic data flow analysis.

In this paper, we describe an extensible compiler-based system that simultaneously advances the state of the art in both directions. The performance of our system is an order of magnitude better than any previous software taint tracking system. The generality of our system allows it to support all of the problems listed in Table 1. Moreover, this generality is easily accessible, as we now explain.

Figure 1 shows the overall architecture of our system. The input is an untrusted program. The output is an enhanced program that enforces some specified security policy, which is selected by the end-user at compile time. To enforce the desired policy, the compiler first performs a static vulnerability analysis that identifies whether the input program might violate the policy; the compiler then performs an analysis that identifies those locations in the input program that require dynamic analysis. The appropriately enhanced program then dynamically enforces the policy by performing a tag-based *dynamic data flow analysis*, discussed in Section 4.1.

The policy itself is defined in an annotation file that describes the policy and the effects of standard library calls on the policy. Thus, the policy is entirely separate from the data flow tracking mechanism, so in addition to the existing security policies that we have already defined, new security policies can be specified without modifying either the compiler or the runtime system.

The key to our system's low overhead is our compiler's ability to identify many innocuous flows of data that provably do not need to be tracked. For example, a program might read data into several different buffers, only one of which is used in a way that violates the policy. It is therefore not necessary to track the other buffers and everything they flow into. In cases where the compiler can prove that no policy violations are possible, the resulting output program contains no instrumentation and thus incurs no runtime overhead. Our compiler can precisely identify innocuous data flow because (1) our system's security policy can be interpreted as a static dataflow analysis that determines whether the policy was violated, and (2) our compiler performs an interprocedural analysis that uses a precise and scalable pointer analysis [23]. By contrast, other current systems do not attempt to statically detect the policy violation, and they perform static analysis that is limited in scope (they are intraprocedural) and precision.

This paper makes three main contributions:

- We introduce a system that accepts untrusted C programs as input and produces C programs that can enforce any policy that can be expressed as data flow tracking. Our system comes with predefined policies for taint and file disclosure, and our system can be easily extended to handle other problems and security policies without modifying our system implementation. Our system uses sophisticated static analysis to minimize the amount of dynamic analysis that is needed.
- We show that our system is general by using it enforce file disclosure vulnerabilities—a privacy problem that taint tracking cannot handle—as well as to perform traditional taint tracking. Our system is as general as GIFT [31], but our system is configured through an existing declarative annotation language [21] rather than by writing wrapper functions or new code.
- We demonstrate our system's performance advantages by evaluating it on both server programs and compute-bound programs. For common open-source server programs, our system's average overhead is 0.65%, compared with 6% for the previous best reported results [48]. For compute-bound SPECint benchmarks, our system is able to prove the absence of format string errors entirely, giving a true overhead of 0%. After inserting vulnerabilities into these programs, our system's average overhead for compute-bound programs is less than 13%, compared with 75-260% for previous systems [48, 39].

The remainder of this paper is organized as follows. Section 2 discusses related work, Section 4 describes our solution, and Section 5 presents our evaluation.

2. RELATED WORK

Our work is a generalization of dynamic taint tracking [46, 37, 38, 44, 16, 10, 12, 48, 39, 31, 11], which has been used to protect against buffer overflows, stack smashing, and format string attacks, and which covers attacks previously addressed separately by various different solutions [15, 14, 4, 13]. Taint tracking is a practical realization of some of the concepts of information flow control [5, 18] and integrity [8], but it typically ignores implicit flows. Much of the previous work in taint tracking has used dynamic binary instrumentation frameworks [37, 12, 39, 11] or specialized hardware [44, 16, 17]. Except where noted below, these systems are not designed for more general data flow tracking problems, and core components would need to be rewritten to accommodate such generality. Instead, these systems have focused on reducing runtime overhead.

Many taint tracking systems based on binary instrumentation, such as TaintCheck [37] or Dytan [11], have enormous overheads, as high as 37×. More recent binary instrumentation systems such as LIFT [39] have significantly improved performance, but overhead for compute-bound programs remains as high as 7.9×.

Xu, *et al.* present a high-performance compiler-based taint tracking system [48], which achieves an average overhead of 6% for server applications. However, to achieve this performance, the system exploits assumptions about the memory layout in 32-bit x86 Linux, allowing it to write the tag map directly to unused memory at fixed addresses. Even with this optimization, their average overhead for compute-bound applications is 75%. Without this optimization, their overhead is "unacceptable" [48].

Hardware-based taint tracking [44, 16, 17] can in most cases eliminate the overhead associated with software-based taint tracking. However, overhead can still occasionally be as high as 23% [44] due to cache performance degradation. Furthermore, most hardware-based systems only defend against memory and control flow errors. Raksha [17] provides additional flexibility by using tag propagation registers and OS traps, but this comes at a high cost, with average overhead exceeding 3×.

Lam and Chiueh's General Information Flow Tracking [31] is a framework that uses a compiler to automatically add code to propagate and check tags associated with data, allowing it to handle a wide range of problems beyond overwrite attacks. GIFT can be extended to handle new policies by defining wrapper functions

and transformations for the the GIFT compiler that implement the policy and related tag management. However, the policies are defined operationally by writing code that implements transformations. Without an understanding of the semantics of the policy itself, the compiler is limited in its ability to analyze and optimize the added code, leading to an average CPU time overheads of 82% for the sandboxing of client applications.

Inline reference monitors [19, 41] use security automata to defend against attacks. However, finite state automata are awkward and unsuitable for addressing problems that require the tracking of information flow. PQL [32] presents a more flexible scheme based on pattern matching of event sequences on objects. This system can track direct information flows and uses static analysis to reduce overhead a including direct information flows. However, their techniques require type safety and work only on heap objects, so their techniques are unsound for programs with flows through scalar variables.

Static analysis can be used to find bugs and potential security vulnerabilities in software [42, 2, 20, 23]. While extremely valuable, such systems still require that software developers expend significant effort to verify and fix any vulnerabilities reported by the tool. For example, CQUAL has been used to find format string vulnerabilities [42]. However, to use their results, programmers must examine the output and the original source program using an advanced graphical interface and attempt to determine manually whether the reported vulnerability is real or a false positive. Moreover, they report a false positive rate of approximately 84%, so considerable programmer effort is wasted. In contrast, our system uses statically identified vulnerabilities to determine where dynamic guards are required, producing a system protected from the vulnerability without additional programmer effort. Thus, our system complements static tools by guaranteeing that possible vulnerabilities are prevented even if the programmer has not yet fixed the problem.

Language-based security solutions range from type-safe C [35, 27] and bounds checking for C [28] to full language support for information flow [33, 40]. These systems can provide strong guarantees at compile time, but they either impose significant performance overhead or require significant developer effort to rewrite and redesign current programs. The "safe-C" approaches cannot handle errors like SQL injection that do not violate language semantics. By contrast, JiF [33] supports complex information flow policies that our system cannot support.

Finally, static analysis can also be used to construct static models of program behavior that can then be enforced dynamically. For example, control [30, 1] and data flow integrity [9] ensure that the program never deviates from statically computed models of control and data flow, respectively. However, these systems are not generalizable to security problems that do not violate control or data flow integrity, and they are of limited use in languages that already provide similar guarantees. In contrast, our system is capable of handling a far broader class of problems that plague even safe languages.

3. MOVING BEYOND TAINT ANALYSIS

Before we describe the details of our system, we will explain the importance of moving beyond taint analysis into a realm of richer data flow tracking problems.

With respect to properties, taint tracking does not provide sufficient information to handle many attacks. In Section 4.3.2, we discuss the problem of file disclosure vulnerabilities. Taint tracking cannot handle this problem because it requires the simultaneous tracking of two different properties. In addition, taint tracking is too simplistic for problems in privacy and access control. For example, to enforce a policy based on labeled security [34], the system must simultaneously track hierarchical levels and non-hierarchical categories. To meet future challenges, systems must be able to track multiple complex properties at the same time.

With respect to policies, most taint tracking systems enforce "low-level" policies that prevent overwrite attacks. While overwrite attacks are a major concern in unsafe languages, they are not a significant concern in managed languages and scripting languages that already guarantee memory safety. For example, SQL injection and cross-site scripting attacks do not overwrite pointers but do occur in safe languages. Moreover, privacy breaches and information breaches are usually accomplished without memory errors, and they represent a serious challenge for current and future web applications. To meet these and future attacks, systems must support higher-level policies that are aware of higher-level application semantics.

With respect to performance, any system the performs taint-specific optimizations is likely to suffer when applied to problems beyond taint. To maintain high performance in a general setting, the techniques and optimizations must be generally applicable to all data flow tracking problems.

4. OUR SOLUTION

Our system takes a C program as input and produces as output a modified version of the program that enforces a selected security policy. Our system does not require hardware or operating system changes, is easy to use and extend, and exploits a powerful interprocedural data flow analysis to eliminate unnecessary instrumentation. While our specific implementation targets C code, our techniques are not specific to taint tracking or C and can be applied to a wide array of current and future problems and languages.

Our system is easy to deploy: the *end user* of our system needs only to recompile a program and select a security policy to create a secure program. A *security expert* can extend our system—which currently includes policies for taint tracking and the prevention of file disclosure vulnerabilities—with new analyses and policies by providing a annotation file that is independent of any specific application.

Our policies use a simple and flexible dynamic model similar to General Information Flow [31]. Our system associates symbolic tags with data objects at runtime, it updates the tags as the program executes, and it enforces policies based on the tag values. Unlike prior systems, our system is explicitly based on *data flow analysis* [29], a technique for computing facts about data by observing how it flows through the program. This design allows our system to both statically check for and dynamically guard against policy violations from the same specification. A static data flow analysis computes an approximate solution that holds over all possible executions of the program because a fully precise solution is undecidable [29]. In contrast, a dynamic data flow analysis [26] computes precise facts but only about the current execution. These complementary characteristics allow our system to use a static data flow analysis, discussed in Section 4.4, to compute a conservative solution at compile time and to refine the result at runtime to enforce a policy efficiently and precisely.

To perform the dynamic data flow analysis that actually enforces the policy, our compiler inserts into the source program calls to a small runtime library that manages tag information along with any required checks necessary to enforce the policy. Since nothing in our system is specific to taint tracking, our system and our optimizations apply to all general data flow tracking problems.

```
1   char input[1024];
2   char buf[1024];
3   char otherbuf[1024];
4   char buf1[1024];
5   ...
6   read_from_network(input);
7   read_from_network(otherbuf);
8   ...
9   memcpy(buf, input, 1024);
10  memcpy(buf1, otherbuf, 1024);
11  process(buf);
12  process(buf1);
13  ...
14  printf(buf);
```

Figure 2: A simple example illustrating the benefits of our static analysis. Current systems must track all objects, while our static analysis can eliminate tracking on all except `buf`.

Eliminating Unnecessary Tracking.

The naive insertion of calls throughout the program inevitably leads to high overhead, so to achieve good performance, it is critical that unnecessary calls be eliminated. To illustrate some of the limitations and difficulties encountered by current systems, consider the code in Figure 2. This code contains a format string vulnerability where a tainted buffer is printed. Assuming a policy that uses taint analysis to guard against format string attacks, current taint tracking systems, including those that perform some static analysis, would track taintedness on all buffers in this example, as well as anything that the `process` function touches and anything that those variables affect. However, very little tracking is actually required. Our system can prove that tracking on `buf1` is not required because it is never passed to `printf` or any other sensitive function. Additionally, if tracking on `buf1` is not required, neither is tracking on `otherbuf`, because `buf1` receives its value only from `otherbuf`. We also do not need to track anything in the call to `process(buf1)` because none of its results is used by `printf`. Moreover, we do not need to track the original `input` buffer because we know that it is always tainted; it is sufficient to simply mark `buf` as tainted at the call to `memcpy`. Finally, we do not need to track anything else that `process(buf)` can affect if none of the resulting values is misused.

The keys to removing this type of unnecessary tracking are an interprocedural static analysis that leverages semantic information about the security policy and a sophisticated interprocedural pointer analysis to perform policy-specific optimizations. Without semantic information about the policy, our system could not distinguish possible violations from safe events. Without a precise pointer analysis, our system could not account for flows between objects in an effective manner. Without a dependence analysis that builds on the pointer analysis and knowledge of the policy, our system could not determine which objects are involved in possible vulnerabilities. Moreover, all of these analyses must be interprocedural to eliminate flows among functions.

More specifically, our system operates by first performing a static data flow analysis and a highly precise and scalable pointer analysis to determine where possible policy violations lie. Our system then instruments the program to ensure that all policy enforcement checks are performed correctly. Because the portion of a program involved in any given attack is typically exceedingly small [36], our system typically adds very little code and incurs negligible overhead. Identifying this portion, however, requires a powerful static analysis.

Our system is built on the Broadway static data flow analysis and error checking system [21], which is a source-to-source translator for C. We use the Broadway annotation language and analysis infrastructure, enhancing it with our own dependence analysis (described in Section 4.4.3) and dynamic data flow analysis component (described in Section 4.1).

The remainder of this section discusses the components of our system in more detail. We will begin with an overview of our compiler-based dynamic data flow analysis system, followed by a discussion of our policy specification mechanism. Finally, we will discuss the details of our static analysis.

4.1 Dynamic Data Flow Analysis

The dynamic data flow analysis that we perform is a typestate analysis [43], which is an analysis that associates an abstract value, called typestate, with objects in a program. Unlike types, the typestate of an object can change during execution. For example, a file handle or a string maintains the same type throughout its lifetime, but its typestate—open or closed, tainted or untainted—can change as the program executes. Typestate is a general model capable of supporting a wide variety of security analyses and policies, including all those supported by general data flow tracking [31].

Our implementation of dynamic data flow analysis treats typestates as flow values in a data flow analysis and represents them at runtime with a map that associates 32-bit tags with data. As execution proceeds and data are used, the tag map is updated in accordance with the property being analyzed. To enforce a particular security policy, checks that use these tags are inserted into the program.

The map is implemented using a very small runtime library that includes functions for initializing, checking, updating, and deleting entries from the map. Our implementation tracks tags at the byte granularity, providing *fine-grained* tracking of data flow properties, which is necessary because the tracking of flow values at the level of variables is unsafe in a type-unsafe language like C, especially in the presence of aliasing. Our map uses a sparse representation similar to tree-like structures previously used for memory leak profiling [25]. Each node in the tree represents an address range, with child nodes representing finer subdivisions of the range of their parent nodes. The leaf nodes contain arrays which record flow values associated with memory at a byte granularity. For example, to record the taintedness of a byte of memory at address a, the library traverses the tree to find the leaf node representing the smallest address range that contains a, and it then record taintedness in that node's array of flow values. In addition, to save memory and decrease lookup times, our implementation also allows us to store flow values in the interior nodes when the entire subtree contains only one flow value, which can occur when large regions are marked entirely with the same typestate.

4.2 Code Instrumentation

To use the map to track flow values at runtime, the compiler instruments the original program with calls to functions that manage the map. This process is straightforward. Like most compilers, our system first transforms C to a simpler intermediate representation before performing analysis and transformations. At this level, the compiler only needs to consider assignments, basic operators, pointer dereferences, and function calls. Our transformation for inserting code is as follows:

- Constants are given the default flow value.
- Assignments transfer the flow value of the source to the target.

- Operators (such as arithmetic operators and array accesses) have the flow value of the $meet$ of the operands. The $meet$ operator in data flow analysis combines flow values based on their position in the lattice [29].
- Any address or pointer dereference that is used or assigned to acts on the corresponding entry in the map.
- In keeping with C's call-by-value semantics, function calls transfer flow values to the arguments in the function body, and function calls return any flow values through the return value.

These rules are analogous to the standard rules for applying data flow analysis [29] and remain the same for the wide variety of security problems that lattices naturally model [18]. When applied to taintedness, these rules are the same code insertion rules used by other compiler-based systems [48, 31] (although our additional analysis and optimizations often allow us to remove considerable amounts of instrumentation). These rules track *explicit flows*, which are information flows that occur because of assignments or arithmetic operations. Like taint tracking systems, our system does not track implicit flows [44, 16, 37, 48, 39, 31].

4.3 Policy Specification

In most taint tracking systems, the semantics of taint analysis are hardcoded into the system. Because our system is designed to handle general data flow problems, our system instead factors out the semantics of the analysis and policy to an external file that contains annotations describing the property to analyze, the policy to enforce, and the effects of library procedures on the property. This file contains the *same information* that would have been hardcoded into a compiler-based taint tracking system, but it provides the capability to extend our system to other problems without changing the core analysis. Unlike in-lined annotations, our annotations define an analysis that is *independent* of the input program, enabling reuse across many programs. A typical user does not have to write any annotations to use an existing policy. The creation of new policy files is a careful activity that is only necessary when defining a new analysis or security policy.

Our system uses the Broadway declarative annotation language [22, 21], which has been previously used for static error checking [23] and library-level optimizations [24]. The annotation file tells the compiler how to perform a specific data flow analysis by supplying the specifics for the rules in Section 4.2. The rules fall into three categories:

- **Defining the Lattice.** The lattice for each typestate property must be defined. The tags used at runtime correspond to the flow values, while the lattice itself defines the meet function that specifies how flow values should be combined when used together in arithmetic and other operations.
- **Describing Effects of Library Calls.** The compiler also needs to know how the various library calls affect tag values. For each external function that affects the flow values, a brief summary annotation must be provided that describes how the function can affect the flow values of globals and arguments.
- **Defining Security Policies.** Lastly, the compiler needs to be given the definition of policy violations. Violations are defined as predicates over flow values that are checked at procedure boundaries, most commonly a check on the flow value of an argument. By default, violations trigger our default error handler, which logs the violation and blocks the operation, but the user can supply a custom error recovery function, which can be application-specific.

```
1  property Taint : { Tainted, { Untainted } }
2                   initially Untainted
3
4  procedure recv(s, buf, len, flags) {
5    on_entry { buf --> buffer }
6    modify { buffer }
7    analyze Taint { buffer <- Tainted }
8  }
9
10 procedure strdup(s) {
11   on_entry { s --> string }
12   access { string }
13   modify { string_copy }
14   on_exit { return --> new string_copy }
15   analyze Taint { string_copy <- string }
16 }
17
18 procedure printf(format, args) {
19   on_entry { format --> format_string }
20   access { format_string }
21   error if (Taint: format_string could-be Tainted)
22           "Error: tainted format string!"
23 }
```

Figure 3: Example syntax for defining a policy that prevents format string attacks. First, the concept of taintedness is defined. Then, we specify the introduction of taint through I/O functions such as `recv()` and the propagation of taint through functions such as `strdup()`. Finally, we prevent the use of tainted format strings in functions such as `printf()`. The forward arrows in the syntax allow us to distinguish between pointers and pointed-to objects.

Our system readily supports domain-specific annotations that go beyond the standard library functions. For example, if the application calls an input sanitization function, we can add an annotation for that function that untaints the sanitized output. Our system can also support policies that depend on concrete values. For example, a naive policy that rejects tainted SQL query strings is inappropriate for detecting SQL injection attacks because query strings always contain tainted characters. To handle SQL injection attacks, we can supply a custom handler in the form of a C function that checks the taintedness of the keywords in the query string, accepting strings with untainted keywords. To ensure soundness, we require that a user-defined handler never reject an event that the typestate analysis accepts, which ensures that the typestate analysis over-approximates the actual policy.

A key advantage of our system is the compiler's access to the semantics of the security analysis. Our annotations define *what* analysis to perform, not *how* to perform the analysis. In addition to being simpler to reason about, declarative policies allow our system to perform an efficient static analysis, described in Section 4.4, that over-approximates dynamic behavior. It is this static analysis that allows our system to achieve the low overheads that we will discuss in Section 5

4.3.1 Specifying Taintedness

In this section, we will use taint tracking and format string attacks to briefly illustrate the annotation language in Figure 3. The annotations on lines 1-16 describe a data flow analysis, in this case a taint analysis, as we now explain.

On line 1, the `property` keyword defines one lattice with two possible values, *Tainted* and *Untainted*. We place *Untainted* above *Tainted* on the lattice so that *Tainted* and *Untainted* values combine to yield *Tainted* values.

We then annotate the library procedures that introduce tainted-

ness, such as `recv` on line 4. Here, the `on_entry` keyword at line 5 allows us to assign a name, `buffer`, to the object pointed to by the pointer `buf`. The `analyze` keyword indicates that when the `recv` routine is invoked, `buffer` (not `buf`, which is a pointer) becomes tainted. In addition, we inform the compiler that this procedure modifies `buffer`.

Similarly, we also annotate library procedures that propagate taintedness, such as `strdup`. Again, the `on_entry` and `on_exit` keywords (lines 11–14) describe relations between pointers and the objects they point to, while the `access` and `modify` lines specify that the function reads from `string` and writes to `string_copy`.

The policy itself is defined on line 21 by using the results of the taint analysis when `printf` is called. Here, we specify that an error occurs if `format_string` (the string that `format` points to) could be tainted. This line specifies the entire policy with respect to the `printf` procedure–other taint-based policies could be added that would reuse all the taint tracking annotations as-is.

Similar annotations have been used in other systems for error checking. For example, CQUAL [42] uses type qualifier annotations on library functions to statically check for format string vulnerabilities. Although slightly less verbose than the Broadway annotation language, their type system permits "reverse flows" and other artifacts that increase the false positive rate. Because our system is based on a more precise data flow analysis, we are able to avoid such anomalies while also being able to leverage far more precise pointer analysis (see Section 4.4.2). The actual annotation burden—which is only incurred when defining new policies—is discussed in Section 5.4

4.3.2 Specifying File Disclosure

To illustrate the flexibility of our system, we also apply it to file disclosure vulnerabilities. File disclosure can occur when a remote user can connect and download the contents of arbitrary files, thus improperly revealing sensitive information. This vulnerability can be present when a program behaves unintentionally like an FTP server; that is, if the remote user can specify the name of a file whose contents are then sent over the network. Note that sending data from files not directly specified by the user is fine, as is sending responses constructed from user input. In essence, file disclosure is a simple privacy protection problem where the goal is to ensure that untrusted users cannot directly specify data to access. These attacks are not well-studied on C programs because overwrite attacks account for the majority of C vulnerabilities. However, these vulnerabilities are common among web applications written in scripting languages such as PHP, Python, and Perl. Thus, our techniques remain relevant and applicable to safe languages.

File disclosure cannot be modeled with only taint tracking because taint tracking does not distinguish between the source of data and the trustedness of data. A taint tracking system could disallow the transmission of tainted data, but such a policy would also prevent legitimate echoes of network input. The taint tracker could also disallow transmission of any file data, but such a policy also eliminates legitimate transfers and would even prevent most query services from operating. To model file disclosure accurately, the system must track both the trustedness (whether the data is under attacker control) and the origin (whether the data comes from a file) of data within the system.

File disclosure is straightforward to model in our system. We first define two properties, *Trust* and *Kind*. *Trust* represents the trustworthiness of the data source, which can be *Internal* to the program, *External* to the program but on the local system, or obtained from a *Remote* source. *Kind* denotes the possible source of the data, be it from a *File*, standard I/O, the network, or otherwise. A file disclosure attack occurs when *File* data with *Remote* trustworthiness is written to a *Remote*ly opened socket. The required procedure summary annotations themselves are similar to those for format string attacks and are omitted here. This policy precisely models the FTP-like behavior described earlier, disallowing file disclosure while permitting other file data or other user-derived data to be sent.

4.3.3 Other Problems

Although we focus on the above two problems in this paper, our system can be used to enforce a wide variety of problems. Lattices are a natural model for many security problems [5, 18, 8]. For example, multilevel security can be implemented with a lattice representing hierarchical levels, such as *Unclassified*, *Classified*, or *TopSecret*, along with properties representing categories, such as *Army*, *Navy*, etc. Library I/O functions would be annotated to call a user-provided helper function to read the appropriate label from the file, while the annotations for operations like string copy would remain essentially identical to those for taint tracking or file disclosure. For additional information on the Broadway language and its capabilities, please refer to prior work [22, 23, 21].

4.4 Static Data Flow Analysis

To avoid the cost of tracking all objects at runtime, our compiler statically performs an interprocedural data flow analysis that identifies program locations where policy violations might occur. Starting from these possible violations, a subsequent interprocedural analysis identifies statements in the program that affect the flow values—and therefore the policy decision—at these violations. Other statements do not require instrumentation because they cannot affect the relevant flow values and thus cannot affect policy enforcement decisions. This analysis is supported by a fast and precise pointer analysis, which is critical because a less precise pointer analysis would identify many more program locations as possibly violating the specified policy [23], leading to higher runtime overhead. We now discuss these steps in detail in the following subsections.

4.4.1 Static Vulnerability Analysis

The first step is to statically check the program to identify all possible violations of the security policy as defined by the annotations [23]. If the compiler can prove that there are no such violations in the program, *no further analysis or code insertion is required*. However, in cases where the compiler identifies possible violations, additional analysis is needed to determine where instrumentation should be inserted.

To perform this first step, our system uses an iterative static data flow analysis that is performed by the Broadway static analysis system [21]. Because the analysis is sound, these locations are the only locations where violations of the policy can occur. In Section 4.4.3, we explain how our system ensures that all of these possible violations are guarded against.

4.4.2 Pointer Analysis

A significant obstacle to interprocedural program analysis is the use of pointers. To reason precisely about the flow of data, the compiler must know which objects a pointer could point to. The limited scalability of pointer analysis has stymied previous attempts to apply interprocedural analysis to dynamic taint tracking [31], so interprocedural analysis is not commonly used.

Our system uses a scalable and precise *client-driven pointer analysis* [23, 21]. The client-driven analysis is able to match the precision of a fully flow- and context-sensitive pointer analysis without requiring significantly more runtime than a fast and imprecise flow-

and context-insensitive analysis. Unlike most pointer analyses, the client-driven analysis cannot be used as a stand-alone pointer analysis. Instead, it requires a *client* that uses the results of the analysis, which in our system is the static data flow analysis that identifies possible policy violations. By identifying locations where imprecision in the pointer analysis affects the precision of the client's results, the client-driven analysis is able to selectively increase precision for the pointer analysis in places where it will improve the results of the client analysis. Because the amount of extra precision is typically small [23], the client-driven analysis is able to avoid analyzing pointer relations that do not affect the client, dramatically improving scalability without sacrificing precision with respect to the client. The client itself must be a lattice-based data flow analysis, so we see now how our annotation language's declarative specification of data flow analysis plays an important role in minimizing runtime overhead.

Finally, we note that the client-driven approach does not impact the soundness of the pointer analysis. Precise pointer analysis is an undecidable problem, so almost all pointer analyses, including ours, compute a conservative over-approximation of the actual result. In particular, our pointer analysis is sound under the assumption that displacements between objects are undefined, a necessary assumption common to C pointer analyses [3].

4.4.3 Data Flow Slicing

The static error checker identifies possible vulnerabilities by location and memory object. Our system must ensure that all the dynamic checks that are required to prevent possible vulnerabilities are performed correctly. We refer to the process of computing the statements that require instrumentation as *data flow slicing*, by analogy with program slicing [47].

We define a *data flow slice* with respect to some object o at some program location l to be the set S of statements and locations that affect a set O of objects, computed by the transitive closure as follows:

- l is in S and o is in O.
- If statement s' defines the flow value of some $v \in O$, then s' is in S.
- If statement $s \in S$ uses the flow value of some o', then o' is in O.

In contrast with a program slice, which is the portion of the program necessary for computing the value of o at location l, a data flow slice is the portion of the program that affects the *flow value* of o at l. For example, a statement that increments a counter will change the counter's concrete value but not its taintedness. Since the flow value does not change, this statement is not part of the data flow slice. Moreover, implicit flows and branch conditions are not part of the data flow slice because they cannot affect the flow values for a tag-based data flow analysis, but they *are* a part of a program slice.

As long as the underlying pointer analysis is sound, data flow slicing is a sound method for identifying statements that affect flow values: A statement can affect flow values only by defining them. If statement s affects the flow value of object o at location l, it is by definition in the data flow slice, and statements that affect o at l through intermediate assignments are also included because the data flow slice is a transitive closure.

Data flow slicing is an interprocedural dependence analysis that tracks dependences in terms of flow values instead of concrete values. Our compiler computes data flow slices by first constructing interprocedural use-def chains, which allows it to identify all possible definitions for any given use of an object. The data flow slice is always a subset of the locations in the use-def chains, as a flow value cannot change without a def (but as we have mentioned, not all defs change the flow value).

Our data flow slicing algorithm is also able to truncate the slice when flow values are definitely known. Since the static data flow analysis is an over-approximation of possible dynamic behavior, a statically computed exact flow value means that the object will always have that flow value at that location at runtime. For server programs and taint tracking, this optimization has the effect of moving instrumentation away from input functions and closer to the data directly involved in checks. In several of our programs, input is read into a buffer, which is immediately copied to another buffer. The copy is then used for subsequent operations. Our static analysis can determine that the copy is always tainted, rendering further backwards tracing unnecessary. The resulting program thus does not need to instrument the input buffer, instead directly marking the copy as tainted.

This optimization has an interesting side effect: At times, the tags of some addresses will not be up-to-date. However, we can guarantee that any piece of information will be up-to-date when it is used to make a security decision. Thus, the system is as secure as a fully instrumented system, but it does not pay the performance penalty of keeping all tags up-to-date at all times.

Another consequence of our technique is that a more complex policy or program does not necessarily result in higher overhead. The actual overhead depends on the policy, the program, and the way in which the program might violate the policy. A more complex policy can result in lower overhead when the portion of the program involved is smaller or off the critical path. In Section 5.3, we see that guarding against file disclosure vulnerabilities can often have even lower overhead than taint tracking despite being a significantly more complex problem. Larger programs also do not necessarily experience higher overhead; in fact, in our results (Section 5.1.3), the highest overhead for server programs is for the smallest program.

Once the data flow slice from a potential vulnerability is computed, it is straightforward to add instrumentation to these program locations. The data flow slice includes all information that impacts the flow value at the potential vulnerability, so the check will evaluate to the same result as a fully-instrumented system.

4.5 Security Discussion

We now examine the security-related assumptions and advantages of our system.

4.5.1 Trusted Computing Base

As with other software taint tracking solutions, our system increases the size of the TCB, in our case adding the compiler to the TCB. Although there are security implications [45] to trusting the compiler, the additional trust required by our approach is mitigated by two factors. First, in typical modern environments, the compiler (usually `gcc` or some other widely used compiler) is *already trusted* to compile the server programs that are actually run. Second, our *source-to-source* translator relies on the user's already trusted compiler for generating binary code. The changes and modifications that our system makes to programs are thus transparent and human-readable, making it difficult to insert undetected malicious code. Thus, our system requires minimal additional trust beyond that which is already present in most deployed systems.

Like any system based on user-defined policies, the policies themselves are also a part of the trusted computing base. If the annotations that summarize the effects of external functions are incorrect or incomplete, the system may miss important data flow. Such an

error is analogous to a bug or omission in a hardcoded taint tracking system. Fortunately, frequently-used external code resides in libraries like the C Standard Library that are relatively robust and whose semantics are well-understood, and we have found that providing accurate annotations for these functions is straightforward.

4.5.2 Attacks Detected

Our system is capable of detecting attacks that depend on the propagation of data through the system. More specifically, we can enforce any typestate policy, which includes traditional taint-based attacks as well as general information flow tracking [31]. These attacks include those that do not overwrite control data or violate data flow integrity and thus are problems even in safe languages.

In our evaluation, we enforce a taint-based policy that prevents format string attacks, similar to the format string policies used by existing taint tracking systems, such as TaintCheck [37], as well as interpreters with taint tracking modes [46, 38]. In addition, our system can enforce a policy that prevents attacker-controlled data leaks such as file disclosure vulnerabilities; this policy cannot be enforced precisely by an ordinary taint tracking system.

Our system only guarantees that violations of the specified policy do not occur. This situation is shared by all enforcement mechanisms—for example, a memory-safe database server can still be compromised by an SQL injection attack because such attacks do not violate memory safety. The soundness of our analysis prevents any attacks that violate the policy. However, if it is possible for the attacker to gain control through an attack that does not violate the policy, it may be possible to compromise the application.

4.5.3 Alternate Attack Channels

Like other taint tracking systems, we do not concern ourselves with *implicit flows*. Implicit flows occur when control flow influences the possible values of data. For example, information may be implicitly passed along branches of the form `if(x==0) y=1; else y=0;` which allows the user to influence the value of `y` by modifying the value of `x`. Taint tracking systems usually do not consider `y` tainted even if `x` is tainted. Although such cases result in implicit information flows that are theoretically exploitable, the majority of attacks depend on direct flow of data [16, 10], which our system does guard against.

Our system also does not defend against attacks that are not based on information flows in program code. For example, distributed denial of service attacks can harm systems without creating any individually anomalous information flows. Information can also be leaked via covert timing channels, which we also do not detect, although our requirement for source code limits the ability of malicious developers to introduce malicious code. Finally, our solution only defends against attacks, not arbitrary memory errors. A buggy program can still experience segmentation faults and other errors using only untainted data.

4.5.4 Defending the Enforcement Mechanism

The design of our system makes it difficult in practice for an attacker to subvert the enforcement mechanism itself. First, like other compiler-based systems [48, 31], the original program is written before the enforcement code is added, so the original program cannot directly access enforcement data. Moreover, unlike taint tracking systems that track taintedness using stack-allocated variables or fixed addresses [48], all of our structures are dynamically allocated on the heap and concealed behind function calls. Pointers to enforcement data never appear in application code, so the attacker cannot obtain a pointer to our enforcement data without sophisticated heap attacks. Thus, the attacker will not be able corrupt enforcement data without first hijacking the program by exploiting some vulnerability that the user's security policy does not guard against. Attacks that the user's policy do guard against are prevented.

For additional protection, our mechanism can be easily combined with various defenses against memory errors. For example, address space randomization [7] or heap randomization [6] can be used to defend our system against corruption attacks.

5. EVALUATION

In this section, we evaluate the effectiveness of our system by using it to prevent format string attacks and file disclosure vulnerabilities. We verify attack prevention, measure static code expansion, and measure runtime overhead for five open-source server programs and four compute-bound SPECint 2000 benchmarks. Since our system is a source-to-source translator, we compile the enhanced C programs using `gcc-3.3` on Linux with the default compiler options and optimization levels that were supplied by the original developers of the benchmark programs. The programs are then run on a 2.4 GHz Pentium 4 with 1 GB of RAM, running Linux 2.6.17. For each benchmark, we use the program's documentation and examples to run the program with a reasonable configuration.

5.1 Taint Analysis for Server Programs

We evaluate our system with a taint checking policy that prevents the use of tainted format strings in exploitable functions. This strict policy is similar to that enforced in the TaintCheck system [37].

Our policy distrusts all inputs that can be under user-control, including the file system and environmental variables. Our policy is significantly stronger than the default "trust everything except network input" policy used by some other systems [37, 39] for servers. This stronger policy is necessary to detect uses of tainted data that are cached in the file system, an actual problem in one of our benchmarks, as we discuss in Section 5.1.1.

We apply our system to five commonly-used open source server programs: `pfingerd`, `muh`, `wu-ftpd`, `BIND`, and `apache`. These programs are, respectively, a finger daemon, an IRC proxy, an FTP server, a name server, and a web server. Several are widely deployed and typically run in privileged mode, so their robustness and integrity are critical.

We use our system to produce a modified version of each program that contains additional code to perform dynamic taint tracking. In our tables, we refer to this version as `DDFA`. The actual analysis time, while not negligible, is no worse than four minutes for `apache`, our largest benchmark with nearly 67K lines of code, and thus does not pose a serious obstacle to deployment.

Finally, we note that these programs were selected in part because our static data flow analysis identified potential vulnerabilities in them. Our test programs were selected from a suite of open-source server programs that was previously used for static program checking research [23]. For nine other programs in this suite, our compiler analysis determines that there are no improper uses of tainted data and therefore no instrumentation whatsoever is required. These programs include `BlackHole`, `privoxy`, `sqlite`, and `pureftpd`, and indeed there are no known applicable tainted-data attacks against our tested versions in the CVE database. For these nine programs, our system does not modify the program and therefore exhibits 0% runtime overhead and 0% code expansion. *Only a system that performs a static interprocedural taint analysis can achieve these overheads.* We have chosen to exclude these nine programs from our results and to instead focus on those programs that have possible vulnerabilities, but these results nevertheless highlight an important advantage of our approach.

Program	Version	Exploit Ref	Detected
pfingerd	0.7.8	NISR16122002B	Yes
muh	2.05c	CAN-2000-0857	Yes
wu-ftpd	2.6.0	CVE-2000-0573	Yes
bind	4.9.4	CVE-2001-0013	Yes

Table 2: Evaluation of our system's ability to detect actual attacks. All attacks are detected successfully.

Program	Original	DDFA	Code Overhead
pfingerd	49655	49655	0%
muh	59880	60488	1.01%
wu-ftpd	205487	207997	1.22%
bind	215669	219765	1.90%
apache	552114	554514	0.43%
Average Code Expansion			0.91%

Table 3: The static code expansion required for dynamic taint tracking, as measured by compiled binary size (bytes).

5.1.1 Security Evaluation

We first evaluate our system's ability to detect attacks. Four of our benchmark programs contain known vulnerabilities that are exploitable. For example, `pfingerd` improperly trusts hostnames, while `muh` does not properly check format strings when reading or writing log files. The SITE EXEC format string vulnerability in `wu-ftpd` is actually the first discovered format string vulnerability [13]. `BIND` improperly writes requests to `syslog` when an authoritative nameserver is malicious. Our particular configuration of `apache` (core only without optional modules) does not contain any known format string vulnerabilities; it is included because our static analysis was not able to completely eliminate that possibility.

To test whether our system correctly detects the use of tainted data, we send malicious input to the instrumented programs. Table 2 shows the vulnerable programs, shows the vulnerability in question, and indicates that in each case our system successfully detects these attacks. In each case, it detects that tainted data is about to be used improperly and identifies the potentially malicious data.

The case of `muh` deserves special attention. The vulnerability exists because `muh` writes logged messages verbatim to disk. Later, when a user requests log information, `muh` reads the message back from disk and prints it directly using `printf`. Thus, if the original message contained dangerous format specifiers, `muh` could be compromised when the message is printed back. If the policy is to trust local files, then this attack will go undetected, which can be a serious problem in servers that cache data on disk. Several taint tracking systems trust local files by default [37, 39]; their performance when applying our more aggressive policy is unknown but likely to be worse due to the greater presence of tainted data. Our system can enforce this stronger policy without fear of incurring significant additional overhead because our interprocedural analysis can frequently prove that most uses of local file data are safe.

5.1.2 Code Expansion

Because our system adds instrumentation to the source program, it introduces some static code expansion over unmodified code. We measure this expansion by comparing the sizes of the original and modified binary executables. Binary code size provides a more accurate measure of code overhead than source code size, because the binary code size includes the effects of standard compiler optimizations.

From Table 3, we see that the average code expansion for our benchmarks is less than 1%. In several cases, the compiled binary size does not actually increase because the added code falls in the padding that `gcc` adds. To place our results in context, LIFT with hot path optimizations can at least double the size of the code due to the need to maintain separate "fast" and "check" copies [39], while compiler-based systems like GIFT [31] report 30-60% increases in binary size.

Program	Original	DDFA	Runtime Overhead
pfingerd	3.07s	3.19s	3.78%
muh	11.23ms	11.23ms	0.0%
wu-ftpd	2.745MB/s	2.742MB/s	0.10%
bind	3.580ms	3.566ms	-0.38%
apache	6.048MB/s	6.062MB/s	-0.24%
Average Overhead			0.65%

Table 4: Runtime overhead for performing dynamic taint tracking on server programs. This table shows the response time or throughput overhead for our `DDFA` system running on a 100mbps ethernet network.

5.1.3 Runtime Overhead

The tracking of data flows incurs a runtime cost. For our set of server programs, we measure this cost by measuring server response time or throughput, as appropriate for the particular program.

The `pfingerd`, `muh`, and `bind` servers deal with short requests, so the end user is most directly impacted by increases in response time. For these programs, we measure the time between the sending of the request and the receipt of the entire response, averaged over one hundred requests. On the other hand, `wu-ftpd` and `apache` are used to serve files of varying sizes, so the primary metric of concern to end users is throughput (MB/sec). We measure throughput by downloading files with sizes uniformly distributed among 4KB, 8KB, 16KB, and 512KB over one minute.

To avoid local resource contention, our benchmarking client runs on a different machine from the server, interacting over a local 100mbps Ethernet connection. As shown in Table 4, our solution has an average overhead of 0.65%. In all instances, the overhead is lost within the noise. In fact, in three instances, average server performance actually improves by small amounts when we perform taint tracking. This improvement may be due to differences in memory layout induced by our runtime system and the resulting effect on cache performance. As a point of comparison, the previous fastest compiler-based and dynamically optimized systems report server application overhead of 3-7% [48] and 6% [39], respectively.

5.2 Taint Analysis for Compute-Bound Applications

In this section, we evaluate our system's performance on compute-bound applications by applying the format string policy to four SPECint 2000 benchmarks, with all I/O marked as tainted. These benchmarks were chosen because it was possible to inject realistic format string vulnerabilities into them, a task that can be challenging for the other SPECint benchmarks. In each case, our static analysis determines that the programs contain no such vulnerability. Thus, *our true overhead for these examples is 0%*.

To study the performance impact that our system would have on

Program	Code Expansion	Overhead
`gzip`	0.0%	51.35%
`vpr`	0.0%	0.44%
`mcf`	0.0%	-0.32%
`crafty`	0.36%	0.25%
Average	0.09%	12.93%

Table 5: Runtime overhead for performing dynamic taint tracking on compute-bound programs. These versions of the SPECint benchmarks were modified to introduce a format string vulnerability.

Program	Code Expansion	Response time
`pfingerd`	0%	0%
`muh`	2.67%	2.13%
`bind`	0.10%	-1.38%
Average	0.92%	0.25%

Table 6: Servers augmented by our system to guard against file disclosure vulnerabilities exhibit negligible overhead and code expansion.

compute-bound programs that do contain vulnerabilities, we manually insert a vulnerability into each of the benchmarks. To ensure that these injected vulnerabilities are realistic and representative of real vulnerabilities, we use the following guidelines in selecting the locations for the artificial vulnerabilities. (1) We choose locations where actual `printf`/`scanf` calls are being made, ensuring that our injected vulnerability appears at a location where it might be possible. (2) We preferentially choose calls that operate on character data, eliminating unrealistic vulnerabilities, such as the use of integers as format strings. (3) Finally, we check that our injected vulnerability is not eliminated by our static analysis.

Table 5 presents our results with the standard SPEC workloads. In all of the benchmarks, we demonstrate significant performance improvements over current software-based systems. The average overhead of 12.9% improves upon the best previously reported averages of 75-260% [48, 39]. Furthermore, in most cases, our system's overhead for compute-bound applications is essentially zero even when the application does contain vulnerabilities. Thus, our approach is less adversely affected by CPU-intensive programs than all current software-based techniques.

The `gzip` benchmark is a worst case for taint tracking systems [44, 48, 39, 17] due to its complex behavior and sensitivity to memory bandwidth. It operates on character data extensively and propagates tainted data everywhere, reducing the flows that our system can statically eliminate and negatively impacting performance. Nevertheless, our system's overhead of 51% represents a significant improvement over prior software systems, with overheads of 106% for a compiler-based system [48] to over 600% for dynamic instrumentation [39], and our result compares favorably with the 31% overhead for the most recent hardware-based solution [17].

5.3 File Disclosure Attacks

In addition to taint tracking, we evaluate our system's ability to prevent file disclosure attacks, as discussed in Section 4.3.2. Table 6 shows our results. For `pfingerd`, our static analysis was able to determine that it contained no FTP-like behaviors and therefore no instrumentation was required. For `muh` and `bind`, out system was unable to rule out this possibility and therefore had to insert a small amount of additional code. However, the delay in response time was so small as to not be consistently measurable.

These results highlight the advantages of our system. First, in some cases all instrumentation can be eliminated, giving 0% overhead. Second, in the cases where some tracking is required, our analysis is able to keep the additional code to a minimum, imposing only a small or negligible overhead. Finally, this example shows that without rewriting the compiler or its static analysis, our system can be applied to complex problems that taint tracking cannot directly handle. Moreover, if existing taint tracking systems were extended to cover richer problems, they would require significantly greater memory usage to track additional bits of data, while our system's memory usage is not adversely affected by these richer policies.

5.4 Policy Annotation Burden

We now briefly evaluate the burden of providing the policy annotations that are required by our system. Our annotations come in three types: (1) *pointer annotations*, which describe pointer relations; (2) *analysis annotations*, which define a data flow analysis; and (3) *policy annotations*, which use the results of the data flow analysis to enforce a policy.

Pointer annotations are common to all policies, because they describe the pointer relations of the arguments of each function, specifying what the function accesses and modifies; this information is used by the pointer analysis. Once a library has been annotated—in our case the Standard C library—pointer annotations need not be rewritten unless the library interface changes. For the Standard C library, there are pointer annotations for 116 procedures, with a median size of 3 lines and an average size of 4.68 lines.

Analysis and policy annotations can differ for different security policies. For the format string policy, there are 44 annotations of these types, with a median size of 6 lines each and an average size of 5.75 lines each. However, the vast majority of these are essentially duplicates. For example, the annotations for each member of the `printf` family of functions are essentially identical. When these "cut-and-paste" duplicates are eliminated, the total number is only 21. For the file disclosure policy, there are 65 such annotations, with a median size of 7 lines each and an average size of 6.52 lines. Again, the majority of these are essentially duplicates. When these are accounted for, there are only 36 unique annotations.

To understand the difference between analysis and policy annotations, we now discuss several different use cases. In the simplest case, the desired policy exists and there is no need to touch any annotation file. In other cases, a security expert may wish to modify an existing policy, for example, by calling a sanitization function when a violation is detected. Here, only the policy annotations require changes to account for the sanitization code. Finally, in the most invasive case, a new data flow analysis must be defined, in which case new analysis and policy annotations must be written.

The annotations themselves are not difficult to write. Our annotation files only use seven major constructs, so the language is easy to understand. All of these constructs are shown in the example in Section 4.3.1, and the annotations shown are representative of the kind that must be written. In any case, the information provided by the annotations is required by any policy-enforcing system; in our system such information is specified by annotation files rather than being imbedded in the code.

6. CONCLUSIONS

In this paper we have presented a compiler-based system that enforces security policies by tracking dynamic data flow through programs. We have demonstrated the generality of our system by using it to enforce two security policies, one that prevents unwanted file disclosure and another that prevents format string attacks. We have also shown that our system produces low overhead when applied

to both compute-bound applications and I/O bound server applications.

Although our current implementation is a source-to-source translator for the C language, our techniques are applicable to other modern languages and even binary code. For example, our static data flow analysis could be implemented in a static binary rewriting system, producing a system that protects binary code from attacks while using static analysis to reduce the runtime cost. Such a system would have to deal with the difficulty of building the control flow graph (*i.e.*, the difficulty of discovering all instructions), so in many cases it would not produce the same low overheads that we report in this paper.

Acknowledgments

This work was supported by NSF grant CNS-0509354, Air Force Research Laboratory contract FA8750-07-C-0035 from the Disruptive Technology Office, and a grant from the Intel Research Council. We thank Vitaly Shmatikov, Lili Qiu, and E Lewis for their helpful comments on this paper.

7. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the ACM Conference on Computer and Communication Security*, pages 340–353, 2005.

[2] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 143–159, 2002.

[3] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering*, pages 332–341, 2005.

[4] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 251–262, 2000.

[5] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report 2547, MITRE, March 1973.

[6] E. D. Berger and B. G. Zorn. DieHard: Probabalistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 158–168, 2006.

[7] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, pages 271–286, 2005.

[8] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ES-TR-76-372, Electronic Systems Division, Hanscom Air Force Base, April 1977.

[9] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 147–160, 2006.

[10] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 378–387, 2005.

[11] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing*, pages 196–206, 2007.

[12] M. Costa, J. Crowcroft, M. Castro, A. Rwostron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of Internet worms. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, pages 133–147, 2005.

[13] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 15–23, 2001.

[14] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, 2003.

[15] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, 1998.

[16] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 221–232, 2004.

[17] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 482–493, 2007.

[18] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[19] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, Ithaca, New York, 2003.

[20] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January/February 2002.

[21] S. Z. Guyer. *Incorporating Domain-Specific Information into the Compilation Process*. PhD thesis, The University of Texas at Austin, Austin, TX, 2003.

[22] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 39–52, 1999.

[23] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *Proceedings of the 10th Annual Static Analysis Symposium*, pages 214–236, June 2003.

[24] S. Z. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE, Special issue on program generation, optimization and adaptation*, 93(2):342–357, January-February 2005.

[25] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, 2004.

[26] J. C. Huang. Detection of data flow anomaly through program instrumentation. *IEEE Transactions on Software Engineering*, SE-5(3):226–236, May 1979.

[27] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and

Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, 2002.

[28] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 4th International Workshop on Automated and Algorithmic Debugging*, pages 13–26, 1997.

[29] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–176, January 1976.

[30] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th Annual USENIX Security Symposium*, pages 191–206, 2002.

[31] L. C. Lam and T.-C. Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 463–472, 2006.

[32] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: A program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, and Applications*, pages 365–383, 2005.

[33] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[34] National Security Agency Information Systems Security Organization. Labeled security protection profile version 1b, October 1999.

[35] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[36] J. Newsome, D. Brumley, and D. Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the Network and Distributed Security Symposium*, 2006.

[37] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed Security Symposium*, 2005.

[38] A. Nguyen-Tong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, pages 295–308, 2005.

[39] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM Symposium on Microarchitecture*, pages 135–148, 2006.

[40] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[41] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[42] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, pages 201–218, 2001.

[43] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[44] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.

[45] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[46] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly & Associates, Sebastopol, California, United States, third edition, 2000.

[47] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[48] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, pages 121–136, 2006.