Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks

Wei Xu Sandeep Bhatkar R. Sekar
Department of Computer Science
Stony Brook University, Stony Brook, NY 11794-4400
{weixu,sbhatkar,sekar}@cs.sunysb.edu

Abstract

Policy-based confinement, employed in SELinux and specification-based intrusion detection systems, is a popular approach for defending against exploitation of vulnerabilities in benign software. Conventional access control policies employed in these approaches are effective in detecting privilege escalation attacks. However, they are unable to detect attacks that "hijack" legitimate access privileges granted to a program, e.g., an attack that subverts an FTP server to download the password file. (Note that an FTP server would normally need to access the password file for performing user authentication.) Some of the common attack types reported today, such as SQL injection and cross-site scripting, involve such subversion of legitimate access privileges. In this paper, we present a new approach to strengthen policy enforcement by augmenting security policies with information about the trustworthiness of data used in security-sensitive operations. We evaluated this technique using 9 available exploits involving several popular software packages containing the above types of vulnerabilities. Our technique sucessfully defeated these exploits.

1 Introduction

Information flow analysis (a.k.a. *taint analysis*) has played a central role in computer security for over three decades [1, 10, 8, 30, 25]. The recent works of [22, 28, 5] demonstrated a new application of this technique, namely, detection of exploits on contemporary software. Specifically, these techniques track the source of each byte of data that is manipulated by a program during its execution, and detect attacks that overwrite pointers with untrusted (i.e., attacker-provided) data. Since this is an essential step in most buffer overflow and related attacks, and since benign uses of programs should never involve outsiders supplying pointer values, such attacks can be detected accurately by these new techniques.

In this paper, we build on the basic idea of using fine-grained taint analysis for attack detection, but expand its scope by showing that the technique can be applied to detect a much wider range of attacks prevalent today. Specifically, we first develop a source-to-source transformation of C programs that can efficiently track information flows at runtime. We combine this information with security policies that can reason about the source of data used in security-critical operations. This combination turns out to be powerful for attack detection, and offers the following advantages over previous techniques:

- *Practicality.* The techniques of [28] and [5] rely on hardware-level support for taint-tracking, and hence cannot be applied to today's systems. TaintCheck [22] addresses this drawback, and is applicable to arbitrary COTS binaries. However, due to difficulties associated with static analysis (or transformation) of binaries, their implementation uses techniques based on a form of runtime instruction emulation [21], which causes a significant slowdown, e.g., Apache server response time increases by a factor of 10 while fetching 10KB pages. In contrast, our technique is much faster, increasing the response time by a factor of 1.1.
- *Broad applicability.* Our technique is directly applicable to programs written in C, and several other scripting languages (e.g., PHP, Bash) whose interpreters are implemented in C. Security-critical servers are most frequently implemented in C. In addition, PHP and similar scripting languages are common choices for implementing web applications, and more generally, server-side scripts.
- *Ability to detect a wide range of common attacks.* By combining expressive security policies with fine-grained taint information, our technique can address a broader range of attacks than previous techniques. Figure 1 shows the distribution of the 139 COTS software vulnerabilities reported in 2003 and 2004 in the most recent official CVE dataset (Ver. 20040901). Our technique is applicable for detecting exploitations of about 2/3rds of these vulnerabilities, including *buffer overflows, format-string attacks, SQL injection, cross-site scripting, command and shell-code injection,* and *directory traversal.* In contrast, previous approaches typically handled smaller attack classes,

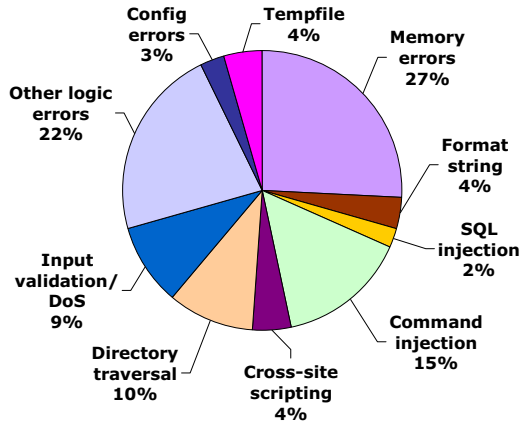

Figure 1: Breakdown of CVE software security vulnerabilities (2003 and 2004)

e.g., [7, 9, 2, 22, 28, 5] handle buffer overflows, [6] handles format string attacks, and [24, 23] handle injection attacks involving strings.

The focus of this paper is on the development of practical fine-grained dynamic taint-tracking techniques, and on illustrating how this information can be used to significantly strengthen conventional access control policies. For this purpose, we use simple taint-enhanced security policies. Our experimental evaluation, involving readily available exploits that target vulnerabilities in several popular applications, shows that the technique is effective against these exploits. Nevertheless, many of these policies need further refinement before they can be expected to stand up to skilled attackers. Section 7.2 discusses some of the issues in policy refinement, but the actual development of such refined policies is not a focus area of this paper.

We have successfully applied our technique to several medium to large programs, such as the PHP interpreter (300KLOC+) and `glibc`, the GNU standard C library (about 1MLOC). By leveraging the low-level nature of the C language, our implementation works correctly even in the face of memory errors, type casts, aliasing, and so on. At the same time, by exploiting the high-level nature of C (as compared to binary code), we have developed optimizations that significantly reduce the runtime overheads of fine-grained dynamic taint-tracking.

Approach Overview. Our approach consists of the following steps:

- *Fine-grained taint analysis:* The first step in our approach is a source-to-source transformation of C programs to perform runtime taint-tracking. Taint originates at input functions, e.g., a `read` or `recv` function used by a server to read network input. Input operations that return untrusted input are specified using *marking specifications* described in Section 4. In the transformed program, each byte of memory is associated with one bit (or more) of taint information. Logically, we can view the taint information as a bit-array `tagmap`, with `tagmap[a]` representing the taint information associated with the data at memory location a. As data propagates through memory, the associated taint information is propagated as well. Since taint information is associated with memory locations (rather than variables), our technique can ensure correct propagation of taint in the presence of memory errors, aliasing, type casts, and so on.
- *Policy enforcement:* This step is driven by *security policies* that are associated with security-critical functions. There are typically a small number of such functions, e.g., system calls such as `open` and `execve`, library functions such as `vfprintf`, functions to access external modules such as an SQL database, and so on. The security policy associated with each such function checks its arguments for "unsafe" content.

Organization of the Paper. We begin with motivating attack examples in Section 2. Section 3 describes our source-code transformation for fine-grained taint-tracking. Our policy language and sample policies are described in Section 4. The implementation of our approach is described in Section 5, followed by the experimental evaluation in Section 6. Section 7 discusses implicit information flows and security policy refinement. Section 8 presents related work. Finally, concluding remarks appear in Section 9.

2 Motivation for Taint-Enhanced Policies

In this section, we present several motivating attack examples. We conclude by pointing out the integral role played by taint analysis as well as security policies in detecting these attacks.

2.1 SQL and Command Injection. SQL injection is a common vulnerability in web applications. These server-side applications communicate with a web browser client to collect data, which is subsequently used to construct an SQL query that is sent to a back-end database. Consider the statement (written in PHP) for constructing an SQL query used to look up the price of an item specified by the variable `name`.

```
$cmd = "SELECT price FROM products WHERE
       name = '" . $name . "'"
```

If the value of `name` is assigned from an HTML form field that is provided by an untrusted user, then an SQL injection is possible. In particular, an attacker can provide the following value for `name`:

```
xyz'; UPDATE products SET price = 0 WHERE
name = 'OneCaratDiamondRing
```

With this value for `name`, `cmd` will take the value:

```
SELECT ...  WHERE name =
'xyz'; UPDATE products SET price = 0 WHERE
```

```
name = 'OneCaratDiamondRing'
```

Note that semicolons are used to separate SQL commands. Thus, the query constructed by the program will first retrieve the price of some item called `xyz`, and then set the price of another item called `OneCaratDiamondRing` to zero. This attack enables the attacker to purchase this item later for no cost.

Fine-grained taint analysis will mark every character in the query that is within the box as tainted. Now, a policy that precludes tainted control-characters (such as semicolons and quotes) or commands (such as `UPDATE`) in the SQL query will defeat the above attack. A more refined policy is described in Section 7.2.

Command injection attacks are similar to SQL injection: they involve untrusted inputs being used as to construct commands executed by command interpreters (e.g., `bash`) or the argument to `execve` system call.

2.2 Cross-Site Scripting (XSS). Consider an example of a bank that provides a "ATM locator" web page that customers can use to find the nearest ATM machine, based on their ZIP code. Typically, the web page contains a form that submits a query to the web site, which looks as follows:

```
http://www.xyzbank.com/findATM?zip=90100
```

If the ZIP code is invalid, the web site typically returns an error message such as:

```
<HTML> ZIP code not found: 90100 </HTML>
```

Note in the above output from the web server, the user-supplied string `90100` is reproduced. This can be used by an attacker to construct an XSS attack as follows. To do this, the attacker may send an HTML email to an unsuspecting user, which contains text such as:

```
To claim your reward, please click <a href="
  http://www.xyzbank.com/findATM?zip=
  <script%20src='http://www.attacker.com/
   malicious_script.js'></script>">here</a>
```

When the user clicks on this link, the request goes to the bank, which returns the following page:

```
<HTML> ZIP code not found:
  <script src='http://www.attacker.com/
   malicious_script.js'></script> </HTML>
```

The victim's browser, on receiving this page, will download and run Javascript code from the attacker's web site. Since the above page was sent from `http://www.xyzbank.com`, this script will have access to sensitive information stored on the victim computer that pertains to the bank, such as cookies. Thus, the above attack will allow cookie information to be stolen. Since cookies are often used to store authentication data, stealing them can allow attackers to perform financial transactions using victim's identity.

Fine-grained taint analysis will mark every character in the zip code value as tainted. Now the above cross-site scripting attack can be prevented by disallowing tainted `script` tags in the web application output.

2.3 Memory Error Exploits. There are many different types of memory error exploits, such as *stack-smashing*, *heap-overflows* and *integer overflows*. All of them share the same basic characteristics: they exploit bounds-checking errors to overwrite security-critical data, almost always a code pointer or a data pointer, with attacker-provided data. When fine-grained taint analysis is used, it will mark the overwritten pointer as tainted. Now, this attack can be stopped by a policy that prohibits dereferencing of tainted pointers.

2.4 Format String Vulnerabilities. The `printf` family of functions (which provide formatted printing in C) take a format string as a parameter, followed by zero or more parameters. A common misuse of these functions occurs when untrusted data is provided as the format string, as in the statement "`printf(s)`." If `s` contains an alphanumeric string, then this will not cause a problem, but if an attacker inserts format directives in `s`, then she can control the behavior of `printf`. In the worst case, an attacker can use the "`%n`" format directive, which can be used to overwrite a return address with attacker-provided data, and execute injected binary code.

When fine-grained taint analysis is used, the format directives (such as "`%n`") will be marked as tainted. The above attack can be then prevented by a taint-enhanced policy that disallows tainted format directives in the format string argument to the `printf` family of functions.

2.5 Attacks that "Hijack" Access Privileges. In this section, we consider attacks that attempt to evade detection by staying within the bounds of normal accesses made by an application. These attacks are also referred to as the confused deputy attacks [13].

Consider a web browser vulnerability that allows an attack (embedded within a web page) to upload an arbitrary file f owned by the browser user without the user's consent. Since the browser itself needs to access many of the user's files (e.g., cookies), a policy that prohibits access to f may prevent normal browser operations. Instead, we need a policy that can infer whether the access is being made during the normal course of an operation of the browser, or due to an attack. One way to do this is to take the taint information associated with the file name. If this file is accessed during normal browser operation, the file name would have originated within its program text or from the user. However, if the file name originated from a remote web site (i.e., an untrusted source), then it is likely to be an attack. Similar examples include attacks on (a) P2P applications to upload (i.e., steal) user files, and (b) FTP servers to down-

load sensitive files such as the password file that are normally accessed by the server.

A variant of the above scenario occurs in the context of *directory traversal* attacks, where an attacker attempts to access files outside of an authorized directory, e.g., the document root in the case of a web server. Typically, this is done by including "`..`" characters in file names to ascend above the document root. In case the victim application already incorporates checks for "`..`" characters, attacker may attempt to evade this check by replacing "`.`" with its hexadecimal or Unicode representation, or by using various escape sequences. A taint-enhanced policy can be used to selectively enforce a more restrictive policy on file access when the file name is tainted, e.g., accesses outside of the document root directory may be disallowed. Such a policy would not interfere with the web server's ability to access other files, e.g., its access log or error log.

The key point about all attacks discussed in this section is that conventional access control policies cannot detect them. This is because the attacks do not stray beyond the set of resources that are normally accessed by a victim program. However, taint analysis provides a clue to *infer the intended use of an access.* By incorporating this inferred intent in granting access requests, taint-enhanced policies can provide better discrimination between attacks and legitimate uses of the privileges granted to a victim application.

2.6 Discussion. The examples discussed above bring out the following important points:

- *Importance of fine-grained taint information.* If we used coarser granularity for taint-tracking, e.g., by marking a program variable as tainted or untainted, we would not be able to detect most of the attacks described above. For instance, in the case of SQL injection example, the variable `cmd` containing the SQL query will always be marked as tainted, as it derives part of its value from an untrusted variable `name`. As a result, we cannot distinguish between legitimate uses of the web application, when `name` contains an alphanumeric string, from an attack, when `name` contains characters such as the semicolon and SQL commands. A similar analysis can be made in the case of stack-smashing and format-string attacks, cross-site scripting, directory traversal, and so on.
- *Need for taint-enhanced policies.* It is not possible to prevent these attacks by enforcing conventional access control policies. For instance, in the SQL injection example, one cannot use a policy that uniformly prevents the use of semicolons and SQL commands in `cmd`: such a policy would preclude any use of the database, and cause the web application to fail. Similarly, in the memory error example, one cannot have a working program if all control transfers through pointers are prevented. Finally, the examples in Section 2.5 were specifically chosen to illustrate the need for combining taint information into policies.

  Another point to be made in this regard is that attacks are not characterized simply by the presence or absence of tainted information in arguments to security-critical operations. Instead, it is necessary to develop policies that govern the manner in which tainted data is used in these arguments.

3 Transformation for Taint Tracking

There are three main steps in taint-enhanced policy enforcement: (i) *marking*, i.e., identifying which external inputs to the program are untrusted and should be marked as tainted, (ii) *tracking* the flow of taint through the program, and (iii) *checking* inputs to security-sensitive operations using taint-enhanced policies. This section discusses tracking, which is implemented using a source-to-source transformation on C programs. The other two steps are described in Section 4.

3.1 Runtime Representation of Taint

Our technique tracks taint information at the level of bytes in memory. This is necessary to ensure accurate taint-tracking for type-unsafe languages such as C, since the approach can correctly deal with situations such as out-of-bounds array writes that overwrite adjacent data. A one-bit taint-tag is used for each byte of memory, with a '0' representing the absence of taint, and a '1' representing the presence of taint. A bit-array `tagmap` stores taint information. The taint bit associated with a byte at address a is given by `tagmap[a]`.

3.2 Basic Transformation

The source-code transformation described in this section is designed to track *explicit information flows* that take place through assignments and arithmetic and bit-operations. Flows that take place through conditionals are addressed in Section 7.1. It is unusual in C programs to have boolean-valued variables that are assigned the results of relational or logical operations. Hence we have not considered taint propagation through such operators in this paper. At a high-level, explicit flows are simple to understand:

- the result of an arithmetic/bit expression is tainted if any of the variables in the expression is tainted;
- a variable x is tainted by an assignment $x = e$ whenever e is tainted.

Specifically, Figure 2 shows how to compute the taint value $T(E)$ for an expression E. Figure 3 defines how a statement S is transformed, and uses the definition of $T(E)$. When describing the transformation rules, we

E	$T(E)$	Comment
c	0	Constants are untainted
v	$tag(\&v, sizeof(v))$	$tag(a, n)$ refers to n bits starting at $tagmap[a]$
$\&E$	0	An address is always untainted
$*E$	$tag(E, sizeof(*E))$	
$(cast)E$	$T(E)$	Type casts don't change taint.
$op(E)$	$T(E)$ 0	for arithmetic/bit op otherwise
$E_1\ op\ E_2$	$T(E_1)\ ||\ T(E_2)$ 0	for arithmetic/bit op otherwise

Figure 2: Definition of taint for expressions

S	$Trans(S)$
$v = E$	$v = E;$ $tag(\&v, sizeof(v)) = T(E);$
$S_1; S_2$	$Trans(S_1); Trans(S_2)$
$if\ (E)\ S_1$ $else\ S_2$	$if\ (E)\ Trans(S_1)$ $else\ Trans(S_2)$
$while\ (E)\ \ S$	$while\ (E)\ \ Trans(S)$
$return\ E$	$return\ (E, T(E))$
$f(a)\ \{\ S\ \}$	$f(a, ta)\ \{$ $tag(\&a, sizeof(a)) = ta; Trans(S)\}$
$v = f(E)$ $v = (*f)(E)$	$(v, tag(\&v, sizeof(v))) = f(E, T(E))$ $(v, tag(\&v, sizeof(v))) = (*f)(E, T(E))$

Figure 3: Transformation of statements for taint-tracking

use a simpler form of C (e.g. expressions have no side effects). In our implementation, we use the CIL [19] toolkit as the C front end to provide the simpler C form that we need.

The transformation rules are self-explanatory for most part, so we explain only the function-call related transformations. Consider a statement $v = f(E)$, where f takes a single argument. We introduce an additional argument ta in the definition of f so that the taint tag associated with its (single) parameter could be passed in. ta is explicitly assigned as the taint value of a at the beginning of f's body. (These two steps are necessary since the C language uses call-by-value semantics. If call-by-reference were to be used, then neither step would be needed.) In a similar way, the taint associated with the return value has to be explicitly passed back to the caller. We represent this in the transformation by returning a pair of values as the return value. (In our implementation, we do not actually introduce additional parameters or return values; instead, we use a second stack to communicate the taint values between the caller and the callee.) It is straight-forward to extend the transformation rules to handle multi-argument functions.

We conclude this section with a clarification on our notion of soundness of taint information. Consider any variable x at any point during any execution of a transformed program, and let a denote the location of this variable. If the value stored at a is obtained from any tainted input through assignments and arithmetic/bit operations, then `tagmap[a]` should be set. Note that by referring to the location of x rather than its name, we require that taint information be accurately tracked in the presence of memory errors. To support this notion of soundness, we needed to protect the `tagmap` from corruption, as described in Section 3.4.

3.3 Optimizations

The basic transformation described above is effective, but introduces high overheads, sometimes causing a slowdown by a factor of 5 or more. To improve performance, we have developed several interesting runtime and compile-time optimizations that have reduced overheads significantly. More details about the performance can be found in Section 6.4.

3.3.1 Runtime Optimizations In this section, we describe optimizations to the runtime data structures.

Use of 2-bit taint values. In the implementation, accessing of taint-bits requires several bit-masking, bit-shifting and unmasking operations, which degrade performance significantly. We observed that if 2-bit taint tags are used, the taint value for an integer will be contained within a single byte (assuming 32-bit architecture), thereby eliminating these bit-level operations. Since integer assignments occur very frequently, this optimization is quite effective.

This approach does increase the memory requirement for `tagmap` by a factor of two, but on the other hand, it opens up the possibility of tracking richer taint information. For instance, it becomes possible to associate different taint tags with different input sources and track them independently. Alternatively, it may be possible to use the two bits to capture "degree of taintedness."

Allocation of `tagmap`. Initially, we used a global variable to implement `tagmap`. But the initialization of this huge array (1GB) that took place at the program start incurred significant overheads. Note that tag initialization is warranted only for static data that is initialized at program start. Other data (e.g., stack and heap data) should be initialized (using assignments) before use in a correctly implemented program. When these assignments are transformed, the associated taint data will also be initialized, and hence there is no need to initialize such taint data in the first place. So, we allocated `tagmap` dynamically, and initialized only the locations corresponding to static data. By using `mmap` for this allocation, and by performing the allocation at a fixed address that is unused in Linux (our implementation platform), we ensured that runtime accesses to `tagmap` elements will be no more expensive than that of a statically allocated array (whose base address is also determined at compile-time).

The above approach reduced the startup overheads, but

the mere use of address space seemed to tie up OS resources such as page table entries, and significantly increased time for `fork` operations. For programs such as shells that fork frequently, this overhead becomes unacceptable. So we devised an *incremental allocation* technique that can be likened to user-level page-fault handling. Initially, `tagmap` points to 1GB of address space that is unmapped. When any access to `tagmap[i]` is made, it results in a UNIX signal due to a memory fault. In the transformed program, we introduce code that intercepts this signal. This code queries the operating system to determine the faulting address. If it falls within the range of `tagmap`, a chunk of memory (say, 16KB) that spans the faulting address is allocated using `mmap`. If the faulting address is outside the range of `tagmap`, the signal is forwarded to the default signal handler.

3.3.2 Compile-time Optimizations

Use of local taint tag variables. In most C programs, operations on local variables occur much more frequently than global variables. Modern compilers are good at optimizing local variable operations, but due to possible aliasing, most such optimizations cannot be safely applied to global arrays. Unfortunately, the basic transformation introduces one operation on a global array for each operation on a local variable, and this has the effect of more than doubling the runtime of transformed programs. To address this problem we modified our transformation so that it uses local variables to hold taint information for local variables, so that the code added by the transformer can be optimized as easily as the original code.

Note, however, that the use of local tag variables would be unsound if aliasing of a local variable is possible. For example, consider the following code snippet:

```
int x; int *y = &x;
x = u; *y = v;
```

If `u` is untainted and `v` is tainted, then the value stored in `x` should be tainted at the end of the above code snippet. However, if we introduced a local variable, say, `tag_x`, to store the taint value of `x`, then we cannot make sure that it will get updated by the assignment to `*y`.

To ensure that taint information is tracked accurately, our transformation uses local taint tag variables only in those cases where no aliasing is possible, i.e., the optimization is limited to simple variables (not arrays) whose address is never taken. However, this alone is not enough, as aliasing may still be possible due to memory errors. For instance, a simple variable `x` may get updated due to an out-of-bounds access on an adjacent array, say, `z`. To eliminate this possibility, we split the runtime stack into two stacks. The main stack stores only simple variables whose addresses are never taken. This stack is also used for call-return. All other local variables are stored in the second stack, also called *shadow stack*.

The last possibility for aliasing arises due to pointer-forging. In programs with possible memory errors, a pointer to a local variable may be created. However, with the above transformation, any access to the main stack using a pointer indicates a memory error. We show how to implement an efficient mechanism to prevent access to some sections of memory in the transformed program. Using this technique, we prevent all accesses to the main stack except using local variable names, thus ensuring that taint information can be accurately tracked for the variables on the main stack using local taint tag variables.

Intra-procedural dependency analysis is performed to determine whether a local variable can ever become tainted, and to remove taint updates if it cannot. Note that a local variable can become tainted *only if* it is involved in an assignment with a global variable, a procedure parameter, or another local variable that can become tainted. Due to aliasing issues, this optimization is applied only to variables on the main stack.

3.4 Protecting Memory Regions

To ensure accurate taint-tracking, it is necessary to preclude access to certain regions of memory. Specifically, we need to ensure that the `tagmap` array itself cannot be written by the program. Otherwise, `tagmap` may be corrupted due to programming errors, or even worse, a carefully crafted attack may be able to evade detection by modifying the `tagmap` to hide the propagation of tainted data. A second region that needs to be protected is the main stack. Third, it would be desirable to protect memory that should not directly be accessed by a program, e.g., the GOT. (Global Offset Table is used for dynamic linking, but there should not be any reference to the GOT in the C code. If the GOT is protected in this manner, that would rule out attacks based on corrupting a function pointer in the GOT.)

The basic idea is as follows. Consider an assignment to a memory location a. Our transformation ensures that an access to `tagmap[a]` will be made before a is accessed. Thus, in order to protect a range of memory locations l—h, it is enough if we ensure that `tagmap[l]` through `tagmap[h]` will be unmapped. This is easy to do, given our incremental approach to allocation of `tagmap`. Now, any access to addresses l through h will result in a memory fault when the corresponding `tagmap` location is accessed.

Note that l and h cannot be arbitrary: they should fall on a 16K boundary, if the page size is 4KB and if 2 bit tainting is used. This is because `mmap` allocates memory blocks whose sizes are a multiple of a page size. This alignment requirement is not a problem for `tagmap`, since we can align it on a 16K boundary. For the main stack, a potential issue arises because the bottom of the

Attack Type	Policy	Comment
Control-flow hijack	$\mathbf{jmp}(addr)$ | $addr$ **matches** $(any+)^t \rightarrow term()$	Tainted values cannot be used as a target of control transfer
Format string	Format = "%[^%]" vfprintf(fmt) |fmt **matches** $any*$ (Format)$^T any* \rightarrow reject()$	Format directives (e.g.%n) should not be tainted
Directory traversal	DirTraversalModifier = ".." file_function($path$) = $open(path,)$ || $unlink(path)$ || ... file_function($path$) | $path$ **matches** $any*$ (DirTraversalModifier)$^T any*$ && $escapeRootDir(path) \rightarrow reject()$	If $path$ contains tainted directory traversal strings (e.g. ".."), then the real path of $path$ should not go outside the top level directories that are allowed to be accessed by the program, e.g. DocumentRoot and cgi-bin for httpd
Cross-site scripting	ScriptTag = "<script" | ... html_print_function(str) | str **matches** (StrIdNum|Delim) $*$ (ScriptTag)$^T any*$ $\rightarrow reject()$	No tainted script tags (e.g. $script$) should be output to HTML.
SQL injection	SqlMetachar = "'" | ";" | "/*" | ... sql_query_function($query$) | $query$ **matches** (StrIdNum|Delim) $*$ (SqlMetachar)$^T any*$ $\rightarrow reject()$	SQL query string should not contain tainted SQL meta-chars
Shell command injection	ShellMetachar = ";" | "&&" | ... shell_command_function(cmd) | cmd **matches** (StrIdNum|Delim) $*$ (ShellMetachar)$^T any*$ $\rightarrow reject()$	cmd argument of $system$ or $popen$ should not contain tainted shell meta-chars

Figure 4: Illustrative security policies

stack holds environment variables and command-line arguments that are arrays. To deal with this problem, we first introduce a gap in the stack in `main` so that its top is aligned on a 16K boundary. The region of main stack above this point is protected using the above mechanism. This means that it is safe to use local tag variables in any function except `main`.

4 Marking and Policy Specification

4.1 Marking Trusted and Untrusted Inputs

Marking involve associating taint information with all the data coming from external sources. If all code, including libraries, is transformed, then marking needs to be specified for system calls that return inputs, for environment variables and command-line arguments. (If some libraries are not transformed, then marking specifications may be needed for untransformed library functions that perform inputs.) Note that we can treat command-line arguments and environment variables as arguments to `main`. Thus, marking specifications can, in every case, be associated with a function call.

Marking actions are specified using BMSL (Behavior Monitoring Specification Language) [29, 3], an event-based language that is designed to support specification of security policies and behaviors. BMSL specifications consist of rules of the form $event\_pattern \longrightarrow action$. We use BMSL in a simplified way in this paper — in particular, $event\_pattern$ will be of the form $event \mid condition$, where $event$ identifies a function. When this function returns, and (the optional) $condition$ holds, $action$ will be executed. The event corresponding to a function will take an additional argument that captures the return value from the function. Both the *condition* and the *action* can use external functions (written in C or C++). Moreover, the *action* can include arithmetic and logical operations, as well as if-then-else. Consider the following example:

```
read(fd, buf, size, rv)|(rv > 0) →
  if (isNetworkEndpoint(fd))
      taint_buffer(buf, rv);
  else untaint_buffer(buf, rv);
```

This rule states that when the `read` function returns, the `buf` argument will be tainted, based on whether the read was from a network or not, as determined by the external function `isNetworkEndpoint`. The actual tainting is done using two support functions `taint_buffer` and `untaint_buffer`.

Note that every input action needs to have an associated marking rule. To reduce the burden of writing many rules, we provide default rules for all system calls that untaint the data returned by each system call. Specific rules that override these default rules, such as the rule given above, can then be supplied by a user.

4.2 Specifying Policies

Security policies are also written using BMSL, but these rules are somewhat different from the marking rules. For a policy rule involving a function f, its *condition* component is examined immediately *before* any invocation of f. To simplify the policy specification, *abstract events* can be defined to represent a set of functions that share the same security policy. (Abstract events can be thought of as macros.)

The definition of *condition* is also extended to support regular-expression based pattern matching, using the

keyword `matches`. We use *taint-annotated* regular expressions defined as follows. A tainted regular expression is obtained for a normal regular expression by attaching a superscript t, T or u. A string s will match a taint-annotated regular expression r^t provided that s matches r, and at least one of the characters in s is tainted. Similarly, s will match r^T provided *all* characters in s are tainted. Finally, s will match r^u provided none of the characters in s are tainted.

The predefined pattern any matches any single character. Parentheses and other standard regular expression operators are used in the usual way. Moreover, taint-annotated regular expressions can be named, and the name can be reused subsequently, e.g., `StrIdNum` used in many sample policy rules is defined as:

```
StrIdNum = String | Id | Num
```

where `String`, `Id` and `Num` denote named regular expressions that correspond respectively to strings, identifiers and numbers. Also, `Delim` denotes delimiters.

Figure 4 shows the examples of a few simple policies to detect various attacks. The *action* component of these policies make use of two support functions: $term()$ terminates the program execution, while $reject()$ denies the request and returns with an error.

For the control-flow hijack policy, we use a special keyword `jmp` as a function name, as we need some special way to capture low-level control-flow transfers that are not exposed as a function call in the C language. The policy states that if any of the bytes in the target address are tainted, then the program should be terminated.

For format string attacks, we only define a policy for `vfprintf`, because `vfprintf` is the common function used internally to implement all other `printf` family of functions. All format directives in a format string begin with a "%", and are followed by a character other than "%". (The sequence "%%" will simply print a "%", and hence can be permitted in the format string.)

Example policies to detect four other attacks, namely, directory traversal, cross-site scripting, SQL injection and shell command injection are also shown in Figure 4. The comments associated with the policies provide an intuitive description of the policy. These policies were able to detect all of the attacks considered in our evaluation, but we do not make any claim that the policies are good enough to detect all possible attacks in these categories. A discussion of how skilled attackers may evade some of these policies, and some directions for refining policies to stand up to such attacks, can be found in Section 7.2. The main strength of the approach presented in this paper is that the availability of fine-grained taint information makes it possible for a knowledgeable system administrator to develop such refined policies.

5 Implementation

We have implemented the program transformation technique described in Section 3. The transformer consists of about 3,600 lines of Objective Caml code and uses the CIL [19] toolkit as the front end to manipulate C constructs. Our implementation currently handles `glibc` (containing around 1 million LOC) and several other medium to large applications. The complexity and size of `glibc` demonstrated that our implementation can handle "real-world" code. We summarize some of the key issues involved in our implementation.

5.1 Coping with Untransformed Libraries

Ideally, all the libraries used by an application will be transformed using our technique so as to enable accurate taint tracking. In practice, however, source code may not be available for some libraries, or in rare cases, some functions in a library may be implemented in an assembly language. One option with such libraries is to do nothing at all. Our implementation is designed to work in these cases, but clearly, the ability to track information flow via untransformed functions is lost. To overcome this problem, our implementation offers two features. First, it produces warnings when a certain function could not be transformed. This ensures that inaccuracies will not be introduced into taint tracking without explicit knowledge of the user. When the user sees this warning, she may decide that the function in question performs largely "read" operations, or will never handle tainted data, and hence the warning can safely be ignored. If not, then our implementation supports *summarization functions* that specify how taint information is propagated by a function. For instance, we use the following summarization function for the `memcpy`. Summarization functions are also specified in BMSL, and use support functions to copy taint information. A summarization function for f would be invoked in the transformed code when f returns.

```
memcpy(dest, src, n) →
  copy_buffer_tagmap(dest, src, n);
```

So far, we had to write summarization functions for two `glibc` functions that are written in assembly and copy data, namely, `memcpy` and `memset`. In addition, `gcc` replaces calls to some functions such as `strcpy` and `strdup` with its own code, necessitating an additional 13 summarization functions.

5.2 Injecting Marking, Checking and Summarization Code into Transformed Programs

In our current implementation, the marking specifications, security policies, and summarization code associated with a function f are all injected into the transformed program by simply inlining (or explicitly call-

CVE#	Program	Language	Attack type	Attack description
CAN-2003-0201	samba	C	Stack smashing	Buffer overflow in call_trans2open function
CVE-2000-0573	wu-ftpd	C	Format string	via SITE EXEC command
CAN-2005-1365	pico server	C	Directory traversal	Command execution via URL with multiple leading "/" characters and ".."
CAN-2003-0486	phpBB 2.0.5	PHP	SQL injection	via topic_id parameter
CAN-2005-0258	phpBB 2.0.5	PHP	Directory traversal	Delete arbitrary file via ".." sequences in avatarselect parameter
CAN-2002-1341	SquirrelMail 1.2.10	PHP	Cross site scripting	Insert script via the mailbox parameter in read_body.php
CAN-2003-0990	SquirrelMail 1.4.0	PHP	Command injection	via meta-character in"To:" field
CAN-2005-1921	PHP XML-RPC	PHP	Command injection	Eval injection
CVE-1999-0045	nph-test-cgi	BASH	Shell meta-character expansion	using '*' in `$QUERY_STRING`

Figure 5: Attacks used in effectiveness evaluation

ing) the relevant code before or after the call to f. In the future, we anticipate these code to be decoupled from the transformation, and be able to operate on binaries using techniques such as library interposition. This would enable a site administrator to alter, refine or customize her notions of "trustworthy input" and "dangerous arguments" without having access to the source code.

6 Experimental Evaluation

The main goal of our experiments was to evaluate attack detection (Section 6.1), and runtime performance (Section 6.4). False positives and false negatives are discussed in Sections 6.2 and 6.3.

6.1 Attack Detection

Table 5 shows the attacks used in our experiments. These attacks were chosen to cover the range of attack categories we have discussed, and to span multiple programming languages. Wherever possible, we selected attacks on widely-used applications, since it is likely that obvious security vulnerabilities in such applications would have been fixed, and hence we are more likely to detect more complex attacks.

In terms of marking, all inputs read from network (using `read`, `recv` and `recvfrom`) were marked as tainted. Since the PHP interpreter is configured as a module for Apache, the same technique works for PHP applications as well. Network data is tainted when it is read by Apache, and this information propagates through the PHP interpreter, and in effect, through the PHP application as well. The policies used in our attack examples were already discussed in Section 4.

To test our technique, we first downloaded the software packages shown in Figure 5. We downloaded the exploit code for the attacks, and verified that they worked as expected. Then we used transformed C programs and interpreters with policy checking enabled, and verified that each one of the attacks were prevented by these policies without raising false alarms.

Network Servers in C.

- `wu-ftpd` versions 2.6.0 and lower have a format string vulnerability in `SITE EXEC` command that allows arbitrary code execution. The attack is stopped by the policy that the format directive `%n` in a format string should not be tainted.
- `samba` versions 2.2.8 and lower have a stack-smashing vulnerability in processing a type of request called "transaction 2 open." No policy is required to stop this attack — the stack-smashing step ends up corrupting some data on the shadow stack rather than the main stack, so the attack fails.

  If we had used an attack that uses a heap overflow to overwrite a GOT entry (which is common with heap overflows), this too would be detected without the need for any policies due to the technique described in Section 3.4 for preventing the GOT from being directly accessed by the C code. The reasoning is that before the injected code gets control, the GOT entry has to be clobbered by the existing code in the program. The instrumentation in the clobbering code will cause a segmentation fault because of the protection of the GOT, and hence the attack will be prevented. Note that the GOT is normally used by the PLT (Procedure Linkage Table) code that is in the assembly code automatically added by the compiler, and is not in the C source code, so a normal GOT access will not be instrumented with checks on taint tags, and hence will not lead to a memory fault.

  If the attack corrupted some other function pointer, then the "jmp" policy would detect the use of tainted data in jump target and stop the attack.
- `Pico HTTP Server (pServ)` versions 3.2 and

lower have a directory traversal vulnerability. The web server does include checks for the presence of "`..`" in the file name, but allows them as long as their use does not go outside the `cgi-bin` directory. To determine this, `pServ` scans the file name left-to-right, decrementing the count for each occurrence of "`..`", and incrementing it for each occurrence of "`/`" character. If the counter goes to zero, then access is disallowed. Unfortunately, a file name such as `/cgi-bin////../../bin/sh` satisfies this check, but has the effect of going outside the `/cgi-bin` directory. This attack is stopped by the directory traversal policy shown in Section 4.

Web Applications in PHP.

- `phpBB2` *SQL injection* vulnerability in (version 2.0.5 of) `phpBB`, a popular electronic bulletin board application, allows an attacker to steal the MD5 password hash of another user. The vulnerable code is:

```
$sql="SELECT p.post_id FROM ... WHERE ...
      AND p.topic_id = $topic_id AND ..."
```

  Normally, the user-supplied value for the variable `topic_id` should be a number, and in that case, the above query works as expected. Suppose that the attacker provides the following value:

```
-1 UNION SELECT ord(substring(user_password,
5,1)) FROM phpbb_users WHERE userid=3/*
```

  This converts the SQL query into a union of two `SELECT` statements, and comments out (using "`/*`") the remaining part of the original query. The first `SELECT` returns an empty set since `topic_id` is set to `-1`. As a result, the query result equals the value of the `SELECT` statement injected by the attacker, which returns the 5th byte in the MD5 hash of the bulletin board user with the userid of 3. By repeating this attack with different values for the second parameter of `substring`, the attacker can obtain the entire MD5 password hash of another user. The SQL injection policy described in Section 4 stops this attack.

- `SquirrelMail` *cross-site scripting* is present in version 1.2.10 of `SquirrelMail`, a popular web-based email client, e.g., `read_body.php` directly outputs values of user-controlled variables such as `mailbox` while generating HTML pages. The attack is stopped by the cross-site scripting policy in Section 4.

- `SquirrelMail` *command injection:* `SquirrelMail` (Version 1.4.0) constructs a command for encrypting email using the following statement in the function `gpg_encrypt` in the GPG plugin 1.1.

```
$command .= " -r $send_to_list 2>&1";
```

  The variable `send_to_list` should contain the recipient name in the "`To`" field, which is extracted using the `parseAddress` function of `Rfc822Header` object in `SquirrelMail`. However, due to a bug in this function, some malformed entries in the "`To`" field are returned without checking for proper email format. In particular, by entering "⟨recipient⟩; ⟨cmd⟩;" into this field, the attacker can execute any arbitrary command ⟨cmd⟩ with the privilege of the web server. By applying a policy that prohibits tainted shell meta-characters in the first argument to the `popen` function, this attack is stopped by our technique.

- `phpBB` *directory traversal:* A vulnerability exists in `phpBB`, which, when the gallery avatar feature is enabled, allows remote attackers to delete arbitrary files using directory traversal. This vulnerability can be exploited by a two-step attack. In the first step, the attacker saves the file name, which contains "`..`" characters, into the SQL database. In the second step, the file name is retrieved from the database and used in a command. To detect this attack, it is necessary to record taint information for data stored in the database, which is quite involved. We took a shortcut, and marked all data retrieved from the database as tainted. (Alternatively, we could have marked only those fields updated by the user as tainted.) This enabled the attack to be detected using the directory traversal policy.

- `phpxmlrpc/expat` *command injection:* `phpxmlrpc` is a package written in PHP to support the implementation of PHP clients and servers that communicate using the XML-RPC protocol. It uses the `expat` XML parser for processing XML. `phpxmlrpc` versions 1.0 and earlier have a remote command injection vulnerability. Our command injection policy stops exploitations of this vulnerability.

Bash CGI Application. `nph-test-cgi` is a CGI script that was included by default with Apache web server versions 1.0.5 and earlier. It prints out the values of the environment variables available to a CGI script. It uses the code `echo QUERY_STRING=$QUERY_STRING` to print the value of the query string sent to it. If the query string contains a "`*`" then `bash` will apply file name expansion to it, thus enabling an attacker to list any directory on the web server. This attack was stopped by a policy that restricted the use of tainted meta-characters in the argument to `shell_glob_filename`, which is the function used by `bash` for file name expansion. In terms of marking, the CGI interface defines the exact set of environment variables through which inputs are provided to a CGI application, and all these are marked as tainted.

6.2 False Positives

The policies described so far have been designed with the goal of avoiding false positives. We experimentally verified that false positives did not occur in our experiments involving the `wu-ftpd` server, the Apache

Server Programs	Workload	Orig. Response Time	Overhead
Apache-2.0.40	Webstone 30 clients downloading 5KB pages over 100Mbps network	0.036 sec/page	6%
wu-ftpd-2.6.0	Download a 12MB file 10 times.	11.5 sec	3%
postfix-1.1.12	Send one thousand 3KB emails	0.03 sec/mail	7%

Figure 6: Performance overheads of servers. For Apache server, performance is measured in terms of latency and throughput degradation. For other programs, it is measured in terms of overhead in client response time.

Program	Workload	Overhead(A)	Overhead(B)	Overhead(C)	Overhead(D)
bc-1.06	Find factorial of 600.	212%	68%	61%	**61%**
enscript-1.6.4	Convert a 5.5MB text file into a PS file.	660%	529%	63%	**58%**
bison-1.35	Parse a Bison file for C++ grammar.	134%	92%	79%	**78%**
gzip-1.3.3	Compress a 12 MB file.	228%	161%	110%	**106%**

Figure 7: Performance overheads of CPU-intensive programs. Performance is measured in terms of CPU time. Overheads in different columns correspond to: (A) No optimizations, (B) Use of local tag variable, (C) B + Use of 2-bit taint value, (D) C + Use of dependency analysis.

web server, and the two PHP applications, `phpBB` and `SquirrelMail`. For `wu-ftpd` and Apache, we enabled the control flow hijack policy, format string policy, directory traversal policy, and shell command injection policy. For the PHP applications, we additionally enabled the SQL injection policy and cross-site scripting policy for the PHP interpreter.

To evaluate the false positives for Apache, we used the transformed server as our lab's regular web server that accepted real-world HTTP requests from Internet for several hours. For the `wu-ftpd` server, we ran all the supported commands from a ftp client. To test `phpBB` and `SquirrelMail`, we went through all the menu items of these two Web applications, performed normal operations that a regular user might do, such as registering a user, posting a message, searching a message, managing address book, moving messages between different mail folders, and so on. No false positives were observed in these experiments.

6.3 False Negatives

False negatives can arise due to (a) overly permissive policies, (b) implicit information flows, and (c) use of untransformed libraries without adequate summarization functions.

We will discuss the policy refinement and implicit flows in Section 7. As for external libraries, the best approach is to transform them, so that the need for summarization can be eliminated. If this cannot be done, then our transformation will identify all the external functions that are used by an application, so that errors of omission can be avoided. However, if a summarization function is incorrect, then it can lead to false negatives, false positives, or both.

6.4 Performance

Figure 7 shows the performance overheads, when the original and transformed programs were compiled using `gcc 3.2.2` with `-O2`, and ran on a 1.7GHz/512MB/Red Hat Linux 9.0 PC.

For server programs, the overhead of our approach is low. This is because they are I/O intensive, whereas our transformation adds overheads only to code that performs significant amount of data copying within the program, and/or other CPU-intensive operations. For CPU-intensive C programs, the overhead is between 61% to 106%, with an average of 76%.

6.4.1 Effect of Optimizations. The optimizations discussed in Section 3.3 have been very effective. We comment further in the context of CPU-intensive benchmarks.

- *Use of local taint variables* reduced the overheads by 42% to 144%. This is due to the reasons mentioned earlier: compilers such as `gcc` are very good in optimizing operations on local variables, but do a poor job on global arrays. Thus, by replacing global `tagmap` accesses with local tag variable accesses, significant performance improvement can be obtained.

  Most programs access local variables much more frequently than global variables. For instance, we found out (by instrumenting the code) that 99% of accesses made by `bc` are to local variables. A figure of 90% is not at all uncommon. As a result, the introduction of local tag variables leads to dramatic performance improvement for such programs. For programs that access global variables frequently, such as `gzip` that has 41% of its accesses going to global variables, the performance improvements are less striking.
- $tagmap$ *optimizations* are particularly effective for

programs that operate mainly on integer data. This is because of the use of 2-bit taint tags, which avoids the need for bit-masking and shifts to access taint information. As a result we see significant overhead reduction in the range of 7% to 466%.

- *Intraprocedural analysis* and optimization further reduces the overhead by up to 5%. The gains are modest because `gcc` optimizations have already eliminated most local tag variables after the previous step.

When combined, these optimizations reduce the overhead by a factor of 2 to 5.

7 Discussion

7.1 Support for Implicit Information Flow

Implicit information flow occurs when the values of certain variables are related by virtue of program logic, even though there are no assignments between them. A classic example is given by the code snippet [25]:

```
x=x%2; y=0; if (x==1) y=1;
```

Even though there is no assignments involving `x` and `y`, their values are always the same. The need for tracking such implicit flows has long been recognized. [11] formalized implicit flows using a notion of *noninterference*. Several recent research efforts [18, 30, 20] have developed techniques based on this concept.

Noninterference is a very powerful property, and can capture even the least bit of correlation between sensitive data and other data. For instance, in the code:

```
if (x > 10000) error = true;
if (!error) { y = "/bin/ls"; execve(y); }
```

there is an implicit flow from `x` to `error`, and then to `y`. Hence, a policy that forbids tainted data to be used as an `execve` argument would be violated by this code. This example illustrates why non-interference may be too conservative (and hence lead to false positives) in our application. In the context of the kinds of attacks we are addressing, attackers usually need more control over the value of `y` than the minimal relationship that exists in the code above. Thus, it is more appropriate to track explicit flows. Nevertheless, there can be cases where substantial information flow takes place without assignments, e.g., in the following if-then-else, there is a direct flow of information from x to y on both branches, but our formulation of explicit information flow would only detect the flow in the else statement.

```
if (x == 0) y = 0; else y = x;
```

The goal of our approach is to support those implicit flows where the value of one variable *determines* the value of another variable. By using this criteria, we seek a balance between tracking necessary data value propagation and minimizing false positives. Currently, our implementation supports two forms of implicit flows that appear to be common in C programs.

- *Translation tables.* Decoding is sometimes implemented using a table look up, e.g.,

  ```
  y = translation_tab[x];
  ```

  where `translation_tab` is an array and `x` is a byte of input. In this case, the value of `x` determines the value of `y` although there is no direct assignment from `x` to `y`. To handle this case, we modify the basic transformation so that the result of an array access is marked as tainted whenever the subscript is tainted. This successfully handles the use of translation tables in the PHP interpreter.
- *Decoding using if-then-else/switch.* Sometimes, decoding is implemented using a statement of the form:

  ```
  if (x == '+') y = ' ';
  ```

  (Such code is often used for URL-decoding.) Clearly, the value of `y` can be determined by the value of `x`. More generally, `switch` statements could be used to translate between multiple characters. Our transformation handles them in the same way as a series of if-then-else statements. Specifically, consider an if-then-else statement of the form:

  `if (x ==` E`) {  ...  y =` E'`; ... }`

  If E and E' are constant-valued, then we add a tag update $tag(y) = tag(x)$ immediately before the assignment to y.

While our current technique seems to identify some of the common cases where implicit flows are significant, it is by no means comprehensive. Development of a more systematic approach that can provide some assurances about the kinds of implicit flows captured, while ensuring a low false positive rate, is a topic of future research.

7.2 Policy Refinement

Policy development effort is an important concern with any policy enforcement technique. In particular, there is a trade-off between policy precision and the level of effort required. If one is willing to tolerate false positives, policies that produce very few false negatives can be developed with modest effort. Alternatively, if false negatives can be tolerated, then false positives can be kept to a minimum with little effort. To contain both false positives and false negatives, more effort needs to be spent on policy development, taking application-specific or installation-specific characteristics.

The above remarks about policy-based techniques are generally applicable to our approach as well. For the format string attack, we used a policy that tended to err on the side of producing false positives, by disallowing all use of tainted format directives. However, it is conceivable that some applications may be prepared to receive a subset of format directives in untrusted inputs, and handle them correctly. In such cases, this application knowledge can be used by a system administrator to use a less

restrictive policy, e.g., allowing the use of format directives other than %n. This should be done with care, or else it is possible to write policies that prevent the use of %n, but allow the use of variants such as %5n that have essentially the same effect. Alternatively, the policy may be relaxed to permit specific exceptions to the general rule that there be no format directives, e.g., the rule:

$\text{vfprintf}(fmt)$ |
fmt **matches** $any*\,(\text{Format})^T\,any*$ &&
(!(fmt **matches** "[^%]*%s[^%]*")) $\rightarrow reject()$

allows the use of a single %s format directive from untrusted sources, in addition to permitting format strings that contain untainted format directives.

The directory traversal policy also tends to err on the side of false positives, since it precludes all accesses outside the authorized top level directories (e.g. DocumentRoot and cgi-bin) of a web server if components of the file name being accessed are coming from untrusted sources. In devising this policy, we relied on application-specific knowledge, namely, the fact that web servers do not allow clients to access files outside the top level directories specified in the server configuration file. Another point to be noted about this policy is that variants of directory traversal attack that do not escape these top level directories, but simply attempt to fool per-directory access controls, are not addressed by our policy.

The control-flow hijack policy is already accurate enough to capture all attacks that use corruption of code pointers as the basis to alter the control-flow of programs, so we proceed to discuss the SQL injection policy. The policy shown in Figure 4 does not address attacks that inject only SQL keywords (e.g., the UNION operation) to alter the meaning of a query. This can be addressed by a policy based on tokenization. The idea is to perform a lexical analysis on the SQL query to break it up into tokens. SQL injection attacks are characterized by the fact that multiple tokens appear in the place of one, e.g., multiple keywords and meta-characters were provided by the attacker in the place of a simple string value in the attack examples discussed earlier in the paper. Thus, systematic protection from SQL injections can be obtained using a policy that prevents tainted strings from spanning multiple tokens. A similar approach is suggested in [24], although the conditions are not defined as precisely. Su et al [27] provide a formal characterization of SQL injection using a syntax analysis of SQL queries. The essential idea is to construct a parse tree for the SQL query, and to examine its subtrees. For any subtree whose root is tainted, all the nodes below that subtree should be tainted as well. In other words, tainted input cannot straddle different syntactic constructs. This is a further refinement over the characterization we suggest, where tainted input should not straddle different lexical entities.

Command injection attacks are similar to SQL injection attacks in many ways, and hence a tokenization-based policy may be a good choice for them as well. For this reason, we omit a detailed discussion of command injection policies. Nevertheless, it should be mentioned that there are some differences between SQL and command injection, e.g., shell syntax is much more complex than SQL syntax. Moreover, we may want to restrict the command names so that they are not tainted.

Note that tokenization is a lexical analysis task that is (almost invariably) implemented using regular expression based specifications. Thus, the above tokenization-based policy is amenable to expression using our policy language. One could argue that a regular expression to recognize tokens would be complex, and hence a policy may end up using a simpler approximation to tokenization. This discussion shows that the usual trade-off in policy based attack detection between accuracy and policy complexity continues in the case of taint-enhanced policies as well. Nevertheless, it should be noted that for a given policy development effort, taint-enhanced policies seem to be significantly more accurate than policies that do not incorporate any knowledge about taint.

Finally, we discuss the cross-site scripting attack. The policy discussed earlier does not address variations of the basic attack, e.g., attackers can evade this policy by injecting the malicious script code in "`onmouseover=malicious()`" or "`<img src="javascript:malicious()">`", which is not a block enclosed by the `script` tag. To detect these XSS variations, one has to understand the different HTML tag patterns in which a malicious script can be injected into dynamic HTML pages, and develop policies to prevent the use of such tainted patterns in HTML outputs.

In summary, although the example policies shown in Figure 4 were able to stop the attacks in our experiments, many of them need further improvement before they can stand up to skilled attackers that are knowledgeable about the policies being enforced. We outlined the ways to improve some of these policies, but a comprehensive solution to the policy development problem is not really the focus or contribution of this paper. Instead, our contribution is to show the feasibility and practicality of using fine-grained taint information in developing policy-based attack protection. The availability of fine-grained taint information makes our policies significantly more precise than traditional access-control policies. Moreover, our approach empowers system administrators and security professionals to update and refine these policies to improve protection, *without* having to wait for the patches of a newly discovered attack avenue.

8 Related Work

Memory Error Exploit Detection. Buffer overflows and related memory errors have received a lot of atten-

tion, and several efficient techniques have been developed to address them. Early approaches such as StackGuard [7] and ProPolice [9] focused on just a single class of attacks. Recently, more general techniques based on randomization have been developed, and they promise to defend against most memory error exploits [16, 2], However, due to the nature of the C language, these methods still cannot detect certain types of attacks, e.g., overflows from an array within a structure to an adjacent variable. Fine-grained taint analysis can capture these attacks whenever the corrupted data is used as an argument in a sensitive operation. (This is usually the case, since the goal of an attacker in corrupting that data was to perform a security-sensitive operation.) Although our overheads are generally higher than the techniques mentioned above, we believe that they are more than compensated by the increase in attack coverage.

Fine-Grained Taint Analysis. The key distinctions between our work and previous fine-grained taint analysis techniques of [22, 28, 5] were already discussed in the introduction, so we limit our discussion to the more technical points here. As mentioned earlier, [28, 5] rely on hardware support for taint-tracking. [22] is closer to our technique than these two techniques. It has an advantage over ours in that it can operate on arbitrary COTS binaries, whereas we require access to the C source code. This avoids problems such as hand-written assembly code. Their main drawback is performance: on the application Apache that they provide performance numbers on, their overheads are more than 100 times higher than ours. This is because (a) they rely on Valgrind, which in itself introduces more than 40 times overheads as compared to our technique, and (b) they are constrained by having to work on binary code, and without the benefit of static analyses and optimizations that have gone into our work. (Here, we are not only referring to our own analyses and optimizations, but also many of the optimizations implemented in the GCC compiler that we used to compile the transformed programs.)

There are several other technical differences between our work and that of [22]. For instance, they track 32-bits of taint information for each byte of data, whereas we use 2 bits. Another important difference is our support for implicit flows, which are not handled in [22].

Dynamic Taint Based Techniques for Detecting Attacks on Web Applications. Independently and in parallel to our work, which first appeared in [33], [23] and [24] have proposed the idea of using fine-grained taint analysis to detect injection attacks on web applications. The implementations of [23] and [24] are very similar, using hand-transformation of the PHP interpreter to track taint data. However, [24] provides a more detailed formulation and discussion of the problem, so we focus on this work here. They explain that these injection attacks are the result of *ad hoc* serialization of complex data such as SQL queries or shell commands, and develop a detection technique called *context-sensitive string evaluation (CSSE)*, which involves checking the use of tainted data in strings. Our work improves over theirs in several ways. First, by working at the level of the C language, we are able to handle many more applications: most server programs that are written in C, as well as programs written in interpreted languages such as PHP, bash and so on. Second, our formulation of the problem as taint-enhanced policy enforcement is more general, and can be applied to stealthy attacks such as those discussed in Section 2 that do not involve serialization problems; and to attacks involving arbitrary types of data rather than being limited to strings. Third, our approach relies on a simple transformation that is shown in Section 3, and implemented using 3.6KLOC of code, while their approach relies on manual transformation of a large piece of software that has over 300KLOC. Other technical contributions of our work include (a) the development of a simple policy language for concise specification of taint-enhanced polices, and (b) support for implicit flows that allow us to provide some support for character encodings and translations.

As discussed in Section 7, Su et al [27] describe a technique for detecting SQL injection attacks using syntax analysis. Their main focus is on providing a precise and formal characterization of SQL injection attacks. However, their implementation of taint tracking is not very reliable. In particular, they suggest a technique that avoids runtime operations for taint-tracking by "bracketing" each input string with two special symbols that surround untrusted input strings. Assuming that these brackets would be propagated together with input strings, checking for the presence of taint would reduce to checking for the presence of these special symbols. However, this assumption does not hold for programs that extract parts of their input and use them, e.g., a web application may remove non-alphanumeric characters from an input string and use them, and this process would likely discard the bracketing characters. In other cases, a web application may parse a user input into multiple fields, and use each field independently, once again causing the special symbols to be lost.

Manual Approaches for Correcting Input Validation Errors. Taint analysis targets vulnerabilities that arise due to missing or incorrect input validation code. One can manually review the code, and try to add all the necessary input validation checks. However, the notion of validity is determined by the manner in which the input is used. Thus, one has to trace forward in the program to identify all possible uses of an input in security sensitive operations, which is a very time-consuming and error-

prone task. If we try to perform the validation check at the point of use, we face the problem that the notion of validity depends on the data source. For instance, it is perfectly reasonable for an SQL query to contain semicolons if these originated within the program text, but not so if it came from external input. Thus, we have to trace back from security-sensitive operations to identify how its arguments were constructed, once again having to manually examine large number of program paths. This leads to situations where validation checks are left out on some paths, and possibly duplicated on others. Moreover, the validation checks themselves are notoriously hard for programmers to code correctly, and have frequently been the source of vulnerabilities.

Information Flow. Information flow analysis has been researched for a long time [1, 10, 8, 18, 30, 20, 25]. Early research was focused on multi-level security, where fine-grained analysis was not deemed necessary [1]. More recent work has been focused on tracking information flow at variable level, and many interesting research results have been produced. While these techniques are promising for protecting privacy and integrity of sensitive data, as discussed in Section 2, the variable-level granularity is insufficient for detecting most attacks discussed in this paper.

Static Analysis. Static taint analysis techniques have been proposed by many for finding security vulnerabilities, including input validation errors in web applications [17, 14, 32], user/kernel pointer bugs [15], format string bugs [26], and bugs in placement of authorization hooks [34]. The main advantage of static analysis (as compared to runtime techniques) is that all potential vulnerabilities can be found statically, while its drawback is a relative lack of accuracy. In particular, these techniques typically detect *dependencies* rather than vulnerabilities. For instance, [17] will produce a warning whenever untrusted data is used in any manner in an SQL query. This may not be very useful if such a dependency is an integral part of application logic. To solve this problem, the concept of *endorsement* can be used to indicate "safe" dependencies. Typically, this is done by first performing appropriate validation checks on a piece of untrusted data, and then endorsing it to indicate that it is safe to use (i.e., no longer "tainted"). However, programmers are still responsible for determining what is "safe" — as discussed before, there is no easy way for them to do this.

An important difference between our work and static analysis is one of intended audience. Static analysis based tools are typically intended for use by developers, since they need detailed knowledge about program logic to determine where to introduce endorsements, and what validation checks need to be made before endorsement. In contrast, the audience for our tool is a system administrator or an outside security engineer that lacks detailed knowledge of application code.

Other Techniques. SQLrand [4] defeats SQL injection by randomizing the textual representation of SQL commands. A drawback of this approach, as compared to the technique presented in this paper, is that it requires manual changes to the program so that the program uses the modified representation for SQL commands generated by itself. Our approach was inspired by the effect achieved by SQLrand, namely, that of distinguishing commands generated by the application from those provided by untrusted users.

AMNESIA[12] is another interesting approach for detecting SQL injection attacks. It uses a static analysis of Java programs to compute a finite-state machine model that captures the lexical structure of SQL queries issued by a program. SQL injection attacks cause SQL queries issued by the program to deviate from this model, and hence detected. A key benefit of this approach is that by using static analysis, it can avoid runtime taint-tracking, and is hence much more efficient than our approach. Although this approach has been demonstrated to work well for SQL injections, the conservative nature of its static analysis and its inability to distinguish different sources of inputs can lead to a higher rate of false positives when applied to other types of attacks.

Perl has a taint mode [31] that tracks taint information at a coarse granularity – that of variables. In Perl, one has to explicitly untaint data before using it in a security sensitive context. This is usually done after performing appropriate validations. In our approach, due to the flexibility provided by our policy language, we have not faced a need for such explicit untainting. Nevertheless, if a user explicitly wants to trust some input, a primitive can be easily added to support this.

9 Conclusion

In this paper, we presented a unified approach that addresses a wide range of commonly reported attacks that exploit software implementation errors. Our approach is based on a fully automatic and efficient taint analysis technique that can track the flow of untrusted data through a program at the granularity of bytes. Through experiments, we showed that our technique can be applied to different types of applications written in multiple programming languages, and that it is effective in detecting attacks without producing false positives.

We believe that a number of software vulnerabilities arise due to the fact that security checks are interspersed throughout the program, and it is often difficult to check if the correct set of checks are being performed on every program path, especially in complex programs where the control flows through many, many functions. By decou-

pling policies from application logic, our approach can provide a higher degree of assurance on the correctness of policies. Moreover, the flexibility of our approach allows site administrators and third parties to quickly develop policies to prevent new classes of attacks, without having to wait for patches.

Acknowledgments

This research was supported in part by an ONR grant N000140110967, NSF grants CNS-0208877 and CCR-0205376, and by a NYSTAR grant.

References

[1] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[2] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, August 2003.

[3] T. Bowen, D. Chee, M. Segal, R. Sekar, T. Shanbhag, and P. Uppuluri. Building survivable systems: An integrated approach based on intrusion detection and damage containment. In *DISCEX*, 2000.

[4] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *International Conference on Applied Cryptography and Network Security (ACNS)*, pages 292–302, 2004.

[5] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2005.

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

[7] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Automatic detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[8] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[9] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/main.html, June 2000.

[10] J. S. Fenton. Memoryless subsystems. *Computing Journal*, 17(2):143–147, May 1974.

[11] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, 1982.

[12] W. Halfond and A. Orso. AMNESIA: Analysis and monitoring for neutralizing SQL-injection. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005.

[13] N. Hardy. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, October 1988.

[14] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *International World Wide Web Conference*, 2004.

[15] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, 2004.

[16] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM Conference on Computer and Communication Security (CCS)*, 2003.

[17] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, 2005.

[18] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *IEEE Symposium on Security and Privacy*, pages 79–93, May 1994.

[19] S. McPeak, G. C. Necula, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for C program analysis and transformation. In *Conference on Compiler Construction*, 2002.

[20] A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, Jan. 1999.

[21] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Workshop on Runtime Verification (RV)*, Boulder, Colorado, USA, July 2003.

[22] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[23] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, 2005.

[24] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID)*, 2005.

[25] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1), Jan. 2003.

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

[27] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2006.

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

[29] P. Uppuluri and R. Sekar. Experiences with specification based intrusion detection. In *proceedings of the Recent Advances in Intrusion Detection conference*, October 2001.

[30] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[31] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. O'Reilly, 1996.

[32] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, 2006.

[33] W. Xu, S. Bhatkar, and R. Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. Technical Report SECLAB-05-04, Department of Computer Science, Stony Brook University, May 2005.

[34] X. Zhang, A. Edwards, and T. Jaeger. Using CQual for static analysis of authorization hook placement. In *USENIX Security Symposium*, 2002.