An Efficient Black-box Technique for Defeating Web Application Attacks*

R. Sekar
Stony Brook University, Stony Brook, NY, USA

Abstract

Over the past few years, injection vulnerabilities have become the primary target for remote exploits. SQL injection, command injection, and cross-site scripting are some of the popular attacks that exploit these vulnerabilities. Taint-tracking has emerged as one of the most promising approaches for defending against these exploits, as it supports accurate detection (and prevention) of popular injection attacks. However, practical deployment of taint-tracking defenses has been hampered by a number of factors, including: (a) high performance overheads (often over 100%), (b) the need for deep instrumentation, which has the potential to impact application robustness and stability, and (c) specificity to the language in which an application is written. In order to overcome these limitations, we present a new technique in this paper called taint inference. This technique does not require any source-code or binary instrumentation of the application to be protected; instead, it operates by intercepting requests and responses from this application. For most web applications, this interception may be achieved using network layer interposition or library interposition. We then develop a class of policies called syntax- and taint-aware policies that can accurately detect and/or block most injection attacks. An experimental evaluation shows that our techniques are effective in detecting a broad range of attacks on applications written in multiple languages (including PHP, Java and C), and impose low performance overheads (below 5%).

1 Introduction

The past few years have witnessed a significant shift in terms of software vulnerabilities: while buffer overflows in C/C++ programs were dominant earlier, CVE reports indicate that the vast majority of today's vulnerabilities are in web applications. So-called *injection vulnerabilities* dominate, including SQL injection, command injection, path traversals, cross-site scripting (XSS), and so on. As shown in Figure 1, more than half the CVE reports in 2006-07[1] correspond to these injection vulnerabilities. If we omit the "other" category, which includes very general vulnerability categories such as configuration and design errors, and focus on well-defined vulnerabilities, about 85% of these are injection vulnerabilities[2]. The techniques developed in this paper target this large fraction of current software vulnerabilities.

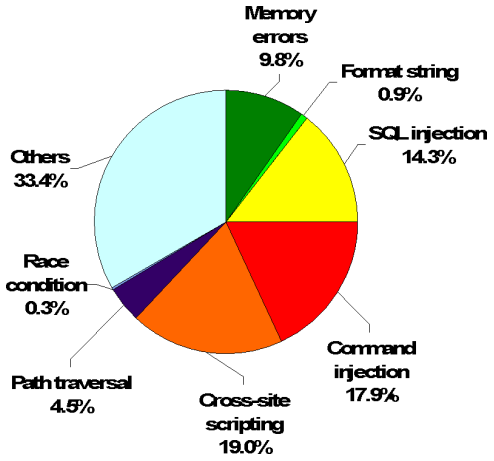

Figure 1. CVE vulnerabilities over 2006-07.

Figure 2 illustrates the general context for injection attacks. The attack target (henceforth called a *target application*, or simply as "application") is a program that accepts requests (inputs) from an untrusted source. It carries out these requests using operations (henceforth called "sensitive operations" or "outgoing requests") on back-end "resources" such as databases, back-end servers, files, and other applications (including command interpreters) running on the target machine. Finally, it outputs a response back to the client. The target application decides which sensitive operations to make, as well as the parameters to these operations.

*This work was partially funded by Defense Advanced Research Project Agency under contract N00178-07-C-2005, by ONR grant N000140710928 and an NSF grant CNS-0627687. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Project Agency, the Naval Surface Weapons Center, ONR, NSF, or the U.S. Government.

[1] Specifically, the chart captures CVE candidates and vulnerabilities from January 2006 to July 2007.

[2] We are including format string vulnerabilities among injection vulnerabilities but not buffer overflows.

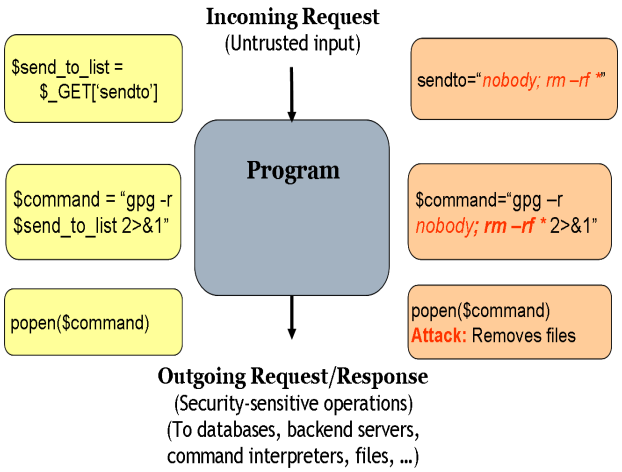

Figure 2. SquirrelMail Command Injection.

To ensure that untrusted users cannot exert undue control over these decisions, the target application should incorporate *input validation* checks on untrusted inputs. However, due to software bugs, these checks may be incomplete, or may be missed on some program paths between the input and output operations. This leads to vulnerabilities that allow an attacker to modify the output operations by "injecting" malicious data at the input. The left-hand side of Figure 2 shows the key statements responsible for a command injection vulnerability in version 1.4.0 of SquirrelMail (a popular web application for email access) with GPG plug-in version 1.1. (These statements are written in the PHP language, and have been abstracted to improve readability.) The right-hand side of the figure shows the input provided by the attacker, and the result of executing the statements on this input. In particular, SquirrelMail uses the program `gpg` to encrypt email body. The `gpg` program needs to access the public key of the recipient, so the recipient name should be provided as a command-line argument. SquirrelMail retrieves the name of the recipient from the "to" field on the email composition form, and constructs a shell command to launch `gpg`. By providing an unexpected value in the "to" field, the attacker is able to execute arbitrary commands on the target machine. The ability to carry out an injection attack hinges on these questions:

1. *Is the attacker able to **exert control** over a sensitive operation performed by an application?*
2. *Is this degree of control **intended** by the programmer (or the administrator) of the target application?*

Fine-grained taint-tracking has been proposed as an effective technique for answering the first question. It involves marking untrusted inputs as "tainted," and as the program uses this data, copying the taint labels together with data values. Taint is tracked at fine granularity, i.e., each byte of data has its own taint bit. As a result, we can determine whether an individual byte within a security-sensitive argument is derived from attacker-furnished data, and hence can be "attacker-controlled." In Figure 2, tainted data bytes are shown using red color and in italics. A number of researchers have developed techniques for fine-grained taint-tracking. Some of these techniques rely on processor-support for taint-tracking [27, 5], while others rely on automated program transformations on source code [28, 13, 10] or binary code [16, 22, 23].

To answer the second question, we need *policies* to capture the degree of control that is intended. For instance, in Figure 2, the SquirrelMail developers intended that the untrusted user be able to control the values of some command-line arguments to `gpg`, but not the introduction of shell metacharacters (such as semicolons) or additional shell commands. Attacks such as command injection, which provide the ability to introduce "commands" rather than "data," can be detected using very general policies that are independent of the web application [4, 21, 26, 28, 10, 25]. For attacks that involve injection of data, (e.g., use of unintended data values in SQL queries, path traversals, etc.) application-specific policies are usually needed.

Drawbacks of Taint-Tracking Techniques. Since taint-based techniques were introduced, they have become the de-facto standard for detecting injection attacks. They have been shown to be very effective and accurate, providing essentially zero false positives and false negatives [10, 28, 21, 17]. However, taint-based techniques suffer from one of more drawbacks that make them difficult to use on production systems:

- *Intrusive instrumentation.* Taint-tracking requires a fine-grained transformation of the target application. Every statement in the application needs to be transformed to introduce additional statements that propagate taint. Such instrumentation can affect the stability and robustness of the target application, making administrators reluctant to use these techniques on production systems.
- *High performance overheads.* Taint-tracking techniques, especially those operating on C [28, 13] or binary code [16, 22, 23], have high overheads, often slowing down programs by a factor of two or more.
- *Language dependence.* Source-code based techniques [21, 17, 10, 28] are language-specific, and need to be redesigned and reimplemented for each language. Even for binary-based techniques, it is not straightforward to apply them across all languages — for instance, applying a machine-code based taint-tracking to Java requires the JVM to be taint-tracked, which can

pose challenges in terms of false positives and false negatives. As a result, previous techniques have either been applicable to Java [10, 25] or to C/C++/compiled binaries, but not both.

Our Approach. In this paper, we overcome the above drawbacks by developing ***a new***, ***low-overhead***, ***non-intrusive*** and ***language-independent approach*** for detecting injection attacks. This approach features:

- an efficient, black-box ***taint-inference*** technique, and
- a flexible and powerful policy framework called ***syntax- and taint-aware policies***.

Our approach *infers taint* by observing data at the input and output interfaces shown in Figure 2. These observations could be made on the network in most cases, but our implementation relies primarily on library interposition. Taint is inferred if the data at the output interface is obtained from the data at the input interface using predefined "transformations." We point out that web applications frequently transform their inputs, e.g., to decode form data and parameters, remove white space, convert upper-case characters to lower-case, etc. Our technique is designed to accommodate such transformations.

Inferring taint from input/output observations may appear to be a hard problem, but we have been able to exploit the characteristics of web applications to make this problem tractable. Incoming requests to web applications use the HTTP protocol, with standardized ways of encoding parameter names and values. Web applications typically retrieve these parameter values, apply simple sanitization or normalization operations on them, and finally use their values within an outgoing request sent to a back-end system. As a result, data flows can be identified by comparing input parameter values against (all possible) substrings of outgoing requests. Our technique relies on approximate (rather than exact) string match so as to be able to identify taint in the presence of simple sanitization or normalization operations used by a web application.

As observed by previous works [17, 21, 9, 26, 28], SQL injection attacks are characterized by the fact that tainted data modifies the lexical and/or syntactic structure of an outgoing SQL query. Other attacks, such as cross-site scripting (XSS), may not change the output structure, but are characterized by certain sensitive components (e.g., a script name or a script body) becoming tainted. Finally, format string and path traversal attacks are characterized by the fact that tainted parameters have impermissible values. In order to detect all these attacks within a uniform framework, we have developed a class of policies that we call as *syntax- and taint-aware policies.* Our policy framework is able to support different languages at the output interface, such as shell, SQL and HTML. Equally important, variations in these languages are handled without having to rewrite code, e.g., our implementation supports variations in SQL across different servers, as well as variations among common shell-scripting languages.

2 Overview of Approach

Figure 3 illustrates the architecture of our system. It "hooks" into existing web server/web application architectures using network-layer or library interposition in order to observe incoming and outgoing requests. These "events" are captured by an event collector that feeds into a syntax analyzer, which in turn feeds into the taint inference and attack detection components.

The event collector is responsible for intercepting incoming as well as outgoing requests. On the input side, our system takes advantage of the plug-in architecture provided by web servers such as Apache and IIS. Specifically, it is possible to register plug-ins that get invoked at key points during the processing of every HTTP request. This plug-in can then examine the data associated with the request, as well as the response. The event collector uses this framework to keep track of "sessions," which are used to limit the scope over which taint inference algorithms are applied. Each session begins when the web server receives a request, and ends when a response is sent back. Whenever the web server invokes the plug-in, it provides information that identifies the session. When intercepting library functions, the event collector uses information such as the thread identifier to keep track of sessions.

The event interceptor incorporates a pluggable architecture for syntax analysis. In particular, each intercepted operation can be associated with a syntax analyzer plug-in. We have currently implemented six plug-ins: two on the input side (for HTTP and XML RPC requests), and four on the output side (HTML, SQL, shell-scripts, and HTTP responses).

The primary goal of syntax analyzers on the input side is to decompose the input into multiple components in such a way as to simplify taint inference. Our syntax analyzer for HTTP performs all the normalization and decoding operations that are necessary on the request, and parses its contents into an URI, form fields, cookies, HTTP header fields, etc. The XML RPC parser carries out a similar function, parsing the request to extract RPC parameters. All this information is captured uniformly as $\langle name, value \rangle$ pairs so as to simplify the design of the rest of the components. Further details about event collection and syntax analysis are provided in Section 3.

Our taint inference algorithm is best understood as operating on a single pair of data items at a time, and identifying whether there is a flow of information from the first

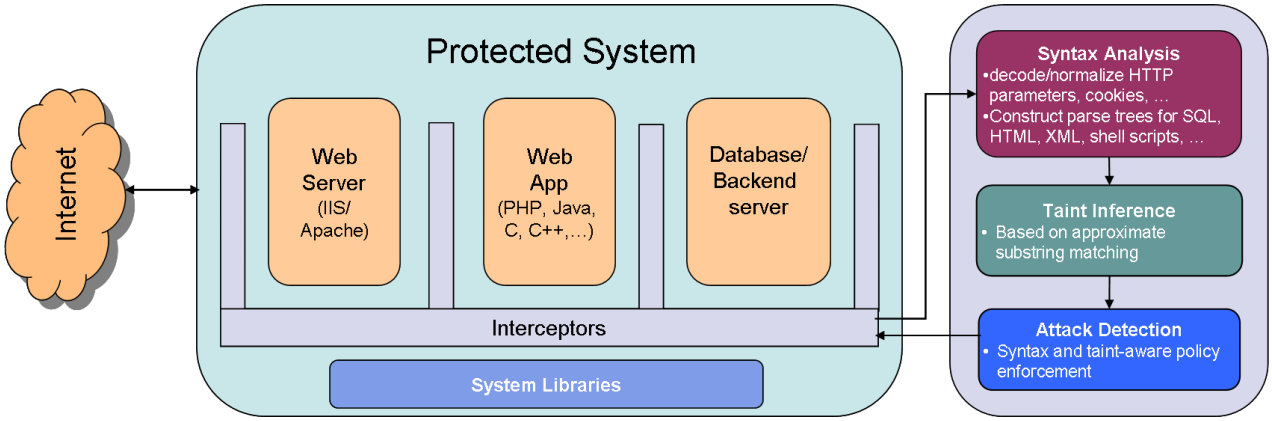

Figure 3. System Architecture.

item to the second. Specifically, given a name-value pair (N, I) from the input syntax analyzer and an output O, the goal of taint inference is to determine if any *substring* of O contains data from I. In principle, different taint inference algorithms could be applied for different types of data, but in practice, we rely on a single inference algorithm that is motivated by the way most web applications operate: they retrieve cookies and form fields from an input request, and use them (after possibly some simple sanitizations and/or edits) to construct an outgoing request. Thus, the value I would likely appear within the outgoing request O, possibly after some slight modifications. Hence our technique infers a taint on a substring o of O if there is an *approximate string match* between I and o, or equivalently, if the edit-distance between I and o is less than a given threshold. The taint-inference algorithm is further described in Section 4.

The goal of output syntax analyzers is to provide support for syntax-aware policies. Our syntax analyzers implement "rough parsers[3]" that recognize key elements of the syntax of the language in question, but are not meant to be full parsers. We rely on rough parsing for several reasons. First, implementing full parsers can be a significant amount of effort for most languages. Second, a rough parser can accommodate variations in the language across different vendor implementations of the same language, for example, variations in SQL syntax across MySQL, Postgres, etc. Similarly, a rough parser can be written that recognizes syntactic components that remain the same across various flavors of shells[4]. The syntax analyzers must also be robust in the face of syntax errors, as they are relatively common in some languages, e.g., HTML. We have designed our parsers with an eye toward error-recovery so as to make them more robust in the face of syntax errors. (This task is also simplified by our decision to rely on rough rather than full parsing.)

An output syntax analyzer constructs an abstract syntax tree (AST), which is a data-structure that is common to all output languages. As described in Section 5, syntax and taint-aware policies are applied against this AST. If there is a policy violation, the output request is blocked by the event interceptor and an error code returned to the caller.

Implementation, optimization and evaluation of our techniques are described in Sections 6 and 7, followed by related work in Section 8 and concluding remarks in Section 9.

3 Event Collection & Syntax Analysis

Our event interceptor for HTTP requests and responses is based on ModSecurity [3], which is a open-source web application firewall. ModSecurity operates as an Apache module, and uses the Apache plug-in infrastructure to intercept every request received by the web server and the response sent back. It can then apply user-specified regular-expression based filters against HTTP requests, cookies, parameters, responses, etc. In order to do this, ModSecurity incorporates code for handling various HTTP requests (GET, PUT, POST, etc.), handling multi-part requests, extracting form parameters and cookies, and so on. In addition, in order to ensure accurate matching against filtering rules, ModSecurity handles various encodings that are commonly used with HTTP (e.g., base64 encoding), and ensures that all elements of HTTP request and response are appropriately normalized. By relying on ModSecurity to handle these tasks, which can be complex and error-prune, we have been able to build a robust implementation of event interception for HTTP requests and responses.[5]

[3] An exception is the HTTP parser, where we have relied on readily available parsers to accurately parse HTTP requests.

[4] Actually, in the case of shell languages, variable substitution, evaluation and parsing are intertwined, so it is not possible to construct a full parser without also implementing a shell interpreter.

[5] Note that weaknesses in parsing, decoding and normalization HTTP requests and responses can be exploited by an attacker to mount evasion attacks to get past our defenses. Such evasion attacks are a significant threat in the context of HTTP requests, which are entirely under the control of an attacker. By relying on ModSecurity, which has been developed (and tested over many years) with an eye towards evasive techniques, our

The input event interceptor extracts various HTTP header fields, URL name, parameters, cookies, etc. It then makes them available to the rest of our system in a standardized format, namely, a pair of strings $\langle name, value \rangle$. In addition, for complex data that need additional parsing, it can invoke additional parser plug-ins. Currently, we do this primarily for extracting parameters from XML RPC requests.

On the output side, we have currently implemented parsers for SQL, shell-scripts and HTML. Both SQL and shell are "command languages" and hence have some commonality, thus leading us to employ similar approaches for implementing parsers for the two languages. Our implementation uses standard parser generator tools Flex and Bison, and required about one week for initial implementation and testing. Our SQL and shell-scripts parsers are quite accurate in terms of recognizing the lexical structure of these languages — we chose to do this because (a) the lexical structure of these languages is significantly simpler (and much less variable across language variants) than the syntactic structure, so accurate lexical analysis is feasible with modest efforts, and (b) injection attacks typically involve changes to the lexical structure (due to the introduction of characters such as quotes, spaces or comment characters), and hence inaccuracies in lexical analysis can lead to false positives and false negatives.

As mentioned before, there is a trade-off between completeness of the parser for a language and other factors such as implementation effort, generality, robustness and performance of the parser. There may be several ways to balance these conflicting concerns. Here, we describe some of the specific choices made in our implementation. Our shell-script parser is capable of parsing nested structures such as if-then-else, loops, parenthesized expressions, and so on, but does not know the interpretation of most operators, or information such as their precedence. Instead, it treats a nested structure as a sequence of elements. At the highest level, the parser assumes that shell commands will consist of a sequence of statements that are separated by characters such as semicolons, newlines, ampersands, pipe characters, etc. The first word in a statement is recognized as a command name, and the others as parameters. The resulting parser is capable of parsing multiple shell languages, and has been tested on several bash and C-shell scripts that contain tens of thousands of lines of code.

A similar approach is used in our SQL parser. It recognizes keywords such as *select* and *union*, as well as operators, comments, literals, numbers, etc. It can also recognize nested structures such as expressions within nested parentheses. It does not currently incorporate knowledge about specifics of each operator or keyword, such as the number of required operands or operator precedence. (However, unlike shell languages, this knowledge could be incorporated into our SQL parser with modest additional efforts.) The resulting parser has been tested against a large number (many thousands) of SQL queries generated by several web applications.

HTML parsing is simplified by the fact that it does not use any infix operators. The goal of the parser is to identify tag names and parameters so that they may be matched against policies for attack detection; the parser is otherwise unconcerned with the significance of tags. The main complexity in the HTML parser arises from the fact that HTML frequently contains syntactic errors, e.g., missing closing tags[6]. A related complication is that closing tags are optional in some cases. To cope with both complications, our parser uses a stack to keep track of currently open tags. When a closing tag does not match the open tag at the stack top, it can skip the top few tags to find a matching tag further down. The parser pretends as if these missing close tags were present, so as to produce a valid parse. But if the matching open tag is not found among the top few, then the closing tag is considered an error, and is discarded. When the end-of-file is encountered, the parser pretends that closing tags corresponding to all open tags on the stack have been seen, thus producing a valid parse tree. The resulting parser has been tested on thousands of web pages. Essentially the same parser was used for parsing XML in the context of XML/RPC.

The output of the parser is an abstract syntax tree (AST). The AST consists of nodes that are linked to substrings of the parsed text. Each node is annotated with a tag that indicates the type of the node, such as a command name, parameter name, parameter value, command separator, etc. Additional details about the AST structure are provided in Section 5.

Our design choice to implement rough parsers led to significant reductions in implementation efforts: the actual implementation of all of the above-mentioned parsers took just over two weeks of a single developer's time.

4 Taint Inference

Our taint inference algorithm is based on the following observations about many web applications:

- Web applications perform sanitization operations on their input parameters. These operations involve deletion (e.g., white-space removal), insertion (e.g., es-

implementation reduces the likelihood of such attacks.

[6]While errors are possible in SQL and shell-commands, they usually occur in the context of attacks. This is because syntax errors in web-application generated SQL and shell commands lead to application failures, and hence are promptly corrected. In contrast, many HTML errors go unnoticed because browsers typically employ robust error-handling and recovery, and can often cope with many syntax errors without visibly altering outputs.

caping special characters or introducing quotes), or substitution (e.g., replacing lower-case characters with upper-case and replacing spaces with underscore). However, a majority of characters are usually left untouched by sanitization.

- A web application assembles an outgoing request from input parameters. Some parts of these requests are statically specified in the web application code, while other parts are derived from inputs.

The first observation suggests that taint inference should be based on *approximate* string-matching rather than exact string matching. The second observation suggests that it should be based on finding *substrings* rather than complete matches. To illustrate these choices, consider the SquirrelMail command injection example again. The vulnerable URL is `/squirrelmail-1.4.0/plugins/gpg/gpg_encrypt.php`. In the exploit we studied, there were about dozen parameters, all of which were parsed and extracted. The one that is of interest is the "send_to" parameter, which had the value

```
alice, bob; touch /tmp/GotYou
```

Based on these input values, SquirrelMail generated the following shell-command:

```
echo '···' | /usr/bin/gpg ··· -r
   alice@ -r bob;touch /tmp/GotYou@ 2>&1
```

Some parts of the command that are irrelevant for the attack have been replaced with "···". In addition, characters copied from the send_to parameter have been highlighted in italics. Note that the input text has gone through a few changes before use: in particular, the recipient list has been separated into its component names, and each of these names prefixed by a "`-r`" and postfixed with an "`@`" symbol. As a result, an exact substring matching algorithm will not identify that the send_to parameter appears in the shell-command, but an approximate substring matching algorithm can detect this with a high degree of confidence.

4.1 Taint Inference and Approximate Substring Matching

We start by recalling the notion of edit distance on strings. Edit distance is given by three cost functions D, I, and S. $D(c)$ denotes the cost of deleting the character c, $I(c)$ denotes the cost of inserting c, and $S(c,d)$ denotes the cost of substituting a character c by another character d. The following conditions apply for all characters c and d:

- $I(c) \geq 0$ and $D(c) \geq 0$
- $S(c,d) \leq D(c) + I(d)$

Let $\{e_1, ..., e_k\}$ be a sequence of edit operations (insertion, deletion or substitution) that yield a string t from another string s. The cost of this sequence is defined to be the sum of costs of the individual edit operations. The *weighted edit distance* $ED(s,t)$ is defined to be the lowest cost among all edit sequences that transform s into t. Next, we define *normalized weighted substring edit distance* between s and t, which, for simplicity, we refer to as *taint distance (TD)*. Let $D(s)$ denote the cost of deleting all characters in s:

$$TD(s,t) = \min_{u \text{ a substring of } t} ED(s,u)/D(s)$$

Note that $TD(s,\epsilon) = 1.0$, where ϵ denotes the empty string.

Definition 1 (Taint Inference Criteria) *Given an input s and output t and a threshold $0 \leq d \leq 1$, taint is inferred if $TD(s,t) \leq d$.*

The problem of selecting thresholds is discussed in Section 7.3.

Edit distance can be computed using a dynamic programming technique in $O(nm)$ time [14, 7] and using $O(nm)$ storage, where $n = |s|$ and $m = |t|$ represent the lengths of the strings involved in the computation. Below, we review this dynamic programming algorithm. Let $s = s_1 s_2 \cdots s_n$ and $t = t_1 t_2 \cdots t_m$, where s_i and t_j denote the i^{th} and j^{th} characters in s and t respectively. Let $M_{k,l}$ denote the weighted edit distance between $s_1 s_2 \cdots s_k$ and $t_1 t_2 \cdots t_l$.

$$\begin{aligned} M_{i,j} \quad = \quad & min(M_{i-1,j-1} + S(s_i, t_j), \\ & M_{i-1,j} + D(s_i), \\ & M_{i,j-1} + I(t_j)) \end{aligned}$$

This same recurrence can be used to handle approximate matching of whole strings as well as substrings — the difference arises only in the base case for M. To support approximate substring matching, $M_{0,j}$ is set to zero, which captures the fact that a match for s may be tried within t starting from any position $j+1$, i.e., there is no "cost" for skipping the first j characters of t. However, there is a penalty for skipping characters in s: it corresponds to the cost of deleting these characters, so $M_{i,0}$ is set to $D(s_1 s_2 \cdots s_i)$. (For matching whole strings (rather than substrings), $M_{0,j}$ will be set to $I(t_1 t_2 \cdots t_j)$)

Based on the definition of M above, we can define TD as follows:

$$TD(s,t) = \min_{0<l\leq m} M_{n,l}/D(s)$$

This equation captures the intuition that we are willing to skip a prefix of t (since $M_{0,j} = 0$) as well as a suffix $t_{l+1} t_{l+2} \cdots t_m$ without incurring any penalty. (However,

skipping a prefix or suffix of s will incur a penalty). Thus, this equation computes the lowest weighted edit distance between s and a substring of t.

Using standard dynamic programming techniques, the above recurrence can be computed in $O(mn)$ time using $O(mn)$ storage — the details can be found in standard algorithm textbooks. (The algorithm remains essentially the same for computing edit-distance, longest common subsequence and global alignment problems.) However, $O(mn)$ storage can be too large, as some of the outputs (e.g., HTML) can be as large as 10^3 to 10^5, while inputs are frequently larger than 10^2. This also means that $O(mn)$ time complexity will be unacceptable in practice. We therefore develop optimization techniques below to speed up taint inference.

4.2 Speeding Up Taint Inference

We present a "coarse filtering" algorithm that, in $O(m)$ time (assuming $n < m$), identifies all possible positions where the slower (edit-distance) algorithm can succeed. The slower algorithm is then invoked only at these positions. Our coarse filtering algorithm is selective enough that in practice, more time is spent in the coarse-filtering algorithm than the slower edit-distance algorithm. In effect, this means that the combined algorithm operates in $O(n + m)$ in practice.

Coarse-filtering algorithms have been proposed in the context of biological computing, and implemented in tools such as BLAST [6] and FASTA [18]. They work well in the context of genomic data, where they tend to identify the best matches. However, their heuristic techniques *cannot provide any guarantee* that the best matches will always be found; or more generally, that all matches within a certain distance threshold will be found. In our application, the inability to provide such guarantees means that some attacks may be missed. Moreover, given the adversarial context of our problem, there is a danger than an attacker may be able to craft inputs that can reliably evade such heuristics. These factors motivated us to develop new techniques for coarse-filtering that can indeed provide such guarantees.

The basic idea behind our technique is to map a string s over an alphabet Σ into a multiset[7] of characters occurring in it. For instance, if $\Sigma = \{0, 1, 2\}$, then a string 021121 will map to the set $\{0, 1, 1, 1, 2, 2\}$.[8] The multiset representation ignores the relative ordering of characters in s, while preserving the number of occurrences of each character.

For the rest of this section we assume $n \leq m$. (If this is not the case, the "sliding window" aspect of the algorithm becomes unnecessary.) We also make the simplifying assumption that $S(c, d) = D(c) + I(d)$ if $c \neq d$.[9] For any string u, let $\mathcal{U}$ denote the multiset of characters occurring in it. A natural way to define distances on multisets is as follows:

$$ED'(u, v) = D(\mathcal{U} - \mathcal{V}) + I(\mathcal{V} - \mathcal{U})$$

where "$-$" stands for the (multi)set difference operator. We have extended D and I in the natural way to multisets, i.e., $D(\{x_1, \ldots, x_k\}) = D(x_1) + \cdots + D(x_k)$. Note that $ED(u, v) \geq ED'(u, v)$.

Based on the distance metric ED', our coarse-filtering technique is as follows. It uses a sliding window of length n to mark off a substring u of t, and computes $ED'(s, u)$. Let $\overline{t_i}$ denote the substring of t that begins at position i and has length n, i.e., $\overline{t_i} = t_i t_{i+1} \cdots t_{i+n-1}$. From the above definition of ED', $ED'(s, \overline{t_1})$ can be computed in $O(n)$ time. As the window is slid to the right by one position, note that $\overline{t_2}$ differs from $\overline{t_1}$ in that t_1 is being removed and t_{n+1} is being added. This change can be factored into ED' computation easily, and so we can compute $ED'(s, \overline{t_2})$ from $ED'(s, \overline{t_1})$ in $O(1)$ time. Thus, ED' for all n-length substrings of t can be computed in $O(m)$ time.

Based on ED' values, we invoke the more expensive algorithm for computing ED under the following condition. Let $[i, j]$ define a maximal range such that $ED'(s, \overline{t_k}) \leq d$ for $i \leq k \leq j$, where d is the distance threshold (not normalized). Then we invoke $ED(s, t_i t_{i+1} \cdots t_{j+n-1})$. We would like to show that this is sufficient to obtain all approximate substring matches between s and t that are within a distance of d. However, with ED' defined as above, this is not true. Consider the example shown in Figure 4, where $D(c) = I(c) = 1$ for all characters, $q > r$, and c^q denotes that a character c is repeated q times (consecutively) in a string.

Note that t can be obtained from s by inserting r-occurrences of b into it. Thus, $ED(s, t) = I(b^r) = r$. However, we show that $ED'(s, \overline{t_i})$ is twice that of $ED(s, t)$. To understand why, note that when $q > r$, every $\overline{t_i}$'s will contain r occurrences of b. Moreover, it will contain fewer (or the same) occurrences of a's and c's as compared to s. Thus, to obtain $\overline{\mathcal{T}_i}$ from $\mathcal{S}$, we need to insert r-occurrences of b into s, and then delete a total of r occurrences of a's and c's. Thus, $ED'(\overline{t_i}) = 2r$.

The example illustrates that we need an alternative metric in place of ED' in order to ensure that all potential $\overline{t_i}$ are identified, and the full dynamic programing algorithm

[7] A multiset differs from a set in that elements could occur multiple times, and the number of occurrences is significant.

[8] Note that this multiset may also be viewed as a $|\Sigma|$-dimensional vector whose magnitude in the i^{th} dimension is given by the number of times the character i occurs in s.

[9] Often, there is no better way to define $S(c, d)$ than that of $D(c) + I(d)$. Exceptions to this occur primarily in the context of upper-case to lower-case (or one white-space to another white-space) character. These can be handled by first mapping such "equivalent" characters into the same character, and then applying the coarse-filtering algorithm — this is what we do in our implementation.

s	t	$ED(s,t)$	$\overline{t_i}$	$D(\mathcal{S}-\overline{\mathcal{T}_i})$	$I(\overline{\mathcal{T}_i}-\mathcal{S})$	$ED'(s,\overline{t_i})$
$a^q c^q$	$a^q b^r c^q$	r	$a^{q-(i-1)} b^r c^{q-r+(i-1)}$	r	r	$2r$

Figure 4. Example to illustrate ED and ED'.

applied on it. We therefore define an alternative distance measure $ED^{\#}$:

$$ED^{\#}(u,v) = min(D(\mathcal{U}-\mathcal{V}), I(\mathcal{V}-\mathcal{U}))$$

The intuitive reason for having to choose min is as follows. In order to reduce the number of substrings considered by the filtering technique, we constrain ourselves to n-length substrings of t. Due to this constraint on length, it may seem that a large number of insertion or deletion operations are needed to obtain u from s. However, the closest match v found by the edit-distance algorithm may be a substring of u or, u may be a substring of v. In the former case, the insertion operations may no longer be needed, while in the latter case, none of the deletion operations may be needed. Since we cannot predict which of these two case will be applicable, we need to take the minimum of deletion and insertion costs as an upper bound for edit-distance.

By replacing ED' with $ED^{\#}$ we obtain a correct and efficient coarse-filtering algorithm. In particular, the following theorem guarantees that this coarse-filtering technique will identify every approximate substring match between s and t that are within the distance threshold d. In addition, the example in Figure 4 shows that setting a threshold lower than $ED^{\#}$ will miss matches in some cases. Together, these two results indicate that $ED^{\#}$ provides a tight characterization of all the instances where the dynamic programming algorithm needs to be tried.

Theorem 2 (Soundness of Filtering Technique) *If there is a substring $u = t_i t_{i+1} \cdots t_j$ of t such that $ED(s,u) \leq d$ then $ED^{\#}(s,\overline{t_i}) \leq d$, and $ED^{\#}(s,\overline{t_k}) \leq d$ for all $i < k \leq j-n+1$.*

Proof: There are two cases to consider: one in which $|u| \geq n$ and another in which $|u| < n$. (Recall that n denotes the length of s.)

Case 1 ($|u| < n$)**:** Let u begin at the i^{th} position in t. Let $\mathcal{D}_i$ denote $\mathcal{S} - \overline{\mathcal{T}_i}$. Note that $\mathcal{D}_i$ represents an "excess" of characters in s that are not present in $\overline{t_i}$. Clearly, all these characters would need to be deleted from s in any edit sequence that yields $\overline{t_i}$ from s. Since u is a substring of $\overline{t_i}$, all these deletions (and possibly more) must be included in any edit sequence that obtains u from s. Thus, the edit distance between s and u must be greater than the deletion cost of $\mathcal{D}_i$, thus yielding the inequality:

$$d \geq ED(s,u) \geq D(\mathcal{D}_i) = D(\mathcal{S} - \overline{\mathcal{T}}_i)$$

Clearly, $D(\mathcal{S} - \overline{\mathcal{T}}_i)$ is greater than or equal to $ED^{\#} = min(I(\overline{\mathcal{T}}_i - \mathcal{S}), D(\mathcal{S} - \overline{\mathcal{T}}_i))$. This completes the proof for Case 1. (Note that the second part of the theorem $ED^{\#}(s,\overline{t_k}) \leq d$ for all $i < k \leq j-n+1$ holds vacuously in this case, since $j-n+1 < i$.)

Case 2 ($|u| \geq n$)**:** For $i \leq k \leq j-n+1$, let $\mathcal{I}_k$ denote $\overline{\mathcal{T}}_k - \mathcal{S}$. Note that $\mathcal{I}_k$ represents an "excess" of characters in $\overline{t_k}$ that are not present in s. These would have to be inserted into s in order to obtain $\overline{t_k}$. Moreover, since $\overline{t_k}$ is a substring of u, obtaining u from s would require the addition of all these characters, and possibly more. Thus, the edit distance between s and u must be greater than the insertion cost of $\mathcal{I}_k$, thus yielding the inequality:

$$d \geq ED(s,u) \geq I(\mathcal{I}_k) = I(\overline{\mathcal{T}}_k - \mathcal{S})$$

Clearly, $I(\overline{\mathcal{T}}_k - \mathcal{S})$ is greater than or equal to $ED^{\#}(s,\overline{t_k}) = min(I(\overline{\mathcal{T}}_k - \mathcal{S}), D(\mathcal{S} - \overline{\mathcal{T}}_k))$. Thus, we have established the theorem in both cases.

5 Syntax and Taint Aware Policies for Attack Detection

As described earlier, attacks are detected using policies that govern the use of tainted data at an output point. It is well-established now [21, 28, 26, 10] that the most effective policy frameworks are those that leverage knowledge about the output language syntax, e.g., SQL syntax in the case of an output parameter that specifies a query to be sent to a database. This is because many types of injection attacks (e.g., command injection and SQL injection) manifest structural changes to outgoing requests as a result of tainted data, as illustrated by an example in Figure 5. The left-hand side of this figure shows the AST constructed by our parser for a shell command executed by SquirrelMail in the absence of attacks, while the right-hand side shows the AST for the shell command executed when SquirrelMail is subjected to the gpg-based command injection attack. While the specifics regarding the AST structure will be described later, it is clear that the structure of the AST is significantly altered by the attack. (In this figure, tainted nodes are in darker (red) color, while untainted nodes are shown in lighter (green) color.)

Some of the previous works [28] have suggested a *lexical confinement* policy to detect SQL and command injection attacks. This policy requires that tainted data should

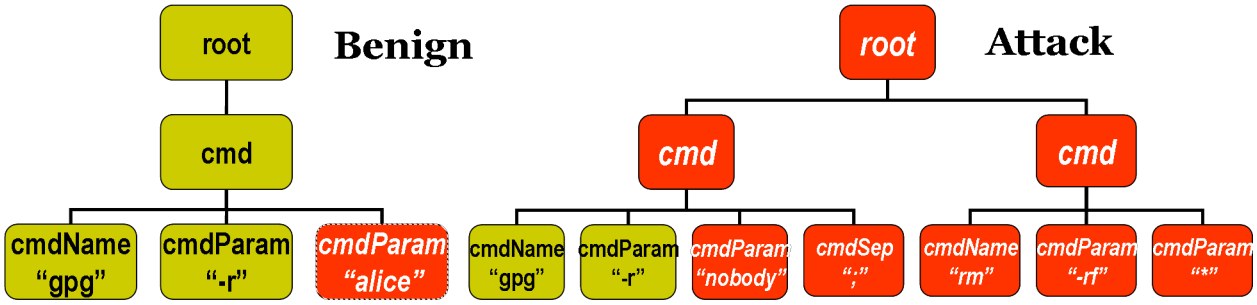

Figure 5. Examples of a benign shell-command (left) and a command involved in an injection attack.

not span multiple tokens. In Figure 5, note that in the benign case, tainted data is confined to a single token that corresponds to a shell parameter. However, in the attack case, tainted data overflows beyond this token — in fact, it creates four additional tokens. A more general criteria was proposed by Su et al [26], which was called *syntactic confinement*. They use an augmented context-free grammar to specify such policies. In particular, their policy identifies grammar symbols (in the grammar of the output language) that can correspond to tainted data. Attacks are detected whenever tainted data is found under parse tree nodes that don't correspond to these grammar symbols. Syntactic confinement is more general than lexical confinement: it can express lexical confinement, but not vice-versa. For instance, a syntactic confinement policy can allow multiple consecutive shell arguments to be tainted, while still preventing command names or command separators from becoming tainted. For most applications, such a policy would block command injection attacks while providing more flexibility in terms of how external input is used in crafting shell command parameters. But this policy cannot be expressed using lexical confinement.

Benefits of Our Policy Framework. The focus of Su and Wasserman's work [26] was to provide a precise characterization of SQL injection and related attacks, so their work does not address many of the practical difficulties in applying their policy framework on a large scale:

- First, their policies are expressed as a modification to a context-free grammar for a particular output language. This means that *the policy writer needs to understand the details of the grammar used in the parser for each of these output languages.* Not only does this make it difficult to specify policies for a single language, but it also means that there is *no uniform way to specify policies across different languages* such as SQL, various flavors of shells, HTML, etc.

- Second, their approach assumes the availability of a full parser for these languages. (Indeed, the selectivity of their policies, in specifying which components of an output can be tainted, relies on being able to syntactically identify those components using a unique grammar symbol.) As mentioned earlier, development of complete parsers for each of these languages requires a lot of effort, and moreover, such parsers will likely need to be modified in order to support variations in the language across different platforms and products.

To overcome the above drawbacks, we have developed a new policy framework that decouples parsers from policy specifications, while still supporting syntax- and taint-aware policies. This is accomplished by defining a simple and generic AST structure as the intermediate representation. In particular, all parsers map their inputs into this AST structure, and policies are specified in terms of ASTs. A second benefit of our policy language is that it can cope with rough parsers. For instance, our HTML parser does not syntactically distinguish between many HTML constructs, yet the policies can apply selectively to some of these constructs (e.g., scripts) by using regular expressions to match against the tag names and values.

The practical benefits of our policy framework are illustrated by Figure 6, which shows the complete set of policies used to detect a wide range of attacks across multiple languages.

Abstract Syntax Tree (AST). The AST is designed to capture the structure of typical languages used at the output, including SQL, shell languages, and HTML. It has a tree structure, where the nodes are tagged with a node type that (roughly) identifies the corresponding language construct, and a value that captures the string value associated with this node. The list of possible node types is meant to be extensible, and it now includes:

root: Corresponds to the root of the parse tree; represents the entire output that was parsed.

cmd: Represents a simple command, such as a shell command, SQL query, or a HTML tag.

Name	Policy	Comments
TaintedCmd	Deny {cmdName|cmdSep}^{T}	Command/SQL injection
TaintedScriptName	Deny {cmd}[{cmdName="script"}^{T}, . . . , {cmdParam}[{paramName="src"}^{T}, {paramVal}^{T}], . . .]	XSS attack
TaintedScriptBody	Deny {cmd}[{cmdName="script"}^{T}, . . . , {cmd|grp}^{T}]	XSS attack
SpanNodeSQL	AllowSpan NONE	SQL injection
SpanNodeShell	AllowSpan {cmd}[. . .,{operator|cmdParam}, {operator|cmdParam}, . . .]	Shell command injection
StraddleTreesSQLShell	AllowStraddle NONE	Most injection attacks

Figure 6. List of Policies Enforced for Attack Detection

cmdName: Corresponds to the name of a command, typically the first word within a command.

cmdParam: Corresponds to a command parameter, usually the words following a command name.

cmdSep: Correspond to constructs that separate commands within a command sequence.

paramName, paramVal: Used to represent parameter names and values in languages that have named parameters, e.g., to represent arguments within a HTML tag.

operator: Represents a unary or binary operator other than any of the above.

grp: Represents a nesting construct such as open and close braces or a begin/end pair.

The parser for a language is responsible for mapping the language constructs into appropriate node types in the AST. This enables a policy writer to formulate policies in terms of these node types, instead of having to understand the details of the underlying grammar which can be much larger. After taint inference, the AST nodes are marked to indicate if they are tainted or not.

Figure 5 shows two examples of ASTs constructed by our shell-language parser. The nodes of the AST in this figure are annotated with the node type. Values, if present, are enclosed within double quotes. Finally, tainted nodes are identified in red color (and italic font).

Policy Language. The policy language is designed to specify constraints on taintedness of AST components and associated values. Our policies have the form:

$$\begin{aligned} Policy &::= Directive \quad Pattern \\ Pattern &::= \text{“\{”} NodeConstraint \text{“\}”}^{[TaintConstraint]} \\ &\quad \text{“[”} ChildConstraints \text{“]”} \end{aligned}$$

The $Directive$ component indicates what is to be done when the $Pattern$ component of the policy matches a given AST or one of its subtrees. Based on observations made earlier about injection attacks, we designed three directives in our policy language. The simplest one is “Deny” which states that any AST that matches the policy specification should be rejected, i.e., the output operation should be blocked. The other two directives relate to our observation about lexical and syntactic confinement policies. By default, our framework enforces both confinement policies, but this can be disabled for specific languages (e.g., HTML) if desired. Exceptions to this default behavior can be specified using the “AllowSpan” and “AllowStraddle” directives. In particular, if tainted data spans two leaves in the AST (thus violating lexical confinement), or two subtrees (thus violating syntactic confinement), that would be considered an attack unless the structure of the AST at that point matches one of the “AllowSpan”-policies.

The “AllowStraddle” directive is used to permit certain kinds of overflows that are almost always indicative of attacks. These involve situations where a subtree of the AST is partially tainted, with tainted data overflowing beyond the right end of this subtree into the subtree that is to the immediate right. This indicates that tainted data completed some syntactic construct, and was responsible for the creation of the syntactic construct that is to the immediate right. In the context of command languages such as SQL and shell, we cannot think of any situation where this should be permitted. Thus, although the directive is supported, we did not specify any exceptions to this policy for SQL and shell languages.

Within a $Pattern$, $NodeConstraint$ is a “|”-separated list of constraints of the form $nodetype$ or $nodetype = RegExpr$, where $nodetype$ is one of the node types in the AST, and $RegExpr$ specifies a regular expression. In order for $NodeConstraint$ to match an AST node, the AST node’s type must match one of those specified in the $NodeConstraint$, and the associated regular expression must match the value associated with the AST node.

$TaintConstraint$, if present, it is simply the letter T, which indicates that a matching AST node must be tainted. Finally, $ChildConstraints$ is a comma-separated list of constraints on children, each of which has the same structure as $Pattern$. While specifying constraints on the children, "..." may be used to skip zero or more of the children of the AST node during the match phase.

A pattern may match at multiple nodes within an AST. In addition, due to the presence of "...", there may be multiple ways of matching at the same AST node. The semantics of our policy is that the directive will be applied for each one of the matches.

Figure 6 shows the actual policies enforced in our system. We have not omitted any policies, nor have we abstracted them in any way. These policies were able to detect all of the attacks described in Section 7. The compactness and simplicity of the policies illustrates the benefits of our syntax and taint-aware policy framework. Note that the attack example in Figure 5 violates the TaintedCmd policy, as well as the SpanNodesSQL and StraddleTreesShellSQL policies. The benign example does not violate any of the policies.

6 Pruning Policies: Omitting Taint Inference and Policy Enforcement for Benign Inputs

This optimization is based on the observation that for a vast majority of inputs, we can determine, based on their structure, that they are "benign" and cannot contribute to attacks. We have developed policies on inputs for this purpose. These policies are called *pruning policies* because they have the effect of pruning away the more expensive taint-inference and output policy enforcement operations. These pruning policies are designed to be *conservative,* i.e., when they deem an input to be benign, it must never lead to the violation of one of the policies shown in Figure 6. When we cannot rule out the possibility that an input could lead to the violation of these policies, then the taint inference and syntax and taint-aware policy enforcement will be performed for such inputs.

Specifically, note that an attack is detected when one of the policies shown in Figure 6 is matched at an output point. Since the portions involved in the match need to be tainted (at least for the policies shown in Figure 6), such a match can occur only if the input satisfies certain characteristics. Ideally, there would be an automated procedure that can derive a necessary set of constraints on the input (but may not be sufficient) for any given output policy to be satisfied. Rather than exploring the development of such an automated approach, we took the more expedient step of manually specifying these input constraints. Development of such *pruning policies* poses only a minimal additional burden on the policy writer when the number of policies is small, as is the case in our approach. For the policies shown in Figure 6, only two pruning policies are needed:

- All of the SQL and command injection related policies require tainted data to span multiple output tokens. Hence, a suitable pruning policy is that an input contain at least one of the lexical separators in the output language, such as space, punctuation, etc.

- Cross-site scripting policy requires a tainted "script" tag. Thus, a pruning policy would check for the presence of this string in the input. Alternatively, a policy that checks for the presence of "<" and ">" in the input may be used.

Since taint inference is based on edit distance, pruning policies should also rely on edit distance with the same thresholds. (As discussed in Section 7.3, the threshold will automatically be set to zero for very small strings. This means that an exact match would be used for the shell and command injection pruning policies.)

Note that the pruning policies are determined by the output operation rather than the input. Thus, different pruning policies may be applied on the same input, depending on whether it is being compared with an SQL output, shell output or HTML output.

As shown in the next section, pruning policies are highly effective in practice. Even though they still involve the use of edit-distance algorithms, the lengths of input parameters (as well as the strings they are matched against) are small and hence lead to performance gains.

7 Evaluation

7.1 Attack Detection

Applications. Figure 7 shows the applications we studied in our experimental evaluation. These applications spanned multiple languages (Java, PHP, and C), and multiple platforms (Apache/modPHP, Apache/Tomcat, Microsoft IIS), and ranged in size from a few to several tens of KLoCs. Some of these applications are popular web applications used by millions of users, such as phpBB and SquirrelMail. Although most applications were tested (for attack detection) with Apache as well as IIS, our discussion here is focused on Apache-based tests.

Figure 7 also shows the attacks used in our experiments. Where 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. However, the effort involved in installing and recreating these "real-world" exploits is significant, which limits the number of attacks that can be used in our evaluation. To mitigate this

Application	Language	Size (lines)	Environment	Attacks	Comments	**Detection**	**False Positives**
phpBB 2.0.5	PHP/C	34K	IIS, Apache	SQL injection	CAN-2003-0486	Yes	None
SquirrelMail 1.4.0	PHP/C	42K	IIS, Apache	Shell command injection	CAN-2003-0990	Yes	None
SquirrelMail 1.2.10	PHP/C	35K	IIS, Apache	XSS	CAN-2002-1341	Yes	None
PHP/XMLRPC	PHP/C	2K	IIS, Apache	PHP command injection	CAN-2005-1921	Yes	None
AMNESIA [8] (5 apps)	Java/C	30K (total)	Apache/Tomcat	SQL injection	21K attacks, 3.8K legitimate	100%	0%
WebGoat [20]	Java		Tomcat	HTTP response splitting Shell command injection		Yes Yes	None None

Figure 7. Applications used in experimental evaluation

problem, we augmented our testing with WebGoat [20], a J2EE application that was designed to illustrate common web application vulnerabilities, and the AMNESIA dataset [8] that has been used for evaluating several previous SQL injection defenses [9, 10, 25, 26]. It consists of five[10] Java applications that implement a bookstore, an employee directory, an intranet portal, a classified advertisement service, and an event listing service. A suite consisting of several tens of thousands of legitimate requests and SQL injection attacks on these application is provided with the data set. Some of these attacks were designed to evade detection techniques, while the others were designed to comprehensively test them.

Policies. For attack detection, we used the policies shown in Figure 6. Command injection, SQL injection, and HTTP Response-splitting attacks are detected by TaintedCmd, SpanNodes and StraddleTrees policies, so we could have done with just one of them. Nevertheless, we listed them all to illustrate our policy language, and to show that there are multiple ways to detect these attacks.

The two XSS policies are designed to capture the two most common forms of XSS attacks. The TaintedScriptBody policy captures so-called reflected XSS attacks, where the script body is provided by an attacker using a form parameter (including those that may be included in a POST body) or a cookie, and is unwittingly inserted by the server into its HTML output. The TaintedScriptName captures a variant of this attack, where the attacker injects the name of a malicious script rather than its body. Note that the XSS policies given above can be further improved by carefully considering all possible ways of injecting scripts, and blocking all those avenues. This is a nontrivial task, as there are many ways of introducing script content into a HTML page. Nevertheless, our policy framework does provide the mechanisms needed to for this purpose.

Note that in order to block the PHP command injection attack, we needed a PHP parser. However, since our shell parser is quite generic, we were able to reuse it for parsing PHP in this case.

HTTP response splitting [19] involves injection of additional HTTP headers, which may in turn enable other attacks such as XSS. A vulnerable HTTP server inserts user-provided data as the value for a HTTP header without first checking that this insertion does not end up creating new headers. Our HTTP parser identifies each HTTP header as a cmd node, and hence a HTTP response splitting results in the violation of TaintedCmd, SpanNode and StraddleTrees policies.

Finally, our policy framework is powerful enough to detect several other types of attacks such as XPath injection and path traversals, although we did not include them in our experiments.

Detection Summary. All of the attacks were detected without generating any false positives. For all attacks except those involving AMNESIA dataset, we manually launched the attack and verified the results.

In the case of AMNESIA, the dataset consisted of 36,753 attacks. Of these, 15,337 attacks resulted in errors that were caught by the application, and hence the application itself did not issue a malicious query to the database. The remaining 21,416 attacks resulted in a malicious query being sent to the database. Of these, 21,106 were recognized by our technique as an injection attack and hence blocked. The remaining 310 were blocked because they caused parse errors in our SQL parser[11]. In summary, our technique was able to block every malicious query sent to the database.

7.2 False Negatives

False negatives may arise due to weaknesses in taint inference, input and output parsing, or policies. Our taint inference techniques can lead to false negatives for applications that perform complex transformations on inputs. It should be noted that the most common transformations arise

[10] In reality, the dataset contains seven applications, but two of them could not be successfully installed in our environment. However, these two applications are qualitatively no different from the other five, so this omission does not invalidate our results.

[11] In reality, many more attacks result in SQL queries with syntax errors. Because ours is a "rough parser," and because of its focus on graceful error recovery, it is able to construct a meaningful AST in spite of those errors, and discover policy violations.

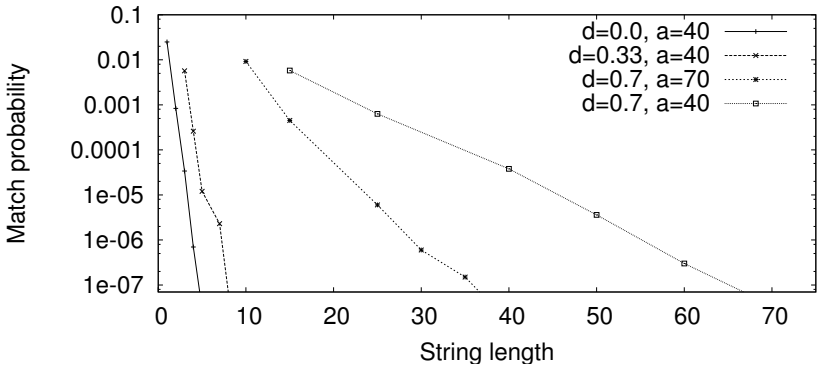

Figure 8. Match confidence.

due to various encodings used in HTTP, but these are already handled by our approach. Other than these, the most common transformations performed by web applications result in small changes to the input, and are hence detected by our approximate matching technique. However, if a web application makes extensive use of application-specific encodings or input-to-output transformations, taint inference may lead to significant false negatives and hence should not be applied to such applications.

False negatives will also occur with second-order injection attacks where the attackers first inject their malicious data into persistent storage (e.g., files or databases), and this data is subsequently retrieved by the server and inserted into its output. These second order attacks can escape taint-based techniques unless taint information is also stored and retrieved from persistent store.

Input and output parsing errors can also lead to false negatives. This can happen in two ways. First, attackers could send erroneous or very complex inputs, hoping to cause a failure in the parsing code used by defensive mechanisms. We mitigate this threat by (a) using rough parsers that parse a superset of the input language, and (b) focusing on error recovery in our parsers. The resulting parsers are sufficiently robust that we are able to simply block inputs that lead to unrecoverable parsing errors in our code.

A second evasion strategy that can lead to false negatives is to exploit the differences between how data is interpreted by the defensive mechanisms versus the actual target of the data. For instance, the well-known MySpace worm (aka Samy worm) exploited the fact that a syntactically incorrect HTML fragment, which contained a newline character in the middle of the string "javascript", was "corrected" by a browser that removed the newline. Unless this (odd) behavior was known and incorporated into our HTML parser, this kind of attack would go undetected. There does not seem to be a general solution to this class of evasions, except to move the policy enforcement to the browser [11].

Finally, false negatives may occur due to policy errors or incompleteness. We remark that the simplicity and generality of the SpanNodes and StraddleTrees policies make it difficult to carry out evasion attacks that lead to false negatives. However, with the XSS attack, there are many ways to inject scripts into web pages. False negatives can occur when the policy misses out some of these ways. This source of false negatives is shared generally by policy-based defenses.

Although the potential for false negatives exist, we remark that many web applications only rely on simple input transformation that are easily handled by our taint inference approach. As a result, no false negatives were observed in our experiments, in spite of the fact that the AMNESIA dataset consists of many thousands of attacks.

7.3 False Positives

False Positives in Taint-Inference. False positives occur when an input and output are within an edit-distance of d without any actual information flow taking place, i.e., as a result of coincidence. To control them, we need to select d such that the probability of coincidental matches is very low, e.g., pick d such that $P(TD(s,t) < d)$ is less than a desired false positive rate, say, 10^{-7}. (Recall that we use s to denote an input parameter and t to denote an output.)

In order to arrive at such probabilities, we need to make assumptions regarding the distribution of characters in s and t. An analysis of typical inputs and outputs of an application may be used to arrive at these distributions, but the difficulty with this approach is that it will differ from one application to another, and moreover, require "training data" for each application. We therefore made the simplifying assumption of uniform distribution, and studied match probabilities in terms of four parameters:

- $\boldsymbol{n}$, the length of s,

- ***m***, the length of t,
- ***d***, the distance threshold, and
- ***a***, the size of the alphabet.

Analytical derivation of match probabilities is known to be a hard problem in the context of approximate string matching. Therefore our approach was to generate strings using a uniform pseudorandom number generator, and experimentally determine the distribution of TD on such strings. Note that with increases in m, the number of substrings that could be matched against s increase linearly, so the match probability can be expected to increase roughly linearly with m. Thus, in our experiments, we did not vary m, but used a fixed value of $m = n$. We used two different alphabet sizes, $a = 40$ (which would correspond roughly to applications that use alphanumeric strings without distinguishing between cases) and $a = 70$. The results are shown in Figure 8.

From the type of results shown in Figure 8, we derived a table of n versus d. This table is used to select a value of d at runtime based on the value of n and m. For instance, if $n = 8$ and $m = 8$, a distance of $d = 0.33$ will result in a false positive rate of about 10^{-7}. If $m = 80$, then the false positive rate increases correspondingly, to about 10^{-6}. (Recall our observation that the probability of coincidental match increases roughly linearly with m.) We limited d to a maximum of 0.3, and the false positive rate to a maximum of 10^{-4}. (This rate of false positive occurs for very short strings ($n = 3$), while the rate can be kept to 10^{-6} or less when $n > 6$.)

False Positives in Attack Detection. In order to have a false positive in attack detection, note that we need to have a false positive in taint inference, and in addition, a policy violation needs to occur. Thus, false positives in attack detection can be expected to be significantly less than those of taint inference.

In our experiments, we did not encounter any false positives. While the thoroughness of false positive testing on applications such as phpBB could be questioned, the false positive results on AMNESIA dataset is quite meaningful. The creators of this dataset crafted a set of about 3.8K legitimate queries, many of which were intended to "look like" attacks. Our technique did not flag an attack on any of these.

Application	# of requests	Response time	Overhead
bookstore	605	20.67	1.7%
empldir	660	17.33	3.4%
portal	1080	31.67	5.1%
classifieds	576	18.00	4.3%
events	900	23.10	3.1%
Total	3821	110.77	3.5%

Figure 9. End-to-end Performance Overhead.

7.4 Runtime Overhead

Performance-related experiments were carried out on a laptop with dual-core 2GHz processor with 2GB memory running Ubuntu (version 6.10) Linux. Our goal was to measure the performance overhead introduced by benign requests.

First, we investigated the performance gains obtained as a result of using our coarse-filtering algorithm described in Section 4.2. These measurements were carried out across the applications shown in Figure 7 across several runs that contained a combination of attacks and benign requests. For this comparison, the CPU time spent within the taint inference algorithm was compared, without considering the time for I/O, parsing, policy enforcement, etc. We observed that

- taint inference was speeded up by a factor of 10 to 20, with the average of about 14,
- taint inference memory requirements were reduced by a factor of 50 to 1000, and
- the time spent in the coarse-filtering algorithm far exceeded the time spent in the edit-distance algorithm.

Next, we investigated the relative contribution of different components of our implementation (excluding the time spent within event interceptors) to the overall performance. We found that

- about 60% of the time was spent in taint inference algorithms,
- about 20% of the time was spent in the parser, and another 20% in policy checking.

These numbers were obtained using profiling tools. These results validate the efforts put into improving the performance of taint inference.

Overall Performance overhead. Finally, we measured the total performance overhead introduced by our approach, after all optimizations (including pruning policies) were factored in. Our focus in this context was on benign requests, the assumption being that the vast majority of requests received by a protected server would likely be benign, and hence the overhead due to benign requests will dominate over those of attacks. For this measurement, we used the AMNESIA dataset because the requests can be launched from a script. (In contrast, user interaction is required for other applications, thus making it hard to make

meaningful performance measurements.) All of the policies shown in Figure 6 were enforced.

Figure 9 shows the results. The "response time" column shows the total time taken (in seconds) to carry out all of the requests in the absence of our defenses. (Not even the event interceptors are in place.) It measures the wall-clock time for sending all the requests and receiving responses. The "overhead" column shows the increase in response time when our defenses are enabled.

These performance numbers reflect the effect of the optimization described in Section 6. This optimization is exceptionally effective: when an input parameter does not match our attack filtering criteria, then it is not processed by taint inference algorithm. Moreover, output parsing and policy enforcement are skipped if none of the parameters in a session match attack-filtering criteria. As a result, this optimization improves performance by an average factor of 5 for these applications.

8 Related Work

Model-carrying code [24] and Dataflow anomaly detection [1] developed techniques for discovering information flows by using runtime comparison of parameter values to different function calls. The focus of these works was to discover equality relationships among relatively short strings (mainly, file names), which is much easier than the taint inference problem described in this paper. Moreover, those works were concerned with building an automata model of program behavior, which is quite different from our goals in this paper of defining policies and detecting injection attacks.

A number of techniques have been developed that rely on taint-tracking for detecting memory corruption attacks [27, 5, 16] and script injection attacks [21, 17, 10]. This prompted much research into efficient techniques for automated taint-tracking [28, 13, 22, 23]. Nevertheless, taint-tracking remains quite expensive, at least in the context of C and binary code, incurring high overheads of 50% or more. More importantly, it requires extensive instrumentation of protected application, which may impact its stability, and/or may need source code access. We have therefore developed a complementary approach that avoids such instrumentation.

Several researchers have made the observation that injection attacks are characterized by changes to the structure of commands. AMNESIA [9] relied on a static analysis to detect the intended structure of SQL queries. Static analysis, since it must operate without knowing runtime inputs, must make some approximations that decrease its accuracy. Candid [25] therefore uses a dynamic analysis to discover intended query structure. Recently, this technique was extended to address XSS attacks [2]. These dynamic approaches, although different from taint-tracking, nevertheless rely on deep instrumentation of protected applications, and hence have drawbacks similar to those of taint-tracking approaches.

Our approach, in many ways, is similar to traditional intrusion detection techniques. These techniques can operate non-intrusively, based on observations that could be made without deep instrumentation, or otherwise adversely affecting the application operation. At the same time, our approach is able to offer greatly improved accuracy over intrusion detection techniques, which have historically suffered from high false positive (or in false negative) rates. Indeed, our approach is able to offer about the same level of accuracy as taint-tracking and related techniques, but without the need for deep instrumentation.

It may appear that taint-tracking techniques have an advantage in terms of being able to reason about information flows in the presence of complex transformations, but this is arguable. Most practical taint-tracking techniques track only data dependence, and ignore control-dependence and implicit flows. This results in missing flows when complex transformations take place. Moreover, when tainted and untainted data are stored and retrieved from the same aggregate data structure, taint-tracking techniques can introduce a number of false positives. In contrast, none of these aspects lead to problems with our approach, but it faces a different set of problems, such as the application of simple functions on input data. Thus, in terms of their ability to handle complex transformations, the strengths and weaknesses of the two approaches complement each other. Moreover, since web applications do not seem to use many complex transformations (except the standard encodings that are already handled by our approach), our taint inference approach is robust enough to reliably detect attacks on them.

The focus of Su et al [26] was on developing a formal characterization SQL injection and related attacks. Our work improves on theirs by providing a policy framework that is language-neutral, and moreover, is decoupled from parser implementations.

The implementation of Su et al [26] does not require application instrumentation. However, it relies on modifying inputs to add "bracketing" characters around each input. It assumes that these characters would propagate unchanged to the output, where they can be stripped away. This technique is fragile and can break many real applications. For instance, in the case of SquirrelMail and phpBB, a number of input parameters are not used in SQL queries, so the technique may have no opportunity to remove the brackets before the parameter value is used. In other cases, the parameter may be used in SQL queries as well as for other purposes, e.g., generating a URL, setting a cookie value, generating a file name, etc. Moreover, applications may make assumptions about the length of their inputs, or may per-

form computations that depend on their value. Finally, even if the bracketing approach could be used, it requires manual assistance to identify the set of characters that could be used for bracketing. In contrast, our approach does not require any modification of inputs or outputs, and is truly a black-box technique.

Microsoft has recently incorporated a defense for common forms of reflected XSS attacks in IE 8 [15]. It shares some similarities with our approach: it is also based on comparing inputs and outputs, and it detects attacks when the HTTP request data matches some of the characteristics of an XSS attack. But there are several differences as well. Whereas our defense is implemented on the server side, their defense is more accessible to an end-user since it resides on the client side. More importantly, the technical approach for recognizing reflection is quite different in the two approaches. Whereas we rely on rough parsing and approximate substring matching, their approach is based on regular expression matching. Specifically, they match outgoing HTTP requests against regular expressions crafted to identify those requests that may be involved in XSS attacks. If there is a match, their approach generates another regular expression that captures the characteristics of a response (to this request) that contains an attack. If the response matches this regular expression, an XSS attack is detected. The regular expression used to match against requests differs from that for responses in order to allow for small changes (such as character removal, insertion or translation) that may occur on the server. Whereas we rely on approximate string matching to handle this problem, their approach seems to rely on the developers to encode possible differences explicitly in terms of the differences between the regular expressions for matching requests and responses.

Privacy Oracle [12] is a recently developed black box approach for discovering information leaks by applications. Although information leaks are often modeled in terms of information flows, this is not the approach taken in Privacy Oracle. Instead, they infer possible flows based on differential testing, i.e., by observing differences in the application output when the input is changed. Their approach has some superficial similarities to our approach in that they make use of sequence alignment algorithms, which are closely related to approximate matching. However, in contrast with our approach where we match inputs with outputs, their approach applies sequence alignment algorithms to compare different outputs so as to discover what portions of the output remain constant and what portions change.

9 Conclusions

In this paper, we presented a new approach for accurate detection of common types of attacks launched on web applications. Our approach relies on a new technique for inferring taint propagation by passively observing inputs and outputs of a protected application. We then presented a new policy framework that enables policies to be specified in a language-neutral manner. As compared to previous works, our approach does not require extensive instrumentation of the applications to be protected. It is robust, and can easily support applications written in many different languages (Java/C/C++/PHP), and on many platforms (Apache/IIS/Tomcat). It is able to detect many types of command injection attacks, as well as cross-site scripting within a single framework, and using very few policies. It introduces significantly lower overheads (typically less than 5%) as compared to previous approaches.

Acknowledgements

We would like to thank Mark Cornwell, James Just and Nathan Li from Global Infotek for numerous discussions on this project, and for helping with experiments involving Microsoft IIS and WebGoat. We would also like to thank Lorenzo Cavallaro for compiling the CVE vulnerability chart shown in Figure 1, and Wei Xu for providing a working version of many of the exploits used in this paper.

References

[1] Sandeep Bhatkar, Abhishek Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *IEEE Symposium on Security and Privacy*, May 2006.

[2] Prithvi Bisht and V.N. Venkatakrishnan. Xss-guard: Precise dynamic detection of cross-site scripting attacks. In *International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment*, 2008.

[3] BreachSecurity. Modsecurity: Open source web application firewall. On the web at http://www.modsecurity.org/.

[4] Gregory Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. Using parse tree validation to prevent sql injection attacks. In *SEM '05: Proceedings of the 5th international workshop on Software engineering and middleware*, pages 106–113, New York, NY, USA, 2005. ACM.

[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] National Center for Biotechnology Information (NCBI). Basic Local Alignment Search Tool (BLAST). On the web at http://www.ncbi.nlm.nih.gov/blast/Blast.cgi.

[7] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[8] William Halfond. SQL injection application testbed. On the web at http://www.cc.gatech.edu/ whalfond/testbed.html.

[9] William Halfond and Alessandro Orso. AMNESIA: Analysis and monitoring for neutralizing sql-injection. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005.

[10] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *SIGSOFT FSE*, 2006.

[11] Trevor Jim, Nikhil Swamy, and Michael Hicks. Beep: Browser-enforced embedded policies. In *WWW*, 2007.

[12] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. Privacy oracle: a system for finding application leaks with black box differential testing. In *CCS*, 2008.

[13] Lap Chung Lam and Tzi cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[14] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(707), 1966.

[15] Microsoft. IE 8 XSS Filter Architecture/ Implementation, 2008. On the web at http://blogs.technet.com/swi/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx.

[16] James Newsome and Dawn 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.

[17] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, 2005.

[18] University of Virginia. Fasta. On the web at http://fasta.bioch.virginia.edu/.

[19] OWASP. Http response splitting. On the web at www.owasp.org/index.php/HTTP_Response_Splitting.

[20] OWASP. Owasp webgoat project. On the web at http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project.

[21] Tadeusa Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID)*, 2005.

[22] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting general security attacks. In *IEEE/ACM International Symposium on Microarchitecture*, December 2006.

[23] Prateek Saxena, R. Sekar, and Varun Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *IEEE/ACM Conference on Code Generation and Optimization (CGO)*, April 2008.

[24] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *ACM Symposium on Operating System Principles*, Bolton Landing, New York, October 2003.

[25] P. Madhusudan Sruthi Bandhakavi, Prithvi Bisht and V.N. Venkatakrishnan. Candid: Preventing sql injection attacks using dynamic candidate evaluations. In *CCS*, 2007.

[26] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–382, New York, NY, USA, 2006. ACM.

[27] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas 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.

[28] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, August 2006.