Joza: Hybrid Taint Inference for Defeating Web Application SQL Injection Attacks

Abbas Naderi-Afooshteh*, Anh Nguyen-Tuong†, Mandana Bagheri-Marzijarani‡, Jason D. Hiser§, Jack W. Davidson¶
Department of Computer Science
University of Virginia, Charlottesville, US
e-mail: *abiusx@virginia.edu, †an7s@virginia.edu, ‡mb3wz@virginia.edu, §dh8d@virginia.edu, ¶jwd@virginia.edu

***Abstract*—Despite years of research on taint-tracking techniques to detect SQL injection attacks, taint tracking is rarely used in practice because it suffers from high performance overhead, intrusive instrumentation, and other deployment issues. Taint inference techniques address these shortcomings by obviating the need to track the flow of data during program execution by inferring markings based on either the program's input (negative taint inference), or the program itself (positive taint inference). We show that existing taint inference techniques are insecure by developing new attacks that exploit inherent weaknesses of the inferencing process. To address these exposed weaknesses, we developed Joza, a novel hybrid taint inference approach that exploits the complementary nature of negative and positive taint inference to mitigate their respective weaknesses. Our evaluation shows that Joza prevents real-world SQL injection attacks, exhibits no false positives, incurs low performance overhead (4%), and is easy to deploy.**

I. Introduction

Despite increasing awareness of security issues in recent years [34], widely-used Web applications remain vulnerable to SQL injections and other common attacks [35], [38]. The impact of such attacks is severe and can lead to full server takeovers [38]. SQL injections have consistently ranked on top of various lists, e.g., #1 on MITRE's 2011 CWE/SANS list of Top 25 Most Dangerous Software Errors [17], and #1 or #2 on OWASP Top 10 Web Application Vulnerabilities for 2007 [34], 2010 [39] and 2013 [38]. Proposed solutions primarily rely on developer awareness of secure coding practices (such as prepared statements and sanitizing inputs), but these practices are routinely ignored or exercised incorrectly. Furthermore, these best practices are rarely retrofitted to the ever-growing base of existing legacy code.

Compounding this problem, popular Web frameworks such as WordPress actively encourage their developer community to extend the base framework with new functionality via a plugin architecture. While the core frameworks are heavily scrutinized and employ best coding practices, the quality of plugins varies widely. Attesting to the low quality of plugins, we collected 50 vulnerable Wordpress plugins. Using a range of SQL injection attacks, we then harvested and adapted a working exploit for each plugin [2].

A well-explored technique for detecting SQL injection attacks is negative run-time taint-tracking, where untrusted data is annotated with taint markings and these markings are maintained as data flows through an application [21], [26], [9], [7], [23], [14]. Security-critical commands in the application can then be checked for the presence of tainted commands, which, if present, indicates a potential attack. Another form of taint-tracking is that of positive taint-tracking, in which taint markings are associated with data that originate from within a program, and are therefore trusted [12], [13]. In this case, a security-critical command that is not marked as positively tainted indicates an attack is being attempted. Figure 1 illustrates the complementary nature of using negative and positive taint to detect attacks. In the figure, − indicates negative taint markings (untrusted), + indicates positive taint markings (trusted), and c indicates critical SQL tokens obtained by parsing the command.

Despite the security effectiveness of taint-tracking techniques, they are rarely deployed. For PHP, our target language, solutions that provide good performance (in the 10% range) require administrator privileges to install custom interpreters or extensions [21], [26], [29], [14]. Further, PHP continues to evolve at a rapid pace, which makes adopting taint-tracking extensions a risky business proposition as these extensions will invariably fall behind releases of the main distribution. Solutions that manage the propagation of taint information at the source code level, e.g., directly in PHP, incur high overhead (in the 200% range) [23].

A low-overhead, emergent alternative approach to taint tracking is *taint inference*. Taint inference techniques seek to infer taint markings, obviating the need for the complex machinery and modeling required to propagate and maintain taint information [28], [22]. Analogously to taint-tracking techniques, taint inference techniques are categorized as negative [28] or positive [22] depending on whether they seek to infer taint markings for untrusted data (negative taint) or trusted data (positive taint) (Figure 1).

The potential disadvantage of taint inference is that it is susceptible to false negatives, i.e., missed attack detection, due to the inherent imprecision in the inference process (Section 3). The key insight underlying our approach is that a hybrid taint inference model that exploits the complementary nature of negative and positive taint techniques results in a much more secure system than using either inference technique in isolation while simultaneously mitigating their respective weaknesses.

The primary contributions of this paper are:

- A convincing demonstration that neither negative taint inference nor positive taint inference is adequately secure. Using novel but straightforward techniques we success-

```
INPUT    : 1' OR 1='1
SQL QUERY: SELECT * FROM users WHERE Username='1' OR 1='1' AND Password='foo'
NEGATIVE :                                     ----------
POSITIVE : ++++++++++++++++++++++++++++++++++++          ++++++++++++++++++++
STRUCTURE: cccccc   cccc       ccccc         c    cc  c    ccc         c
```

Fig. 1: Negative and positive taint markings for a SQL query. An attack is detected when a critical part of the query structure is negatively tainted, or when it is *not* positively tainted. Note that attack detection is orthogonal to whether the taint markings are obtained via traditional taint-tracking techniques or via taint inference.

fully mutated 51 out of 53 real-world exploits to bypass negative taint inference. For positive taint inference we developed an automated evasion tool to adapt 14 out of 53 real-world exploits to bypass positive taint inference.

- The development of a novel hybrid taint inference model that synergistically combines negative and positive taint inference, resulting in a more secure system than either. Attacks that evade negative taint inference are detected by positive taint inference, and vice-versa.
- A comprehensive evaluation of a hybrid taint inference prototype called Joza.[1] We show that Joza incurs less than 5% overhead, with no false positives, is easy to deploy, and thwarts a wide range of SQL injection attack types, including 53 instances of novel attacks designed to bypass positive or negative taint inferencing.
- The development of WP-SQLI-LAB [2], an open-source fully-automated SQL injection test suite.

The rest of this paper is organized as follows. The threat model is presented in Section II. Section III describes the hybrid taint inferencing model, discusses positive and negative taint inference techniques, including their complementary strengths and weaknesses in detail. Section IV presents a high-level architecture of Joza and its deployment model. We present the security evaluation in Section V, followed by performance evaluation in Section VI. Section VII discusses related work, while Section VIII provides concluding remarks.

II. Threat Model

Our threat model assumes software is intended to be benign, but likely contains flaws. The program, when run, accepts untrusted input, possibly from many sources such as files, environment variables, HTTP request bodies, HTTP request headers, databases and others. The input is then used to create SQL queries that are issued to the database. Most inputs to the program are benign and cause the queries to behave as intended, but malicious inputs may exploit the program to violate the security policy intended for the SQL queries. An SQL injection occurs when attacker-controlled inputs are interpreted as SQL keywords, built-in functions, or delimiters, or when they change the programmer-intended syntactic structure of a command [36], [28].

[1]Joza is the Persian/Arabic equivalent of the Gemini zodiac constellation, which is Latin for twins.

We considered using a strict definition of SQL injection attacks such as the one defined by Ray and Ligatti [27], [29]. Unfortunately, many programs, such as those that incorporate advanced search functionality, would break as they allow field and table names to be specified through user inputs [6], [30], [31], [3]. We assume a more pragmatic stance, which permits these common programming practices, but the techniques presented can be easily adjusted to enforce a user's desired policy.

III. Taint Inference Models

To motivate the key insights underlying the Joza hybrid taint inference model, we first present the strengths and weaknesses of current negative and positive taint inference models.

A. Negative Taint Inference (NTI)

Negative taint inference (NTI) infers taint markings by correlating application inputs with query strings [28]. The pseudo-code for the NTI inference algorithm is as follows:

```
query q = intercept_query()
for each input source, S
   for each input p, in S
      diff_ratio = substring_distance(q, p)
      if diff_ratio < threshold
         mark_negative_taint(q, p)
```

NTI employs an approximate string matching algorithm to make allowance for common and small string transformations performed by an application, such as stripping whitespace and performing case-conversions. Function `substring_distance` computes a difference ratio which is the string distance between an input and a query divided by the length of the matched query substring. A difference ratio of zero means that the input string appears unchanged inside the query. If the `diff_ratio` is below a threshold the algorithm infers that a match has occurred. As will be discussed shortly, selecting a proper threshold is not straightforward.

Finding the minimum substring distance is a computationally expensive algorithm. In its simplest form, every substring of the query is compared to the input using the Levenshtein edit-distance algorithm [15]. This simple form has a computational cost of $\mathrm{O}\left(n^2 \times m^2\right)$ where n is the length of the input parameter and m is length of the query. The running time of the algorithm is $\mathrm{O}\left(l \times n^2 \times m^2\right)$ where l is the number of input parameters. This algorithm is impractical for long queries

```
  INPUT:       http://example.com/show_record.php?id=380
A QUERY:       SELECT * FROM records WHERE ID=380 LIMIT 5
  NTI MARKING:                                ---
  STRUCTURE:   cccccc   cccc         ccccc   c    ccccc

  INPUT:       http://example.com/show_record.php?id=-1 OR 1=1
B QUERY:       SELECT * FROM records WHERE ID=-1 OR 1=1 LIMIT 5
  NTI MARKING:                                ---------
  STRUCTURE:   cccccc   cccc         ccccc   c   cc  c  ccccc

  INPUT:       http://example.com/show_record.php?id=-1/*'''''*/OR 1=1
C QUERY:       SELECT * FROM records WHERE ID=-1/*\'\'\'\'\'*/OR 1=1 LIMIT 5
  NTI MARKING:
  STRUCTURE:   cccccc   cccc         ccccc   c  cccccccccccccccc  c  ccccc
```

Fig. 2: NTI Markings. Part A: benign input, Part B: malicious input (attack detected), Part C: evasive input (attack undetected).

composed of large user inputs, such as when a user posts a multi-page blog entry or uploads a file, or when a visitor posts a sizable comment.

Numerous optimizations exist for this algorithm, such as computing distances using dynamic programming and using heuristics to skip implausible comparisons [28]. The optimizations used in Joza's NTI component are explained in the performance evaluation (Section VI).

Figure 2 shows the taint markings inferred for various inputs sent to a vulnerable application. In part A of Figure 2, the query is deemed safe as no critical token has been marked as negatively tainted. Part B of Figure 2 illustrates how NTI detects an attack. NTI infers that `-1 OR 1 = 1` is negatively tainted as it precisely matches the value of the input parameter `id`. Because the critical tokens `OR` and = are tainted, NTI detects a potential attack.

1) Strengths:

Low Overhead and Low Implementation Complexity. NTI performs well when there is a strong correspondence between application inputs and queries. NTI has negligible memory and processor footprint for small inputs and queries, and only needs to be computed when input is provided to the application [28].

2) Weaknesses:

Sensitivity to Threshold Value. As previously noted, NTI uses an approximate string matching algorithm to allow for transformations of the input. The sensitivity of the string matching algorithm is tuned by specifying a threshold value that is proportionally related to the edit distance between an input value and its match in the query string. Setting the threshold value too high yields the inference of too many taint markings, which causes false positives. On the other hand, setting the threshold value too low yields too few taint markings, which causes false negatives. Selecting an optimum threshold value for an application or across a set of applications is not straightforward.

Evasion via Application-level Transformations. Any input transformation applied inside an application can potentially result in the bypass of NTI as it breaks the correspondence between inputs and query strings. For example, a common data transformation is to use a Base64 encoding where binary data is converted to human-readable characters for transfer over ASCII-based protocols.

Most web applications apply some form of input manipulation for the purpose of validation, sanitization or normalization. For example, Wordpress enforces *Magic Quotes*, a deprecated PHP facility that escapes quotes, backslashes and double quotes with additional backslashes. Wordpress also trims whitespace from input provided by authenticated users.

For applications that perform similar transformations to Wordpress, an attacker can craft an injection payload that includes a comment block, inside of which an arbitrary number of special characters (e.g., quotes in the case of Wordpress) can be added. The web application will then transform and include these escaped quotes in a comment block inside the SQL query, resulting in a higher string edit distance than the specified threshold, effectively bypassing negative taint inference.

An attacker can also leverage whitespace trimming (a common operation) by appending an arbitrary number of whitespaces, and rely on the fact that these whitespaces will be removed by the web application. Again, the net effect is a higher string edit distance than the specified threshold. Note that evasions can be done via any transformation of input inside the application code, and are not limited to the examples discussed here.

Part C of Figure 2 illustrates NTI evasion. The edit distance between the input and the matched portion of the output is five (the number of backslashes added by magic quotes). Dividing by the length of the entire matched portion (22) yields a 22.7% difference ratio, which is not small enough to cause a match for a threshold of 20%. An adversary evades NTI by adding enough quotes to drive the difference ratio higher than the threshold value.

Payload Construction. Concatenation of two or more inputs by an application enables attackers to construct an attack payload that potentially evades NTI. The following PHP code and sample input demonstrate this attack:

```
$input=$_GET['q1'].$_GET['q2'].$_GET['q3'];
$query="SELECT * FROM data WHERE ID=".$input;
```

```
  QUERY:        SELECT * FROM records WHERE ID=380 LIMIT 5
A PTI MARKING:  ++++++++++++++++++++++++++++++++   ++++++++
  STRUCTURE:    cccccc   cccc          ccccc   c    ccccc

  QUERY:        SELECT * FROM records WHERE ID=-1 UNION SELECT username() LIMIT 5
B PTI MARKING:  ++++++++++++++++++++++++++++++++                      ++++++++
  STRUCTURE:    cccccc   cccc          ccccc   c  ccccc ccccccc cccccccccc ccccc

  QUERY:        SELECT * FROM records WHERE ID=-1 OR 1=1 LIMIT 5
C PTI MARKING:  ++++++++++++++++++++++++++++++++   ++  + ++++++++
  STRUCTURE:    cccccc   cccc          ccccc   c  cc  c  ccccc
```

Fig. 3: PTI Markings. Part A: benign input, Part B: malicious input (attack detected), Part C: malicious input (attack undetected).

Input: q1=1 O   q2=R TR   q3=UE
Query: SELECT * FROM data WHERE ID=1 OR TRUE

Note that taint markings inferred from different inputs cannot be combined to detect an attack as it would introduce too many false positives. For example, by combining common one letter inputs such as `O` and `R`, all queries containing the word `OR` would be incorrectly inferred as negatively tainted. Also to alleviate false positives that would result from matching very short inputs (such as single letters), NTI detects an attack only if an input matches at least one whole SQL token.

B. Positive Taint Inference (PTI)

In contrast to negative taint inference, positive taint inference (PTI) infers the parts of a SQL query string that should be trusted. The PTI technique works by reconstructing security-critical commands using string fragments extracted from the program. PTI was successfully used previously to thwart OS command injection attacks [22]. We generalize this work and adapt PTI to cover SQL injections for web applications.

The PTI inference process is conceptually simple and is shown with the following pseudo-code:

```
Let F be the set of string fragments
    extracted from program P
query q = intercept_query()
for each string fragment f in F
  for each position, i, in q
    if f == q[i..i+len(f)]
     mark_positive_taint(q[i..i+len(f)]);
```

The set of string fragments, `F`, is extracted by processing the application and all plugins to identify string literals contained in the application.

As shown, this algorithm is computationally expensive, running in $O\left(n \times m^2\right)$ where n is the number of fragments and m is the length of the query. Section VI-A describes optimizations to speed up the inference process.

Consider the following vulnerable PHP program:

```
$postid=$_GET['id'];
$query = "SELECT * FROM records WHERE ID=" .
    $postid;
$query = $query . " LIMIT 5";
$result = mysql_query($query);
```

For this example, the string fragment extraction process yields the following fragments:

```
id
SELECT * FROM records WHERE ID=
 LIMIT 5
```

Note that the space before `LIMIT 5` is part of the fragment extracted from the program and can be important in the matching process.

Figure 3 illustrates positive taint markings (denoted with +). In part A of Figure 3, the query is deemed safe as all critical tokens are positively tainted. Part B of Figure 3 illustrates the case when an attack payload such as *-1 UNION SELECT username()* is the application input. This payload extracts the database username, but is detected by PTI because three critical tokens (`UNION`, `SELECT` and `username()`) are not marked as positively tainted.

To prevent attackers from combining fragments to form a critical token, PTI requires that critical tokens be fully contained within a single fragment. For example, PTI does not allow the critical token `OR` to be created by combining the single-letter fragments `O` and `R`. Additionally, PTI treats SQL comments as one critical token and requires that comments be fully contained in one fragment.

1) PTI Strengths:

Input-Independence. A distinguishing feature of positive tainting techniques in general is that the process of obtaining the taint markings is intrinsic to a program. This process is not affected by external input and therefore is not subject to control by an adversary [12], [13], [22]. To reinforce this key point, note that the algorithm used by PTI to infer taint markings for a query depends only on string fragments extracted from the program. Independence from external inputs means that PTI is immune to issues that plague negative taint-tracking techniques, e.g., correctly identifying all sources of untrusted data, correctly propagating taint markings throughout execution of a program, and precisely modeling complex string functions such as regular expressions replacement functions.[2]

PTI is resistant to *second order attacks*, such as when the injection payload is cached into a file, and then retrieved by the application and fed into a query. PTI is also resistant to

[2]For example, Diglossia [29], PHPrevent [21] and the PHP taint-tracking extension [14] do not model functions such as `preg_replace` precisely.

```
  INPUT      : http://example.com/show_record.php?id=-1 OR 1=1
  QUERY      : SELECT * FROM records WHERE ID=-1 OR 1=1 LIMIT 5
A NTI MARKING:                                ---------
  PTI MARKING: +++++++++++++++++++++++++++++++   ++  + ++++++++
  STRUCTURE  : cccccc   cccc         ccccc   c   cc  c  ccccc

  INPUT      : http://example.com/show_record.php?id=-1/*'''''*/OR 1=1
  QUERY      : SELECT * FROM records WHERE ID=-1/*\'\'\'\'\'*/OR 1=1 LIMIT 5
B NTI MARKING:
  PTI MARKING: +++++++++++++++++++++++++++++++                ++  + ++++++++
  STRUCTURE  : cccccc   cccc         ccccc   c  cccccccccccccccc  c  ccccc
```

Fig. 4: Part A: Attack that is undetected by PTI but is detected by NTI, Part B: Attack that is undetected by NTI, but is detected by PTI.

mixed input-source attacks, such as when an injection payload is constructed inside the application by concatenating harmless inputs from different sources. Furthermore, input-independence enables extensive use of caching for performance optimization, since a query can be analyzed once and the analysis result cached indefinitely.

Encoding-Resistance. Encodings performed by the database engine and application logic are frequent in web applications. For example many web applications store encoded or encrypted data in cookies, sessions and databases for subsequent use. Firewalls and intrusion detection systems typically operate on user-input at the network level and have no visibility into the actual value of these inputs. PTI can access the original data, because the data is eventually decoded and used in a SQL query.

2) PTI Weaknesses:

Application-dependent Attack Surface. The set of extracted string fragments forms the vocabulary with which an attacker can craft an exploit. For example, in part C of Figure 3, the attacker-supplied input, `1 OR 1 = 1`, would erroneously be deemed safe if the program contained both the string fragments `OR` and `=`. In general, longer attack payloads that require multiple critical tokens have a higher probability of detection than shorter attacks.

C. Hybrid Taint Inference Model

The complementary nature of negative and positive taint inference techniques is concretely illustrated in the examples of Figure 4. Part A of Figure 4 shows an attack payload that is undetected by PTI but detected by NTI. Conversely, part B of Figure 4 shows an attack payload that is undetected by NTI but detected by PTI.

PTI is susceptible to short attack payloads built with only a few critical tokens. These payloads are likely intercepted by NTI, since they are of short length and appear mostly unchanged in the output. NTI is susceptible to long payloads constructed by leveraging application-specific transformations. These payloads are typically intercepted by PTI since they are composed of a large number of critical tokens or use large blocks filled with transformable data (such as whitespaces or comments).

To exploit the complementary nature of PTI and NTI, we combine them in one system so that even attacks explicitly designed to bypass one, will be detected by the other. If either algorithm detects an attack, an attack is reported. If neither technique detects an attack, no attack is reported. Thus, the combination mitigates the security weakness of each individual technique.

Combining NTI and PTI in a hybrid model also means composing false positive rates and overhead rates. Previous studies of NTI and PTI have shown performance overhead rates to be less than 5% with no false positives [28], [22]. Sections V and VI experimentally demonstrate that Joza, our system that implements a hybrid NTI and PTI model, retains favorable performance characteristics without incurring false positives.

IV. Joza System

Figure 5 provides an architectural overview of the Joza system. Joza consists of two major analyses components, PTI Analysis and NTI Analysis. The PTI Analysis component implements the positive taint inference algorithm, whereas the NTI component implements the negative taint inference algorithm. All commands intended for the backend database management system (DBMS) are intercepted and first sent to the PTI Analysis component, and then to the NTI Analysis component before being allowed to proceed to the DBMS.

A. Installation

Joza is initially installed by adding the preprocessing component to the entry point of a web application. In the case of Wordpress, this step can be done by placing Joza in the plugins directory of Wordpress and configuring the Wordpress plugin manager to run Joza automatically on every request.

A web application in PHP is typically a collection of PHP source code files residing in one top-level directory and several subdirectories. Joza recursively parses all source code files reachable from the top directory and extracts string literals from each file to form the final set of string fragments. These fragments will subsequently be used by the PTI Analysis component. In the case of format strings or other strings with placeholders, Joza breaks them down into multiple fragments.

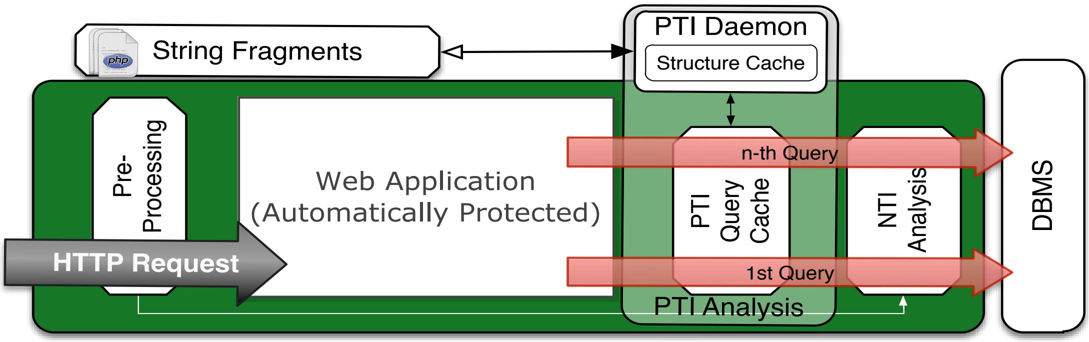

Fig. 5: Joza Architecture

For example, the string "`SELECT * from users where id = $id and password=$password`" would be broken down into two fragments:

```
SELECT * from users where id =
 and password=
```

Note that only fragments that contain at least one valid SQL token need to be retained.

To intercept queries, the installation process wraps all standard PHP functions and classes that interact with backend databases, e.g., `mysql*` and PDO*. These wrappers are implemented using a source-level transformation to replace all calls to database functions with calls to equivalent Joza wrappers.

B. Preprocessing

The preprocessing component defines Joza wrappers and stores a copy of all inputs to the web application to preserve them for NTI analysis. This step is required as many web applications modify user-input before it reaches NTI analysis. The preprocessing component also invokes the installer whenever new or modified files are found in the application (e.g when the application is updated or a new plugin is installed), to keep the set of string fragments complete and enable Joza to intercept all queries sent to the database by the application.

C. PTI Analysis Component

The PTI Analysis component sends intercepted queries to a PTI daemon. The daemon performs two primary functions. The first is to parse intercepted queries to extract critical tokens and keywords. The second is to infer which parts of the intercepted query should be trusted using the PTI algorithm described in Section III-B, and return whether the query is deemed safe or not. As an optimization, the PTI Analysis component maintains a query cache to store safe queries. For applications such as Wordpress with a workload heavily-skewed towards reads, this caching mechanism dramatically boosts performance (Section VI).

1) PTI Daemon: The PTI Daemon is a native binary application that loads the PTI dynamic library as well as the string fragments into memory, connects to the web application and waits for incoming queries. Once a query arrives, its structure and the result of its taint analysis is communicated back to the web application. Multiple daemon processes can coexist together. The lifetime of a single daemon instance can range from a single web application instance (comprising of multiple database queries) to hours.

The daemon is launched on demand (as a binary process) by the PHP application and communicates with the PHP application using named or anonymous pipes. In its shortest lifespan, the daemon lives for the duration of one web request, communicating via anonymous pipes and terminating alongside the application. To allow longer lifetimes, the daemon is launched independently of the launching web application (e.g. using `nohup`) and communicates to the web application instances using named pipes.

To improve performance, the daemon also includes a query structure cache which caches abstract syntax trees of parsed queries without storing contents of data nodes. This optimization is discussed in more detail in Section VI.

2) PTI Query Cache: The PTI query cache uses an in-memory hashtable in the backend database to cache the PTI analysis result of a query (i.e whether the query is safe or not). Because many queries of a web application are constant and do not rely on any user-input, caching improves performance significantly without noticeably increasing the memory footprint of the daemon.

D. NTI Analysis Component

To implement the NTI algorithm described in Section III-A, Joza must first make a copy of all inputs including cookies contained in HTTP headers, as well as HTTP GET and POST values. While computing the necessary substring distance between inputs and the intercepted query can be expensive, PHP directly supports this computation using a built-in Levenshtein edit-distance algorithm [15]. Once the negative taint markings have been inferred, the NTI Analysis Component reuses the critical tokens and keywords previously obtained by the PTI Daemon, and can then determine whether a query is safe.

E. Attack recovery

A query is safe if and only if both PTI and NTI components deem the query safe. When an attack is detected, Joza supports two recovery policies: error virtualization and termination. The error virtualization policy returns an error code as if the query had failed and relies on the application logic to handle this error gracefully. The termination policy forces the application to exit. The default Joza policy is to assume a conservative security posture; Joza uses termination, which typically results in a blank HTML page returned to the end user.

F. Architecture Rationale

The twin requirements for Joza to exhibit low overhead and be easy-to-deploy, i.e., without requiring administrator privileges, motivate our decision to implement the PTI algorithm as a user daemon. Two alternative designs for the PTI algorithm are PHP extensions and a pure PHP implementation.

A PHP extension is a native library linked against a specific version of PHP headers and is not compatible with other PHP versions, and would therefore require the PTI daemon to be updated as frequently as the PHP interpreter. Loading or installing PHP extensions requires administrative privileges, which is impractical in many deployment scenarios, e.g., shared hosting environments.

A pure PHP implementation of a SQL parser and the PTI algorithm was also tested, but rejected, as the resulting overhead ranged from 20% to 200%.

As for NTI, moving the analysis to the daemon would not benefit performance, because NTI requires all inputs of the application and communicating them to the daemon would incur more overhead than the performance gain, especially when processing sizable inputs (such as file uploads).

V. SECURITY EVALUATION

To evaluate Joza's security, we created WP-SQLI-LAB, an open-source security testbed consisting of a recent Wordpress version (v3.8) packaged with 50 plugins publicly reported to be vulnerable to SQL injection attacks [2]. The plugins represent a diverse set of applications, including social media, e-commerce, image galleries and forums. Exploits were obtained from various public sources, including CVE reports, security research blogs and other security-related websites.

Attack Type	NO. of Plugins
Union Based	15
Standard Blind	17
Double Blind	14
Tautology	4

TABLE I: Classification of WP-SQLI-LAB attack types

Table I lists the type of exploits collected and their frequency in the testbed. A union-based exploit allows attackers to replace the expected result of a query with a data record obtained by a query of their choosing. This type of exploit allows easy extraction of any information from the database. A standard-blind exploit returns errors if the query returns no results, and valid results otherwise. This type of exploit allows an attacker to extract desired data by binary searching each character using conditional payloads generated by automated tools (such as SQLMap) or manually. Double-blind exploits seek to determine the validity of an injected payload by observing the application's response time. With a judicious choice of payload, a double-blind exploit can leak vital information such as passwords. Again, typical attacks using this exploit are carried out using a binary search to leak data one valid character at a time. Tautologies such as `1 OR 1=1` can result in the leakage of information or bypassing of authentication code.

A. NTI and PTI Evaluation

The goal of our first experiment was to evaluate the effectiveness of NTI and PTI individually using our testbed. To the best of our abilities, we developed exploits for the testbed without consideration for either NTI or PTI.

As shown in Table II, the NTI component detected 49 out of the 50 original exploits. (NTI failed to detect an attack in a plugin that used a Base64 encoding of its inputs.) The PTI component detected all 50 original exploits. These results corroborate the effectiveness of taint inference techniques previously reported [28], [22].

To further evaluate the effectiveness of NTI and PTI, we used a powerful penetration tool (SQLMap [10]) on four of the 50 plugins. The four plugins were selected such that each of the exploit types in Table I was present. On average, SQLMap generated 40 valid attack payloads for each plugin. Both NTI and PTI detected all attack variants.

Exploits	NTI	PTI
Testbed	49/50	50/50
Generated by SQLMap	160/160	160/160

TABLE II: Baseline effectiveness of NTI and PTI

The results in Table II were encouraging as both NTI and PTI defeated almost 100% of the attacks. However, a sophisticated attacker would actively seek to take advantage of the weaknesses identified in Section III. In the next set of experiments, we explored the design space of attacks targeted explicitly to evade either NTI or PTI.

NTI Evasion. Since NTI is susceptible to application-induced transformations, we leveraged the Wordpress implementation of magic quotes (magic quote adds an extra backslash for every quote).

We mutated the original attacks by incorporating comment blocks that included quotes. Regardless of the threshold used by NTI for determining a match, an attacker can evade NTI by simply adding enough quotes to ensure that the attack input is above the threshold. Thus, changing the sensitivity threshold used by NTI would not be an effective remedy. Figure 6C shows such an attack. This novel evasion approach resulted in the complete bypass of NTI.

Fragment
UNION
AND
OR
SELECT
CHAR
,
#
- -
;
/* */
␣)
(
GROUP BY
ORDER BY
CAST
WHERE 1
INSERT
INSERT INTO
=
users WHERE ID =
:-)
)))
">?
To:
&#8727
<iframe
tail -c
<td rowspan=

TABLE III: Sample fragments in Wordpress

```
A. SELECT * FROM wp_proplayer WHERE (POST_ID='-1') union all select NULL,NULL,char('65')
B. SELECT * FROM wp_proplayer WHERE (POST_ID='-1') UNION SELECT 1,2,CHAR('65')
C. SELECT * FROM wp_proplayer WHERE (POST_ID='-1') union/*\'\'…\'*/all select NULL,NULL,char('65')
D. SELECT * FROM wp_proplayer WHERE (POST_ID='-1') UNION/*\'\'…\'*/SELECT 1,2,CHAR('65')
```

Fig. 6: Real-world exploit for one of WP-SQLI-LAB vulnerabilities. Part A shows the original exploit, part B shows the exploit mutated using Taintless to bypass PTI, part C shows the exploit adapted for NTI evasion and part D depicts the mixture of PTI and NTI evasions in the exploit.

PTI Evasion. To exploit the application-dependent attack surface of PTI, we created *Taintless* [1], an automated evasion tool that reconstructs attack payloads using string fragments available in an application. Taintless replaces certain SQL tokens with their equivalents (e.g. UNION with UNION ALL, CHAR with string literals), matches the letter case of attack tokens with those available in the application, removes those tokens not found inside the application that can be safely removed from the attack payload, and also matches the type and number of whitespaces with those available in the application.

Using Taintless, we succesfully adapted 13 out of 50 exploits in the testbed to evade PTI detection. Figure 6B shows one of these adapted exploits.

Table III lists example fragments extracted from Wordpress and the 50 plugins. Since these fragments include `OR` and = (among many SQL tokens), PTI does not detect an attack with a payload of `OR 1 = 1`. To understand how common simple injection payloads are in real-world applications, we analyzed 100 recently reported SQL injection vulnerabilities (containing exploit codes) listed by MITRE CVE from 2012 to 2014. Of these only 4 were tautologies (vulnerable to simple payloads) [20].

B. Hybrid Model Evaluation

The previous section evaluated the security of NTI and PTI individually. We now evaluate Joza, a system where both NTI and PTI are combined. Joza detects all attacks in the testbed, even attacks successfully adapted to evade NTI and PTI (Table IV).

One such attack is shown in Figure 6. Part A of the figure shows the original exploit in the resulting query, while parts B and C display adaptations to bypass PTI and NTI respectively. Part D shows an unsuccessful attempt at evading both taint inference techniques in a single exploit as each technique detects the adaptation used to bypass the other.

In general, the Joza PTI component stops the practice of using NTI evasion in an attack payload, as PTI requires that the entire evasion block originate from a single fragment. On the other hand, the susceptibility of PTI to malicious payloads that contain a small number of critical tokens available in the application is compensated by NTI.

Joza's hybrid taint inference algorithm dramatically raises the bar for mounting a successful SQL injection attack. To evade Joza one must construct an attack that evades both NTI and PTI. An example would be an attack against a plugin where NTI fails to detect the attack because the string distance is too high, and PTI also fails to detect the attack because the attack uses fragments previously extracted from Wordpress and the plugins. Despite our best efforts, we have not been able to create such an attack against the 50 plugins in our testbed.

To further demonstrate the effectiveness of our approach, we used Joza to protect Drupal, Joomla and osCommerce, popular applications with well-known, recently reported vulnerabilities.

The Drupal vulnerability [19] is based on encoded user-input used to construct prepared statements in the web application. Prepared statements are used to prevent SQL injection attack by sending the query to be prepared by the database engine first, and then separately sending user data to the database engine to be used in named or anonymous placeholders defined in the prepared query. Use of prepared statements would remove the attackers' ability to modify a query, and any input provided by an attacker would be treated as data by the backend database. Unfortunately, prepared statements are not a panacea. In this case, user input was used to construct the placeholder names in the query sent to the database to be prepared, allowing an attacker to provide carefully crafted input to modify the original command to the database, regardless of the data parameters.

Joomla was vulnerable to a very complicated double blind SQL injection attack which used encoded input to instantiate an object of a particular class inside the application [18]. This object would construct an SQL query based on its member variables (which could be overridden by the attacker), and execute the query on destruction.

osCommerce was susceptible to a tautology attack that extracted sensitive information from the database [8].

PTI or NTI were not sufficient to detect all three of these attacks on popular highly scrutinized web applications, but Joza successfully detected and prevented them.

False Positives. To evaluate false positives, we developed a script to perform a full crawl of the Wordpress application testbed, including posting random comments and performing random searches. We also manually clicked through various parts of Wordpress and did not uncover any false positives. We ran SQLMap on Wordpress configured with the plugins and verified that all attacks detected by Joza were true positives, i.e., valid attacks.

Plugin / Application	Version	CVE/ OSVDB	SQL Vulnerability	NTI Original Exploit	NTI Mutated Exploit	PTI Original Exploit	PTI Mutated Exploit	Joza
A to Z Category Listing	1.3	86069	Tautology	Yes	No	Yes	No	Yes
AdRotate	3.6.6	2011-4671	Tautology	No	No	Yes	No	Yes
Advertizer	1.0		Double Blind	Yes	No	Yes	Yes	Yes
Ajax Gallery	3.0		Double Blind	Yes	No	Yes	Yes	Yes
Allow PHP in posts and pages	2.0.0		Double Blind	Yes	No	Yes	Yes	Yes
Community Events	1.2.1	75252	Union Based	Yes	No	Yes	No	Yes
Contus HD FLV Player	1.3	74573	Tautology	Yes	No	Yes	No	Yes
Count per Day	2.17	75598	Union Based	Yes	No	Yes	Yes	Yes
Couponer	1.2		Union Based	Yes	No	Yes	Yes	Yes
Crawl Rate Tracker	2.02		Blind	Yes	No	Yes	Yes	Yes
Easy Contact Form Lite	1.0.7		Tautology	Yes	No	Yes	No	Yes
Event Registration plugin	5.43		Blind	Yes	No	Yes	Yes	Yes
Eventify	1.7.f	86245	Union Based	Yes	No	Yes	No	Yes
Facebook Promotions	1.3.3		Double Blind	Yes	No	Yes	Yes	Yes
File Groups	1.1.2	74572	Blind	Yes	No	Yes	Yes	Yes
FireStorm Real Estate Plugin			Union Based	Yes	No	Yes	No	Yes
GD Star Rating	1.9.10	83466	Blind	Yes	No	Yes	Yes	Yes
Global Content Blocks	1.2	74577	Double Blind	Yes	No	Yes	Yes	Yes
iCopyright	1.1.4		Blind	Yes	No	Yes	Yes	Yes
IP-Logger	3.0		Union Based	Yes	No	Yes	Yes	Yes
Js-appointment	1.5	74804	Double Blind	Yes	No	Yes	Yes	Yes
KNR Author List Widget	2.0.0		Blind	Yes	No	Yes	Yes	Yes
Link Library	5.2.1	84579	Blind	Yes	No	Yes	Yes	Yes
Media Library Categories	1.0.6		Union Based	Yes	No	Yes	Yes	Yes
Mingle Forum	1.0.31	75791	Double Blind	Yes	No	Yes	Yes	Yes
MM Duplicate	1.2		Blind	Yes	No	Yes	Yes	Yes
MyStat	2.6		Double Blind	Yes	No	Yes	Yes	Yes
OdiHost Newsletter	1.0	74575	Blind	Yes	No	Yes	Yes	Yes
Paid Downloads	2.01	86247	Blind	Yes	No	Yes	Yes	Yes
post highlights	2.2		Union Based	Yes	No	Yes	No	Yes
Profiles	2.0.RC1		Blind	Yes	No	Yes	Yes	Yes
ProPlayer	4.7.7		Union Based	Yes	No	Yes	No	Yes
PureHTML	1.0.0		Double Blind	Yes	No	Yes	Yes	Yes
SCORM Cloud	1.0.6.6		Double Blind	Yes	No	Yes	Yes	Yes
SearchAutocomplete	1.0.8		Union Based	Yes	No	Yes	No	Yes
SH Slideshow	3.1.4	74813	Blind	Yes	No	Yes	Yes	Yes
Social Slider	5.6.5	74421	Blind	Yes	No	Yes	Yes	Yes
UPM Polls	1.0.3		Union Based	Yes	No	Yes	No	Yes
VideoWhisper Video Presentation	1.1		Double Blind	Yes	No	Yes	Yes	Yes
Facebook Opengraph Meta			Union Based	Yes	No	Yes	Yes	Yes
Paypal Donation Plugin		74838	Blind	Yes	No	Yes	Yes	Yes
WP Audio Gallery Playlist	0.12		Union Based	Yes	No	Yes	No	Yes
WP Bannerize	2.8.7	76658	Blind	Yes	No	Yes	Yes	Yes
WP DS FAQ	1.3.2	74574	Double Blind	Yes	No	Yes	Yes	Yes
WP eCommerce	3.8.6	75590	Tautology	Yes	No	Yes	Yes	Yes
WP FileBase	0.2.9	75308	Blind	Yes	No	Yes	Yes	Yes
WP Forum Server	1.7.8	2012-6625	Union Based	Yes	No	Yes	No	Yes
WP Menu Creator	1.1.7	74578	Double Blind	Yes	No	Yes	Yes	Yes
yolink Search for WordPress	1.1.4	74832	Union Based	Yes	No	Yes	Yes	Yes
Zotpress	4.4		Double Blind	Yes	No	Yes	Yes	Yes
Joomla	3.0.1	2013-1453	Double Blind	No	No	Yes	Yes	Yes
Drupal	7.31	2014-3704	Union Based	Yes	No	Yes	Yes	Yes
osCommerce	2.3.3.4	103365	Tautology	Yes	No	No	No	Yes

TABLE IV: Joza security effectiveness evaluated using original and mutated real-world exploits on the Wordpress testbed. Joomla, Drupal and osCommerce were evaluated using only the original exploits.

VI. Performance Evaluation

The performance evaluation of Joza was carried out using Wordpress, a popular content management system that powers 22% of the top 10 million websites [32] . All evaluations were performed on a 4-core iMac using Mac OS X 10.10 with 24 GB RAM running at 2.9 GHz.

A. PTI Optimization

To measure the performance of the Joza PTI component, we setup a fully functional Wordpress site populated with 1001 unique URLs. Crawling the entire website resulted in approximately 20,000 SQL queries as Wordpress requires multiple database queries to render a page.

Our initial implementation of PTI initiated a new process to detect SQL injections. To make PTI fit for practical use, we dramatically increased its performance by running PTI as a daemon process and by performing two primary optimizations. The first optimization was to use a most-recently-used caching policy for fragments that match a query to take advantage of the SQL query working set of a Web application [22]. The second optimization was to first parse the query to determine the critical set of tokens before attempting to match these tokens. When coupled with the first optimization, benign queries are therefore quickly matched, while malicious queries may require scanning the entire set of fragments.

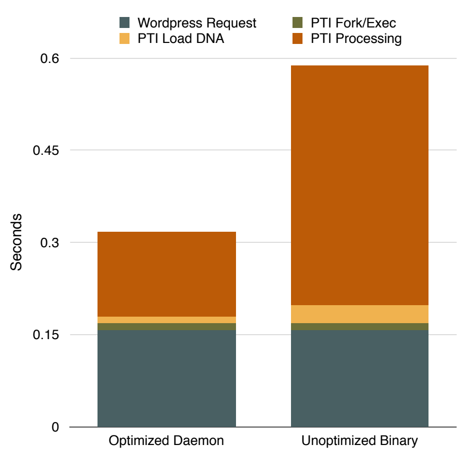

Fig. 7: Performance breakdown of optimized PTI daemon compared to binary PTI on top of Wordpress core

Figure 7 illustrates Joza's PTI performance breakdown for a Wordpress request. The unoptimized version is clearly dominated by PTI processing. The optimized daemon reduces this processing time by 66%.

	Original	With PTI	Exact Cache	Structure Cache
Read	0.2170	0.4440	0.2378	0.2255
Write	0.3319	0.8538	0.4441	0.3725

TABLE V: Average Read/Write time for a Wordpress request with PTI (seconds).

Table V characterizes performance overhead based on whether a Wordpress request is a read or a write request. A typical read request is to read a Wordpress post, whereas a write request might be to post a comment. Note that both types of request may result in multiple database queries.

For read requests, the use of a query cache to store previous PTI decisions, i.e., whether a given query has previously been deemed safe, reduces overhead to less than 4%. For write requests, the query cache also improves performance over the non-cached version, but incurs 34% overhead. The reason a Wordpress write request still benefits from caching is that posting a comment results in multiple database queries, some of which are database reads and so may have been cached.

Another caching mechanism was introduced to increase performance of write and other dynamic queries. The query structure cache caches the structure of the SQL query abstract-syntax-tree without the content of data nodes. This caching mechanism caches the safety result of all queries except those dynamically generated inside the application (such as advanced search). With this caching in place, write requests incur only a 12% overhead.

To support Joza's goal of ease-of-deployment, we deliberately chose not to implement PTI as a direct PHP extension as it would have required administrator privileges to install or load. Our results estimate that implementing PTI as a PHP extension would incur only 0.2% overhead for read requests and 3.2% for write requests (as described in Section C).

B. NTI Optimization

A naive implementation of NTI's string matching algorithm would be too slow for practical use. Fortunately, previous work provides several optimizations [33], [28]. Joza uses PHP's internal Levenshtein distance function for short inputs and queries. As an internal PHP function, its implementation runs at native speed instead of being emulated by the PHP interpreter. When input or query length is larger than that supported by PHP's Levenshtein function, Joza uses an optimized Levenshtein function written in PHP that requires linear memory and time.

C. Joza Overall Evaluation

Figure 8 displays the time spent on PTI and NTI for a full site crawl (read), random comment posting (write) and random searching. NTI and PTI overheads and the total overhead of Joza can be observed in the figure for different types of requests.

The performance of Joza depends on the relative frequency of reads vs. write requests. Table VI shows overhead for a variety of workloads. A workload consisting of 10% writes and 90% reads results in an overall overhead of 5%, whereas a workload of 99% reads and 1% writes results in an overall overhead of 4%.

We also estimate the cost of our design decision to implement Joza completely at the user-level. This estimation is based on not including daemon spawn and communication times in the calculations. A Joza system implemented as a direct PHP extension would incur only 1.7% overhead even with a

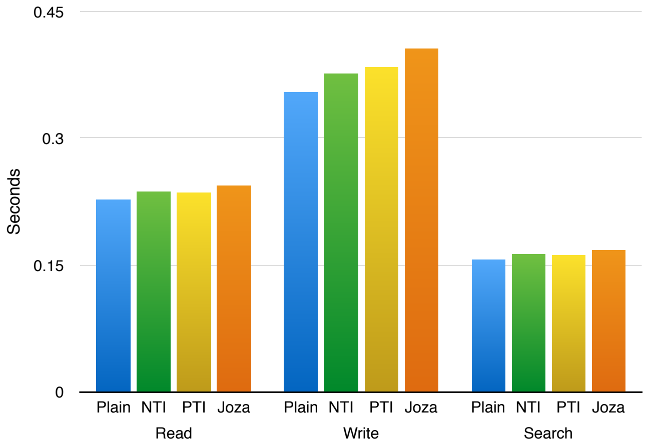

Fig. 8: Comparison of read/write/search times with and without Joza's protection in Wordpress

workload consisting of 50% write requests, which would make Joza well-suited for performance-critical deployment scenarios with full administrative privileges.

Table VII lists the average number of new blog posts, pages, comments and RPC posts (posts written or read via third party applications) over the last five years, as well as the average number of annual page views on all blogs hosted on Wordpress.com [41], [40]. From these statistics, we compute the typical read/write workload for Wordpress.com, the official website for hosting Wordpress sites. On average, less than one percent of all requests involve writes, which would result in less than 4% overhead on average when protected by Joza.

Writes	Reads	Plain Time	Protected Time	Overhead
50%	50%	0.2744	0.2990	8.96%
10%	90%	0.2284	0.2402	5.16%
5%	95%	0.2227	0.2328	4.53%
1%	99%	0.2181	0.2269	**4.03%**

TABLE VI: Overhead of Joza on different workloads.

In practice, Joza's overall performance would be further improved by using content-caching engines. Heavily-trafficked Wordpress sites often make use of such caches. With content-caching enabled, only the first request to a URL results in the execution of database queries to serve up the requested page. Subsequent requests would be mostly served by retrieving a static cached copy, thereby reducing the demand on Joza's processing time.

VII. Related Work

The appeal of taint inference techniques is that they obviate the need for propagating taint information during program execution. Previous PTI work focused on defeating OS command injection attacks for x86 binaries [22]. The work reported here widens the attack classes covered by PTI to include SQL injection attacks targeted towards web applications.

The vast majority of work in taint tracking uses a form of negative taint tracking, i.e. they track external (untrusted) data as it flows through a program and check whether such data is used in a security-sensitive operation [11], [21], [26], [9], [42], [12], [13], [7]. Livshits provides an extensive review of dynamic taint tracking projects [16] and their potential pitfalls, including the difficulty of propagating taint markings across functions correctly. For example, neither PHPrevent [21], nor the PHP taint-tracking extension [14] model taint accurately across string functions that support complex regular expression patterns, e.g. `preg_replace`. Failure to model such functions accurately can result in increased false negative or false positive rates. Joza sidesteps this issue completely as it does not propagate taint markings across functions.

While most taint-tracking approaches keep track of external data, Halfond et al. use positive taint tracking to track internal (trusted) data [12], [13]. The primary tradeoff is that positive taint tracking potentially results in higher false positive rates (breaking application functionality), whereas negative taint tracking tilts towards higher false negatives (missing attacks). Halfond advocates the use of positive taint tracking as it provides a more conservative security posture.

CANDID and Diglossia detect command injections using shadow computations instead of tracking taint information directly. CANDID builds shadow query strings in which user input is replaced with known non-attack strings such as a sequence of 'a' characters [4]. Any structural difference in the parse tree of the real and shadow queries reveals an attack. Diglossia uses a complementary approach to generate shadow queries. Instead of transforming strings derived from external inputs, Diglossia remaps strings that originate from within the application into an alternative character set [29]. To detect an attack, Diglossia checks that the parse trees are syntactically isomorphic, and that all SQL code in the shadow parse tree is encoded with the alternative character set. Since CANDID and Diglossia seek to delineate data from code, they are also subject to the complexity of modeling complex string functions. For example, Diglossia does not model `preg_replace()`.

Despite the large body of research with ample evidence of the effectiveness of taint-tracking techniques in defending web applications, taint-tracking is not widely deployed or used. To the best of our knowledge, Perl and Ruby are the only two major programming languages that provide built-in support for dynamic taint tracking [24], [25]. One reason for the lack of deployment is that propagating taint information often requires changes to the underlying run-time system [21], [26], [9], which hinders deployment as such changes typically require administrator privileges. Another potential reason is the perceived high cost of taint-tracking. While this perception is true for some projects (e.g. 2.2X for the ASPIS project on Wordpress [23]), others have reported average performance overhead in the 10-15% range when measured against various web application workloads [12], [13], [21], [26], [7].

SQLRand [5] uses randomization of critical SQL tokens to implement an alternate and secret SQL instruction set. This randomization is then reversed at run-time so that the database processes the original query. Without knowledge of the key used for randomization, an attacker cannot inject valid SQL tokens. Weatherwax relaxes the SQLRand requirement that the

	Posts	Pages	Comments	RPC	Views	Total Dynamic	Writes %	Reads %
2014	40,537	6,789	50,341	5,767	14,426,095	103,434	0.71	**99.29**
2013	48,7281	84,409	668,469	74,257	144,777,605	1,314,417	0.90	**99.10**
2012	351,612	79,046	468,318	40,725	112,329,943	939,701	0.83	**99.17**
2011	176,507	56,033	147,738	32,928	79,614,461	413,206	0.52	**99.48**
2010	145,696	39,519	126,097	21,155	49,021,659	332,467	0.67	**99.33**

TABLE VII: Wordpress.com statistics, raw numbers are divided by 10^3

randomization key be kept secret by using redundant parallel execution in such a manner that a SQL token valid in one variant is guaranteed invalid in the other [37]. However, this approach is subject to the same limitations as SQLRand in that it requires the complete and accurate identification of all SQL tokens, a process which is very difficult to automate, or error-prone if done manually.

VIII. Conclusions

This paper has shown that taint inference techniques offer many practical advantages including speed and ease of deployment, but the individual security of these approaches is weak. To address this weakness, we have developed a novel hybrid taint inferencing approach that synergistically combines the strengths of negative taint inference and positive taint inference. To illustrate the power of the hybrid approach, a prototype system, called Joza, was developed to automatically protect PHP-based applications. This paper discusses the architecture and implementation of Joza, which seamlessly and synergistically incorporates both negative and positive taint inference methods. Using Joza and Wordpress as a testbed, the paper shows that the hybrid approach is extremely effective at thwarting SQL injection attacks on Web applications without requiring developer effort and does so with negligible performance overhead.

Acknowledgment

This research was supported by the Air Force Research Laboratory (AFRL) contracts FA8650-10-C-7025 and FA8750-13-2-0096, the U.S Department of Commerce (DOC) grant 01-79-14214, and the Commonwealth Research Commercialization Fund (CRCF) grant MF13-071-CS. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL, DOC, CRCF, or the U.S. Government.

References

[1] Anonymized-for-review. Taintless: Taint tracking and inference analysis and breaking tool. In *Black Hat USA*, 2014.

[2] Anonymized-for-review. WP-SQLI-LAB: Wordpress SQL injection lab, February 2014.

[3] A. Axelsen. Wordpress advanced search widget.

[4] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. Venkatakrishnan. Candid: preventing SQL injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 12–24. ACM, 2007.

[5] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *Applied Cryptography and Network Security*, pages 292–302. Springer, 2004.

[6] M. Chartier. Wordpress wp-advanced-search plugin.

[7] E. Chin and D. Wagner. Efficient character-level taint tracking for java. In *Proceedings of the 2009 ACM Workshop on Secure Web Services*, SWS '09, pages 3–12, New York, NY, USA, 2009. ACM.

[8] E. DB. oscommerce 2.3.3.4 (geo_zones.php, zid param) sql injection vulnerability.

[9] A. Futoransky, E. Gutesman, and A. Waissbein. A dynamic technique for enhancing the security and privacy of web applications. *Proc. Black Hat USA*, 2007.

[10] B. D. A. Guimaraes and M. Stampar. SQLmap, Februry 2014.

[11] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, 2005.

[12] W. G. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185. ACM, 2006.

[13] W. G. Halfond, A. Orso, and P. Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. *Software Engineering, IEEE Transactions on*, 34(1):65–81, 2008.

[14] X. Hui. PECL PHP taint tracker.

[15] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.

[16] B. Livshits. Dynamic taint tracking in managed runtimes. *Microsoft Research Technical Report*, 2012.

[17] B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey. 2011 CWE/SANS top 25 most dangerous software errors. *Common Weakness Enumeration*, 7515, 2011.

[18] MITRE. Cve-2013-1453.

[19] MITRE. Cve-2014-3704.

[20] MITRE. Cve details (SQL injection).

[21] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. *Automatically hardening web applications using precise tainting*. Springer, 2005.

[22] A. Nguyen-Tuong, J. D. Hiser, M. Co, J. W. Davidson, and J. C. Knight. To B or not to B: Blessing OS commands with software DNA shotgun sequencing. In *10th European Dependable Computing Conference*, 2014.

[23] I. Papagiannis, M. Migliavacca, and P. Pietzuch. PHP Aspis: using partial taint tracking to protect against injection attacks. In *2nd USENIX Conference on Web Application Development*, page 13, 2011.

[24] Perl taint mode.

[25] Ruby taint feature.

[26] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, pages 124–145. Springer, 2006.

[27] D. Ray and J. Ligatti. Defining code-injection attacks. In *ACM SIGPLAN Notices*, volume 47, pages 179–190. ACM, 2012.

[28] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.

[29] S. Son, K. S. McKinley, and V. Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1181–1192. ACM, 2013.

[30] TC.K. Wordpress advance wp query search filter plugin.

[31] TC.K. Wordpress ultimate wp query search filter plugin.

[32] W. Techs. Historical trends in the usage of content management systems for websites, February 2014.

[33] E. Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1):100–118, 1985.

[34] A. van der Stock, J. Williams, and D. Wichers. Top 10 2007. Technical report, OWASP, 2007.

[35] Verizon. The 2013 data breach investigations report. Technical report, Verizon, 2013.

[36] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM Sigplan Notices*, volume 42, pages 32–41. ACM, 2007.
[37] E. Weatherwax. *Modeling Secretless Security in N-variant Systems*. PhD thesis, University of Virginia, 2009.
[38] D. Wichers. Top 10 2013. Technical report, OWASP, 2013.
[39] J. Williams and D. Wichers. Top 10 2010. Technical report, OWASP, 2010.
[40] Wordpress.com. Wordpress.com posts statistics, 2014.
[41] Wordpress.com. Wordpress.com traffic statistics, February 2014.
[42] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, pages 121–136, 2006.