Towards a SQL Injection Vulnerability Detector Based on Session Types

António Silvestre[a] , Ibéria Medeiros[b] and Andreia Mordido[c]
LASIGE, Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa, Portugal
asilvestre@lasige.di.fc.ul.pt, ivmedeiros@ciencias.ulisboa.pt, afmordido@ciencias.ulisboa.pt

Keywords: SQL injection Vulnerabilities, Session Types, Type Checking, Static Analysis, Software Security

Abstract: Vulnerabilities in web applications pose a risk for organisations. Among them, SQL injections (SQLi) give the attacker access to private data by submitting malicious SQL queries to the database via invalidated entry points. Although there are various techniques for detecting SQLi, static analysis is widely used as it inspects the application code without executing it. However, static analysis tools are not always precise. In this work, we explore an avenue that links the detection of SQLi to type checking, thus providing stronger guarantees of their existence. We propose a novel approach which consists of interpreting the *behaviour* of a web application as if it was a communication protocol and uses *session types* to specify this behaviour. We leverage FreeST, a functional programming language for session types, to implement FREESQLI, a seminal detector of SQLi in PHP web applications. The tool translates the PHP code into FreeST code and capitalizes on FreeST's type checker to verify protocol adherence and detect inconsistencies associated with the presence of SQLi.

1 INTRODUCTION

Despite efforts to prevent vulnerabilities in web applications, the number of detected vulnerabilities has increased yearly and continues to be a security threat for organizations. SQL injections (SQLi) are among the top three vulnerabilities listed in the OWASP Top 10 Web Application Security Risks[1]. SQLi attacks occur when a system fails to sanitize or validate user inputs correctly, which allows malicious SQL queries to be submitted to the database. According to CVEDetails[2], the number of reported SQLi has increased from 740 in 2021 to 1790 in 2022 and is still rising. This data highlights the importance (and growing prevalence) of this vulnerability on web applications, despite being a well-researched topic.

Over the years, some programming languages have developed mechanisms to prevent these attacks, but they are still susceptible to human error (Halfond et al., 2006). For this reason, there has been an effort to develop tools to automatically detect vulnerabilities by inspecting the application code and assisting developers in identifying exploitable code fragments. Yet, these tools can produce many false positives, leading to developers spending time correcting nonexistent vulnerabilities.

PHP is the most widely used language on the server-side of web applications, accounting for roughly 76.5%[3]. It is a dynamically typed language since variable data types are determined in runtime, and the programmer does not need to specify them (Achour et al., 2023). As PHP does not require type specifications, it can pose security risks to applications if the intended types do not match. The absence of strict type enforcement makes data validation from external sources (e.g., user inputs) more difficult. For instance, if a numeric field is not correctly validated, an attacker can inject a handcrafted string containing suspicious characters, leading to security flaws within the application.

Unlike PHP, strongly typed languages require the programmer to specify types that govern the program specification explicitly and throw an exception or an error at *compile time* (i.e., *before* execution) whenever the program does not meet the specified type. Following the numeric field example, in these languages, a type mismatch is identified when the field receives a string. Although type systems alone can not eliminate the need for content validation, (e.g., user inputs), they tend to lead to fewer fault errors than dynamic languages.

[a] https://orcid.org/0000-0003-4227-2752
[b] https://orcid.org/0000-0003-4478-8680
[c] https://orcid.org/0000-0002-1547-0692

[1] https://owasp.org/Top10/

[2] https://www.cvedetails.com/

[3] https://w3techs.com/technologies/overview/programming_language

In this work, we resort to more expressive types – the so-called *session types* – which extend the traditional notion of *data types* and enable the specification of communication protocols between different parties by specifying types of data exchange and their directions (Vasconcelos, 2012; Thiemann and Vasconcelos, 2016). Session type systems provide additional guarantees of protocol adherence, which is paramount for our approach to the detection of SQLi: using session types we can, for instance, specify messages exchanged between the web application and the database and detect if the latter is receiving unsanitised queries from the former, which indicates the presence of a vulnerability for SQLi.

In this paper, we propose an approach to detect SQLi vulnerabilities in PHP web applications through a novel static analysis technique based on session types to define the nature of interactions between servers, clients, and databases. The approach leverages a strongly typed language called FreeST (Almeida et al., 2019), which has a robust type system based on session types. More specifically, the approach converts PHP code into FreeST code, which must include the translation of the program and the inference of the expected type. Translation includes an analysis of the application's behaviour (over its interaction with the database and the user) and the specification of a communication protocol (read, session type) that specifies this behaviour. Once the code has been translated, the FreeST compiler identifies inconsistencies between the expected types and the respective values. These inconsistencies allow us to identify vulnerabilities for SQLi, which will then be appropriately collected and reported to the developer. So far we have implemented a proof of concept of our approach in the FREESQLI prototype.

The main contributions of this paper are: (1) A novel static analysis approach that uses session types to detect vulnerabilities for SQL injections; (2) The FREESQLI tool that identifies SQLi vulnerabilities in PHP code, using the FreeST language.

2 BACKGROUND

Our work lives at the intersection of vulnerability detection for SQLi and programming languages. So, in this section we introduce some background related to SQLi and detection mechanisms, but also related to programming languages, session types and the language that underpins our work – FreeST.

SQL Injections. An SQLi is a security vulnerability that occurs when an attacker injects SQL commands and meta-characters into the application's unsanitised entry points, such as user forms, cookies and server variables (Halfond et al., 2006). These meta-characters when inserted in an existing SQL query, they can change its structure so that it is interpreted as a different SQL query, allowing the attacker to bypass authentication, access the database, view, change, or delete sensitive data, or even take control of administrative functions. One example of a vulnerable PHP code to SQLi can be seen in Listing 1.

```
1 $u = $_GET['username'];
2 $p = $_GET['password'];
3 $qry = "SELECT * FROM users WHERE uname='$u' AND
       passwd='$p'";
4 $r = mysqli_query($conn, $qry);
```

Listing 1: Example of PHP code vulnerable to SQLi.

This example verifies if a user belongs to the database after he provides their credentials on the login page and through two entry points (lines 1 and 2). Both entry points are injectable and do not go through sanitisation. An attacker can access all users' information by simply providing the string `' OR 1=1;--` on the `username` field. This input alters the final query to:

```
SELECT * FROM users WHERE uname='' OR 1=1;-- '
       AND passwd=''
```

The use of `--` meta-characters serves as a comment delimiter, which prevents the remaining content from being treated as part of the query. Consequently, the interpreted query would be:

```
SELECT * FROM users WHERE uname='' OR 1=1;
```

The tautology `'' OR 1=1` in the `WHERE` clause makes the query always true, which is logically equivalent to the query `SELECT * FROM users;`. Next, this query is sent to the database through the *sensitive sink* `mysqli_query` to be executed. In this context, a sensitive sink is a function that when executed with abnormal data (like injected SQL code) can return unexpected values.

According to Halfond et al. (Halfond et al., 2006), different types of SQLi have distinct goals and characteristics. However, they occur essentially because of unsanitised input fields, which made languages adopt mechanisms to prevent them. PHP was endowed with input sanitisation functions and prepared statements to mitigate the lack of validation on entry points (Achour et al., 2023). Nonetheless, since these mechanisms are applied by the developer, they are prone to human error. For this reason, there has also been an effort to develop tools to detect or prevent this kind of vulnerability (Gould et al., 2004; Halfond and Orso, 2005; Medeiros et al., 2016b; Medeiros et al., 2016a).

Static Analysis Tools. Although programming languages provide defensive mechanisms to prevent vulnerabilities, such as the sanitization functions (e.g., `mysqli_real_escape_string` in PHP), they can not guarantee that the software produced is secure since human errors remain inevitable. Supplementary tools are, therefore, essential for validating the code produced and supporting developers throughout the development process.

Static analysis is a strategy that involves analysing the application source, intermediate or binary code without executing it to detect errors and vulnerabilities. There is a wide variety of static analysis tools, but usually, they follow a similar approach. They start by taking the code and building a model that represents it. This model is created based on different approaches, often inspired by compiler strategies. Once the model is complete, they analyse it based on the knowledge and rules they have of what they are programmed to and generate a report containing the results (Chess and West, 2007). However, as these tools depend on programmed knowledge, they may produce false positives (FP) by detecting nonexistent vulnerabilities and occasionally missing actual vulnerabilities, resulting in false negatives (FN). In security, we know that FP and FN always have a negative effect on the programmer's confidence. It is thus crucial to minimise both and maximise true positives when developing a static analyser.

Types and Type Systems. Modern software engineering recognises a wide range of formal methods to ensure the desired and correct behaviour of a system. Of all formal methods, type systems are best established and prevalent (Pierce, 2002). Type systems, often associated with programming languages, try to prevent errors from occurring during the execution of a program by detecting inconsistencies between types and values at compile time. When designed, they should be easily verifiable and transparent, which means they must be able to verify if a program is typed correctly, and their behaviour should be predictable. Type systems enable the establishment of the relationship between the program and the types used to annotate them (Cardelli, 1996).

Programming languages can be categorised based on their type systems. PHP is a weakly and dynamically typed language. This means that while each variable has a specific type, the programmer does not need to declare their type because it is inferred at runtime. Due to this dynamic nature and the lack of strict type enforcement, PHP becomes more susceptible to potential vulnerabilities, like attackers injecting malicious code in a string to be used in a numeric field.

As previously stated, types function as annotations within computer programs. While different programming languages have unique types, certain types, such as integers, strings, booleans, and others, are commonly used across multiple languages. Some languages can feature more complex types like arrays, objects, or session types, which are especially pertinent to this work.

Session Types and FreeST. Session types enable the specification of heterogenously typed channels, defining communication protocols between two parties (Vasconcelos, 2012). Session types can be seen as sequences of input and output operations defining the messages being exchanged: $!T$ represents the sending of a value of type T, whereas $?T$ represents the reception of a value of type T. Session types can also offer some flexibility in the form of internal or external choices: $+\{\ell : T_\ell\}_{\ell \in L}$ represents the internal choice of a label ℓ, followed by the execution according to T_ℓ, whereas $\&\{\ell : T_\ell\}_{\ell \in L}$ offers a set of choices labelled with $\ell \in L$ and with *continuation* T_ℓ. This structure of sequential operations makes them ideal for modelling protocols in distributed scenarios (Dardha et al., 2012).

FreeST is an experimental concurrent programming language[4] that uses a core of linear functional programming and enables the creation of channels for inter-thread communication. The most notable feature of FreeST is its robust type system founded on context-free session types that regulate communication on channels (Almeida et al., 2019). *Context-free* session types (Thiemann and Vasconcelos, 2016) enable the specification of communication protocols whose traces are defined by context-free languages, while *regular* session types only enable the specification of tail recursive protocols. The detail "context-free" *vs* "regular" is not important in our setting, so we will use FreeST for its implementation of *session types* (although we could choose any other language that implements session types[5][6]).

Table 1 presents the different session types and primitives within the FreeST language. These types allow the definition of communication protocols between threads, with each type being linked to a specific primitive. FreeST features several communication primitives: `send` and `receive` are used for transmitting and receiving values, `select` is used to make choices within the channel, and `match` is used to align with the selected option. The `close` primitive is used for channel termination, while the `wait` primitive is

[4] https://freest-lang.github.io/

[5] https://rss.rd.ciencias.ulisboa.pt/tools/sepi/

[6] https://docs.rs/session_types/latest/session_types/

Table 1: FreeST session type constructor and primitives.

Session Type Constructor	Behaviour	FreeST Primitive
!T	send value of type T	`send`
?T	receive value of type T	`receive`
+{l: T, ..}	select a choice	`select`
&{l: T, ..}	offer a set of choices	`match`
T ; U	do T, then U	-
Close	close the channel	`close`
Wait	wait for channel to be closed	`wait`
Skip	neutral element of ;	-

used to wait for channel termination. The `Skip` type is the neutral element of the sequential composition `;`.

For example, consider a scenario where a server thread is responsible for receiving two integers, checking if their sum exceeds ten and providing the result. The communication between the client and server threads can be read as: *send the first number, send the second number, and receive a boolean value*. This sequence can be translated into a FreeST protocol using the type constructors presented in Table 1, as shown below:

```
type protocol = !Int; !Int; ?Bool; Close
```

Type Checking. Type checking is one of the most (involuntarily) used static analysis techniques by developers. It involves checking and validating the types of values in a programming language and ensuring, at compile time, that the variables or expressions used in operations are compatible with their expected types. Although most programmers are familiar with type-checking, they may not pay much attention to it since the rules are already defined by the programming language and enforced by the compiler, transparently to the user. Regardless, type checking, like any other static analysis method, is not foolproof and can produce FP and FN. Programmers often overlook these imperfections, yet they can significantly impact vulnerability detection (Chess and West, 2007).

3 FREESQLI PROTOTYPE

We aim to develop a static analysis tool to detect SQLi in PHP code using session types offered by FreeST. To enable and guarantee the use and practicality of our tool, session types, FreeST and its compiler are used in a way that is completely transparent to the user, so that FreeST goes unnoticed.

We call our tool FREESQLI, and its approach operates in two distinct phases: the *translation* and the *vulnerability detection*, represented in Figure 1. The translation phase translates the PHP code into FreeST

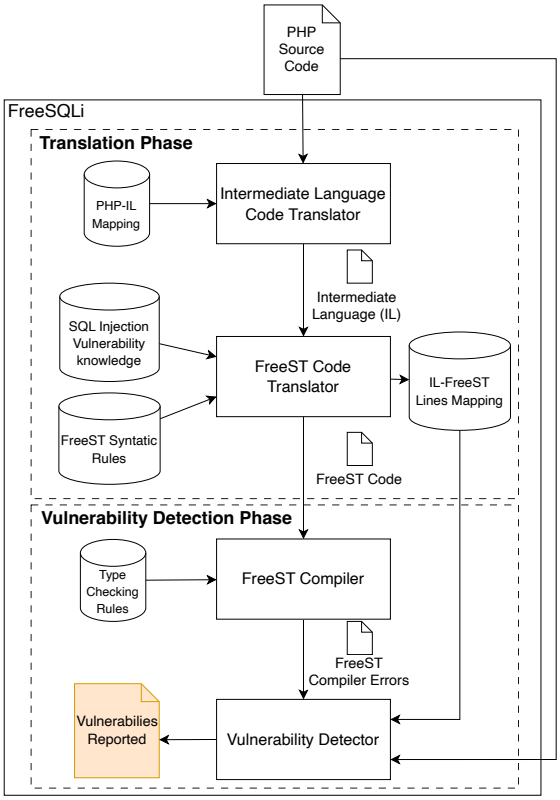

Figure 1: Architecture of the proposed solution.

code, passing through an intermediate language (IL). The conversion of PHP into IL enables us to remove irrelevant syntax from the PHP and simplify the translation to FreeST. The IL code is then transformed into FreeST code, including the program definition and the protocol specification (i.e. the type that should govern the program). This translation registers the mapping of the IL lines into FreeST lines to facilitate the generation of a comprehensive report in the next phase.

Once we have the FreeST code, we are able to compile it and proceed to the vulnerability detection phase. The FreeST compiler checks for protocol compliance and identifies type mismatches, generating a list of errors in case of any inconsistencies. These errors, the source code, and the corresponding mapping are then used to pinpoint the vulnerabilities in PHP code and develop a comprehensive report.

We now give a more detailed explanation of each phase, illustrated by a running example of how FREESQLI detects SQLi vulnerabilities.

Running Example. Consider the PHP code fragment in Listing 2, which consists of an SQL query where we limit the number of returned results to 20 and establish an offset, specified by the user. The code aims to divide a result set into pages, but it overlooks

the proper sanitation of the `$offset` variable. This oversight creates an opportunity for an attacker to inject malicious code, and since the query relies on this variable, the code becomes vulnerable to SQLi.

```
1 $offset = $_GET['offset'];
2 $q  = "SELECT id, name FROM products ORDER BY
      name LIMIT 20 OFFSET $offset;";
3 $r = mysqli_query($conn, $q);
```

Listing 2: Example of a PHP code vulnerable to SQLi.

Translation Phase. In this phase, we focus on converting the PHP source code into FreeST code. In PHP (as in other languages), some functions *behave* similarly. For example, the entry point functions, like `$_GET` and `$_POST`, invoke different HTTP request methods, but both consist of receiving user inputs. With this in mind and to simplify the translation process, we start this phase by converting source code into an IL. The IL allows us to remove unnecessary PHP code (e.g., comments) without oversimplifying the program, which could prevent us from detecting certain vulnerabilities.

The *Intermediate Language Code Translator*, shown in Figure 1, converts the PHP code into IL through a lexer. This translation assigns tokens to individual PHP functions based on their behaviour, using the *IL-PHP mapping*. Since we are translating to FreeST, we focus on the *behaviour* of the functions, and the IL already establishes the connection to the communication protocols (later specified through session types). Always taking the application's point of view, PHP entry points functions will be translated into the `receive` token, PHP functions that consist of sending data (queries) to the database (e.g., `mysql_query`) will be represented in IL by the `send` token. We also need tokens that allow us to represent sanitisation functions (`sanit_f`), type checking (`typechk`) and other functions that do not fall into the previous categories (e.g., `replace_str`). Table 2 presents an excerpt of the mapping between these tokens and the anticipated behaviour corresponding to each function. (Note that we are still working with a small fragment of the PHP language and that we will extend the translation to other primitives soon.)

Variable names are not altered by the translation and do not have a specific token since it could lead us to lose information. SQL queries are represented as the token `query` and its variables since we assume their safeness depends on the variables they use.

Coming back to our running example, the `$_GET` and `mysqli_query` functions are represented as the `receive` and `send` tokens, respectively, meaning that the protocol will receive data provided by `$_GET` and it will send data (to the database) through `mysqli_query` function. Therefore, line 1 of the code would be translated to `$offset receive`, while line 3 would become `$r send $conn $q`. The query on line 2 would also become `$q query $offset`. The complete IL conversion is shown in Listing 3.

```
1 $offset receive
2 $q query $offset
3 $r send $conn $q
```

Listing 3: Conversion of PHP code of Listing 2 to IL.

Once we obtain the IL code, the translation into FreeST occurs through the *FreeST Code Translator*, involving two essential steps: (1) specification of the (translated) FreeST program, and (2) inference of the expected protocol, by means of a session type.

The first step leverages the IL tokens, and the translator parses the lines of IL code into FreeST using predefined templates based on the *FreeST syntactic rules*. In the second step, the translator identifies and specifies protocol definitions. Here, we use session types to describe the protocol of the server's communication with the user and the database. When a server receives data from the user via entry points, he receives *unsafe* data since it comes from an untrusted source. On the other hand, when the server queries a database, it should send *safe* data to avoid possible attacks. For this purpose, we endow the FreeST language with two new types: `Safe` and `Unsafe`, similar to *tainted* and *untainted* variables used in the literature (Shankar et al., 2001; Dahse and Holz, 2014). These new types represent whether a variable can be tampered with. For example, user-input variables fall under the unsafe category, whereas data supplied to a sensitive sink should fall under the safe category. The sanitisation functions (e.g., `mysqli_real_escape_string`) modify the variable type from `Unsafe` to `Safe`. We chose the terms `Safe` and `Unsafe` over *tainted* and *untainted* for clarity and their potential use in FreeST features beyond identifying vulnerabilities. Considering this, we can observe in Table 3 the correlation between the IL tokens that involve communication and their expected behaviour as a FreeST session type. These correlations allow us to define the correct communication protocol based on the IL's tokens. For example, the IL `receive` token represents user input, categorised as `Unsafe` due to its untrusted source. This means that the FreeST session type corresponds to `?Unsafe` (i.e. *"receive unsafe"*), which means we will use the primitive `receive` when translating to FreeST code, as indicated in Table 1.

In our running example, each IL instruction of Listing 3 is transformed following a template, result-

Table 2: PHP-IL Mapping: IL tokens and respective PHP functions.

IL token	Description	Example of PHP primitives/functions
receive	entry point	$_GET, $_POST, $_COOKIE, $_REQUEST
send	sensitive sink	mysql_query, mysql_unbuffered_query, mysql_db_query, mysqli_query, mysqli_real_query
sanit_f	sanitization function	mysql_escape_string, mysql_real_escape_string
replace_str	replace string function	str_replace, str_ireplace, substr_replace, preg_replace, strtr,
typechk	type checking function	is_array, is_bool, is_callable, is_float, is_int, is_null, is_string
query	SQL query	-

Table 3: Correlation between IL and FreeST session types.

IL token	Description of expected behaviour	FreeST session type	FreeST primitive
receive	Receive unsafe input from a user	?Unsafe	`receive`
send	Make a safe query to the database	!Safe	`send`

ing in the FreeST code of Listing 4.

```
1 let (offset, protocol) = receive protocol in
2 let q = query offset in
3 let protocol = send q protocol in
```

Listing 4: Code in Listing 3 translated to a FreeST program.

```
1 type Protocol = ?Unsafe; !Safe; Close
2
3 server: Protocol -> ()
4 server protocol =
5     let (offset, protocol) = receive protocol in
6     let q = query offset in
7     let protocol = send q protocol in
8     close protocol
```

Listing 5: The resulting FreeST code of Listing 2.

It is important to note that in FreeST, the `receive` primitive returns a tuple with the received result and the continuation type (e.g., `(offset, protocol)` in Listing 4, line 1), while the `send` primitive only returns the continuation type. In the second step, we specify a communication protocol (i.e., a type). The type is defined based on specific communication tokens like `send` and `receive`. For instance, a `receive` token appears in line 1, followed by a `send` token in line 3. In this particular case, the communication involves receiving an `Unsafe` type, sending a `Safe` type, and closing the channel. Consequently, the type for this program would be:

```
type Protocol = ?Unsafe; !Safe; Close
```

Once the protocol is defined and the lines are converted based on templates, we add other necessary FreeST syntax to allow the program compilation. As the translated instructions are not in the same line as in the IL, we need to annotate the program with the type. So, to correctly report the vulnerable part of the code, we store a mapping between the IL and FreeST lines for future reference (*IL-FreeST Lines Mapping* in Figure 1). The final FreeST code for our running example is depicted in Listing 5.

Vulnerability Detection Phase. In this phase, we detect SQLi vulnerabilities by relying on the FreeST compiler to verify if the generated FreeST code follows the inferred protocol. Using its type checking rules, the FreeST compiler evaluates if the protocol is being followed by the (translated) FreeST program and generates a list of errors in case of type mismatches. These errors may occur when, for example, the server sends `Unsafe` data (to the database) using the primitive `send` while the protocol requires the server to do it with `Safe` values. Such discrepancies suggest the existence of SQLi vulnerabilities, indicating the presence of one or more unsanitised variables.

As part of our protocol adherence verification, we have modified the compiler to include types `Safe` and `Unsafe` and have developed a FreeST module that specifies the functions needed to properly infer the type of queries based on the types of their arguments. This module specifies, for instance, that a query with one argument with type `Unsafe`, has type `Unsafe`:

```
query : Unsafe -> Unsafe
```

However, if *all* arguments in a query are `Safe`, the query has type `Safe`. In the case were the query has two arguments, we have:

```
query : Safe -> Safe -> Safe
```

In short, if some variable in a query statement has type `Unsafe`, the query has type `Unsafe`. Sanitisation functions transform `Unsafe` values into `Safe` values:

```
sanitise : Unsafe -> Safe
```

Once the FreeST errors are collected and the mapping between IL and FreeST lines is identified, a report with all SQLi detected is generated. The report includes queries vulnerable to injections alongside the unsanitised variables used within these queries. Also, it specifies the exact line where these vulnerabilities occur for clear identification and resolution.

Coming back to our running example, we need to compile the resulting FreeST code in Listing 5. Following the program execution, we notice that the

`offset` variable is assigned type `Unsafe`; as `q` is defined from a query with an argument of type `Unsafe`, it also has type `Unsafe`. And so, in line 7 of the program, we send a value of type `Unsafe`, when we were expecting to send a value of type `Safe` (as indicated in the type in line 1). As a result, the compiler detects this error and generates the following message: *Could not match expected type Safe with actual type Unsafe for expression q*, providing details in which lines this error appears. This message indicates that we tried to send an `Unsafe` query to the database when it should be `Safe`. Because the query uses the `offset` variable, which is an entry point that was not sanitised, we are in the presence of an SQLi. Based on this compilation error, the vulnerability detector detects this vulnerability and generates a report to the user with the necessary information to correct it:

```
----------------------------------------
 SQL injections detected: 1

 Queries: "SELECT id , name FROM products
 ORDER BY name LIMIT 20 OFFSET $offset;"

 Vulnerable variables: $offset
----------------------------------------
```

4 USAGE EXAMPLE

Listing 6 contains a code example that conventional static analysis tools may wrongly identify it as vulnerable to SQLi. The input received from the user undergoes the `substr` function, which extracts the first four characters. This substring is not dangerous because it contains less than five characters (Medeiros et al., 2016a) and does not compromise the query, making this example a false positive (FP) for certain tools (e.g., (Dahse and Holz, 2014)).

```
1 $u = $_POST['user'];
2 $u1 = substr($u, 0, 4);
3 $q = "SELECT * from users WHERE user='".$u1."'";
4 $r = mysqli_query($con, $q);
```

Listing 6: PHP example without a SQLi vulnerability.

```
1 type Protocol = ?Unsafe; !Safe; Close
2
3 server: Protocol -> ()
4 server protocol =
5     let (u, protocol) = receive protocol in
6     let u1 = sanitise u1 in
7     let q = query u1 in
8     let protocol = send q protocol in
9     close protocol
```

Listing 7: The resulting FreeST code of Listing 6.

Our approach does not classify this code as an FP because the `substr` function is considered a sanitisation function if it extracts less than five characters. The translated FreeST code for this example is in Listing 7. The compiler does not detect any inconsistency because the `sanitise` function converts the `Unsafe` variable `u` in a `Safe` variable `u1`.

5 RELATED WORK

SQL injections are well-researched and documented threats, and over the years, various techniques and tools have been developed to prevent or detect them (Gould et al., 2004; Halfond and Orso, 2005; Medeiros et al., 2016b; Medeiros et al., 2016a). These auxiliary tools aim to help developers reduce the number of vulnerabilities or accelerate the code review process. However, their adoption in real-world applications can vary since developers tend to prioritize tools that seamlessly integrate into their systems and offer minimal false-positive rates (Oyetoyan et al., 2018). In this section, we provide an overview of several such techniques and tools.

JDBC-Checker (Gould et al., 2004) is a static code checker that statically verifies the accuracy of types in dynamically created SQL queries. It was one of the earliest static analysis tools, and even if its primary purpose is not to prevent general SQLi attacks, it can aid in blocking attacks that exploit type discrepancies within dynamically generated query strings. It can help address one of the main issues leading to SQLi vulnerabilities: the lack of proper validation of input types. Yet, it is worth noting that JDBC-Checker is ineffective against more general SQLi, which involve syntactically and type-correct queries.

Another tool that appeared at an early stage was AMNESIA (Halfond and Orso, 2005). It uses a model that combines static analysis with runtime monitoring to enhance security against SQLi attacks. The process involves a static phase that constructs models of legitimate query types for each database access point using static analysis. Subsequently, in the dynamic phase, queries undergo interception pre-database execution and verification against the built models. Queries conflicting with these models are flagged as SQLi and prevented from execution. The effectiveness of this tool is tied to the accuracy of the built query models and can influence both FP and FN.

More recently, some tools have focused on detecting various vulnerabilities, not only SQLi, by leveraging some machine learning techniques. One of these tools is WAP (Medeiros et al., 2016b). The tool uses taint analysis to detect vulnerabilities and machine

learning to ensure that identified vulnerabilities are genuine and not FP. The tool breaks down the code into smaller parts using a Lexer and organises it into an abstract syntax tree (AST). During the taint analysis process, WAP looks for vulnerable entry points and verifies if they compromise any sensitive sink, considering possible entry point sanitisation throughout the code. The tool corrects the code automatically.

Unlike WAP, DEKANT (Medeiros et al., 2016a) differs in how it detects vulnerabilities in PHP code. Instead of using an AST, it employs taint analysis and machine learning, breaking the code into tokens and converting them into an intermediate language (similar to our approach). DEKANT learns to spot vulnerabilities by observing Hidden Markov Model (HMM) sequence models, enabling it to autonomously detect flaws without relying on explicit detection methods.

The main drawback of employing machine learning on DEKANT and WAP is their reliance on the quality of their training datasets. If a poor set is used, it can lead to large numbers of FP and FN. As a result, they cannot offer guarantees regarding their prediction or detection abilities, something we want to reach with our approach.

6 CONCLUSION

In this paper, we propose a novel approach that analyses PHP code's *behaviour* to identify vulnerabilities for SQLi. Looking at the program from the application's point of view, we use session types to identify communication protocols between the application, the user, and the database. Our approach uses the FreeST programming language in the backend and capitalises on its compiler and expressive types to identify type mismatches and infer the existence of vulnerabilities for SQL injections. Also, we present FREESQLI, a proof of concept of our approach.

In the near future, we will extend the fragment of PHP that we are considering to include the `if` primitive and we will prove results of correctness of our translations. In addition, we will adapt our FreeST implementation so that `Safe` is considered a subtype of `Unsafe`, which will enable us to identify second-order SQLi. We will also evaluate the performance of our tool. To do this, we will use NIST SARD[7], which offers small synthetic applications. The test suite will include different scenarios, such as instances where traditional static analyzers produce false positives and negatives and patches for vulnerable cases.

[7]https://samate.nist.gov/SARD/

ACKNOWLEDGMENTS. This work was supported by FCT through the project SafeSessions PTDC/CCI-COM/6453/2020 (http://doi.org/10.54499/PTDC/CCI-COM/6453/2020) and LASIGE Research Unit UIDB/00408/2020 (https://doi.org/10.54499/UIDB/00408/2020) and UIDP/00408/2020 (https://doi.org/10.54499/UIDP/00408/2020)

REFERENCES

Achour, M. et al. (2023). PHP Manual. https://www.php.net/manual/en/.

Almeida, B., Mordido, A., and Vasconcelos, V. T. (2019). FreeST: Context-free session types in a functional language. *arXiv preprint arXiv:1904.01284*.

Cardelli, L. (1996). Type systems. *ACM Computing Surveys (CSUR)*, 28(1):263–264.

Chess, B. and West, J. (2007). *Secure programming with static analysis*. Pearson Education.

Dahse, J. and Holz, T. (2014). Simulation of Built-in PHP Features for Precise Static Code Analysis. In *NDSS*, volume 14, pages 23–26.

Dardha, O., Giachino, E., and Sangiorgi, D. (2012). Session types revisited. In *PPDP*, pages 139–150.

Gould, C., Su, Z., and Devanbu, P. (2004). JDBC checker: a static analysis tool for SQL/JDBC applications. In *Proceedings. 26th ICSE*, pages 697–698.

Halfond, W. G. J. and Orso, A. (2005). AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *IEEE/ACM ASE*, page 174–183.

Halfond, W. G. J., Viegas, J., and Orso, A. (2006). A Classification of SQL-Injection Attacks and Countermeasures. In *IEEE ISSSE*.

Medeiros, I., Neves, N., and Correia, M. (2016a). DEKANT: A Static Analysis Tool That Learns to Detect Web Application Vulnerabilities. In *Proceedings of the 25th ISSTA*, page 1–11.

Medeiros, I., Neves, N., and Correia, M. (2016b). Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE TR*.

Oyetoyan, T. D., Milosheska, B., Grini, M., and Soares Cruzes, D. (2018). Myths and facts about static application security testing tools: an action research at Telenor digital. In *XP 2018*, pages 86–103.

Pierce, B. C. (2002). *Types and programming languages*. MIT press.

Shankar, U., Talwar, K., Foster, J. S., and Wagner, D. (2001). Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*.

Thiemann, P. and Vasconcelos, V. T. (2016). Context-free session types. In *ICFP*, pages 462–475. ACM.

Vasconcelos, V. T. (2012). Fundamentals of session types. *Information and Computation*, 217:52–70.