Virtual Static Security Analyzer for Web Applications

Mihail Brinza
Instituto Superior Técnico
Lisbon, Portugal
mihail.brinza@tecnico.ulisboa.pt

Miguel Correia
Instituto Superior Técnico
Lisbon, Portugal
miguel.p.correia@tecnico.ulisboa.pt

João Pereira
Instituto Superior Técnico
Lisbon, Portugal
joao.d.pereira@tecnico.ulisboa.pt

***Abstract*—In the past decades web applications have been popular victims of injection attacks such as SQL injection or cross-site scripting. In order to prevent these attacks, we need automatic vulnerability detection tools. However, modern existing tools are complex, consist of thousands of lines of code, and are often bound to a single language. This complexity makes them hard to understand and to port to a new language.**

To reduce the complexity of current static analyzers, we propose a new solution that supports the addition of new languages without much effort. In order to achieve this, our solution does not analyze the source code AST directly, instead, it traverses the source code AST and builds a generic AST (GAST) from it. Then, we analyze the GAST to find vulnerabilities. This way we can decouple the analysis and the source code parsing. Furthermore, GAST is just an abstraction that only represents what is needed to perform taint analysis, ignoring the rest. To add support for a new language we just need to generate a parser using ANTLR4 [1] and write a converter for that AST, which is usually less than 110 lines of code.

The approach was implemented in the GT tool, which supports Java, PHP, Python and JavaScript, and was tested against several web applications written in the same languages.

***Index Terms*—Security, Static Analysis, Taint Analysis, Antlr4, Information Flow**

I. Introduction

In the past couple of decades, research on web security and bug finding tools have seen a big increase. This is due to the fact that software vulnerabilities can have devastating effects on companies and/or its clients [2]. In 2017, hackers have compromised the sensitive information of 145 million American customers from Equifax, one of the three major consumer credit reporting agencies in the U.S.A, leading to hundreds of millions of dollars of loss to the company [3].

Web applications are popular victims of security attacks since they accept user input, which can be malicious, and incorporate it into dynamically generated code. For example, a user may have to fill a form, post a comment, or submit a username and a password for authentication. The application then takes this user-provided input and inserts it into a dynamically generated program in another language (e.g., a new client-side script, or an SQL or JavaScript query to a back-end database). If the user input reaches these scripts/queries without first being validated and sanitized, then there is probably a vulnerability.

Code injection attacks such as SQL injection or cross-site scripting were considered the top security problem in 2017 by OWASP [4]. These attacks occur when a malicious user manages to inject his code into dynamically generated scripts/queries, usually by adding meta-characters to the input. By doing this, an attacker could change the behavior of the application, steal data, compromise database integrity and/or bypass authentication and access control, violating system correctness, security, and privacy properties.

Injection vulnerabilities are caused mainly by poor user input sanitization, the use of languages where it is easy to write insecure code (e.g., PHP, C) and programmers that do not have much knowledge about software security [5].

In order to detect and prevent injection attacks, we need automatic detection mechanisms. While researchers have tried many approaches over the past decades, (e.g., static and dynamic taint analysis, symbolic execution, etc.) the dominant trend is towards increasingly complex tools. However, the more complex a tool is, the worse it scales, the harder it is to maintain and understand, and the more assumptions it makes, limiting the programs it can analyze.

One big problem of the increasing complexity in vulnerability detection tools is that most of the times they are not *portable*[1]. Although most of the programming languages we use nowadays to build web applications have a lot of similarities between them, vulnerability detection tools still seem to struggle when it comes to supporting more than one language. Many of them are wedded to a specific language [6]–[13], a specific compiled code [14]–[16] (e.g., x86 binary, bytecode) or they depend on modified runtime engines [6], [17]. To port one of these tools to another language basically requires to implement it again from scratch.

[1]In this paper we use the term *portable* to refer to the approaches that either support different languages or to which it is easy to add support for a new one.

Since there is a wide range of languages available to build web applications, the lack of portability of detection tools can be considered a problem and a limitation. However, there are some approaches that solve this issue to some degree which we discuss in section V.

In this paper we present a new static taint analysis approach, implemented in the *Generic Taint analyzer* (GT) tool, that aims to solve the problem of portability while being context-sensitive and keeping low rates of false positives and negatives. Our solution is not bound to a specific language and can be extended to support a new language with a relatively small amount of work and lines of code. Traditional static taint analyzers parse the code and then analyze the resulting abstract syntax tree (AST). The nodes that the AST consists of are specific to the parsed language, making the tools bound to that language. However, most of the languages we use nowadays to build web applications are similar. Languages usually consist of classes, attributes, functions or methods, statements, expressions, etc... Even between languages such as PHP and Java, that are apparently very different, we find many of these structural similarities. Based on this fact, our approach converts the source code AST to a simple, generic abstract syntax tree (GAST) that can represent a large set of languages used in web applications. More importantly, the GAST does not represent all the details of a language. Instead, it only has the aspects that are actually used in the analysis. This way, we use the same code to find vulnerabilities, regardless of the language being analyzed. The GAST allows us to keep the source code parsing and the analysis decoupled.

Due to the GAST, the analysis is divided into three main steps, where each one uses a different module (*parser, converter, analyzer*):

1) Use a *parser* specific to the language to get the source code AST
2) Convert the source code AST to a GAST using a specific *converter*
3) Run an *analyzer* on the GAST to find vulnerabilities. In our case, the *analyzer* is a static taint analyzer

The only modules that are bound to each language are the *parser* and the *converter* (which converts the source code AST to the GAST). Since parsing several different languages is a complex problem, we use ANTLR4 (ANother Tool for Language Recognition) to generate the *parser*, parse the source code and build the AST. ANTLR4 is a parser generator with a big community that provides grammars under the MIT license for virtually any language. This way, when adding support for a new language we only need to generate a new *parser* with ANTLR4 and program a new *converter*, which in our opinion requires a small amount of work. While implementing the GT tool, we started by adding support for PHP and Java. Later, we extended the tool to support JavaScript and Python. Converters for Java, PHP, Python and JavaScript are all less than 110 lines of code each.

The main contributions of this paper are: (1) an approach for improving portability of static security analyzers by converting the source code AST to a GAST; (2) a tool that implements this approach written in Java for applications written in Java, PHP, Python and JavaScript.

II. Input Validation Vulnerabilities

This section briefly presents a list of vulnerabilities considered in this work. We can divide them in four categories [18] - *query manipulation, client-side injection, file and path injection* and *command injection*. The main problem of all these vulnerabilities lies in the improper validation of user input. Our work focuses on this kind of vulnerability.

A. Query Manipulation

These vulnerabilities are associated with the construction of queries or filters that are executed by some other engine (e.g., a database management system). If the query is constructed with unsanitized inputs, then it is possible to modify the normal behavior.

All vulnerabilities in this category can be prevented by sanitizing user input, so it does not contain metacharacters that can alter the behavior of the engine:

1) SQL Injection: vulnerability caused by the use of string building functions to create SQL queries. An attack consists of mixing normal characters with metacharacters. In the example of listing 1, a malicious user can provide a username `"admin' --"` causing the query to execute without the need of a password.

```
1 $name = $_GET['username'];
2 $pass = $_GET['password'];
3 $query = "SELECT * FROM users
4     WHERE name='$name' AND password='$pass'";
5 $result = mysql_query($query);
```

Listing 1. PHP code vulnerable to SQL Injection

2) NoSQL Injection: Similarly to relational databases, non-relational are also vulnerable to injection attacks.

3) XPath Injection: This vulnerability is very similar to SQL Injection, but in this case, data is injected in XML documents, which are often used to store data or configurations. To prevent this vulnerability, it is enough to check if the input contains malicious characters.

4) LDAP Injection: (Lightweight Directory Access Protocol) Injection is also exploited by providing metacharacters to string-building functions. LDAP Injection attacks aim to modify the structure of the filter and retrieve data from a directory.

B. Client-Side Injection

The vulnerabilities in this category allow an attacker to execute malicious code in the victim's browser. This kind of attack is not against the application itself but against the user and can be prevented by either sanitizing or encoding the input.

1) Cross-site scripting (XSS): There are three types of XSS attacks: reflected or non-persistent, stored or persistent and DOM-based. In this paper we only consider the first two types.

A program vulnerable to reflected XSS can have a single line, *"echo $_GET['user'];"* The attack consists of convincing the user to click on a link to the web application with a malicious script which will be reflected by the *echo* instruction. (e.g., *www.a.pt?user=<script>*malicious code*</script>*).

A stored XSS consists of two steps: first, the attacker inserts a malicious script in the server, then later, the server returns that script to one or more users.

2) Header injection: allows an attacker to break the HTTP response with *"\n"* and *"\r"*. This allows the attacker to inject malicious code in headers or even a new HTTP response. It can be avoided by sanitizing these characters.

3) Email injection: Very similar to Header Injection, it has the goal to manipulate email components (e.g., sender, destination, message) by injecting the line termination character. In this case, sanitizing the input solves the problem as well.

C. File and Path Injection

This category consists of vulnerabilities where the user can control the path or the URL of the file included. Thus allowing him to inject malicious files (*Remote file inclusion*) or read arbitrary files from the server (*Directory/Path traversal*).

D. OS Command injection

This category consists of vulnerabilities where the user can control the input to functions that execute OS commands (e.g., *shell_exec* in PHP). Allowing the attacker to execute arbitrary code in the server.

III. The GT tool

In this section we explain in-depth our approach at solving the complex problem of developing a portable static taint analyzer. Our solution, besides being portable, also aims to be able to perform a context-aware analysis, being as sound and complete as possible.

As stated before, static analyzers are most often bound to a single language and depend on every detail of that language. However, there are a lot of similarities between programming languages, and our approach explores just that. If we take a look at the top trending languages in 2020, ranked by IEEE Spectrum [19], used to build web applications, we can divide them in two categories:

1) Dynamically typed - Languages that do not check or enforce type-safety at compile-time [20]. Instead, type-checking is done during runtime (e.g., Python, JavaScript, PHP and Ruby).
2) Statically typed - Type checking is done at compile-time. (e.g., Java, C#, GO and Dart).

The languages from each category have many similarities between them. Take for instance JavaScript and PHP, which are very common on web applications. Both of them are object-oriented, have methods, functions, attributes, variables, expressions, etc... Even the *control flow statements* are practically the same (e.g., *if, switch, while, for, do while* etc.). Furthermore, the data flow is almost identical, using assignments.

Now, if we compare Java and C# the same is true. Moreover, even comparing languages between categories (e.g., Java and PHP) we see that many of their features overlap. The main difference is that we know the types when analyzing the source code. So from our studies, we found out that when it comes to static taint analysis we can abstract almost any web programming language.

With this in mind, our approach introduces a new way of building static analyzers, which consists of having a simple, generic AST (GAST) that can represent the structure of the source code of a large set of languages found in web applications. The GAST, similarly to the micro-grammars approach [21], does not depend on every detail of the languages. Instead, it only represents what is absolutely needed to perform a data flow analysis (in our case, static taint analysis). Section III-B explains the abstractions that were made to build the GAST.

With the addition of the GAST, our analysis gains an extra step comparing to traditional tools. Usually, analyzing a program consists of first parsing the source code and then traversing the resulting AST to find vulnerabilities. By contrast, our approach adds a new, additional step, which consists of converting the source code AST to a GAST. Then, we perform the taint analysis on the GAST to detect vulnerabilities in the code. This way, the module that performs the taint analysis is completely independent of the language being analyzed.

Next, we describe the architecture and data flow of GT. Then, we present the structure of the GAST and how to build it. Finally, we discuss the features of our taint analysis and the compromises and choices that were made.

A. Architecture and Data Flow

In order to implement a tool that can support several languages simultaneously, we need a decoupled and modular architecture. The architecture of GT, represented in figure 1, consists mainly of four modules: *Parser*, *AST Converter*, *GAST Builder* and *Taint Analyzer*.

1) Parser: takes as input the source code and produces a source code AST, specific to the language. This module is language-dependent, meaning that every time we add a new language, we need a new parser for that language. Since parsing a full-blown language is a complex task, we delegate it to ANTLR4, which is a widely used parser generator that uses *LL* for parsing [1]. ANTRL4 has a big community and provides grammars under the MIT license for virtually any language. This way, thanks

to ANTRL4, our parsers consist simply of generated code. Furthermore, ANTRL4 also generates tree walkers, which we use to traverse the AST.

2) *AST Converter:* traverses the source code AST using the generated tree walker and raises events to the *GAST Builder* (e.g., entering and exiting class, methods or functions declarations). These events allow the *GAST Builder* to create the GAST. This module is the only one needed to program whenever adding support for a new language.

3) *GAST Builder:* reacts to the events from the *AST Converter* and internally builds the GAST. Section III-C shows more in-depth the process of building the GAST.

4) *Taint Analyzer:* module that takes as input the GAST from the *GAST Builder* and a configuration file where we can specify the value of several parameters, such as:

- Sensitive functions - functions that when called with *tainted* arguments may result in a vulnerability (e.g., *mysql_query* in PHP)
- Sanitization functions - functions that perform input sanitization and whose return value is always *untainted* (e.g., *mysqli_real_escape_string* in PHP)
- Entry points - the function which is invoked by the server when the user makes a request. We also specify which arguments of the entry point function are *tainted*

After traversing the GAST, GT produces a vulnerability report.

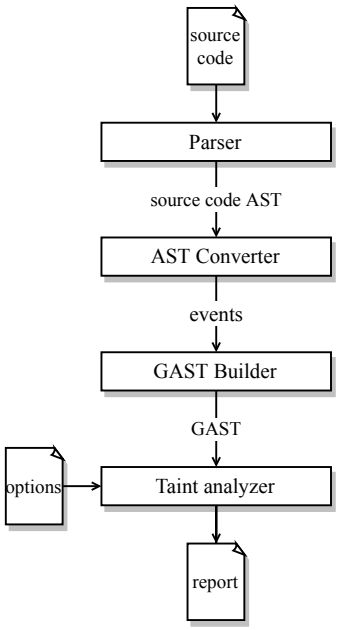

Fig. 1. GT - data flow and architecture overview

B. GAST

Similarly to the micro-grammars solution [21], the GAST aims to be an intermediate structure that does not depend on every detail of any language. Instead, it abstracts most of the complexity by using generic statements. Each generic statement has a correspondence with a concrete statement of almost any programming language. In this section we describe the structure of the GAST by presenting every element that can be part of it.

1) *Constant:* comprises *strings*, *integers*, *floats*, *null* values etc. Basically, everything that is directly hardcoded in the source code is considered a constant. Since this element never changes, it is impossible to be *tainted*.

2) *Variable:* as the name says, it represents the variables in the code. Contains the name of the variable, and if the language is statically typed, it also contains the type of the variable. It can become *tainted* through assignments.

3) *Attribute:* represents an attribute in a class. It has a name and a type. Also, it can be *tainted*.

4) *Parameter:* represents a function or a method parameter. It can be *tainted* if it is the parameter of the entry point function, or if the argument passed to a function is *tainted*.

5) *Expression:* represents a generic abstraction of any expression (e.g., arithmetic, logical, bitwise, comparison etc.). It consists of a list of expressions. Take for instance the listing 2 where to each variable, *a, b, c* we assign an expression. In our representation, all these expressions are equal. They all consist of an expression containing two members: a variable named *"x"* and a constant with value *"5"*. This allows us to abstract any operator due to the fact that in data flow analysis, operators (except the assignment operators) do not influence *taint* propagation. For instance, if the variable *x* in listing 2 is *tainted*, then, all variables will be *tainted*, regardless of their operator.

Since an expression consists only of a list of other expressions, the *taint* propagation is a result of a logical OR of all elements in the expression. Meaning that if any element in the list is *tainted*, the whole expression is marked as *tainted*.

```
1 boolean a = x == 5;
2 int b = x + 5;
3 int c = x * 5;
```

Listing 2. Expression assignment examples

6) *Function Call:* this is a subtype of *expression* with the difference that it has a name, referencing the function it is invoking. The arguments are just a list of expressions that can contain anything. For instance, the function call *executeQuery*(*getQuery*(*"name"*)) is named *executeQuery* and has as argument another function call named *getQuery*. The latter having as argument a constant with value *"name"*.

A function call is considered *tainted* if any of the following conditions is true:

- the called function is in the source code and it returns a *tainted* value

- the called function is in a library and any of its arguments is *tainted*

This element also represents method calls that do not have an object as a source (e.g., methods from the same class or any superclass that are not preceded by the keywords *this* or *super* in Java).

7) *Assignment:* statement that represents, as the name indicates, an assignment. Consists of two expressions, one on the left-hand side and another on the right-hand side. It is the main way of propagating *taint* marks. Whenever the expression assigned on the right is evaluated as *tainted*, the mark is also propagated to the expression on the left. Usually, the expression on the left-hand side is just a variable. Listing 2 contains examples of assignment statements.

8) *Return:* statement that represents the end of the data flow for a path. The returned value is represented by an *expression*, which can be *tainted*.

9) *Throw:* statement very similar to the return statement. The only difference is that when a return statement is found in the callee function, the data flow is transferred to the caller function. Whilst in the case of a throw statement, the data flow is transferred to a catch block. The thrown expression can be *tainted*.

10) *Method Call:* this element represents method calls. It consists of a source, which can be an object or a class (if the method is static). Listing 3 shows examples of method calls where the source is an object and a class respectively. A method call is *tainted* if any of the following conditions is true:

- the method is not in the analyzed code and the source is *tainted*
- the method is in the source code and returns a *tainted* value

```
1 context.getUsers();
2 MyClass.myStaticMethod();
```

Listing 3. Method call examples

11) *New:* expression that represents an object creation. It works mostly as a function call to the constructor. In most languages, the constructor has the same name as the class. But in some languages, the name of the constructor is different from the name of the class (e.g., PHP, Python). For these cases, we keep a configuration file with the names. For instance, *new Foo*() in PHP would result in the call of *__construct*() function of the class *Foo*.

12) *Attribute access:* represents a direct access to an object attribute (e.g., *context.myProperty*).

13) *Code block:* represents a block of code. Consists of a list of statements. Listing 2 is an example of a code block with three statements.

14) *Conditional statement:* this element represents loops (e.g., *for, while, do while* etc.). Consists of a code block and a condition, which is an *expression*.

15) *Try Catch:* statement composed of a *try* code block, a list of *catch* code blocks and a *finally* code block.

16) *If:* statement that represents control flow statements (e.g., *if-else, if-elseif-else* and *switch*). Each option in the control flow has its own code block (e.g., *if-else* has two code blocks - one for *if* and another for the *else*).

17) *Function:* element that represents a method or a function. It has a name, a return type, a list of parameters and a code block.

18) *Class:* element that represents a class. It has a name, a list of attributes and a list of methods. Furthermore, it can have a superclass.

19) *File:* this is the root element of any GAST. It has a code block (for languages like PHP or JavaScript), a list of classes, a list of functions and a list of imported files (used to trace calls to imported function).

One important note to keep in mind is that not every language will use every feature from the GAST. For instance, the file element can have statements directly in the root block. Now, this is a feature that it is only used by languages such as PHP and JavaScript. By contrast, Java and C# do not allow code outside classes, so they do not make use of this feature.

C. Building the GAST

In order to build the GAST, we first parse the source code using a parser generated by ANTLR4 to obtain the source code AST. Then, we convert the AST to the GAST representation. To do this, we use a tree walker also generated by ANTLR4. The tree walker, which conceptually is a visitor, traverses every node of the AST and for each node it invokes a function when it enters or exits that node. So, in order to convert the AST to GAST we need to override some of these methods in the *AST Converter* to pass state to the *GAST Builder*. This is due to the fact that the *GAST Builder* keeps the current context in a stack and these methods indicate what is the new context to push and when to pop it. This way the *AST Converter* is just a class that overrides a set of methods from the generated tree walker.

Listing 4 shows part of the PHP *AST Converter*. In the example, we override two methods from the generated tree walker. The first method is invoked when the tree walker enters a *Func* node and the second when it exits the same node. So, when the tree walker reaches a *Func* node, it invokes the method *enterFunc* from the *AST Converter*. The latter then invokes a method from the *GAST Builder* that pushes the function element to the stack. This way, the *GAST Builder* is able to keep track of the context. For instance, if the tree walker encounters a *Parameter* node while in the *Func* node, the parameter would be added to the element on the top of the stack, which in this case would be a function. The function is then popped from the stack when tree walker exits the *Func* node, invoking the method *exitFunc* which calls a

```
1  @Override
2  public void enterFunc(FuncContext ctx) {
3      gastBuilder.addFunction(ctx, ctx.name());
4  }
5
6  @Override
7  public void exitFunc(FuncContext ctx) {
8      gastBuilder.exitFunctionOrMethodDeclaration();
9  }
```

Listing 4. Function declaration example

method from the *GAST Builder* that pops the function (line 8).

One important note is that we only push to the stack statements (e.g., functions, classes, assignments, conditional statements, etc.), which are nodes in the AST, while elements such as variables, constants or parameters are not pushed. This is due to the fact that they represent leaves in the AST, so they would be pushed and popped right away. This simplification allows us to override less methods from the tree walker when writing the *AST Converter* (e.g., we do not need to override *exitVariable*).

Let us now consider listing 5 where line 1 corresponds to the assignment being built and lines 3-9 to the sequence of calls made by the *AST Converter* to the *GAST Builder*. Due to lack of space, the signatures of the overridden methods are omitted. When the tree walker enters the assignment, the *AST Converter* calls a method that adds an assignment to the stack (line 3). Next, it enters a variable and since the assignment is on the top of the stack, the variable becomes the left side of the assignment (line 4). Then, the tree walker enters the expression *"x == 5"* and the *AST Converter* calls a method that adds an expression to the stack (line 5). After that, it enters a variable and later a constant, which will both be added to the expression, since it is the top of the stack (lines 6, 7). Finally, the tree walker first exits the expression and then the assignment, calling *exitStatementOrExpression* twice (lines 8, 9). Figure 2 shows the GAST representation of the assignment.

```
1  boolean a = x == 5;
2
3  gastBuilder.addAssignment(ctx);
4  gastBuilder.addVariable(ctx.VarName());
5  gastBuilder.addExpression(ctx);
6  gastBuilder.addVariable(ctx.VarName());
7  gastBuilder.addConstant(ctx.getText());
8  gastBuilder.exitStatementOrExpression();
9  gastBuilder.exitStatementOrExpression();
```

Listing 5. Assignment build call sequence

Converting the AST is as simple as identifying the methods needed to override from the generated tree walker, and then call the methods from the *GAST Builder*. For example, the *AST Converter* for PHP has 67 lines of code (counting only statements), and from these

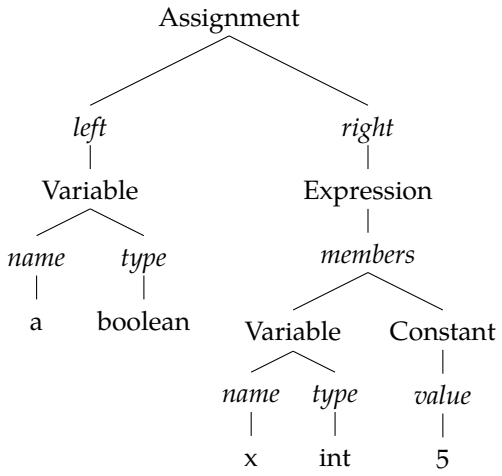

Fig. 2. GAST resulting from executing listing 5

67 statements, there are 29 that are different, meaning that a lot of the methods invoked are the same (e.g., *exitStatementOrExpression* is invoked 23 times).

D. Analysis features

Our approach is to use a context-aware static taint analysis to find all potential security vulnerabilities. To be able to statically find vulnerabilities, it is necessary to know what *objects* each variable may refer to, a general problem known as *pointer*, *points-to* or *alias analysis* [22]. Also, the tool must perform a *path aware taint* propagation. Furthermore, it must be able to detect function/method calls between different files.

Next, we discuss the importance of the *alias analysis*. Then, we present our solution to a *path aware* analysis and the method GT uses to find function/method calls from different files. Finally, we describe our approach to handle loops.

1) Pointer information: to illustrate the importance of pointer information, consider the example from listing 6. Assume that *param* is tainted and that *executeQuery* is a sensitive function. In this example, a more conservative approach may assume that *buf1* and *buf2* may reference the same object, thus marking both calls to *executeQuery* as tainted. Instead, GT traces the data flow through assignments made in each execution path. Thus, being able to identify that line 9 is a safe call and line 13 is a vulnerability.

Pointer analysis has been subject of much compiler research over the last decades [23], [24]. Since determining what heap object a given variable may point to is undecidable, our approach computes only an approximation based on the data flow through assignments. Meaning that, in much more complex cases, when GT is unsure to which reference an object is pointing to, it assumes that they all point to the same instance.

2) Path Analysis: in order to perform a precise static taint analysis it is very important to be able to track

```
String param = req.getParameter("name");
StringBuffer buf1;
StringBuffer buf2;

...

buf1.append(param);
String query = buf2.toString();
con.executeQuery(query);

buf2 = buf1;
query = buf2.toString();
con.executeQuery(query);
```

Listing 6. Taint propagation example

the data through different paths, ideally all of them. However, in practice, it is almost impossible since static path analysis is a very complex problem. The path is often decided at runtime due to features like `instanceof`, dynamic dispatch or reflection in Java [25]. Furthermore, these features differ between languages. Because of this, we can not have the most precise path analysis for each language. Instead, we perform an approximate path analysis based only on the *control flow statements*, ignoring their conditions. This means that we only look at the structure of the code, and for each conditional statement we propagate the data flow twice: one assuming that the flow enters that path and another assuming it does not. This assumption has the disadvantage of propagating *taint* marks through impossible paths.

Consider now the example from listing 7 and assume that the variable *input* is *tainted*. Observing the code, we can easily identify an SQL injection vulnerability at line 7, since at line 3 the *query* is concatenated with the *input*. In this example, GT propagates the *taint* marks through two paths: the first executing the *if* and the second the *else*. Finally, it reports the vulnerability at line 7. Also, it mentions that this vulnerability only happens if the expression *"input != null"* at line 2 is true. In more complex cases, for each vulnerability GT returns the call stack and the conditions that need to be met.

```
String query = "SELECT * FROM users WHERE name=";
if (input != null){
    query = query + input;
} else {
    query = query + "Bob";
}
con.executeQuery(query);
```

Listing 7. Path propagation example

Let us now consider the example from listing 8. This example is almost identical to listing 7, with the exception of line 2. This assignment makes the instruction from line 4 unreachable, meaning that in practice, the code has only one possible path. However, GT has exactly the same output as the previous example: two paths and one vulnerability. This is due to the fact that

```
String query = "SELECT * FROM users WHERE name=";
input = null;
if (input != null){
    query = query + input;
} else {
    query = query + "Bob";
}
con.executeQuery(query);
```

Listing 8. Path propagation example with unreachable branch

GT, and static analysis tools in general, struggle with detecting whether a condition can be true or not.

This way, we perform an approximate path analysis based on the *control flow statements* and their code blocks. The advantage of this kind of path detection is that it can be applied to any language. Furthermore, from our testing, most of the times it is enough to detect vulnerabilities, even though it is not the most accurate.

3) Cross-file function referencing: in the last decades, web applications have become increasingly more complex, consisting of many files. For this reason, in order to perform a precise static analysis, we need to be able to perform *taint* propagation between files. However, languages have different ways of importing code. For instance, Java imports packages, which consist of a set of classes, and PHP imports files directly [26], [27].

To mitigate this problem, our approach supports two generic ways of importing code:

1) File inclusion - usually used by dynamically typed languages, such as PHP, Python and JavaScript. Each file has a list of imported files. When GT finds a call to a function that is not found in the file, it searches in all imported files for that function. If more than one is found, it analyses all of them.
2) Type tracking - works for most of the object-oriented languages, such as Java and C#. Consists of checking the type of the target of the method call and then checking if that class is in the source code. If the class is found it tries to find the method. If the method is not found, it goes to the superclass.

From our testing and analysis, most of the times GT correctly propagates the flow to other functions.

4) Loops Analysis: loops have always been tricky for static analysis tools. This is due to the fact that in many cases it is impossible to know how many times a loop will execute, if any. Many times they are influenced by the user. For this reason, our tool takes a simplistic approach to deal with loops, which is analyzing each loop twice. This approach helps to mitigate cases where a variable only becomes *tainted* after the first iteration. Either way, for more flexibility we left the value configurable, so we could change it depending on the program we want to analyze.

To illustrate this issue, consider listing 9. In this example, if the *while* loop executes once, there is no

```
1 $name = $GET_["name"]
2 $query = "SELECT * FROM users WHERE name="
3 while(true){
4     mysql_query($query);
5     $query += $name;
6 }
```

Listing 9. Vulnerability in while loop

vulnerability. This happens because in the first iteration *$query* is not *tainted* upon executing *mysql_query*. However, after executing line 5 once, *$query* becomes tainted which makes the next call to *mysql_query* unsafe. By propagating the taint more than once, GT is able to detect this kind of vulnerability.

IV. Experimental Evaluation

The objective of this section is to show that GT is capable of finding vulnerabilities in web applications written in different languages and that the effort needed to add a new language to the tool is relatively small. First, we present the results of analyzing several web applications in Java, PHP, Python and JavaScript. Then, we discuss the effort needed to add support for another language.

A. Vulnerability detection

In order to show the ability of GT to analyze and find vulnerabilities in web applications, we tested GT against two types of web applications. First, we chose 5 open source applications from GitHub that are deliberately insecure, with documented vulnerabilities. The criteria used to choose them was the number of stars that each application has on GitHub, essentially choosing the most known ones. To run the tests, we had to manually identify the entry points and sensitive functions for each application, meaning that GT analyzed each application several times, once for each entry point. Second, we also tested GT against some real-world open-source web applications. Since these applications are much bigger, we assumed that every file can is a web page that can receive user input, tainting the variables that might be influenced by the user (e.g., $_GET[*] in PHP). Tables I and II show the results of our analysis. In the data, we only include files with the extension that we analyzed (e.g., *.java and *.php). Furthermore, we excluded comments and blank lines from the line count.

GT analyzed 1946 files and 315031 lines of code and managed to find 71 documented vulnerabilities, such as SQL injection, cross-site scripting, command injection and file inclusion. The analysis times were quite low. The application that took the longest to analyze was *PhpMyAdmin* with 82 seconds. However, the analysis times varied a lot depending on the entry point, meaning that the longer the path through which the data flows, the longer the analysis time.

TABLE I
Deliberately insecure web applications

Application	#loc	Language	Files	Vuln. Found
WebGoat 8	13898	Java	320	11
Vulnado	423	Java	11	3
Dvja	950	Java	21	4
DVWA	19651	PHP	358	18
OWASP Vwa	1018	PHP	27	17
Vulnerable-node	4207	JavaScript	13	5
Dvna	771	JavaScript	14	0
Vulpy	2373	Python	57	6
Dvpwa	674	Python	21	7
Total	43965		842	71

TABLE II
Real-world web applications

Application	#loc	Language	Files	Vuln. Found
SquirrelMail 1.5	46214	PHP	376	0
PhpMyAdmin	224852	PHP	728	0
Total	271066		1104	0

In our tests, the tool had 4 false negatives in Dvna because all the dangerous code is in anonymous functions that are called by referencing variables, since we do not track the value of each variable, this is a limitation.

GT only raised 5 false positives in the applications from table I due to data flow propagation through impossible paths. We may assume that the low number of false positives is due to our conservative taint analysis and to the relatively simple web applications that we have tested. We can not make any assertions about the false positives from the applications from table II since GT did not report any vulnerabilities and also, we are not aware of any existing vulnerabilities in the versions we tested.

B. Portability

Since the objective of this work is to support several languages with as little effort as possible, the portability of the tool is also a metric that we tested. To test the portability, we first implemented the tool to support PHP analysis, and then we added support for Java, which is a substantially different language. While adding support for PHP and Java we were also developing the other modules, so it is hard to tell how much time was spent strictly adding support for each language. However, later, after the tool was built, we added support for another two very popular languages: JavaScript and Python. Adding support for each of them took us roughly 7 hours. Table I shows that our implementations are able to find vulnerabilities. In our opinion, the main challenge

when adding support for a new language is identifying which elements from the grammar are important to the analysis. After that, we just have to write the converter and some unit tests to make sure the converter works properly.

Table III presents the order in which the languages were added, the number of lines of each converter, the number of unique statements and the number of hours spent developing each one of them. The number of unique statements is meant to show that the converter does not have much logic, it basically consists of calls to the *GAST Builder*. For example, the converter for Java has 108 lines, 30 of which are repeated (e.g., *exitStatementOrExpression*() is invoked 19 times), leaving us with 78 unique statements.

TABLE III
CONVERTERS SIZE AND IMPLEMENTATION EFFORT

Impl. order	Language	#loc	Unique statements	Hours to implement
1	PHP	67	49	–
2	Java	108	78	–
3	JavaScript	50	41	7
4	Python	61	49	7

V. Related Work

There is a rich body of work in the area of vulnerability detection, we just summarize the main detection methods by discussing representative papers, while leaving many others unreferenced due to lack of space.

A. Taint Analysis

The most commonly used *data flow analysis* technique. Consists of tracking the flow of sensitive information by marking user input as *tainted* and then propagate the *taint marks* recursively to the variables that are influenced by other *tainted* data. Then, it checks if *tainted* data reaches *sensitive functions* (e.g., *eval, my_sql_query* in PHP). If it does, then there is probably a vulnerability that could be exploited. Taint analysis can be applied to either source, binary or intermediate code.

B. Static Analysis Tools

Static analysis tools are used to automate the detection of bugs and vulnerabilities. Nowadays, they are often part of the development process, with their use being automated by continuous integration pipelines [28]. The reason is that they are a cheap way of detecting issues in code, giving the developers more confidence in their software.

SonarQube [29] is a widely used commercial static analysis tool. Performs static analysis based on a set of rules that can be defined by the user. It is able to detect bugs (e.g., possible *null references*) or bad practices in source code (e.g., empty *catch* blocks in Java). Furthermore, it also performs static taint analysis to find vulnerabilities. The taint analysis supports 4 languages, while other features support more than 20. However, since SonarQube is a commercial tool and it is not open source, we can not make any assertion about its complexity.

FlowDroid [9], is a precise static taint analyzer specifically tailored for Android and Java applications. Analyzes apps' bytecode and configuration files to find vulnerabilities. FlowDroid is precise because it models the lifecycle of android apps and it is context, field, object and flow-sensitive. Pixy [8] performs taint analysis on PHP source code and extends it by using *alias analysis*. This way it considers the existence of two or more variables that identify the same object.

Another very interesting idea is from A. Notzli, and D. Engler [21], who managed to implement an effective bug-finding static checker, which is a static analysis tool that has the goal to find bugs in a program (e.g., *null pointers*, *deadlocks*), and that it is orders of magnitude less complex. This was possible by using a new source code parsing technique based on incomplete *micro-grammars*, instead of depending on every syntax detail of a language or its compiler. Traditional checking systems use parsers designed to parse a complete language syntax, thus rejecting any input that does not lead to a valid parse. On the other hand, micro-grammars parsing for bug finding has two main differences from traditional parsing.

1) When a traditional parser finds a non-matching input to its specifications, it returns an error. By contrast, when a micro-grammar parser hits a non-matching input, it simply slides forward by one token and tries again.
2) Micro-grammars allow developers to perform fine-grained input skipping by using *wildcard* non-terminals that lazily match any input up to a suffix.

The implementation of this static checker is very modular and it is composed of a lexer, parsers and checkers. There is a parser for each non-terminal, for example the parser for *C* is composed of smaller parsers for *if* statements, *while* loops, *for* loops, etc. Then all these small parsers compose the parser for *C*. This modularity brings another big advantage, the possibility of reusing a lot of these small parsers between languages, for example, C and Dart share many parsers. The use of micro-grammars and modular architecture make the tool relatively easy to port to other languages. This somehow inspired our work, since our tool also does not depend on every detail of a language.

Andromeda [30], is a demand-driven static taint analysis tool that supports Java, C# and JavaScript. It is flow and context-sensitive. Furthermore, it extends its analysis by being integrated with Framework For Frameworks

(F4F), which is a solution for augmenting taint analysis with precise framework support [31].

C. Dynamic Analysis Tools

Dynamic analysis tools detect vulnerabilities at runtime. Most commonly, they rely on some kind of *code instrumentation* or transformation. Dytan [14] instruments x86 compiled code with the ability to do *taint* propagation at runtime. It is precise since it does *taint* analysis at byte level. Instrumenting x86 compiled code allows it to support any language that compiles to x86 (e.g., C and C++). Phosphor [17] applies a similar technique to Java bytecode. However, it propagates the *taint* marks at variable level, improving performance over Dytan at the cost of some precision. Moreover, it supports any language that runs on top of Java virtual machine (e.g., Java, Kotlin or Scala).

VI. Conclusion

In this paper we presented an approach that aims to reduce the complexity of static security analyzers. To do this, we build a generic AST (GAST) from the source code AST. The GAST is not bound to any language and can represent a large set of languages used in web applications. This way, we perform our analysis on the GAST, allowing us to decouple the source code parsing and the vulnerability detection. The solution was implemented in a tool called GT, using Java with parsers and tree walkers generated by ANTLR4. Our evaluation shows that GT can find vulnerabilities in different types of languages (e.g., statically and dynamically typed).

References

[1] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[2] Rahul Telang and Sunil Wattal. An empirical analysis of the impact of software vulnerability announcements on firm stock price. *IEEE Transactions on Software engineering*, 33(8):544–557, 2007.

[3] Hal Berghel. Equifax and the latest round of identity theft roulette. *Computer*, 50(12):72–76, 2017.

[4] J. Williams and D. Wichers. OWASP Top 10 - 2017 rcl - the ten most critical web application security risks. Technical report, OWASP Foundation, 2017.

[5] Smriti Jain and Maya Ingle. Review of security metrics in software development process. *International Journal of Computer Science and Information Technologies*, 2(6):2627–2631, 2011.

[6] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, pages 1181–1192, 2013.

[7] Ioannis Papagiannis, Matteo Migliavacca, and Peter Pietzuch. Php aspis: using partial taint tracking to protect against injection attacks. In *2nd USENIX Conference on Web Application Development*, volume 13, 2011.

[8] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6–pp. IEEE, 2006.

[9] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[10] Paulo Jorge Costa Nunes, José Fonseca, and Marco Vieira. phpsafe: A security analysis tool for oop web application plugins. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 299–306. IEEE, 2015.

[11] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 171–180. IEEE, 2008.

[12] Johannes Dahse and Thorsten Holz. Simulation of built-in php features for precise static code analysis. In *NDSS*, volume 14, pages 23–26. Citeseer, 2014.

[13] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.

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

[15] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[16] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Dta++: dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[17] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. *ACM Sigplan Notices*, 49(10):83–101, 2014.

[18] Ibéria Vitória de Sousa Medeiros. *Detection of vulnerabilities and automatic protection for web applications*. PhD thesis, Doctoral dissertation, Universidade de Lisboa, 2016.

[19] Stephen Cass. The top programming languages: Our latest rankings put python on top-again-[careers]. *IEEE Spectrum*, 57(8):22–22, 2020.

[20] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, 2009.

[21] Fraser Brown, Andres Nötzli, and Dawson Engler. How to build static checking systems using orders of magnitude less code. *ACM SIGPLAN Notices*, 51(4):143–157, 2016.

[22] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J Fink, and Eran Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 196–232. Springer, 2013.

[23] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[24] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, 2001.

[25] Christian Hammer, Rüdiger Schaade, and Gregor Snelting. Static path conditions for java. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 57–66, 2008.

[26] Atanas Rountev, Scott Kagan, and Michael Gibas. Static and dynamic analysis of call chains in java. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11, 2004.

[27] Mark Hills, Paul Klint, and Jurgen J Vinju. Static, lightweight includes resolution for php. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 503–514, 2014.

[28] Sikender Mohsienuddin Mohammad. Continuous integration and automation. *International Journal of Creative Research Thoughts (IJCRT), ISSN*, pages 2320–2882, 2016.

[29] G Campbell and Patroklos P Papapetrou. *SonarQube in action*. Manning Publications Co., 2013.

[30] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 210–225. Springer, 2013.

[31] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: taint analysis of framework-

based web applications. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 1053–1068, 2011.