TÉCNICO LISBOA

Virtual Static Security Analyzer for Web Applications

Mihail Brinza

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): João Pereira
Miguel Correia

Examination Committee

Chairperson:

Supervisor:

Member of the Committee:

December 2020

Acknowledgments

I had the pleasure of spending the last 5 years as a student of Instituto Superior Técnico. Throughout this period, I had the opportunity to learn and evolve a lot, not only from a technical standpoint but also from a personal one. For this reason, I am very grateful to the university and all my professors.

During the time I was doing my MSc thesis, Prof. João Pereira and Prof. Miguel Pupo Correia were always willing and available to help and discuss the choices I was making. I am deeply grateful for all the advice, time spent helping me, and all the feedback given on my work.

Also, I must thank my family and friends for all the support received. Finally, I want to thank Miguel Ferreira who took the time to read this work to help with the writing.

Resumo

Nas últimas décadas, as aplicações web têm sido um alvo muito popular de ataques informáticos. Para mitigar esse problema precisamos de formas automatizadas de detetar vulnerabilidades em código fonte. No entanto, as ferramentas modernas são muito complexas, sendo constituídas por milhares de linhas de código. Para além disso, as suas implementações estão muitas vezes presas a uma determinada linguagem. Essa complexidade faz com que as ferramentas sejam muito difíceis de compreender e de serem extendidas para suportarem novas linguagens.

Para reduzir a complexidade dos analisadores estáticos atuais, esta tese propõe uma nova solução genérica, que suporta a adição de novas linguagens sem muito esforço de programação. A nossa solução, ao contrário dos analisadores estáticos tradicionais, não analisa a AST do código fonte diretamente. Em vez disso, percorremos a AST do código fonte e construímos uma AST genérica (GAST) a partir desta. Depois disso, a análise para encontrar vulnerabilidades é feita com base na GAST. Desta forma conseguimos desacouplar a parte da análise e do *parse* do código fonte. Para além disso, GAST apenas contém o que é realmente necessário para fazer a análise, ignorando o resto. Para adicionarmos suporte a uma nova linguagem é necessário apenas gerar um *parser* usando ANTRL4 [Par13] e escrever um conversor para a respetiva AST. Habitualmente um conversor é composto por menos de 110 linhas de código.

A nossa solução foi implementada na ferramenta GT, que supporta PHP, Java, JavaScript e Python, e foi testada contra várias aplicações web.

Palavras-chave: segurança, análise estática, fluxo de dados, vulnerabilidades

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 [Par13] 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.

Keywords: Security, Static Analysis, Taint Analysis, Antlr4, Information Flow

Contents

List of Tables

List of Listings

List of Figures

Chapter 1

Introduction

1.1 Motivation

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 [TW07]. 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 [Ber17].

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 [WW17]. 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 [JI11].

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 [SMS13, PMP11, JKK06, ARF+14, NFV15, WS08, DH14, LL05], a specific compiled code [CLO07, NS05, KMPS11] (e.g., x86 binary, bytecode) or they depend on modified runtime engines [SMS13, BK14]. To port one of these tools to another language basically requires to implement it again from scratch.

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 2.2.

1.2 Overview

In this thesis, 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

[1]In this work 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.

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 thesis 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 Java, PHP, Python and JavaScript applications. The tool is open-source and available at `https://github.com/Mihail1996/GT-tool`. Moreover, there is also a scientific paper based on this work that was submitted to the ISSTA 2021 conference.

1.3 Thesis Outline

To get a better understanding of the proposed work, chapter 2 starts by presenting a list of injection vulnerabilities that we address in this work. Then, section 2.2 explains the concept of *taint analysis*. After that, sections 2.3, 2.4 and 2.5 present a set of static and dynamic approaches at detecting vulnerabilities that somehow influenced our work.

Chapter 3 aims to describe the developed work in further detail. Section 3.1 starts by presenting the architecture of the GT tool, describing each module. Then, section 3.2 goes more in-depth explaining the GAST by presenting each node that is part of it. Section 3.3 follows by

explaining how the GT tool builds the GAST. Finally, section 3.4 presents our taint analysis and the compromises that were made.

Chapter 4 shows the results of our work. Section 4.1 first presents the results of analyzing several web applications and then states how portable our tool is. Then, section 4.2 talks about the limitations of our implementation.

Finally, chapter 5 concludes our work by stating how our goals were achieved and pointing out what can still be improved.

Chapter 2

Background

In this chapter, we start by introducing the vulnerabilities considered in this thesis. Then, we present taint analysis and describe a few static and dynamic approaches that somehow try to solve the problem of portability.

2.1 Input Validation Vulnerabilities

This section briefly presents a list of vulnerabilities considered in this work. We can divide them in four categories [dSM16] - *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.

2.1.1 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 meta-characters that can alter the behavior of the engine.

SQL Injection

This vulnerability is caused by the use of string building functions to create SQL queries. An attack consists of mixing normal characters with meta-characters. In the example of listing 2.1, 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 $query = "SELECT * FROM users WHERE name='$name' AND password='$pass'";
4 $result = mysql_query($query);
```

Listing 2.1: PHP code vulnerable to SQL Injection

NoSQL Injection

Non-relational (NoSQL) databases are used in many large-scale web applications. There are several NoSQL database engines that implement them. MongoDB [Mon] is the most popular engine implementing the document store model [DBE], so we will focus on vulnerabilities from web applications that connect to a MongoDB instance. Mongo executes queries in JSON format, so a NoSQL Injection (NoSQLI) vulnerability is caused by the use of string building functions to create the JSON query.

```
1 let username = req.query.username;
2 query = { $where: 'this.username == '${username}'' }
3 User.find(query, function (err, users) {
4     res.render('userlookup', { title: 'User Lookup', users: users });
5 });
```

Listing 2.2: JavaScript code vulnerable to NoSQL Injection

Listing 2.2 shows an example of code vulnerable to NoSQLI. In the example, the program takes the username query parameter from the request URI and inserts it in the query. The program then sends the query to the database and returns to the user the result of the query. In this example, if a user passes as argument a string like `' || 'a'=='a` the query will become `$where: 'this.username == '' || 'a'=='a''` which will naturally evaluate to true, and thus returning all values. To remove this vulnerability it is enough to sanitize or escape the user input.

XPath Injection

This vulnerability is very similar to SQL Injection, but in this case, the data is injected in XML documents, which are often used to store data or configurations. Listing 2.3 shows an example of a PHP script vulnerable to XPath injection. The script takes a username and a password (lines 2, 3) and inserts them into an XPath query (line 4). An attacker could provide as username `admin' or 1=1`, causing the script to return information about the admin user without providing a password. To prevent this vulnerability, it is enough to check if the input contains malicious characters.

```
1 $users = simplexml_load_file("users.xml");
2 $name = $_POST[username];
3 $pass = $_POST[password];
4 $query = "//User[UserName/text()='".$name."' And Password/text()='".$pass."']";
5 $result = $users->xpath($query);
```

Listing 2.3: PHP code vulnerable to XPath Injection

LDAP Injection

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

2.1.2 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.

Cross-site scripting (XSS)

There are three types of XSS attacks: reflected or non-persistent, stored or persistent and DOM-based. 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.

Header injection

This vulnerability 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.

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.

2.1.3 File and Path Injection

This category considers vulnerabilities related to file accesses from web applications, the file system and to URL locations different than the web application.

Remote file inclusion

PHP allows a script to include files, which can be vulnerable if the file name comes from user input. In the example of listing 2.4, if a malicious user provides as parameter *country* the URL *http://www.site.pt/hack*, would cause the execution of *hack.php* in the server.

```
1 $country = $_GET['country'];
2 include($country . '.php');
```

Listing 2.4: PHP script vulnerable to remote file inclusion

Encoding the input or making it impossible for a user to deliberately influence the filename prevents this vulnerability.

Directory/Path traversal

This vulnerability allows an attacker to read arbitrary files from the server. To access them the attacker builds an URL containing path metacharacters, such as ".." and "/". In the example of listing 2.4, if *"../../../etc/passwd%00"* is passed as input, the file *"/etc/passwd"* is sent to the attacker (the null character %00 truncates additional characters, .php in this case).

2.1.4 Command injection

This category consists of vulnerabilities that allow an attacker to inject operating system commands or PHP code directly.

OS command injection

This vulnerability consists of executing arbitrary system commands defined by the attacker. Consider the following example that uses a script to count the words of a file: *"$words = shell_exec ("/usr/bin/wc" . $GET_['file']);"* (*shell_exec* executes system commands and *wc* is a system command to count words). An attacker could retrieve a file from the server by giving as input *"file.txt; cat /etc/passwd"*. The resultant instruction *"$words = shell_exec("/usr/bin/wc file.txt; cat /etc/passwd);"* executes the *wc* and *cat* commands. The second command shows the content of a file.

PHP code injection

This vulnerability is caused by the *eval* function in PHP. This function runs the code that it receives as a string in its parameter. Consider the example from listing 2.5 that uses the *eval* function to concatenate the string *"Hello"* with the name provided by the user. The attacker can do a command injection attack by providing a username and a command separated by a semicolon (e.g., *"Bob; cat /etc/passwd"*). To avoid this attack, the input must be sanitized, but it is not that simple. For this reason, the use of *eval* function is not advised.

```
1 $msg = 'Hello';
2 $x = $_GET['username'];
3 eval('$msg =' . $msg . $x . ';');
4 echo $msg;
```

Listing 2.5: PHP script vulnerable to code injection

2.2 Vulnerability detection

There are many different approaches to prevent injection vulnerabilities. A very popular one is *taint analysis* which can be divided in two categories: *static taint analysis* and *dynamic taint analysis* [SAB10]. In this thesis, we will focus on the first one to detect vulnerabilities. There is also a second type of dynamic prevention which is *parse tree validation* [BWS05, LYWS09].

2.2.1 Taint analysis

This 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 sinks*. If it does, there is a vulnerability that could be exploited.

Taint analysis has three main components:

1. *Data entry point* - the input that comes from untrusted sources is marked as *tainted*;
2. *Taint propagation* - the *taint* marks are then propagated according to a propagation policy;
3. *Sensitive sinks* - Every function that can be exploited (e.g., *eval*(), *my_sql_query*()). The tools then check if the data that enter *sensitive sinks* is *tainted* or not.

There are two types of taint propagation policies: *explicit* and *implicit information flow*. Listing 2.6 is an example of *explicit information flow*. Assume that the value of the parameter *a* is tainted,

then the taint will be propagated to variable w since a is involved directly in the computation of w;

The second type of taint propagation is less intuitive. It refers to situations in which a tainted value affects the value of another variable indirectly. Consider the code in listing 2.7 and assume the value of parameter a is tainted. Although a is not involved directly in the computation of the value of variable x, the value of a affects the value of x through dependency control.

```
1  void foo(int a){
2      int w;
3      w = a * 2;
4  }
```

Listing 2.6: Explicit information flow

```
1  void foo(int a){
2      int x;
3      if (a > 5){
4          x = 1;
5      }
6      else{
7          x = 2;
8      }
9      print(x);
10 }
```

Listing 2.7: Implicit information flow

2.3 Static Analysis Tools

Static analysis tools use pointer and taint analysis to find data flows from *entry points* to *sensitive sinks*. They can also verify if a *sanitization function* is called on tainted inputs. Since *static taint analysis* must make a lot of simplifications, it is prone to false positives and negatives. However, since it can be applied to either source or compiled code, static analysis tools are often 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 [Moh16]. The reason is that they are a cheap way of detecting issues in code, giving the developers more confidence in their software. Next, we discuss some static analysis tools.

2.3.1 Micro-grammars analysis

F. Brown, A. Notzli, and D. Engler [BNE16] 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 than traditional checkers. They achieved this improvement 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. This approach is called *sliding window*, and it allows the parser to match all the specifications described by the micro-grammar without getting stuck and skip all those that do not match, consuming all the input. This way their static checker only focuses on the most important aspects of a language, which it is used to detect bugs, and ignores the rest.

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. For example, the micro-grammar "$S \rightarrow$ *if* (*wildcard*)" when applied to a file with five *if* statements produces five parse trees where lists of tokens match to each *wildcard* node. For example, parsing "*if* ($x == 1$)" would result in a normal *if* node and a *wildcard* node containing "[*"x"*, *"=="*, *"1"*]". On the other side, a traditional parser creates a tree with one node for every token.

Wildcards are implemented based on the *SkipTo*(P) rule, represented in Figure 2.1, which skips tokens until it reaches a suffix P. Figure 2.1 also shows a micro-grammar example where *Rule1* accepts any token that starts with the character *a* and ends with *cd*. It would accept *abcd* for example, but not *abd*.

The *Slide'* rule, which corresponds to *Sliding Window* implementation, slides one token until it can find something to accept. Because of this rule, *Rule1* is able to accept inputs such as *"Xabcd"*, even though the input does not start with the character *a*.

The implementation of this static checker is very modular and it is composed of a lexer, parsers and checkers. The parsers are recursive and also modular. There is a parser for each non-terminal, for example the parser for *C* is composed by smaller parsers for *if* statements, *while*

$AnyToken \rightarrow token, token \in language$
$SkipTo(P) \rightarrow P \mid AnyToken\ SkipTo(P)$
$Slide' \rightarrow SkipTo(P)\ Slide' \mid \epsilon$
$Rule1 \rightarrow a\ SkipTo(cd)$

Figure 2.1: *Wildcard* and *Sliding Window* micro-grammar

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.

According to the authors, to extend their static checker to another language, one has to do the following:

1. Specify a lexer by supplying a list of keywords, operators, and a set of regular expressions for identifiers, literals and comments. A typical specification is around ten lines of code.

2. Build parsers recursively for each non-terminal. In this step, the developer may be able to reuse parsers from another language with similar syntax.

3. Develop checkers.

The use of micro-grammars and the architecture modularity makes the tool relatively easy to port to other languages.

The static checker presented in the paper focuses on a specific style of bug checking, called *belif-style*. This style assumes that programmers do not want to crash their programs, so from this, we can extract beliefs which are facts implied by the code. For example, the operation x/y implies that y can not be 0. Therefore, if there is a flow that contradicts this belief, it is considered that there is an error.

The fact that this approach does not depend on every detail of the language somehow inspired our work.

2.3.2 SonarQube

SonarQube [CP13] 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.

2.3.3 FlowDroid

FlowDroid [ARF+14] 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.

2.3.4 Pixy

Pixy [JKK06] is one of the first tools that processes PHP code. It performs taint analysis on PHP source code and extends it by using *alias analysis*, which takes into account the existence of aliases i.e., of two or more variable names that reference the same variable. It is able to detect SQL injection and Cross-site scripting in PHP code that does not use objects.

2.3.5 Andromeda

Andromeda [TPC+13] 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 [SAP+11].

2.4 Dynamic Analysis Tools

Dynamic taint analysis tools instrument applications with the ability to track the source of inputs in runtime. The applications are then able to determine if the data that reaches *sensitive sinks* contain any untrusted (*tainted*) inputs. Furthermore, *dynamic analysis* can also check whether input sanitization was done correctly or not, contrary to *static analysis*. Since *dynamic taint analysis* is done at runtime, analyzing applications results in adding performance overhead.

2.4.1 Dytan

Dytan [CLO07] is a dynamic taint analysis framework that aims to be generic and customizable. It is a very precise tool since it does taint tracking at a byte level, meaning that it can tell exactly which byte of a variable is tainted. It instruments compiled code (*x86*) with *taint tracking* ability, so it does not depend on the availability of the source code. Furthermore, it supports both *explicit* and *implicit information flow* propagation. In the article, the authors successfully use Dytan to detect SQL Injection, buffer overflows and format string attacks.

Dytan allows users to specify three main aspects of the analysis:

1. *Taint Sources* - Allows to set which sources are to be marked as *tainted*. The user can choose from variables, memory offsets, data from specific functions, from a type or specific I/O streams.

2. *Propagation Policy* - User can choose between two different propagation policies:

   - *Explicit information flow* - a tainted variable is directly involved in the computation of another variable's value.
   - *Implicit information flow* - a tainted variable affects the value of another indirectly, through information flow.

3. *Taint Sinks* - The user can set variables, functions, function parameters or instructions (ex *jump*;) as *sensitive sinks*.

In spite of being customizable, it is still bound to compiled code, so it can not be applied to an interpreted language or to languages that compile their code to an intermediate language, such as Java or C#. Furthermore, since the granularity of its taint tracking is a byte, it adds a runtime slowdown ranging from 30x-50x (where a slowdown of 1x means that the system now takes twice as much time to run). This makes the tool inviable to be run in real-world systems.

Another reason for this huge overhead is that Dytan taint tracks all implicit flows, which generates a lot of unnecessary taints. Dta++ [KMPS11] presents an improvement to this problem by identifying a minimum set of implicit flows in the program that potentially cause undertainting, and then generate *taint* propagation rules to solve it.

Taint Check [NS05] is a tool similar to Dytan. It also instruments compiled *x86* code, but it is inferior compared to Dytan because it only detects buffer overflows and string format attacks and also lacks customizability.

2.4.2 Phosphor

Phosphor [BK14] is a more recent tool, that implements *taint tracking* for two JVM (Java Virtual Machine) based languages: Java and Scala. It does it by instrumenting *bytecode*, without requiring the source code. This way *Phosphor* is able to run on top of unmodified JVMs (Figure 2.2 shows its high-level architecture).

It has two main differences from Dytan:

- It does tracking at variable level instead of byte level.
- It does not support *implicit information flow* propagation.

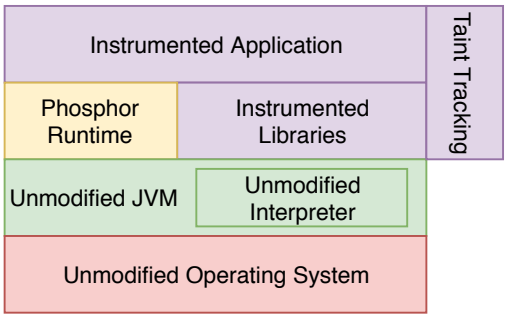

Figure 2.2: *Phosphor's* high-level architecture

One big advantage of the first one is that the overhead added is 1x in time and 2-3x in memory consumption. This is a big improvement and makes the tool more viable to be run in real systems. The cost of the much lower overhead is less precision. Tainting at variable level means that *Phosphor* is only able to tell whether a variable is tainted or not, but it is unable to tell exactly which character is causing the attack. The second difference is more of a limitation, but the authors say that the tool could be extended to support *implicit information flow* propagation.

Although *Phosphor* supports a set of languages and could be easily extended to support Kotlin as well, it is only portable to languages that run on top of JVM. In this case, *Phosphor* is bound to JVM bytecode, and to port it to a different language, such as PHP, is very difficult.

2.5 SQL parse tree validation

SQL parse tree validation [BWS05, LYWS09] is a type of dynamic analysis where the tool does not need compiled code nor source code, thus being language agnostic. Instead, it puts a *proxy* between the application server and the database. This *proxy* intercepts all the queries to the database and does an *SQL parse tree validation*. This way, it can protect applications written in different languages. *Parse tree validation* consists of building the parse tree of the query and validate it against the parse tree of a known benign query. Benign queries are then forwarded to the database whilst the malicious ones are dropped.

Consider the following query *"SELECT 'name' FROM students WHERE id = '12';"* and the parse tree from Figure 2.3. When the input is 12 the parse tree is composed only of the green leaves. If a malicious user gives as input the id *"12' OR 1 > 0 –'"*, the parse tree would be composed by the green and red leaves. This way, the tool detects that the parse tree is different and reports the attack.

Although being an interesting approach for detecting *SQL* and *NoSQL* injection attacks, its scope is limited. We can not apply this technique to detect Cross-site scripting or vulnerabilities in language functions (e.g., PHP code injection).

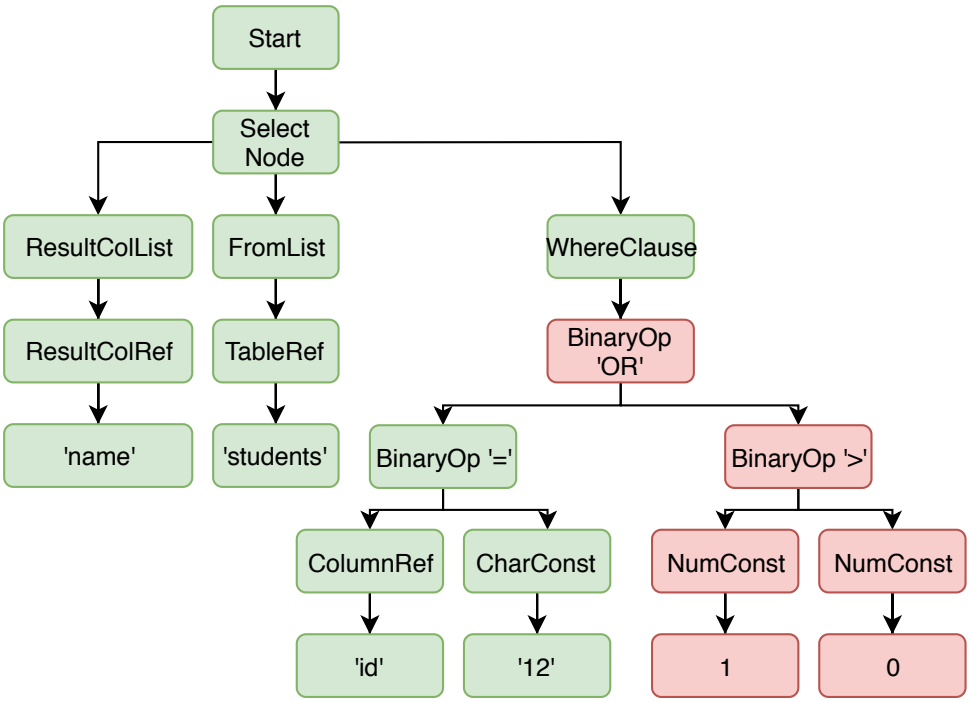

Figure 2.3: SQL query parse tree

2.6 Summary

This chapter first introduced a list of injection vulnerabilities considered in this work. Then, it presented a few static and dynamic approaches at detecting vulnerabilities.

Chapter 3

The GT Tool

In this chapter 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 [Cas20], 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 [Tra09]. 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 [BNE16], 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 3.2 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.

3.1 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 3.1, consists mainly of four modules: *Parser*, *AST Converter*, *GAST Builder* and *Taint Analyzer*.

3.1.1 Parser

This module 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 [Par13]. 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.

3.1.2 AST Converter

Module responsible for traversing the source code AST using the generated tree walker and raising events to the *GAST Builder* (e.g., entering and exiting class, methods or functions decla-

rations). 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.

3.1.3 GAST Builder

Module that reacts to the events from the *AST Converter* and internally builds the GAST. Section 3.3 shows more in-depth the process of building the GAST.

3.1.4 Taint Analyzer

Module that takes as input the GAST from the *GAST Builder* and a configuration file. Then, it traverses the GAST using a visitor and propagates the taint marks to find vulnerabilities according to the configuration file. Listing 3.1 shows an example of a configuration file. In this file, we can specify the value of several parameters, such as:

- Source code location - we can set the directory where the source code is and the file extensions that we want to analyze (lines 2, 3)
- Entry points - the file and the function where the data flow propagation should start. In the example, it corresponds to lines 6-14, where the entry point is a function called *functionEntryPoint* from the class *EntryPointClass* in the file *EntryPointClass.java*. We also specify which arguments are *tainted*: *"taintedArg","userInput"*
- Sensitive functions - functions that when called with *tainted* arguments may result in a vulnerability. In the example we specify a function named *executeQuery* from the class *Statement* (lines 15-20)
- Sanitization functions - functions that perform input sanitization and whose return value is always *untainted*. In listing 3.1 we configure a sanitization method called *sanitizeInputMethod* from the class *SanitizerClass*
- Loops settings - number of times we should analyze a loop. We explain this choice more in-depth in section 3.4.4.

After traversing the GAST, GT produces a vulnerability report in JSON format. Listing 3.2 shows an example of a report when analyzing listing 3.10. In the report, GT presents the vulnerabilities and the time that took to analyze the program. Furthermore, for each vulnerability it presents a list of conditions that need to be met to reproduce the vulnerability.

```
{
    "directoryPath": "path/to/app",
    "fileExtension": ".java",
    "numberOfTimesToAnalyzeCycles" : 2,
    "specification": {
        "filename": "EntryPointClass.java",
        "function": {
            "name": "functionEntryPoint",
            "type": "EntryPointClass"
        },
        "taintedVarsOrArgs": [
            "taintedArg",
            "userInput"
        ],
        "sensitiveFunctions": [
            {
                "name": "exectuteQuery",
                "type": "Statement"
            }
        ],
        "sanitizationFunctions": [
            {
                "name": "sanitizeInputMethod",
                "type": "SanitizerClass"
            }
        ],
        "returnTaintedIfTaintedSource": true,
        "taintedAttributes": [
            "taitedAttributeName"
        ]
    }
}
```

Listing 3.1: Example of file settings

```
{
    "vulnerabilities": [
        {
            "file": "EntryPointClass.java",
            "line": 7,
            "vulnerableMethod": "executeQuery",
            "functionCallStack": [],
            "conditions": [
                {
                    "line": 2,
                    "condition": "input != null"
                },
            ]
        }
    ],
    "timeToProcessMilliseconds": 2074
}
```

Listing 3.2: Vulnerability report when analyzing listing 3.10

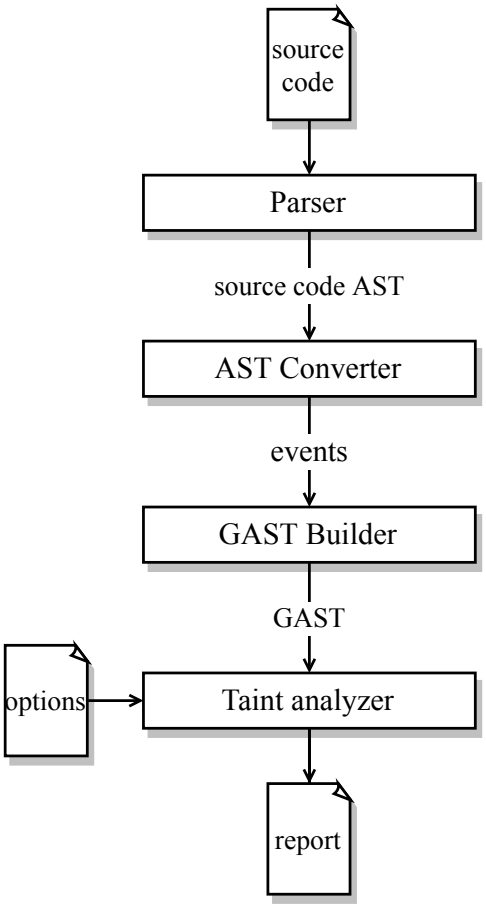

Figure 3.1: GT - data flow and architecture overview

3.2 GAST

Similarly to the micro-grammars solution [BNE16], 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.

3.2.1 Elements

Constant

Element that 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*. Figure 3.2 presents the GAST representation of a constant.

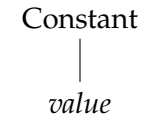

Figure 3.2: GAST representation of a constant

Variable

This element abstracts, as the name says, 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. Figure 3.3 shows the GAST representation of a variable.

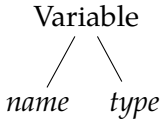

Figure 3.3: GAST representation of a variable

Attribute

Represents an attribute in a class. It has a name and a type. Also, it can be *tainted*. It is very similar to a variable, as we can see in its representation from figure 3.4. In our implementation an attribute is a subtype of *Variable*.

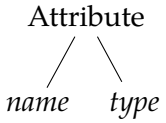

Figure 3.4: GAST representation of an attribute

Parameter

Element that abstracts the parameters of methods and functions. It can be *tainted* if it is the parameter of the entry point function, or if the argument passed to a function is *tainted*. Just like the *attribute*, it is also a subtype of *variable*. Figure 3.5 presents its GAST representation.

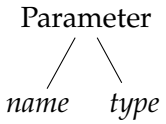

Figure 3.5: GAST representation of a parameter

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 3.3 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 3.3 is *tainted*, then, all variables will be *tainted*, regardless of their operator.

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

Listing 3.3: Expression assignment examples

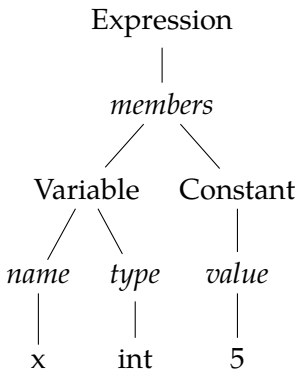

Figure 3.6: GAST representation of the expressions from listing 3.3

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*.

Function Call

This element 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. Figure 3.7 shows the GAST representation of a function call.

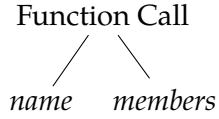

Figure 3.7: GAST representation of a function call

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"*. Figure 3.8 represents the described tree.

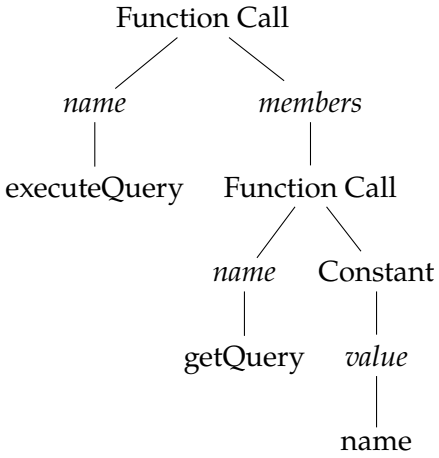

Figure 3.8: GAST representation of *"executeQuery*(*getQuery*(*"name"*)*"*

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).

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 3.3 contains examples of assignment statements and figure 3.9 shows the GAST representation of an assignment.

Return

This statement, represented in figure 3.10, represents the end of the data flow in a path. The returned value is represented by an *expression* which can be *tainted*.

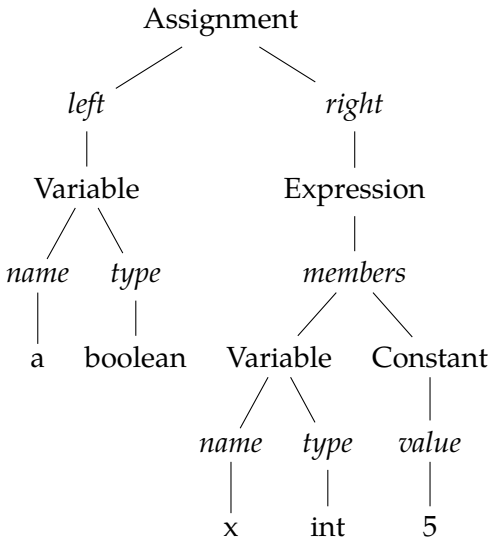

Figure 3.9: GAST representation of *"boolean a = x == 5;"*

Figure 3.10: GAST representation of a return statement

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*. Figure 3.11 shows the GAST representation of the throw statement.

Figure 3.11: GAST representation of a throw statement

Method Call

Element that represents method calls. It consists of a source, which can be an object or a class (if the method is static). Listing 3.4 shows examples of method calls where the source is an object and a class respectively. Figure 3.12 shows the GAST representation of the first example of listing 3.4.

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*

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

Listing 3.4: Method call examples

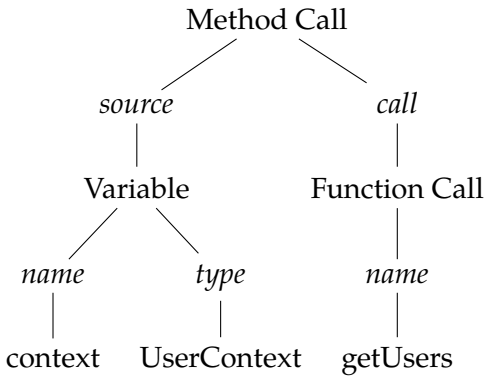

Figure 3.12: GAST representation of *"context.getUsers();"*

- the method is in the source code and returns a *tainted* value

New

Expression that represents an object creation. It is a subtype of *Expression* and works mostly as a function call to the constructor. Figure 3.13 shows the GAST representation of the *new* expression.

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*.

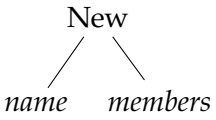

Figure 3.13: GAST representation of a new expression

Attribute access

Represents a direct access to an object attribute (e.g., *context.myProperty*). Figure 3.14 shows its GAST representation.

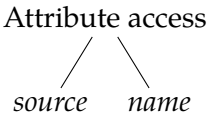

Figure 3.14: GAST representation of an attribute access

Code block

Element that represents a block of code. Consists of a list of statements. Listing 3.3 is an example of a code block with three statements and figure 3.15 shows the GAST of listing 3.3.

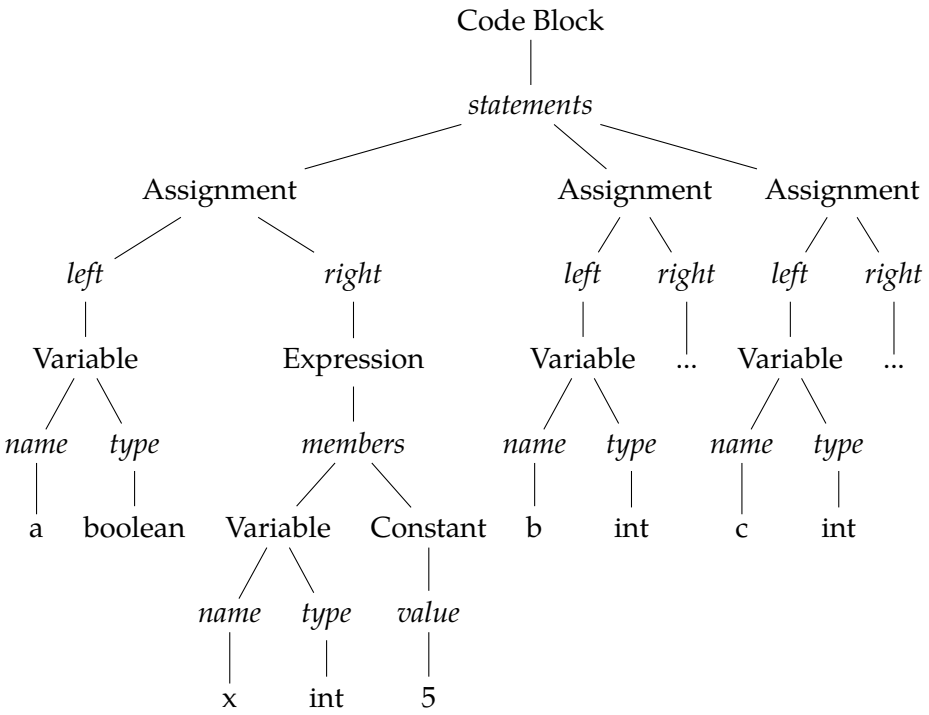

Figure 3.15: GAST representation of the code block from listing 3.3

Conditional statement

Element that abstracts loops (e.g., *for, while, do while* etc.). Consists of a code block and a condition, which is an expression. Figure 3.16 shows its GAST representation.

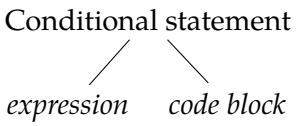

Figure 3.16: GAST representation of a conditional statement

Try Catch

Statement composed of a *try* code block, a list of *catch* code blocks and one *finally* code block. Figure 3.17 shows its GAST representation.

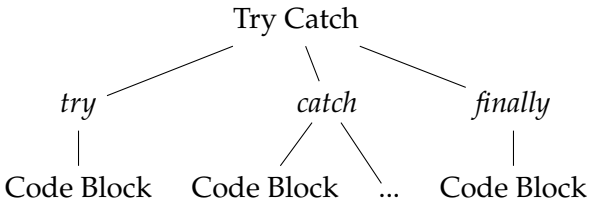

Figure 3.17: GAST representation of a try catch statement

If

Statement that abstracts 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*). Figure 3.18 shows its GAST representation.

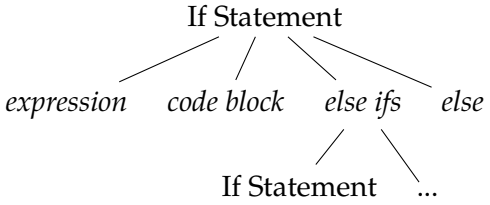

Figure 3.18: GAST representation of an if statement

Function

Element that represents a method or a function. It has a name, a return type, a list of parameters and a code block. Its representation can be seen in figure 3.19.

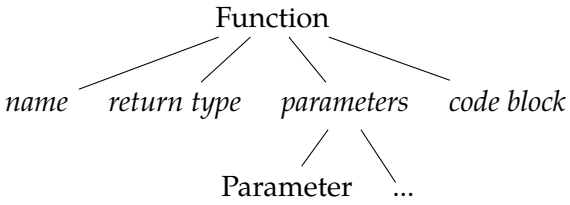

Figure 3.19: GAST representation of a function

Class

This element represents a class in the GAST. It has a name, a list of attributes and a list of methods. Furthermore, it can have a superclass, as shown in figure 3.20.

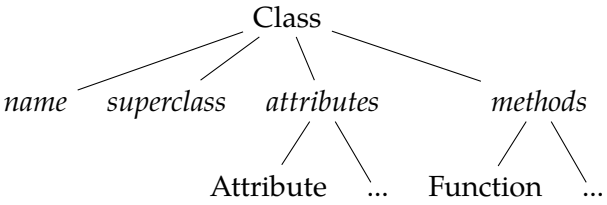

Figure 3.20: GAST representation of a class

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). Its structure is represented in figure 3.21

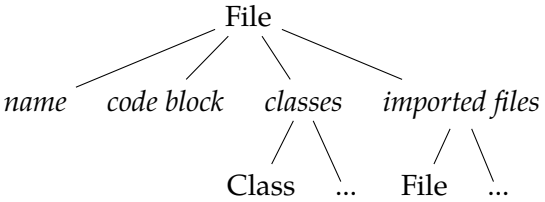

Figure 3.21: GAST representation of a file

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.

3.2.2 Structure

As an example of the structure of the GAST, consider listings 3.5 and 3.6. These listings present code with the exact same functionality written in two different languages (Java and PHP). The code is quite simple – a class with a method that executes an SQL query given an id. Parsing both listings with traditional parsers would result in quite different ASTs, despite the code being very similar. However, when converting the source code ASTs to GAST, their representation is almost identical. Figure 3.22 represents the GAST from listings 3.6 and 3.5 where the circled branches only appear in the Java representation since it is statically typed.

3.3 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

```
1 class MyDbClass
2 {
3     function getAddressById($id)
4     {
5         $sql = "SELECT address FROM users WHERE id = $id";
6         return executeSQLQuery($sql);
7     }
8 }
```

Listing 3.5: MyDbClass in PHP

```
1 public class MyDbClass
2 {
3     public static String getAddressById(String id)
4     {
5         String sql = "SELECT address FROM users WHERE id =" + id;
6         return executeSQLQuery(sql);
7     }
8 }
```

Listing 3.6: MyDbClass in Java

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 3.7 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 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

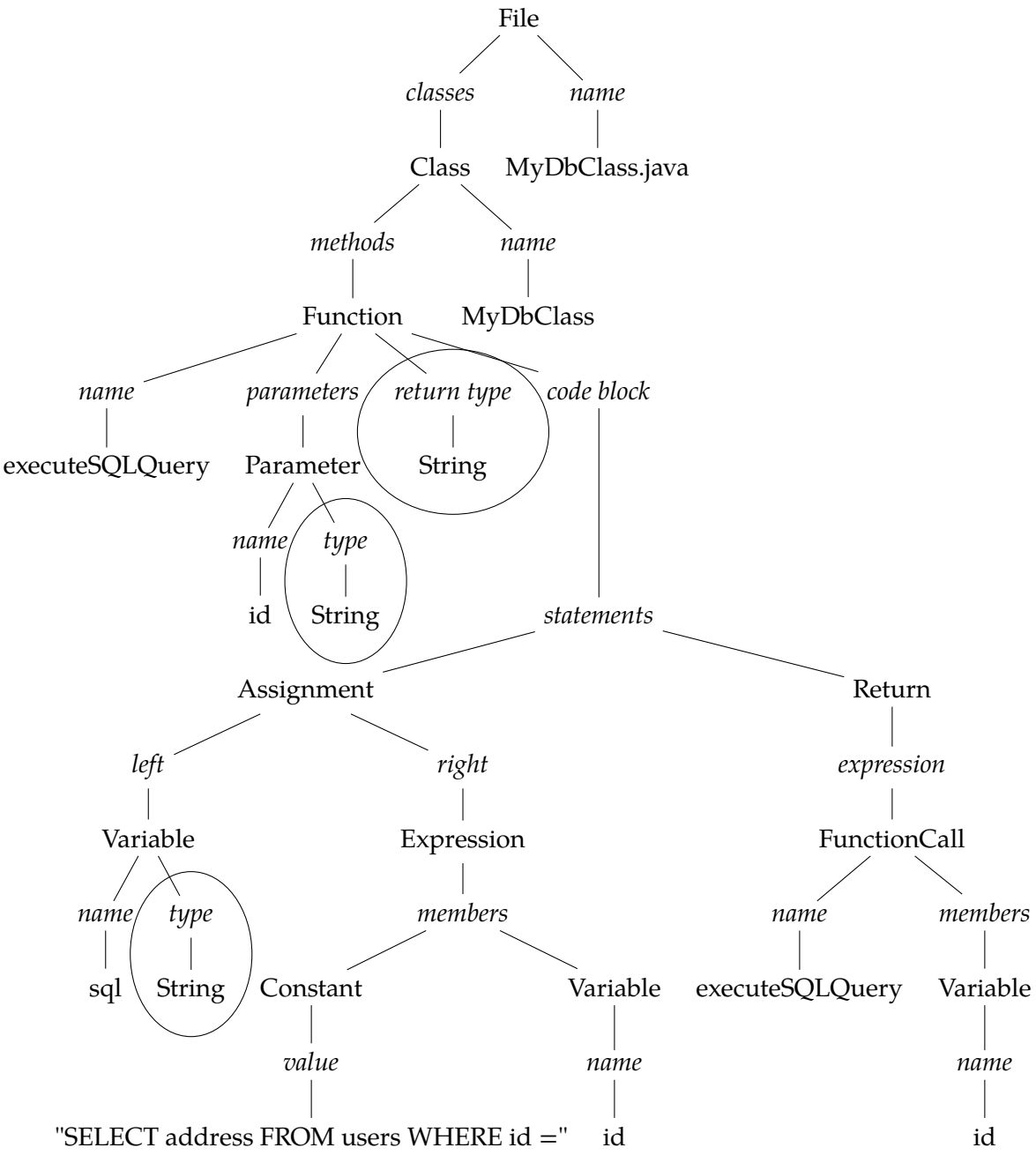

Figure 3.22: GAST representing the code from listing 3.6 and 3.5

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 3.8 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

```
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 3.7: Function declaration example

```
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 3.8: Assignment build call sequence

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 3.9 represents the resulting GAST and figure 3.23 represents the stack states when building the tree.

Converting the AST is as simple as identifying the methods needed to override from the generated tree walker, and then call the functions from the *GAST Builder*. For example, the converter for PHP has 67 lines of code (counting only statements), and from these 67 statements, there are 29 that are different, meaning that a lot of the functions invoked are the same (e.g., *exitStatementOrExpression* is invoked 23 times).

3.4 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*[SCD+13]. Also, the tool must perform a *path aware taint* propagation. Furthermore, it must be able to

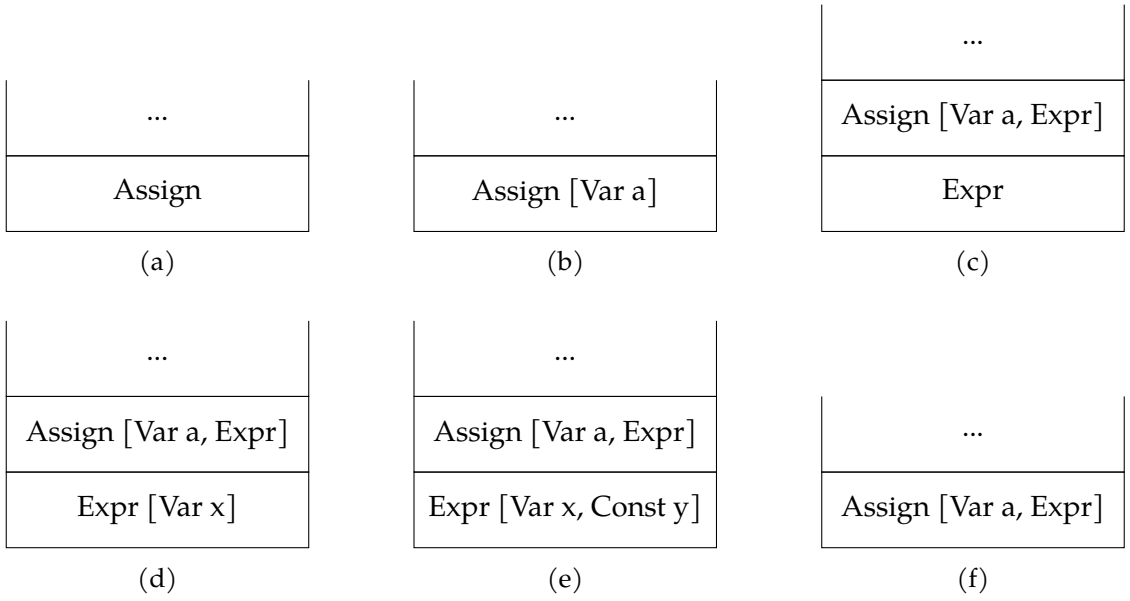

Figure 3.23: Stack state when executing code from listing 3.8

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.

3.4.1 Pointer information

To illustrate the importance of pointer information, consider the example from listing 3.9. 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 [SNQDAB16, Hin01]. 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.

```
1  String param = req.getParameter("name");
2  StringBuffer buf1;
3  StringBuffer buf2;
4
5  ...
6
7  buf1.append(param);
8  String query = buf2.toString();
9  con.executeQuery(query);
10
11 buf2 = buf1;
12 query = buf2.toString();
13 con.executeQuery(query);
```

Listing 3.9: Taint propagation example

```
1 String query = "SELECT * FROM users WHERE name=";
2 if (input != null){
3     query = query + input;
4 } else {
5     query = query + "Bob";
6 }
7 con.executeQuery(query);
```

Listing 3.10: Path propagation example

3.4.2 Path Analysis

In order to perform a precise static taint analysis it is very important to be able to track 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 [HSS08]. 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 3.10 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.

Let us now consider the example from listing 3.11. This example is almost identical to listing 3.10, 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 GT, and static analysis tools in general, struggle with detecting whether a condition can be true or not.

```
1 String query = "SELECT * FROM users WHERE name=";
2 input = null;
3 if (input != null){
4     query = query + input;
5 } else {
6     query = query + "Bob";
7 }
8 con.executeQuery(query);
```

Listing 3.11: Path propagation example with unreachable branch

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.

3.4.3 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 [RKG04, HKV14].

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.

3.4.4 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. They are often 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.

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

Listing 3.12: Vulnerability in while loop

To illustrate this issue, consider listing 3.12. In this example, if the *while* loop executes once, there is no 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.

3.5 Summary

This chapter presented the GAST structure and how it manages to abstract different languages by describing each node that can be part of it. Furthermore, we also introduced the architecture of the GT tool and its taint analysis features.

Chapter 4

Evaluation

This chapter presents the results of our taint analysis. We discuss the ability of GT to detect vulnerabilities and the effort needed to add support for new languages. Finally, we talk about some limitations that our implementation has.

4.1 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.

4.1.1 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 11 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 two real-world open-source web applications. Since these applications are much bigger, we assumed that every file from the application 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 4.1 and 4.2 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. We ran the tests on a computer with a Ryzen 1600 processor (6 cores, 12 threads at 3.6GHz) and 16GB of RAM.

Table 4.1: Deliberately insecure web applications

Application	**#loc**	**Language**	**Files**	**Vulnerabilities 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
Goof	571	JavaScript	8	3
NodeGoat	2697	JavaScript	49	4
Vulpy	2373	Python	57	6
Dvpwa	674	Python	21	7
Total	47233		899	78

Table 4.2: Real-world web applications

Application	**#loc**	**Language**	**Files**	**Vulnerabilities found**
SquirrelMail 1.5	46214	PHP	376	0
PhpMyAdmin 4.9.5	153576	PHP	740	1
Total	199790		1116	1

GT analyzed 2015 files and 247023 lines of code and managed to find 79 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.

Besides testing GT against web applications, we also have a set of 81 unit tests, with simple vulnerabilities, that run on each build of the tool. This way we have more confidence when we make a change to the taint analyzer.

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. Also, we had 1 false negative when testing PhpMyAdmin.

GT only raised 5 false positives in the applications from table 4.1, 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.

4.1.2 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 4.1 shows that our implementations are able to find vulnerabilities. Table 4.1 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 4.3 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 4.3: Converters size and implementation effort

Implementation 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

4.2 Limitations

In this section we present the limitations of our tool. Most of them are related to the nature of static taint analysis, such as propagating the data flow through impossible paths or the pointer analysis problem. Besides the limitations already discussed we also found another related to lambda functions. Since our taint analysis only propagates *taint* marks, we can not track the implementation of lambda functions. For example, consider listing 4.1, written in JavaScript, and assume that the *userInput* is tainted. In this case, we have an arrow function (lambda equivalent in JavaScript) x that accepts a parameter *query* and then executes this query against the database. This is a very simple program, with only two lines, however, GT struggles to find the vulnerability from line 3. This flaw is due to the fact that in our implementation, x is just a variable, it does not know that x is an object that holds a function. Furthermore, this variable x could be passed around as an argument to other functions. So, in complex cases, we quickly lose track of x and are unable to know what is the implementation of the function that x is holding.

```
1 var x = query => db.executeQuery(query);
2
3 x.call("SELECT * FROM users WHERE name=" + userInput);
```

Listing 4.1: JavaScript lambda example

This problem does not have an easy solution since it would require the tool to precisely track the flow of data in the application, which statically is virtually impossible.

Chapter 5

Conclusions

In this work, we presented a new approach to static taint analysis that supports the addition of new languages with little programming effort. We were able to achieve this by taking advantage of the fact that the programming languages used in web applications have many similarities between them. With this in mind, instead of analyzing the source code directly, we first parse the source code and then build a generic AST (GAST) based on it. After that, we traverse the GAST to find vulnerabilities, thus decoupling the analysis from the parsing. The GAST does not represent every detail of a language, instead, it contains only what is needed to perform the analysis. This allows it to be able to represent a large set of programming languages used in web applications. The only parts bound to the language being analyzed are the parser, which in our implementation consists of generated code, and the module that converts source code AST into the GAST, which is usually less than 110 lines of code.

The solution was implemented in the GT tool, using Java with parsers and tree walkers generated by ANTLR4. GT supports Java, PHP, Python and JavaScript and was tested against several web applications written in different languages. Based on the results of our tests and the number of languages supported, we consider that our goal was successfully achieved.

5.1 Future work

The presented work leaves room for several possible improvements that were not possible to develop due to time constraints. GAST could be extended to support even more languages and more features (e.g., adding support for lambda functions). The taint analyzer could also be extended. For instance, we could make it more accurate by using a more precise pointer analysis. Furthermore, we could also track the value of each variable, removing this way the lambda limitation described in section 4.2.

Bibliography

[ARF+14] 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.

[Ber17] Hal Berghel. Equifax and the latest round of identity theft roulette. *Computer*, 50(12):72–76, 2017.

[BK14] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. *ACM Sigplan Notices*, 49(10):83–101, 2014.

[BNE16] 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.

[BWS05] G. T. Buehrer, B. W. Weide, and P. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, pages 106–113, September 2005.

[Cas20] Stephen Cass. The top programming languages: Our latest rankings put python on top-again-[careers]. *IEEE Spectrum*, 57(8):22–22, 2020.

[CLO07] 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.

[CP13] G Campbell and Patroklos P Papapetrou. *SonarQube in action*. Manning Publications Co., 2013.

[DBE] Db-engines ranking - popularity ranking of database management systems. `https://db-engines.com/en/ranking`. (Accessed on 12/17/2020).

[DH14] 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.

[dSM16] Ibéria Vitória de Sousa Medeiros. *Detection of vulnerabilities and automatic protection for web applications*. PhD thesis, Doctoral dissertation, Universidade de Lisboa, 2016.

[Hin01] 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.

[HKV14] 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.

[HSS08] 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.

[JI11] 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.

[JKK06] 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.

[KMPS11] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Dta++: dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[LL05] 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.

[LYWS09] Anyi Liu, Yi Yuan, Duminda Wijesekera, and Angelos Stavrou. Sqlprob: a proxy-based architecture towards preventing sql injection attacks. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 2054–2061. ACM, 2009.

[Moh16] Sikender Mohsienuddin Mohammad. Continuous integration and automation. *International Journal of Creative Research Thoughts (IJCRT), ISSN*, pages 2320–2882, 2016.

[Mon] The most popular database for modern apps | mongodb. `https://www.mongodb.com/1`. (Accessed on 12/17/2020).

[NFV15] 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.

[NS05] 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.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[PMP11] 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.

[RKG04] 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.

[SAB10] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.

[SAP+11] 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.

[SCD+13] 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.

[SMS13] 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.

[SNQDAB16] 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.

[TPC+13] 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.

[Tra09] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, 2009.

[TW07] 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.

[WS08] 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.

[WW17] J. Williams and D. Wichers. OWASP Top 10 - 2017 rcl - the ten most critical web application security risks. Technical report, OWASP Foundation, 2017.