Whitebox Fuzzing for Web Application Security

Anabela Sofia Martins Borges

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Prof. Miguel Nuno Dias Alves Pupo Correia
Prof. Ibéria Vitória de Sousa Medeiros

Examination Committee
Chairperson: Alberto Manuel Rodrigues da Silva
Supervisor: Miguel Nuno Dias Alves Pupo Correia
Member of the Committee: João Carlos Serrenho Dias Pereira

June 2018
Dedicated to someone special...
Acknowledgments

I would like to express my gratitude to my advisors Miguel Correia and Ibéria Medeiros for the support and guidance given during this year.

I would like to thanks to Miguel Guerra for being my rock during these past years. This would have been much more difficult without you at my side.

I would like to thank my family for the support given all my life, and for being there when I needed.

Finally a special thanks to my friends, Artur, Raposo, Arvela, Gonçalo, Margarida, and the other Gonçalo, for being there to help when it was needed.
Resumo

Mesmo já havendo bastante pesquisa sobre segurança na web, a segurança em aplicações web continua a ser um problema relevante. O maior problema com este tipo de aplicações deve-se ao código fonte, que é escrito em linguagens inseguras, levando à existência de vulnerabilidades. Uma maneira de detectar vulnerabilidades no código fonte é utilizando análise estática, que é uma maneira de analisar código sem que este seja executado. No entanto, esta abordagem é falível, uma vez que este tipo de análise é efectuada em código sem este ser executado, fazendo com que haja falsos positivos e falsos negativos. Numa tentativa de melhorar estes resultados, pode-se usar concolic execution, a qual permite a uma ferramenta saber os valores injectados numa aplicação durante a sua execução, e analisar os caminhos de execução originados por estes valores.

Neste sentido, pode-se obter uma análise estática mais refinada e consequentemente, encontrar-se mais caminhos para serem explorados, utilizando-se a técnica de programação com constraints. Programação com constraints é uma linguagem declarativa que expressa a lógica entre variáveis sem descrever os caminhos percorridos pela aplicação.

BaZINGA é uma ferramenta que foi desenvolvida de modo a conseguir detectar vulnerabilidades em aplicações web escritas em PHP. Para se poder desenvolver esta ferramenta, foi necessário estudar várias combinações das técnicas faladas anteriormente.

Palavras-chave: whitebox testing, fuzzing, vulnerabilidades, segurança em aplicações web, constraint programming
Abstract

Despite the research done in web security in the past few years, the security of web applications remains an important concern. The major issue with these applications comes from the source code itself, that is often written in unsafe languages, leading to the existence of vulnerabilities. One way of finding vulnerabilities in the source code is by performing static analysis, which is a way of analyzing the code without its execution. However this approach is unreliable, since it does the analysis on the code without running it, which generates false positives and false negatives. In an attempt to improve these results, concolic execution can be used, which allows the tool to know the inputs injected during the application’s execution, and analyze the control flow originated from these values.

With this in mind, it can be achieved a more refined static analysis, and consequently find more execution paths by using constraint programming. Constraint programming is a declarative language that expresses the logic between the variables without describing its control flow.

BaZINGA is a tool that was developed to detect vulnerabilities in web applications written in PHP. To be possible to develop this tool, it was needed to study several combinations of these techniques.

Keywords: whitebox, fuzzing, vulnerabilities, web application security, constraint programming
Contents

Acknowledgments ................................................................. v
Resumo ................................................................. vii
Abstract ............................................................... ix
List of Tables ............................................................ xiii
List of Figures ............................................................. xv
Nomenclature ............................................................. 1
Glossary ................................................................. 1

1 Introduction ................................................................. 1

2 Background ................................................................. 3
   2.1 Web applications surface vulnerabilities ..................................... 3
   2.2 Static analysis for security .................................................... 4
      2.2.1 Pixy ............................................................... 5
      2.2.2 String-taint analysis to find XSS ................................... 5
      2.2.3 WAP ............................................................. 6
   2.3 Fuzzing ................................................................. 7
      2.3.1 KameleonFuzz ...................................................... 7
      2.3.2 LangFuzz ......................................................... 8
   2.4 Grey-box testing for security ............................................... 9
      2.4.1 Dowser .......................................................... 9
      2.4.2 jĀk ............................................................... 9
   2.5 Constraint languages and solvers for security testing .................. 10
      2.5.1 Satisfiability and solvers ......................................... 10
      2.5.2 Symbolic execution and white box using solvers ................ 13

3 BaZINGA Design ............................................................ 19
   3.1 The BaZINGA approach .................................................. 19
List of Tables

5.1 Samate results .................................................. 37
5.2 DVWA results .................................................... 38
# List of Figures

2.1 PHP script vulnerable to XSS. .................................................. 4  
2.2 Aliases in PHP [JKK06]. ...................................................... 5  
2.3 String-taint analysis [WS08]. ............................................... 6  
2.4 A constraint in JaCoP ........................................................... 11  
2.5 Code snippet [GLM12] ........................................................... 14  
3.1 BaZINGA architecture .......................................................... 21  
3.2 Example of a trace of an execution of the code from Figure 2.1. .......... 24  
3.3 Example of transition from a condition to the intermediate language. .... 26  
4.1 Example of transition from the intermediate language to the stacks. ........ 32
Chapter 1

Introduction

Despite much research done in web security in the past few years, the security of web applications remains an important concern [WW17, Imp17]. A major issue with these applications comes from the source code itself, that is often written in unsafe languages like PHP or JavaScript [JKK06, MNC14], leading to the existence of vulnerabilities. Conspicuous examples of vulnerabilities are those that allow SQL injection and cross-site scripting (XSS) attacks [WW17].

The most sustainable approaches for protecting web applications are those that allow removing such vulnerabilities. Static analysis tools detect vulnerabilities in source code, thus allow cleaning vulnerabilities [JKK06, WS08, MNC14, MNC]. Testing, including fuzzing, achieve a similar goal by executing and exercising the applications, with similar problems [MFS90, GLM12, DCKV12, DRRG13, DRRG14, WCWL17]. Symbolic execution may also be used to find vulnerabilities in code, but differently from the other approaches it exercises applications with symbolic values [GLM+08, CDE+08]. There are other approaches that dynamically block attacks [BK04, BWS05, HO05, MS15, MBNC16], among which the widely adopted web application firewalls [WAS06, OWA08]. These tools and mechanisms are invaluable, but they tend to generate too many alarms and/or miss some vulnerabilities or attacks.

The goal of the dissertation is to improve the detection of vulnerabilities in PHP code, the language most used to implement web applications, at least until recently [Imp12]. For that purpose, we present a whitebox fuzzing approach and a tool called BaZINGA that combines static analysis and concolic execution [GLM+08, CDE+08] with fuzzing. Concolic execution combines both concrete execution (i.e., normal code execution) and symbolic execution, envisaging that the former drives the latter with the assistance of constraint programming [KS, MSKS14, DMB08, BFT15, TCJ14].

The approach involves injecting inputs in the web application under test (fuzzing) to perform concolic execution, leveraging from constraint programming to guide this injection and achieving
a concrete execution of the application, and uses the code from concrete executions to analyze it statically for discovering vulnerabilities, and to solve the constraint paths found in it by symbolic execution for obtaining new inputs to be injected next. The approach works in loop, iteratively to inject the new inputs and in order to cover all existing code branches.

This document presents the implementation of the BaZINGA tool and an experimental evaluation with synthetic code and open source applications. In the evaluation, the tool was able to detect all vulnerabilities as it was able to explore different control flow paths. In the evaluation the tool was also able to cover 100% of the code of the analyzed applications.

The contributions of the dissertation are:

1. an approach for improving software security based on a combination of concolic execution and static analysis with fuzzing to detect and identify vulnerabilities in software;

2. the BaZINGA whitebox fuzzing tool that implements the approach.

The remaining of this thesis is organized as follows. Section 2 presents the related work about detection of vulnerabilities in software using different approaches. Section 3 presents BaZINGA design, and Section 4 its implementation. Section 5 presents and discusses the evaluation results. Section 6 concludes the dissertation and presents ideas for future work.
Chapter 2

Background

There are many ways to analyse a program in order to automatically identify problems in the source code. A program analysis can be achieved by using static analysis, dynamic analysis or a combination of both.

This section addresses these topics in order to get better understanding of the related work regarding this thesis. The idea is to present the various tools that do analysis of vulnerabilities, from the simpler ones that only use one technique to more complex ones that are hybrid.

Section 2.1 presents some of the existent vulnerabilities in web applications. Section 2.2 presents the topics of static analysis, tainted analysis and various tools that use static analysis. Section 2.3 exposes tools that use fuzzing. Section 2.4 presents several tools that use grey-box testing. Section 2.5 starts by presenting the topics of satisfiability, solvers and several solvers. The second part of this section presents symbolic execution and some tools that use it.

2.1 Web applications surface vulnerabilities

This subsection aims to give a brief presentation of web application vulnerabilities, focusing on the reflected cross site scripting (XSS) and SQL injection (SQLI) classes of vulnerabilities, which are the ones most exploited [WW17], and those handled by the BaZINGA tool.

Despite the efforts that have been made to protect web application code, the difficulty of protecting the user inputs (e.g., $_GET) remains, leaving many applications vulnerable and an easy target for attackers. Attackers inject malicious inputs though the application attack surface and verify if these inputs exploited some vulnerability existent in the code. Vulnerabilities associated to user inputs are called input validation vulnerabilities because user inputs are improperly validated or sanitized, or surface vulnerabilities because they are exploited through inputs that are inserted through the attack surface of a program.
SQLI and XSS are two classes of vulnerabilities of this kind. SQLI is associated to malcrafted user inputs that combine normal characters with metacharacters or metadata (e.g., ', OR) which are used in SQL queries without any protection, and then that queries are sent to databases to be executed through a sensitive sink (e.g., mysqli_query). XSS is also associated to malcrafted user inputs, but differently from SQLI, they are injected under scripts (e.g., JavaScript scripts) and used in output functions (e.g., echo), for example, allowing the exploitation of vulnerabilities that reflects the browser data of the victim computer to the attacker.

Figure 2.1 shows a PHP script vulnerable to XSS. The program receives the name and age of a user (lines 2 and 3) to check if their age ranges from 6 to 80 years old (line 5) by outputting a welcome message in such a case or an error message. The code contains three vulnerabilities, in lines 4, 6, and 8. All these lines contain the echo sensitive sink and the $age or the $name variables as parameter, which received the user inputs.

```php
1 if( $_GET["name"] || $_GET["age"] ) {
2 $name = $_GET["name"];
3 $age = $_GET["age"];
4 echo "Welcome ". $name. "<br />";
5 if (($age > 5)&&($age<= 80)) {
6 echo "You are ". $age. " years old<br />";
7 } else {
8 echo $age . "is not within range<br />";
9 }
```

Figure 2.1: PHP script vulnerable to XSS.

The exploitation of such vulnerabilities can have devastating consequences and costs for organizations.

2.2 Static analysis for security

One of the ways to detect vulnerabilities is to use a static analysis tool to check the source code without executing it [CM04, JKK06]. A static analyser uses a set of rules to verify the existence of vulnerabilities, so if a rule does not exist in this set, the vulnerability is never found.

Like it was said in Section 2.1, input validation vulnerabilities are vulnerabilities that exist due to improper handling of user input. XSS is an input validation vulnerability that allows attackers to execute scripts, typically in JavaScript, in a victim’s browser. There are three types of XSS: reflected or non-persistent, stored or persistent, and DOM-based (Document Object Model). A reflected attack is done by clicking a URL usually obtained from an e-mail or a web page. This URL points to a vulnerable web application but it also contains a script to execute the attack. After clicking, the browser executes the script. A stored attack is similar but instead
of having the script on a URL, the script has been previously stored on the web application.

2.2.1 Pixy

Jovanovic et al. [JKK06] implement the concept of alias analysis in Pixy, an open source tool for identifying input validation vulnerabilities using static analysis, in order to detect vulnerabilities in web applications, more specifically cross-site scripting (XSS) and SQL injection.

The authors introduce the notion of alias analysis due to the existence of false positives (non-vulnerabilities) and false negatives when in the presence of aliases of a variable in PHP. In alias analysis, it is considered the relationship between variables i.e., two or more variables are aliases if they share the same memory location. The variables can be must-aliases, if they do not depend of the path taken during the program, or may-aliases, if variables are only aliases in some paths.

In PHP, aliases can be created by using the reference operator &. This operator can be used directly in assignments or in parameters of functions to make a call-by-reference. Figure 2.2 shows an alias relationship between \$a and \$b that is created in line 2. If the static analyser does not know of the existence of alias, it does not know that the assignment in line 3 also affects \$a and as a result it misses the XSS vulnerability that is shown in line 4.

```
1: \$a = 'untainted';
2: \$b = & \$a;
3: \$b = $tainted;
4: echo \$a;
```

Figure 2.2: Aliases in PHP [JKK06].

With the introduction of aliases, it was possible to discover new vulnerabilities with a lower rate of false positives (50%). The majority of the remaining false positives (38 of 57) were due to impossible paths.

2.2.2 String-taint analysis to find XSS

Wassermann et al. [WS08] use a string-taint analysis to find invalid string values, followed by a policy for web pages to include only trusted scripts. This approach starts by translating output statements into assignments, and translates the program into a static single assignment that encodes data dependencies. Next, due to the encoded data dependencies, the control structures are dropped, the assignment statements are translated into a grammar production, and untrusted data sources are labeled. From this phase, it generates an extended context free grammar (CFG). Figure 2.3 shows an example of a program after the first and second phases of
a string-taint analysis. In the last phase, it constructs a CFG from the extended one.

```php
$data1 = $REQUEST['module'];
$data2 = preg_replace(
    '/([\s\"]+[\n\w]\s*=\*/i,
    "HordeCleaned=", $data1);
$module = $data2;

if ($use) {
    $output1 = '<br>';
} else {
    $hiddenfields = "<input value='$module'/>";
    $output2 = '<div>' . $hiddenfields;
}
$output3 = $output1 . $output2;
```

Figure 2.3: String-taint analysis [WS08].

The biggest difficulty in preventing XSS is that every browser supports different ways to invoke JavaScript, so every way to invoke it has to be taken into account. Also, a web application may allow that every HTML character may have some use, so no character can be ignored. In order to know the ways the HTML document invokes the JavaScript interpreter, several documents were examined. Next, with the information gathered, they created lexical rules for the invocation of the interpreter.

Wassermann et al. found unreported vulnerabilities in Claroline, a PHP web application. The tool found 32 true input validation vulnerabilities where CVE-2005-1374 only lists 10. However, it failed to analyse some applications because in some cases it could not resolve certain aliases relationships and in other cases it exceeded the memory limit.

2.2.3 WAP

Data mining is a technique to automatically obtain information from large data sets. Medeiros et al. [MNC14] describe the Web Application Protection (WAP) tool, which complements taint
analysis with the usage of data mining to detect and eliminate input validation vulnerabilities in PHP 5 code. Because taint analysis tends to report many false positives, the usage of data mining is helpful in trying to predict these types of mistakes. When using data mining to discover a false positive, WAP needs to collect the attributes of the code that are usually associated with the existence of false positives. After that, a classifier is used to signal a vulnerability as false positive or not.

WAP involves detecting and correcting vulnerabilities, and for that it needs to find problematic information flows and then modify the code to block them.

In the taint analysis step, the code is parsed and generated an abstract syntax tree (AST). At the beginning, every symbol is untainted with the exception of the entry points. While going through the AST, a tainted execution path tree is built, where each branch of this tree corresponds to a tainted variable. If a variable is tainted, the state of tainted is passed to the symbols that depend of it. The state of tainted only stops being propagated if the variable is sanitized.

For the data analysis step, parts of code are classified as vulnerable or not manually. Afterwards machine learning is used to configure the WAP with the acquired informations, which will then be used to analyse the code. WAP does the correction of the code after it detects vulnerabilities. That step is done by the taint analyser upon returning the data about vulnerabilities. The fix is then inserted in the code and finally returned to the user. WAP was tested with 35 large scale applications and 294 vulnerabilities were found, with at least 28 false positives.

2.3 Fuzzing

Black-box fuzzing is a type of technique that, without knowing the source code, gives to the program random inputs to find vulnerabilities in it. There are two types of fuzzing: generative and mutation. Generative fuzzing creates random inputs, that may follow some construction rules. Mutation fuzzing deduces inputs from previous data, while making some random modifications. The specific values (inputs) that can be used to parameterize or replace part of an entry point in an attack are called *fuzz vectors*.

2.3.1 KameleonFuzz

KameleonFuzz [DRRG14] is an extension of LigRE [DCKV12, DRRG13], a black-box fuzzer that performs control flow to detect reflected and stored XSS vulnerabilities. It is composed by four steps:

1. Control flow inference
2. Approximate taint flow inference

3. Chopping

4. Evolutionary fuzzing

The first three steps are components of LigRE. The last step is subdivided in two sub-steps: malicious input generation, and precise taint flow inference.

In the control flow inference step, the KameleonFuzz constructs a control flow model from input parameters in order to interact with the application.

In the approximate taint flow inference step, it infers the data flow from the control flow model. It generates paths to navigate in the control flow model, and submits these paths to the application to observe data flows.

In the chopping step, it selects from the paths created in the previous step, the shortest ones between the same two points.

In the malicious input generation sub-step of the evolutionary fuzzing component, it uses an attack grammar to generate fuzzed values. An attack grammar is a grammar composed of attack vectors, which are ways to carry out an attack. This attack grammar, in particular, is a tree that generates rules based on XSS exploits. To create a fuzzed value from an attack grammar, it needs to travel through the rules, and if needed, make choices. In the second sub-step, precise taint flow inference, it verifies if the fuzzed value triggers a vulnerability.

LigRE+KameleonFuzz encountered more XSS vulnerabilities than other black-box scanners. From a total of 35 vulnerabilities, LigRE+KameleonFuzz was able to find 32.

2.3.2 LangFuzz

LangFuzz [HHZ12] is a black-box fuzzing tool for script interpreters. It generates semi-random code according to a received grammar, which will then be injected in an interpreter. It is considered semi-random code because the generation process uses code known to cause invalid behaviors in a specific interpreter. LangFuzz is not tied to a specific programming language, so it uses a context-free grammar based on general language assumptions. It uses mutation fuzzing as a primary approach due to its ability to generate complex programs without the need to have semantic rules for a specific language.

The mutation of code in LangFuzz has two phases, a learning phase followed by a mutation phase. In the learning phase, it receives files to be parsed, then build a fragment pool with the parsed files. In the mutation phase, it selects one file to be parsed, while also selecting randomly some fragments from the fragment pool. Then it replaces parts of the file with fragments of the
same type.

LangFuzz is a fuzzer that can easily be adapted to different languages and it is useful as a complement for other tools when trying to find security issues.

2.4 Grey-box testing for security

A tool is said to do grey-box testing if it combines a static analyser with fuzzing. Unlike the black-box fuzzing, a grey-box fuzzing tool has knowledge of the source code, as such it can choose the values to injects from its knowledge of the source.

2.4.1 Dowser

Dowser [HSNB13] is a grey-box fuzzer that combines taint analysis, program analysis (static analysis) and symbolic execution, in order to find buffer overflow vulnerabilities. Haller et al. use this combination because when a static analysis or a symbolic execution tool executes a complex program, for example, programs with chained if statements, it cannot find the bugs because of its undecidability.

First, Dowser performs a data flow analysis on the program and ranks the instructions that access arrays/buffers in loops. In the second step, Dowser selects the instruction with higher ranking. Then, it uses taint analysis to select the inputs that affect the possible vulnerable memory. On the third step, for each instruction that accesses an array, it uses symbolic execution to try to lead the program towards an overflow. In the final step, Dowser uses Google’s AddressSanitizer to detect if there is an overflow.

Dowser has several novelties, but the main contribution is a new fuzzing approach (described above) that is usable in real applications and discovers complex bugs that in other way would be almost impossible to find.

2.4.2 jÅk

Pellegrino et al. [PTBR15] propose jÅk, a tool that combines web application crawling with dynamic program analysis.

jÅk is composed by four modules: dynamic analysis, crawling, attacker, and analysis modules.

It uses dynamic program analysis to monitor the execution of the JavaScript program. This type of analysis can be done in three ways: modifying the JavaScript interpreter to inspect and monitor the execution of the program, inserting calls to functions on the client-side source code,
or by monitoring the behavior of a program by catching calls to APIs inside the JavaScript execution environment.

The typical web application crawler finds new URLs by matching patterns with the HTML content of websites. However jÅk does not use a typical crawler, uses a model-based crawler, which is a crawler that creates and maintains a model of the web application. This model is used to choose which path to explore.

From a seed URL, jÅk gets the client-side program and passes it to the dynamic analysis module which returns traces that are used to create a list of forms and URLs. After having the lists, the attacker and analysis modules search for vulnerabilities on the server side. So, for each URL, the attacker module prepares the URLs to carry the attack. After that, jÅk passes the URL as input to the dynamic analysis module. Finally, the analysis module analyses the execution trace, which was returned from the execution of the dynamic analysis, to decide if there is a vulnerability or not.

2.5 Constraint languages and solvers for security testing

*Constraint Programming* is a paradigm where the relations between the variables are expressed by constraints. Constraints specify the properties of a solution, so constraint programming is a declarative language that expresses the logic without describing its control flow.

The constraints are typically used in specific domains such as boolean, integer, rational, linear, and finite domains. All of these domains are commonly solved by satisfiability modulo theories and the boolean domains are solved by the satisfiability problems.

This sections starts by presenting the satisfiability problems and tools that solve that type of problems, the solvers. In the second part, it discusses various tools that use symbolic execution, white box testing and solvers.

2.5.1 Satisfiability and solvers

The boolean *satisfiability problem* (SAT), abbreviated as satisfiability, is the problem of determining if there is an interpretation that satisfies a boolean expression. That is, if it is possible to replace the values true and false of the variables of a given boolean formula in a way that the formula is always true. For example, the \( p \land \neg q \) expression is satisfiable because when the value of \( p \) is true and the value of \( q \) is false the expression is true, however, the \( (q \land \neg p) \land (\neg q \land \neg p) \) expression is always unsatisfiable for the same values of \( p \) and \( q \) variables.

There are extensions to SAT, in which one of the most famous is the *Satisfiability Modulo Theories* (SMT). SMT is a decision problem that studies methods for checking if a logic formula
is satisfiable in respect to some background theory. The major difference between SMT and SAT is that the predicates are not exclusively binary values in SMT, and as such some of the binary variables can be replaced by predicates that have a set of non-binary values such as inequalities. For example, the $x + 2y \geq 4$ inequality is a non-binary predicate that is not binary.

A solver is a tool that is capable of solving satisfiability problems.

**JaCoP** JaCoP [KS] is a Java-based solver that provides a constraint programming language that solves SAT problems, so it has primitive constraints, such as equality and inequality. It also uses global constraints such as logical, reified and conditional constraints, and it defines decomposable constraints. If a constraint $C$ is reified by a boolean $b$ then $b$ is true if $C$ holds, i.e. $C \iff b$. Decomposable constraints are constraints that are defined using other constraints, and they can also be called recursive constraints.

In JaCoP, a finite domain variable (a variable that can only have a set of values as domain) is created by using the `IntVar(store, "var", db, de)`, where `store` is an instance of class `Store` which is a class that saves the values of the variables, "var" is the name of the variable and `db` and `de` are the limits of the domain. The domain of a variable can be expanded by using the method `<var>.addVar()`. To create constraints, it uses the function `impose` if it is a primitive or a global constraint, or it uses `imposeDecomposition` if the type of the constraint is decomposable. To verify the consistency of the constraints, the function `consistency` is used. There is a function, `imposeWithConsistency`, that modifies the domain while verifying its consistency.

In Figure 2.4 there is an example of how to create a constraint in JaCoP. First, it needs to create the objects, in this case `x1` and `x2`. After that, it creates a constraint. This constraint means that the the value of `x1` and `x2` has to be the same, and the only way possible with their domains is if `x1` and `x2` have the value 3.

```plaintext
IntVar x1 = new IntVar(store, "x1", 2, 3)
IntVar x2 = new IntVar(store, "x2", 3, 4)
store.impose(new XeqY(x1, x2))
```

Figure 2.4: A constraint in JaCoP

**MiniZinc** MiniZinc [MSKS14] is a solver based on SMT that was designed for decision problems over integers and real numbers, while also being capable of performing optimizations. MiniZinc does not solve the constraint problem; however it can give annotations to the solver to help it solve the problem. MiniZinc was created to interact with different solvers.

MiniZinc creates a variable by using `var type: name;` or `var domain: name;`, where
domain is the domain of variable named name and type is the type of the variable. A constraint `exp1 op exp2;` creates a constraint that compares `exp1` with `exp2` using the operator `op`. The function `assert(boolean, string)` dictates that when the boolean expression is false, the execution of the program is aborted and prints `string` as the error message. To solve the existent constraint, it uses `solve satisfy`. If it wants to maximize or minimize the satisfy value, it adds `maximize expression` and `minimize expression` respectively.

The instantiation of arrays is done by using:

```
array[domain_1, ..., domain_n] of variable: name,
```

where `variable` is one of the definitions for variable. A loop is called by using:

```
forall (var_1, ..., var_n in domain where expression) (body),
```

where `domain` is a set of values that iterate `var_1` and `where expression` is a optional condition for comparing `var_1`.

**Z3**

Z3 [DMB08] is a SMT solver that targets problems in software verification and analysis. The front-ends interact with Z3 by using either the textual format, such as the SMT-LIB format, or the binary API. Z3 can also be called in diverse languages by using an API in that language.

SMT-LIB [BFT15] is a library and solver that was created with the purpose of having available common standards that facilitate the evaluation and comparison between the different SMT solvers. SMT-LIB defines a language for interacting with SMT solvers, using a list of functions, that in the end are capable of verifying if the formulas created are satisfiable.

**Z3 is composed by:**

- Simplifier, the first step of the Z3, which simplifies the input formulas like contextual simplifications such as `q = 2 ∧ f(q)` which simplified is `q = 2 ∧ f(2)` because `q` in `f(q)` is substituted with `2`.

- Compiler, which converts the AST in a different data structure made of clauses and congruence-closure nodes.

- Congruence closure core, that receives assignments to atoms from a SAT solver. The atoms can be equalities or theory specific atomic formulas, such as inequalities.

- SAT solver

- Theory solvers, composed by several solvers such as linear arithmetic, bit-vector, arrays and tuples, that are used to solve equality assignments.
The assignments of equalities from the SAT solver are propagated using the congruence closure core. After the propagation, these assignments may point to one or more theory solvers. The core also propagates effects from the theory solvers.

**S3** Trinh *et al.* propose S3 [TCJ14], a solver which can be seen as an extension of Z3. As an extension of Z3, it supports the primitive type string, i.e., supports variables with size unknown, and is capable of using multiple theories, meaning it has the capability of evaluating strings and non-strings simultaneously. Z3-str acts as the part of Z3 that solves the part of string theory for the SMT theory solver. S3 uses Z3-str-star (Z3-str-*), an improved version of Z3-str, that besides the functionalities of Z3-str, it supports recursive functions. There is a need for improvement in the Z3-str because Z3-str does not have informations about the equalities, so it does not know anything about comparing the size of a string.

S3 reduces a given formula $F$ into a *reduced* $F$ formula using certain rules. This *reduced* $F$ is a list of constraints that Z3-str-* can solve. When the reduced formula is a logical formula, it needs to be normalized. The rules differ based on the type of constraint that is given. For regular expressions there are four rules. If it is a string constant, the expression is the constant. The union verifies if a string expression is a member of a regular expression, and the concatenation splits the expression in two sub-expression and verifies if the two expressions belong to the original expression. The last rule is the case where the regular expression repeats itself 0 or more times. There are also reduction rules for String operations, such as `search` and `replaceAll`.

### 2.5.2 Symbolic execution and white box using solvers

The way of analysing the source code of a program in order to determine which inputs cause each part of a program to execute is called symbolic execution. When the execution path diverges, the symbolic execution creates a fork in the path and it adds a constraint. A constraint represents the point in the program where the path diverges, and depending on either this condition is true or false, the chosen path is different.

**SAGE** Godefroid *et al.* [GLM+08] propose Scalable, Automated, Guided Execution (SAGE), a whitebox fuzzing tool that detects vulnerabilities on x86 Windows programs, more specifically on file-reading applications.

SAGE is an implementation of a search algorithm. The algorithm is divided in two parts: search and expand execution. The search part places the seed input in a list and runs the program to check if there is any bug in the first execution of the program. After that, an input from the list is chosen and expanded to generate the new elements of the list using the function
**ExpandExecution.** Finally, the program runs using as input every new element generated. At the end of each run, it verifies if there is any error, gives scores to the inputs, and sorts the inputs according to the scores. The scores given to the new inputs depend on the block coverage. The bigger is the block coverage, the bigger is the score. The next input to be expanded is the one with the biggest score.

**ExpandExecution** executes the program symbolically using the input and generates a path constraint. The input of order \( j \) is modified by making the path constraints from 0 to \( j - 1 \) equal to the input evaluated and \( j \) false.

```c
void func(char input[4]) {
    int count = 0;
    if(input[0] == 'b') count++;
    if(input[1] == 'a') count++;
    if(input[2] == 'd') count++;
    if(input[3] == '!') count++;
    if(count >= 3) abort();
}
```

Figure 2.5: Code snippet [GLM12]

Assuming that the code in the Figure 2.5 receives as initial input `good`, after the first symbolic execution of the program it will generate inputs that differ from the original input in one character, for example `goo!` or `bood`.

SAGE introduces a new algorithm called *generational search*, that instead of systematically trying to find all the possible paths, uses the same input until the possible paths from this input are exhausted, starting then with a new seed.

**KLEE** Cadar et al. [CDE+08] proposed a new symbolic execution tool named KLEE, for detecting vulnerabilities in C. During the execution of the program, when KLEE detects an error or a return call, it produces a test case that follows the same path constraint as the one that was used during the execution.

The symbolic execution of most instructions in KLEE is simple, as it only needs to execute the instruction. However in case of conditional branches and potentially dangerous operations, it needs to do more. When there are conditional branches, it receives a boolean expression and alters the path of the symbolic process depending if the expression can be evaluated or not. In the constraint solver, if a branch condition can be evaluated as true or false, it chooses the appropriate path, otherwise, both paths are added to the paths that are possible to explore. The same process is applied to the potentially dangerous operations; however it creates a branch depending on the value that causes the error.
WAPTEC  Whitebox Analysis for Parameter Tampering Exploit Construction (WAPTEC) is a tool proposed by Bisht et al. [BHSV11], that checks the existence of vulnerabilities in web applications by using symbolic execution and dynamic analysis. It targets vulnerabilities in Linux, Apache, MYSQL, and PHP (LAMP).

WAPTEC’s objective is to identify the inputs that the client rejects but the server accepts. To solve this problem, there are two phases. In the first phase, it finds paths that lead to sensitive operations, and in the second phase, it finds inputs that lead to the sensitive operations but that were rejected by the client. It is considered that every input accepted or rejected by the client will, respectively, be accepted or rejected by the server.

In the first phase, WAPTEC begins by analysing the client’s code (that can be written in either HTML or JavaScript) and extracting constraints. Then it uses a constraint solver to discover an input that satisfies the existent constraints and sends this input to the server. When the server receives the input, the code is executed using a trace analyser to check if it reaches a sensitive sink or not. If this input does not reach a sensitive sink, it is because the server has more constraints than the client, so WAPTEC analyses the execution trace to find and add more constraints, modifying the input while taking into account these new constraints. This process of finding new inputs and constraints repeats until it is found an input that reaches a sensitive sink. Once it reaches a sensitive sink on the server, WAPTEC will try to find more by adding a negation for every constraint found.

In the second phase, for each path that leads to a sensitive sink, WAPTEC tries to generate an input that should not be accepted by the client but that could be accepted by the server. In order to do that, it needs to find solutions that are accepted by the server but are incompatible with the client, so it has to be a solution of ¬Constraints\textsubscript{client} ∧ Constraints\textsubscript{server}.

PHPQuickFix and PHPRepair  Samimi et al. [SSA+12] propose two tools that target the problems of HTML generation in PHP programs. The first tool is PHPQuickFix which is a static analyser that checks the existence of simple bugs in HTML.

PHPQuickFix tries to parse each string constant and inline HTML fragment in the program without knowing the context. While it parses the program, it maintains a stack of the elements that are currently open. When it finds an end tag that does not pair with the first element of the stack, it proposes the insertion of a quick-fix on the place that is currently being analysed. However, because PHPQuickFix only considers each constant in isolation, it cannot detect errors that spread through different parts of the code.

PHPRepair is the second tool proposed. It is a solver based on an execution of a test case, where a test is the concatenation of n statements and has an expected output s\textsubscript{1}, \ldots, s\textsubscript{n} = output.
With the replacement of each statement $s_i$ with a constraint $c_i$, the solution to the equation of the constraints $c_1, \ldots, c_n$ is the way to repair the program to satisfy the test.

**Praspel** Enderlin et al. [EGB13] developed a constraint solver for arrays in Praspel, a language and a framework for contract-based testing in PHP, based on realistic domains.

Realistic domains are domains used for test generation, and their purpose is to specify the values that can be assigned to a data in a given program. The two features that a realistic domain has are predicability and samplability. Predicability is the way to verify if is a given predicate belongs to a realistic domain, and samplability is the capability of generating a domain based on values, for example, a regular expression.

Initially, it transforms the arrays in conditions following certain rules. After that it transforms the conditions in constraints. The possible constraints are the array size, the domain and co-domain, and if the pair value exists in the defined values. Finally after knowing the possible constraints that each value of an array can have, it gives to each variable a value (labeling). For that it generates a value that needs to be propagated and consistent. If it detects an inconsistency within the existent context, it generates a constraint in this context. If the context exchanges, the constraint is removed.

**AFLFast** AFLFast [BPR16] is an extension of American fuzzy lop (AFL) [afl], which is an implementation of Coverage-based Greybox Fuzzing (CGF). CGF is a fuzzing tool that tries to make path exploration more effective without analyzing the code. CGF lightly uses instrumentation to cover more information. It uses this information to decide which input to fuzz and for how long, and which generated input is kept for fuzzing. AFL continues to choose inputs until there is a timeout.

The input chosen to fuzz is the smallest and fastest for a certain path. After choosing an input, it chooses a $n$ which is the number of times that the fuzzer will mutate the input. Finally, it mutates the input $n$ times and saves the value if it is considered interesting.

AFLFast chooses the next input by calling the function ChooseNext, and it chooses the value for $n$ by calling the function AssignEnergy. In the function ChooseNext, it chooses the next input based on the number of times the input has been fuzzed, and in how many times the path that this input is going to exercise, has been exercised before. In the function AssignEnergy, the value $n$ is chosen taking into account the same conditions that were used to choose the input.

**Driller** Driller [SGS+16] is an hybrid vulnerability searching tool, that uses fuzzing and symbolic execution. Driller has two parts, the fuzzing and the symbolic execution components. For
the fuzzing component, it uses AFL without any modification because the improvements made
with the Driller were all based on the interaction between the symbolic execution and the fuzzer.
AFL was chosen because of the way it chooses the inputs to test, the way it creates new inputs,
and the way it does block transition coverage. Due to the nature of a fuzzer, it is unlikely for
it to find a specific value for a variable so there is a need for symbolic execution. The symbolic
execution is invoked when the fuzzer has gone through a predefined number of inputs without
finding a new transition.

When symbolic execution is invoked, the fuzzer passes the inputs that are considered inter-
esting. These inputs have a trace in order to identify which conditions the fuzzer was incapable
of satisfy. So when the symbolic execution discovers this kind of condition, the symbolic exe-
cution module generates an input that can access the new path. After the symbolic execution
processes all the traces, it returns the new information to the fuzzer, which will then explore
the new paths.
Chapter 3

BaZINGA Design

This chapter details the BaZINGA approach. The next four sections present, respectively, the approach, the process of executing guided data flows, trace processing, and the process of solving and validating constraints.

3.1 The BaZINGA approach

We propose a whitebox fuzzing approach that involves a combination of static analysis and concolic execution with fuzzing to detect vulnerabilities in web applications without accessing the source code of the web applications directly.

On one hand, black-box fuzzing allows injecting (random) inputs in applications, checking if some of them exploits a vulnerability. However, actually finding vulnerabilities with black-box fuzzing depends on the injected inputs, and it is difficult to hit all the inputs that are necessary to exploit all vulnerabilities, since this technique is totally agnostic to the source code of the application, and the control flow paths of the source code that are exercised depend on the inputs. For these reasons, this technique is known to have a high false negative rate.

Black-box fuzzing is not capable of identifying the location in the application source code of a vulnerability being exploited. In contrast, when using this technique, it is possible to obtain concrete executions, i.e., real data flow execution paths that we call traces, without false positives.

On the other hand, static analysis interprets the source code of an application, tracking the entry points, and verifying if any of them is used without being sanitized as parameter of a function susceptible to be exploited by an input coming from such entry point (e.g., mysqli_query and echo functions in PHP). This technique tends to generate false positives due to its undecidability [Lan92], since it does not know the values that entry points can take.
In contrast, symbolic execution is a method of analyzing a program, to determine what inputs cause each part of a program to execute. This technique involves obtaining the constraint condition of an application. For that, it solves logical expressions involving constraints (conditions) using a constraint solver that determines what values satisfy such constraints.

Resorting to symbolic execution it is possible to obtain the constraints of a program and resolve their domains and co-domains, finding the ranges of values that a variable (entry point) can take, determining which values cause each constraint of an application to execute new branches, and therefore increase the code coverage. However, doing symbolic execution of a large application has limitations, as the number of feasible paths in a program can grow exponentially, since this technique represents an input value symbolically.

Concolic execution, for its part, involves symbolic execution, but it is guided by concrete executions. This guidance mitigates the limitations of the symbolic execution, since it is employed in a trace of the application for the values that were inserted.

Bringing together these techniques it is possible to mitigate the disadvantages of each one. Our approach aims to detect and identify vulnerabilities in code of web applications using symbolic execution with oriented fuzzing to achieve real execution of data flows. This way, it aims to cover all data flow paths of an application resorting to the combination of these techniques with inputs generation guided by constraint programming to identify vulnerabilities in these data flows paths leveraging a technique similar to static analysis.

The approach works in a loop that goes through two modes: runtime and static. Before starting the loop, the application surface has to be discovered. This surface is composed of entry point subsets, i.e., sets of entry points that are processed together (e.g., the entry points that correspond to a form in a web page).

In runtime mode, the approach involves fuzzing the application with the input generation guided by constraints solved in static mode, except in the first round of the loop in which random inputs are used. During fuzzing, the application is monitored for getting the concrete execution sequence of instructions – a trace.

Next, in static mode, the trace is analyzed, looking for vulnerabilities and extracting the existent constraint paths. The constraints are solved for getting their domains and co-domains and the values that satisfy the logical expressions constituted by them. Afterwards, a new loop iteration takes place. Each entry point subset is explored in order to get all possible traces, using the successive solved constraints and different combinations between them for the entry points, i.e., using stratified values from co-domains and/or negating them. The loop ends when there are no more paths to discover.
The approach is composed by the 7 phases shown in Figure 3.1 and described next. The runtime mode comprises the first four phases, and the static mode the remaining three phases. The loop comprises phases 1 through 6.

0. *Surface web application scanner*: scans the web application surface to retrieve the entry point subsets, the submission method (e.g., GET, POST) for each entry point and the target URL.

1. *Input description generator*: generates a description of each domain/co-domain found (from the information of the *Constraint solver* phase) for each input of a given entry point subset. A description includes, for example, the input type, and their limits. In the first loop iteration as there is no information about entry points’ domains/co-domains, the inputs generated are randomly as usually a fuzzer does (represented by *initial input vector* in Figure 3.1).

2. *Fuzzer*: generates inputs based on the input descriptions (received from the previous phase) and inject them in the web application, using the URL and parameters that characterize the subset in analysis. Different combinations using the descriptions are made to ensure that all possible combinations for the entry points are explored (e.g., negate some descriptions or get some stratified value from descriptions), and ensuring in this way the execution of all data flow paths.

3. *Monitor*: monitors the application for the inputs injected, analyzing the behavior of the web application, and collecting a trace of the execution of the program, i.e., the lines of code executed of the web application.

Figure 3.1: BAZINGA architecture

21
4. **Constraint extractor/vulnerability detector**: analyzes the trace collected by the monitor in two fashions, statically and symbolically. By tracking the entry points (inputs), check if some of them (1) reaches any sensitive sink, and therefore, detecting and identifying a vulnerability, and (2) is used in any condition, and therefore, extracting the constraint paths. The vulnerabilities found are reported with its identification in the source code, and the exploit that allows its exploitation.

5. **Constraint solver**: solves the constraint paths extracted by the trace analyzer. For each constraint, a solver receives it and determines what are its limits. For the constraint path received, i.e., for each logical expression, that is constituted by constraints and the logical operators that connects them, the solver determines which are the values that satisfy the expression.

6. **Domains validator**: validates the results of the solver, verifying if the injected input values are within of the limits found by the solver for the respective entry points.

### 3.2 Executing Guided Data Flows

The process of executing guided data flows involves several phases. The first phase is extracting the entry point subsets of a web application surface before the application starts being monitored. The next phase, it injects inputs through entry points generated from input descriptions. Finally, in the last phase it collects the traces of concrete executions generated by the injected inputs. This process constitutes the first four phases of the presented approach – *surface web application scanner, input description generator, fuzzer, and monitor* (see Figure 3.1) – and the approach’s runtime mode.

#### 3.2.1 Collecting entry points

The injection of inputs (fuzzing) of a web application is done through an URL with the inputs assigned to parameters. The URL structure is `<url>?<param>=<value>&...&<param>=<value>`, where `<url>` represents the base URL, `<param>` the parameters and `<value>` the values assigned to the parameters. These parameters are the web application entry points and are defined for example in HTML forms. For the fuzzing to be done, first the entry points must be discovered, meaning that the application surface has to be scanned in order to discover its parameters, e.g., its forms.

To collect the entry points, when the application starts being monitored, a surface scan is made a single time. For each form found by the scanner, it is retrieved the information needed
to compose a URL containing the base URL, the parameters and the method to submit the request (e.g., GET, POST). At the end, a set of URLs is obtained, each one characterized by a subset of entry points, i.e., a set of parameters.

After extracting the entry points, each entry point subset is exercised in loop in order to explore different input combinations that exercise the different code branches.

### 3.2.2 Generating input descriptions

An input description characterizes an input domain and is composed by the input type and their limits. Its format has to be understandable by the fuzzer. A guided data flow execution means that a program was executed for specific and known inputs. To achieve such concrete inputs and inject them by using fuzzing, we need to describe the input domains/co-domains solved by the constraint solver (see Section 3.4). In this context, the domains and co-domains are the variables and the values that they can take, respectively.

This phase starts the approach’s loop. In the first iteration of the loop, as there is no information about concrete executions with the entry points subset in analysis, it is used an initial input vector to create an input description that is compatible with the URL of the subset under looping, and that the fuzzer is capable of understanding. For that the entry points encountered by the scanner are used to generate the input descriptions.

Later, for the next loop iterations, the various input domains and co-domains are generated from the values discovered in the previous iterations, as explained above. From them, specific and concrete inputs can be generated and injected in the application, resulting in concrete executions.

### 3.2.3 Fuzzing

Although a fuzzer can use generated-based or mutation-based methods for generating inputs without any restriction, we want a fuzzer input-description-based generating strict inputs to perform guided injections. Therefore, the fuzzing phase receives the URL characterizing the entry points subset and the generated input descriptions. The fuzzer generates the inputs based on these descriptions, and composes an URL with them.

Each iteration of the loop represents a different base URL, which alongside its entry points represent the values injected in the application. In the first loop iteration, the initial input vector is used to the fuzzer assign random values to the URL’s entry points, which will then be used to create an injectable URL.

In the next iterations, the fuzzer assigns values to the entry points from the input domains
and co-domains. The values chosen are limit values of the co-domains which are used in order to find different paths, and consequently making it possible to find more vulnerabilities. Afterwards, the fuzzer uses this new URL to inject the values in the web application.

### 3.2.4 Monitoring

For performing static analysis and symbolic execution, a monitor is used for observing the behavior of the web application, verifying when a concrete execution is finished, and registering the actions taken by the application, i.e., to obtain the trace. This can be done using a debugger or another tracing scheme.

---

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td><code>$_GET[&quot;name&quot;] = &quot;Bob&quot;; &quot;$_GET[&quot;age&quot;] = 81</code></td>
</tr>
<tr>
<td>02</td>
<td>`$_GET[&quot;name&quot;]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>`if ($_GET[&quot;name&quot;]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td><code>$_GET[&quot;name&quot;] = &quot;Bob&quot;; &quot;$name = $_GET[&quot;name&quot;];&quot;</code></td>
</tr>
<tr>
<td>04</td>
<td><code>$_GET[&quot;age&quot;] = $_GET[&quot;age&quot;];&quot;</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>06</td>
<td><code>Welcome Bob&lt;br /&gt;</code></td>
</tr>
<tr>
<td>07</td>
<td><code>#$BEGINELSE</code></td>
</tr>
<tr>
<td>08</td>
<td><code>The age: 81 is not within range&lt;br /&gt;</code></td>
</tr>
<tr>
<td>09</td>
<td><code>#$ENDIF</code></td>
</tr>
<tr>
<td>10</td>
<td><code>#$ENDIF</code></td>
</tr>
</tbody>
</table>

---

Figure 3.2: Example of a trace of an execution of the code from Figure 2.1.

---

The monitor handles differently non-if instructions and if instructions:

- **not-if instructions**: logs the line executed and the value of the variables called;

- **if instructions**: generates two log entries during the execution of the if-condition. The first entry is written before the execution of the if-condition. Its structure is:

  `<expr_1>: <value_1>&&...&&<expr_n>: <value n> [if(<expr_1>...<expr_n>)].`

Inside of the brackets is the if-condition, which is composed by one or more `<expr>` that are not evaluated. Before the brackets there are several `<expr>` followed by `<value>`. The symbol `<value_k>` is the value of the `<expr_k>` evaluated. The second entry is added after the execution of the if condition. If the if-condition is true, it logs `#BEGINIF`. In case of it being false and entering the else block, it logs `#BEGINELSE`. After the execution of the if block, it logs `#ENDIF`.  

---

24
Figure 3.2 shows a sample of a trace obtained when the name and age entry points of the code from Figure 2.1 take the values Bob and 81, respectively.

3.3 Analyzing Traces

This is the point when the static mode begins and corresponds to the fifth phase of the approach – constraint extractor/vulnerability detector. The trace is analyzed to verify if a vulnerability was exploited, and to extract the constraint path. For this process, the lines of the trace are read and represented by a syntax tree.

3.3.1 Detecting Vulnerabilities

To search for vulnerabilities, we used a method similar to taint analysis. Taint analysis is the most used technique from static analysis to discover bugs in software. Taint analysis is used directly in the source code of an application, however in our approach we analyzed the logs of the lines executed.

BaZINGA tracks the entry points of a program, and if a variable is assigned to an entry point, it marks the variable as tainted. During the analysis it verifies if the tainted variables stay tainted until it reaches a function that is susceptible to be exploited for some malicious input (a sensitive sink). If this happens, it encounters a vulnerability.

First, the analyzer verifies if the syntax tree element matches with an entry point from the entry points subset, and then it verifies if they are sanitized (i.e., invalidation of malicious inputs). In case of matching the entry point but not with a sanitization operation, the analyzer changes the entry point taintedness to tainted, meaning that that entry point can carry a malicious input, otherwise the entry point taintedness remains untainted.

The next step is verifying if there is an attribution, which happens when a tree branch contains a assign operator (i.e., $ = character) and on its left a variable (e.g., $name in PHP). In case of existing an attribution, the analyzer verifies the existence of entry points, or of a tainted variable on the right side of the assign operator. If there is a tainted variable, it verifies if it is sanitized. Then depending if the information on the right side, the variable created from this attribution is tainted or not.

Finally, the last step is to verify the existence of sensitive sinks, which is done by comparing the line with a list the existing sensitive sinks. In the case of existing a sensitive sink and in this line there is either a tainted variable or an entry point, it finds a vulnerability. When it finds a vulnerability, it saves the vulnerable line, the line and the URL that exploited it, and reports that a vulnerability was found giving these informations.
Following the process for the trace of the Figure 3.2, the analyzer found the name and age entry points in lines 2 and 3, which are assigned to $name$ and $age$ variables; so these variables’ state is tainted. Next the $name$ variable is used as parameter of the echo sensitive sink in line 4, and both variables are used with the same function in line 8. Therefore, the analyzer detects two XSS vulnerabilities, and identifies them.

3.3.2 Extracting Constraint Paths

Extracting the constraint path means that the analyzer looks for the if-blocks in syntax tree. The analyzer finds the lines containing #BEGINIF, #BEGINELSE, and #ENDIF. When it finds one, it adds it to a list that will be delivered to the constraint solver. Afterwards, it searches for the if-conditions and translates them to an intermediate language, i.e., to a language that the constraint solver can understand. Therefore, when it finds an if-condition, it identifies the conditions belonging to it and the logic operators that form the logic expression. At the end, a stack is created with the elements which will be passed to the solver.

Figure 3.3 shows an example of this process. When a parenthesis is opened or closed, the analyzer adds to list which one occurred. The same happens when it finds an AND or an OR logic operator. When a boolean expression is found, it adds to the list COND plus how many expressions it has already found.

```plaintext
if(($age > 5) && ($age <= 80))
  ( -> OPEN
    $age > 5 -> COND0
    ) -> CLOSE
  && -> AND
  ( -> OPEN
    $age <= 80 -> COND1
    ) -> CLOSE

OPEN COND0 CLOSE AND OPEN COND1 CLOSE
```

Figure 3.3: Example of transition from a condition to the intermediate language.

3.4 Solving and Validating Constraints

After analyzer identifying the constraint path in the trace, the path is evaluated by a solver to discover which values satisfy the path, and validates such evaluation. These tasks correspond to the last phases of the approach, namely constraints solver and domains validator.
3.4.1 Solving constraints

After the analyzer identifies the constraint paths in the trace and represents each if-condition in an intermediate language (and stored in lists), a constraint solver analyzes each one of the constraint paths in order to find the possible domains for the if-conditions. Such representations are needed to create the conditional expressions in the solver’s language. Afterwards, a Satisfiability Modulo Theories (SMT) solver is used to evaluate the path, resulting the co-domains that satisfy the path.

The constraint path is evaluated following the assumption behind the concolic execution technique, more precisely the way of evaluating the constraint path represented by symbolic execution is made. In other words, concolic execution says given a constraint path constituted by different conditions, each condition remains its state, excepting the last one which is negated. Considering that the last condition of the path was the one that decided the direction taken by the data flow, negating it makes that a new path constraint is obtained, and so new domains for the involved variables in this new path will be discovered. With this in mind, the constraint solver uses this technique to find the constraints.

For instance, given the constraint path \((\text{\$age} \neq 0) \land (\text{\$age} > 5) \land (\text{\$age} \leq 80)\) composed by two conditions placed in different levels of an application. Evaluating this path we obtain the values between 5 and 80 plus the values different than 0, and so if we look at the values surrounding of the last constraint (4, 5, 6, 79, 80, 81), we can check that there are at least two data flow executions, one when the condition is true, and another when it is false. In each iteration of the constraint solver, if it finds a last constraint different from the previous iterations, the constraint solver adds new paths to explore.

3.4.2 Validating domains

A way for validating the resulting domains provided by the solver is verify if the injected inputs fall in such domains. This validation allows us to check if the solver performs well and consequently if the trace analyzer extracts the constrains from the trace correctly.

Recalling that the symbolic execution aims to represent symbolically a set of well specified variables, i.e., the entry points, and that for that the code of the application is instrumented for these variables, the resulting constraint path contains these variables and their dependencies. The domains are associated with these variables, and so they can be validated. Since we look at the values surrounding the values of the constraints (see Section 3.4.1), if the values found by the solver are correct, it validates that the value of the input of the last iteration.
Chapter 4

The BaZINGA Implementation

This chapter presents the implementation of the BaZINGA tool, describing its modules and how they work.

BaZINGA is a whitebox fuzzing tool, which was implemented in the Java language. It is composed by several components – trace generator, trace analyzer, and constraint solver and validator. The trace generator is used for obtaining the resulting traces from the execution of the web application guided by fuzzing. The trace analyzer detects SQL injection (SQLI) and cross site scripting (XSS) vulnerabilities in PHP web applications. The constraint solver and validator searches for constraint paths, and solves them. The following three subsections are regarding to these modules, explaining their implementation and functionality.

In order to initialize BaZINGA, as stated before, we need to find the entry points of the web application that it is going to be tested. For that, we use Wapiti 2.3.0 [Sur], a tool that scans a web application looking for entry points, which then applies fuzzing using the discovered entry points to search for vulnerabilities. Although Wapiti is a fuzzer, we only used the scanner component. During its search, Wapiti logs the paths it discovers into a xml file, which contains the discovered URLs, its entry points, and submitted method. After the execution of Wapiti, BaZINGA starts running.

4.1 Trace Generator

The trace generator module starts by using the xml file produced by Wapiti to know the entry points of the application. For that, we use javax.xml.bind, a java library for parsing xml files. For the module to be able to store the values from the file, we implemented some classes that can translate the xml file into java objects. The translation process is done using unmarshal(File f), that transforms the File f into an instance of a class, which in our case...
is root. The library matches an element of a xml file with its java class counterpart by using
@XmlElement(name = String s), with String s being the element’s value. For the library
to assign values to the class, it needs to know the elements and the attributes, which can be
obtained by using @XmlElement(name = String s) and @XmlAttribute(name = String s),
respectively. In case of an element/attribute having their own elements/attributes, it needs to
have its own class.

Once the translation is done, the entry point subsets are composed. Next, the input descrip-
tions are generated based on the domains provided by the constraint solver and validator (see
Section 4.4) or on the inputs coming with the Wapiti results (for the first loop iteration).

Afterwards, the module uses the entry point subsets and the input descriptions for generating
inputs, composing final URLs, and then sending request connections for the URLs, injecting thus
the inputs.

4.2 Instrumentation

From the injection in the web application, it generates a log of the lines of code executed. To
generate this log, the PHP application files are instrumented. To do this, we used monolog
[Bog], a tool that sends the logs of the lines of code we instrumented to a file.

The web application is instrumented to log any execution that involve entry points and
their dependencies, are involved (e.g., assigns, if statements). After every line of code of the
web application, we included a piece of code that adds to the trace information about the line
executed. There is an exception for the if-condition that adds the log line before its execution,
as said in the Section 3.2.4.

To implement that, we modified the web application in order to be capable of logging the
lines executed. For that the web application needs to have a line with use Monolog\ILogger;
to be capable of logging and one line with use Monolog\Handler\StreamHandler; for it to be
capable of saving the log in files.

After that, it creates and initializes the logger. To initialize, it uses
$log->pushHandler(new StreamHandler(<path>, Logger::TYPE))
where <path> is the location of the log file and TYPE is the type of log used. We used
INFO which logs the information received without stopping the execution of the web application
being tested.

After that, like said in 3.2.4, it was added a line after each original line of code, containing
$log->info(), so that everything that is between the parenthesis is added to log.
4.3 Trace Analyzer

The module receives the traces for identifying vulnerabilities and extracting constraint paths by applying the process described in Section 3.3. We implemented our own parser in Java, that works as explained in that section.

In this phase, it uses the log generated by the execution of the application, to be capable of finding the vulnerabilities and the paths needed to explore. To achieve that, it splits the log by line, and then verifies the existence of certain elements, in each line, for it to be capable of cataloging the lines. To do the cataloging, it verifies the existence of keywords in the testing line.

To find if-conditions, it searches for the keyword if. In case of finding a line that contains this keyword, it splits the line in several pieces in order to find the several existent constraints.

In order to find attributions, it searches for =. When it finds a line with an attribution, it splits the line using character =. With this, it finds the variable’s name, and variable’s value. It is this value that is evaluated in order to verify if it is tainted.

4.4 Constraint Solver and Validator

After creating the list for every if-condition, the list of if-blocks in the log, and composing the constraint paths, BaZINGA analyses each one of them in order to find the possible domains for the if-conditions, using a solver interpreter.

The constraint solver and validator module uses these lists to create the conditional expressions in the solver’s language. We use Z3 [DMB08], a Satisfiability Modulo Theories (SMT) solver, which can only be used after a Context is created. It is the Context that saves every information about the solvers, variables, and expressions. The variables are created by calling Context’s function \textit{mkConst(Symbol s, Type t)}, being Symbol s the name of the variable and Type t the type of the variable. To create a symbol, it needs to call \textit{mkSymbol(String s)}, and the types are chosen from the different kind of types. In case of being an Integer it uses \textit{get-IntSort()}. Expr is the generic type, it is from Expr that the other expressions are extended. The other expressions are ArithExpr, and BoolExpr. ArithExpr represents the numeral variables, and BoolExpr represents the boolean expressions.

Finally, it is by using the solver that Z3 is capable of solving SMT problems. The solver has every expression and finds a solution that is contained in the domain of its expressions. In BaZINGA, a solver is used to find the domains of the if-conditions, and for each block of if-conditions it creates a solver.
BaZINGA’s solver phase starts by creating three stacks, one for the parenthesis, one for the conditional expressions, and one for the conditional operations. Then it starts iterating the lists, one at a time. In Figure 4.1, we can see the behavior of the stacks. The interpreter starts by verifying what is the value of each String. If the value of the String is OPEN and the expression’s Stack is empty, it puts the value OPEN in the parenthesis stack, as we can see in the first point of the Figure 4.1. However, if expression’s Stack is not empty, it removes the value from that Stack and puts it in the parenthesis’ Stack, along with value OPEN, as seen in point 6.

<table>
<thead>
<tr>
<th>list: OPEN COND0 CLOSE AND OPEN COND1 CLOSE</th>
<th>parenthesis</th>
<th>expression</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. list: COND0 CLOSE AND OPEN COND1 CLOSE</td>
<td>OPEN</td>
<td>COND0</td>
<td></td>
</tr>
<tr>
<td>3. list: CLOSE AND OPEN COND1 CLOSE</td>
<td>OPEN</td>
<td>COND0</td>
<td>AND</td>
</tr>
<tr>
<td>4. list: AND OPEN COND1 CLOSE</td>
<td>COND0</td>
<td>COND0</td>
<td>AND</td>
</tr>
<tr>
<td>5. list: OPEN COND1 CLOSE</td>
<td>COND0</td>
<td>COND1</td>
<td>AND</td>
</tr>
<tr>
<td>6. list: COND1 CLOSE</td>
<td>COND0</td>
<td>COND0</td>
<td>AND</td>
</tr>
<tr>
<td>7. list: CLOSE</td>
<td>COND0</td>
<td>COND1</td>
<td>AND</td>
</tr>
<tr>
<td>8. list: -</td>
<td>BOOL0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.1: Example of transition from the intermediate language to the stacks.

If the value found on the list is COND and expression’s Stack is empty, it adds COND to that Stack, as seen in point 3. In case of the expression’s Stack is not empty, it pops the element
in operations’s Stack and creates a BoolExpr using both conditions and the value on the top of the operation’s Stack, and it adds BOOL in the expression’s Stack. In case of finding an operator in the list, such as AND and OR, it pushes into the operation’s Stack this value, as in the point 5.

If the value of the list is CLOSE, it starts by popping the value on the top of parenthesis’s Stack. Then it verifies if the parenthesis’s Stack and the expression’s Stack are empty. In case they are not empty, it verifies if it is a BOOL or a COND on the top of parenthesis’s Stack. After this, it removes the elements on the top of the three Stacks, and creates a BoolExpr using them, which is saved. The last step, is to push BOOL into the expression’s Stack, as showed in the passage from point 7 to 8.

After processing an intermediate language element, it processes the if-block list. In case the if-block ends, it creates a solver using every BoolExpr created during this block. In case of finding a beginning of if-block, it saves the BoolExpr created during the processing of the intermediate language. When it finds two beginnings of block, it starts processing a new intermediate language list.

When BaZINGA interprets all the elements of the intermediate language list, it creates another list, containing the values of the conditions of the solver. With this list it creates a list of variables with values of the conditions of the solver. With the iteration of this list, the internal loop of BaZINGA starts. This loop starts by injecting an URL, using the values of the list of variables. Then from the log created from this injection, it searches for vulnerabilities and new paths to explore. The values that are injected in the web are chosen from the top to the bottom of the variable’s list. The value injected depends on the type of variable, if it is a numeral or a String, it has a different behavior. In case of being the type String it injects the value of the variable and a different value. In case of being a numeral, the value injected is $v - 1$, $v$, and $v + 1$, with $v$ being the value of the variable.

4.5 Limitations

During the implementation of BaZINGA, we chose to use Wapiti and monolog tool. We use Wapiti as it is not in the scope of this thesis to implement a web application scanner. However this decision comes with its own limitations, since Wapiti is a tool that is not capable of finding every type of entry point, BaZINGA is only capable of finding vulnerabilities in applications for which Wapiti can encounter entry points. Monolog is an application that logs information from the application, so for it to be useful it needs to instrument the applications. However due to the scope of this work, it was decided to focus time and resources on the creation of the
prototype, leaving the automatization for future work.
Experimental Evaluation

The objective of the experimental evaluation was to answer the following questions:

1. Is BaZINGA able to detect vulnerabilities in synthetic and real web applications?
2. Is BaZINGA able to solve constraint paths correctly?
3. How much code coverage does BaZINGA achieve?

In order to validate our approach, we evaluate BaZINGA with two sets of web applications for detecting SQLI and XSS vulnerabilities. In Section 5.1 the synthetic applications are used, and in Section 5.2 the evaluation uses real software. Both sections answer to questions 1 to 3.

5.1 Example Application

To test our tool, we decided to use an example application that we created with common vulnerability patterns.

The applications under test contains several functionalities:

- if-statements (with and without chained ifs);
- entry point subsets;
- three vulnerabilities dispersed in different branches of the if-statements.

The code of this application can be seen in Figure 2.1

For BaZINGA to be capable of finding all the vulnerabilities in this application, it has to inject in it at least two values, as it has two different branches.

Imagining that the first values that BaZINGA injects are name = "name" and age = "100". With these values it enters the if-block in the line 1, as it has values in both variables. In the
lines 2 and 3 it encounters two entry points, and as it has in both lines assignments, the variables being assigned in both lines, $name and $age, become tainted.

In the line 4, there is an echo function. This function is a sensitive sink, this means if in the same line there is a tainted variable or an entry point without it being sanitized, it finds a vulnerability. As in this line there is the variable $name, and like we said previously, it is tainted, it finds a XSS vulnerability.

After that, there is an if-condition in the line 5. This if-condition returns false, as $age is greater than 5 and greater than 80, so it executes the statements in the else-block. When executing the else-block, it encounters an echo function, in the line 8. This time, it encounters another XSS vulnerability due to the tainted variable $age. With this the application ends its execution.

Afterwards, BaZINGA searches for more paths that it can explore. To accomplish that, it looks at the if-conditions, and uses the values present in these conditions. In this application, the conditions are $age > 5 and $age <= 80. BaZINGA starts by using the right value, which is $age > 80. With this value, in the next three iterations it will inject in the variable age values around the value 80, which are 79, 80, and 81.

The first values that BaZINGA from this condition are name = "name" and age = "79". Until the line 5, it acts the same way it acted in the previous execution of the application. However, the if-condition in the line 5 is true, so it executes the statements in the if-block. In the line 6, it encounters a XSS vulnerability as there is a sensitive sink (echo) and a tainted variable ($age).

After ending the iteration with age = 79, it injects 80 which makes the application behave the same way as it behaved in the previous iteration. In the next iteration it injects age = 81, which makes the application behave the same way as when it was injected 100.

In the next iterations, it uses the left condition to search for more paths. It injects age = 4, and age = 5 (these two values make the application behave the same way as the values 81, and 100). The last value injected is age = 6, which makes the application execute the if-block.

Since BaZINGA did not find any other condition, it stops executing and return the vulnerabilities encountered, and the correspondent line where they were found.

This application was written to test if BaZINGA is capable of finding different paths in an application. The application has two entry points in the lines 2 and 3, followed by the first vulnerability in the line 4. After that, there is an if-else-block. It is in this block that tests the functionalities of the tool.

To find all the vulnerabilities, the tool has to explore all the existing branches of the appli-
cation. This is done by exploring the domains of the if-conditions, which are used by the tool in order to enter the possible paths of a if-condition. The tool can reach all the possible paths if the application chooses its paths from the entry point’s values.

BaZINGA was capable of finding every path of the application by using the constraints of the if-conditions. Being capable of changing the path by using the constraints, is what makes BaZINGA capable of finding all the vulnerabilities existing in the application. Also, we verified that the tool covered all data flows contained in the applications.

As BaZINGA was able to detect all vulnerabilities and exercise all paths, we obtained a preliminary positive answer to the three questions above.

5.2 Open Source Applications

To test BaZINGA with real applications, we used two open source projects, SAMATE [sam] and DVWA [dvw], both highly used for testing purposes.

SAMATE is a set of small applications with different kinds of vulnerabilities. Each kind of vulnerability has three different files to test. As shows the Table 5.1, the files with its name contains the number 0 do not have any protection against vulnerabilities, whereas the files containing the number 2 in its name are protected. In total, SAMATE files contain 9 vulnerabilities, 6 SQLI vulnerabilities, and 3 XSS vulnerabilities.

<table>
<thead>
<tr>
<th>File</th>
<th>Entry Points</th>
<th>Sanitization</th>
<th>Sensitive Sinks</th>
<th>VULs existent</th>
<th>VULs found</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql_lox0 1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>sql_lox1 2</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>sql_lox2 2</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>xss_lox0 1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>xss_lox1 2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>xss_lox2 2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: Samate results

The Table 5.1 shows the results of testing SAMATE. These results show that each file has at least an entry point, and that only finds vulnerabilities (VULs) when there are no sanitization operations. The number of vulnerabilities correspond to the number of sensitive sinks. BaZINGA detected all vulnerabilities correctly, presenting neither false positives and false negatives.

The second application tested was DVWA, which is a vulnerable PHP/MySQL web application containing several classes of vulnerabilities. However the ones that we tested were XSS and SQLI. Each class of vulnerability is configured to three levels of security. To test our tool we use the first two levels, the low and medium levels. The Table 5.2 shows the results of the evaluation. Such as in the SAMATE, when the tool finds entry points sanitized it does not report a vulnerability. However in the XSS medium case, which the entry point apparently is
sanitized, the tool reported it as vulnerable because the sanitization operation is not capable of completely protect the vulnerability. Therefore, the tool was capable to detect all existent XSS and SQLI vulnerabilities, without failing one.

<table>
<thead>
<tr>
<th>File</th>
<th>Entry Points</th>
<th>Sanitization</th>
<th>Sensitive Sinks</th>
<th>VULs existent</th>
<th>VULs found</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql/low</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sql/medium</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>xss/low</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>xss/medium</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.2: DVWA results

Answering to questions 1 to 3, the obtained results in both experiments using synthetic and real code suggest a positive answer to questions 1 to 3.
Chapter 6

Conclusions

This dissertation presents a whitebox fuzzing approach and the BaZINGA tool that implements it to improve the way of detecting vulnerabilities in web applications developed in PHP. The approach presented combines concolic execution and static analysis with fuzzing to cover the utmost data flows contained in applications. Concrete execution of data flows are guided by input injection generated from solved constraints using constraint programming. Vulnerabilities are found by using static analysis under the executed data flows. BaZINGA was evaluated with synthetic and real web applications, and the results showed that the tool is able to detect vulnerabilities, and achieves a great code coverage.

6.1 Future Work

In the future, the goal is to improve the tool in order to have more functionalities, like the ones that would make this thesis’ limitations inexistant.

For that, it is needed to create a way to instrument automatically the web applications tested. This functionality would allow the tool to run without the need of human intervention, making the instrumentation needed for the application to work, more reliable.

Another functionality is the creation of a web application surface scanner. This would allow the application to encounter entry points in other types of forms, making tool capable of finding vulnerabilities in more applications.
Bibliography


[dvw] DVWA - Damn Vulnerable Web Application. Available at http://www.dvwa.co.uk/.


43


