BaZINGA: Whitebox Fuzzing for Detecting Web Application Vulnerabilities

Anabela Borges
INESC-ID, Instituto Superior Técnico,
Universidade de Lisboa - Portugal
anabela.borges@tecnico.ulisboa.pt

***Abstract*—Despite much 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. This paper aims to improve the detection of vulnerabilities in PHP code. For that purpose it presents a whitebox fuzzer called BaZINGA that combines static analysis and concolic testing with fuzzing. We provide experimental results of running BaZINGA with code samples and open source applications.**

I. Introduction

Despite much research done in web security in the past few years, the security of web applications remains an important concern [1], [2]. A major issue with these applications comes from the source code itself, that is often written in unsafe languages like PHP or JavaScript [3], [4], leading to the existence of vulnerabilities. Conspicuous examples of vulnerabilities are those that allow SQL injection and cross-site scripting (XSS) attacks [1].

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 [3], [5], [4], [6]. Testing, including fuzzing, achieve a similar goal by executing and exercising the applications, with similar problems [7], [8], [9], [10], [11], [12]. Symbolic execution also finds vulnerabilities in code, but differently from the other approaches it exercises applications with symbolic values [13], [14]. There are other approaches that dynamically block attacks [15], [16], [17], [18], [19], among which the widely adopted web application firewalls [20], [21]. These tools and mechanisms are invaluable, but they tend to generate too many alarms and/or miss some vulnerabilities or attacks.

The objective of the paper is to improve the detection of vulnerabilities in PHP code, the language most used to implement web applications, at least until recently [22]. For that purpose, we present a whitebox fuzzing approach and a tool called BaZINGA that combines static analysis and concolic execution [13], [14] 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 [23], [24], [25], [26], [27]. 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.

The paper presents the implementation of BaZINGA and an experimental evaluation with synthetic code and open source applications. The evaluation has shown that the tool was able to detect all vulnerabilities as it is able to explore different control flow paths. The evaluation has also shown that the tool was able to cover 100% of the code of the analyzed applications.

The contributions of the paper 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; and (2) the BaZINGA whitebox fuzzing tool that implements the approach.

The remaining of the paper is organized as follows. Section II presents the approach, Sections III presents more details about it, and Section IV its implementation. Section V presents and discusses the evaluation results. Section VI discusses the related work about detection of vulnerabilities in software using different approaches. Section VII concludes the paper.

II. Web Applications Surface Vulnerabilities

This section 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 [1], 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 though inputs that are inserted through the attack surface of a program.

```
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 }}
```

Fig. 1. PHP script vulnerable to XSS.

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 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.

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

III. 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 by black-box fuzzing depends on the injected inputs, and it is difficult to hit all those 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 these inputs. For this reason, this technique is known to have a high false negatives rate. Moreover, if a vulnerability is exploited, black-box fuzzing does not identify its location in the application source code. In contrast, 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 analyzes the source code of an application, tracking the entry points and verifying if some of them is used without any protection as parameter of a function susceptible to be exploited by it (e.g., the *mysqli_query* and *echo* functions in PHP). However, the technique tends to generate false positives due to its undecidability [28], 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. Symbolic execution involves obtaining the constraint paths of an application, solving 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 (conditions) of a program and resolve their domains and co-domains, finding the ranges of values that a variable (entry point) can take, determining what values cause each part of an application to execute new branches, and increasing the code coverage. However, doing symbolic execution of a large application has limitations because the number of feasible paths in a program can grow exponentially, since the technique represents the input values symbolically. Concolic execution (or testing), for its part, involves symbolic execution, but guided by concrete executions. Such guiding mitigates the limitations of the former, since symbolic execution 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 concolic execution with fuzzing to achieve real execution of data flows. In this way, is its aim to cover the utmost possible of data flow paths of an application resorting of this combination with inputs generation guided by constraint programming, and to identify vulnerabilities in that data flows paths leveraging 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., of 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 loop) 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 2. The runtime mode comprises the first four phases, and the static mode the remaining three phases. The loop comprises

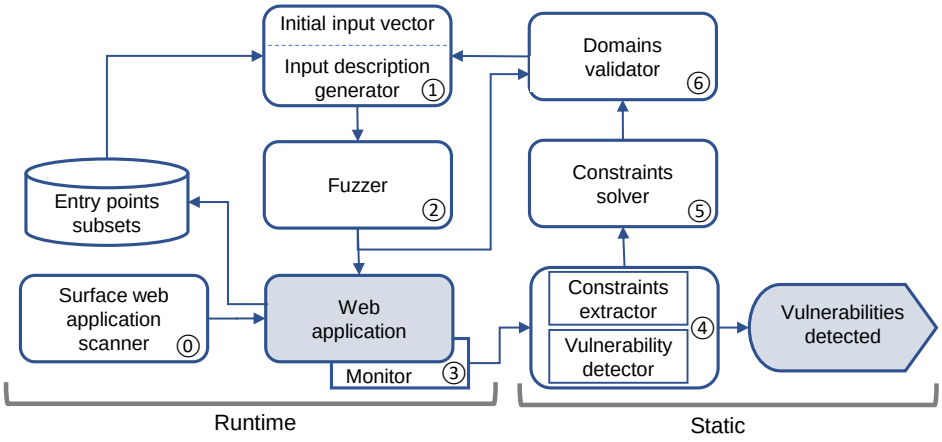

Fig. 2. BaZINGA architecture

phases 1 through 6.

0) *Surface web application scanner*: scans the web application surface to retrieve the entry point subsets, and for each one the submission method (e.g., GET, POST) and the target URL.
1) *Input description generator*: generates a description of each domain/co-domain found (in the *Constraint solver* phase) for each input of a given entry point subset. A description includes, for example, the input type, and their limits. However, in the first loop iteration no information about entry points domains/co-domains is known, so the inputs are generated randomly as usually a fuzzer does (represented by *initial input vector* in Figure 2).
2) *Fuzzer*: generatew 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*: monitorw 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 web application code lines that are running.
4) *Constraint extractor/vulnerability detector*: analyzes the trace 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 all constraints extracted and logical operators that connects them, resulting logical expressions, the solver solves the expression, determining 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.

IV. BaZINGA Design

This section details the approach, describing the actions performed in each phase. The next three sections present, respectively, the process of executing guided data flows, trace processing, and the process of solving and validating constraints.

A. Executing Guided Data Flows

The process of executing guided data flows involves first extracting the entry point subsets of a web application surface when the application starts being monitored, then to inject inputs trough entry points generated from input descriptions, and to collect the traces of concrete executions generated by those inputs. This process constitutes the first four phases of the presented approach – *collecting entry points*, *generating input descriptions*, *fuzzing*, and *monitoring* (see Figure 2) –, and the approach runtime mode.

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 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.

2) Generating input descriptions: 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 IV-C). A *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.

This phase starts the approach's loop. In the first iteration of the loop, no information about concrete executions with the entry points subset in analysis is known, therefore, an *initial input vector* is used 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/co-domains resulting from the previous iteration are described as explained above. From them, specific and concrete inputs can be generated and injected in the application, resulting concrete executions of guided data flows.

3) Fuzzing: Although a fuzzer can use generated-based or mutation-based methods for generating inputs and 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. Afterwards, the fuzzer uses this new URL to inject the values in the web application.

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.

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`.

Figure 3 shows a sample of a trace obtained when the `name` and `age` entry points of the code from Figure 1 take the values `Bob` and `81`, respectively.

B. Analyzing Traces

This is the point when the static mode begins and corresponds to the fifth phase of the approach – *analyzing traces*. 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.

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. It 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 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).

First, the analyzer verifies if the syntax tree elements matches with an entry point from the entry points subset, and then it verifies if they were 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 trace that contains it, 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, the analyzer found the `name` and `age` entry points in lines 2

```
;; A representation of a line with an if-condition
01: $_GET[\"name\"]: Bob&&$_GET[\"age\"]: 81 ["if( $_GET[\"name\"] || $_GET[\"age\"] )"]
;; A representation of a log where the if-condition is true
XX: ["#BEGINIF"]
;; An example of an attribution where is given the value 5 to the value name
02: Bob ["$name = $_GET[\"name\"];"]
03: 81 ["$age = $_GET[\"age\"];"]
;; Another example of an if-condition
04: Welcome Bob<br /> ["echo \"Welcome \". $name. \"<br />\";"]
05: $age > 5: &&$age<= 80: ["if (($age > 5)&&($age<= 80))"]
;; A representation of a log where the if-condition is false
07: ["#BEGINELSE"]
08: The age: 81 is not within range<br /> ["echo \"The age: \" . $age . \"is not within
    range<br />\""] []
09: ["#ENDIF"]
09: ["#ENDIF"]
```

Fig. 3. Example of a trace of an execution of the code from Figure 1.

and 3, which are assigned to `$name` and `$age` variables; so these variables state taint. 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.

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 4 shows an example of this process. As we can see in Figure 4, 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.

```
if(($age > 5) && ($age <= 80))

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

OPEN COND0 CLOSE AND OPEN COND1 CLOSE
```

Fig. 4. Example of transition from a condition to the intermediate language.

C. Solving and Validating Constraints

After analyzer identifying the constraint path in the trace, the path is evaluated by a solver for evaluate which values satisfy the path, and validates such evaluation. These tasks correspond to the last phases of the approach, namely *constraint solving* and *validating domains*.

1) Solving constraints.: 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, 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 discover. For instance, given the constraint path `($age != 0) && ($age > 5 && $age <= 80)` composed by two conditions placed in different levels of an application, it is transformed in `($age != 0) && ($age <= 5 || $age > 80)`. Evaluating this path we obtain the values between 0 and 5 plus the values higher than 80, and so we can take 81 for getting a new data flow execution. Notice that the values less or equals to 0 are not considered because the first condition was already evaluated when the data flow reaches the second condition.

After the analyzer identifies the constraint path in the trace and represents each if-condition in an intermediate language (and stored in lists), a constraint solver analyses each one of them 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.

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. However, since the last condition of the constraint path is negated (see Section IV-C1), the validation for some inputs values fails. To deal with this, the variables figuring in the last condition are picked up, and verified if the inputs values associated to them are those that fail. If so, the respective domains are validated.

V. The BaZINGA Implementation

This section presents the implementation of the BaZINGA describing its modules and how they work.

BaZINGA is a whitebox fuzzing tool which was implemented in the Java language. It is composed by three 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. 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 we stated before, we need to find the entry points of the web application that it is going to test. For that, we use *Wapiti 2.3.0* [29], a tool that scans a web application looking for entry points, for then applying 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.

A. 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 `@XmlRootElement(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 descriptions are generated based on the domains provided by the *constraint solver and validator* (see Section V-C) 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.

From the injection in web application, it generates a log of the lines of code executed. To generate this log, the PHP web application files are instrumented as explained in Section IV-A4, and using the *monolog* tool [30], a tool that sends the logs of the lines of code we instrumented to files, or other descriptor.

B. Trace Analyzer

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

C. 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 [25], 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 *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 *mkSymbol(String s)*, and the types are chosen from the different kind of types. In case of being an Integer it uses *getIntSort()*. 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 5, 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 5. 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 points 4 and 5.

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.

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.

```
1.
list: OPEN COND0 CLOSE AND OPEN COND1 CLOSE
parenthesis |   expression |    operation

2.
list: COND0 CLOSE AND OPEN COND1 CLOSE
parenthesis |  expression  |    operation
OPEN        |              |

3.
list: CLOSE AND OPEN COND1 CLOSE
parenthesis |   expression |    operation
OPEN        |     COND0    |

4.
list: AND OPEN COND1 CLOSE
parenthesis |   expression |    operation
            |   COND0      |
5.
list: COND1 CLOSE
parenthesis |   expression |    operation
COND0       |              |    AND
OPEN        |              |

6.
list: CLOSE
parenthesis |   expression |    operation
COND0       |   COND1      |    AND
OPEN        |              |

7.
list: -
parenthesis |   expression |    operation
            |   BOOL0      |
```

Fig. 5. Example of transition from the intermediate language to the stacks.

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.

VI. 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 VI-A the synthetic applications are used, and in Section VI-B the evaluation uses real software. Both sections answer to questions 1 to 3.

A. Example Application

To test our tool, we decided to use a set of example applications that we created with common vulnerability patterns.

The applications under test contain several functionalities, if-statements (with and without chained ifs), and entry point subsets. Also, they have four vulnerabilities in different branches of the if-statements: 2 SQLI and 2 XSS.

To find all the vulnerabilities, the tool has to explore all the existing branches of the application. 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.

B. Open Source Applications

To test BaZINGA with real applications, we used two open source projects, SAMATE [31] and DVWA [32], both highly used for testing purposes.

SAMATE is a set of small applications with different kinds of vulnerabilities. Each kind of vulnerability has two different files to test. As shows the Table I, 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.

File	Entry Points	Sanitization	Sensitive Sinks	VULs existent	VULs found
sql_lod0	1	0	2	2	2
sql_lod1	2	0	4	4	4
sql_lod2	2	2	4	0	0
xss_lod0	1	0	1	1	1
xss_lod1	2	0	2	2	2
xss_lod2	2	2	2	0	0

TABLE I
SAMATE RESULTS

The table I shows the results of testing SAMATE. These results show that each file has at least an entry point, and that only finds vulnerabilities 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 II 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.

File	Entry Points	Sanitization	Sensitive Sinks	VULs existent	VULs found
sql/low	1	0	1	1	1
sql/medium	1	1	1	0	0
xss/low	1	0	1	1	1
xss/medium	1	1	1	1	1

TABLE II
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.

VII. RELATED WORK

One of the ways to detect vulnerabilities is to use a *static analysis tool* to check the source code without executing it [33]. Jovanovic *et al.* [3] implement the concept of alias analysis in Pixy, a tool for identifying input validation vulnerabilities using static analysis, in order to detect XSS and SQL injection vulnerabilities in web applications. Wassermann *et al.* [5] use string-taint analysis to find invalid string values, followed by a policy for web pages to include only trusted scripts. Data Mining is a technique to automatically obtain information from large data sets. Medeiros *et al.* [4] 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.

Black-box fuzzing is a testing technique that, without knowing the source code, gives to the program random inputs to find vulnerabilities in it [7]. KameleonFuzz [11] is an extension of LigRE [9], [10], a black-box fuzzer that performs control flow to detect reflected and stored XSS vulnerabilities. LangFuzz [34] 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.

Some tools do *grey-box testing* by combining static analysis with fuzzing. BaZINGA falls in this category. Dowser [35] is a grey-box fuzzer that combines taint 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. jÄk combines web application crawling with dynamic program analysis [36]. jÄk is composed by four modules: dynamic analysis, crawling, attacker, and analysis modules.

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 *boolean satisfiability problem* (SAT) is the problem of determining if there is an interpretation that satisfies a boolean expression. 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. A *solver* is a tool that is capable of solving satisfiability problems.

JaCoP [23] 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. MiniZinc [24] is a solver based on SMT that was designed for decision problems over integers and real numbers, while also being capable of performing optimizations. Z3 [25] 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. SMT-LIB [26] 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. Trinh *et al.* proposed an extension of Z3 called S3 [27]. S3 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.

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*. Godefroid *et al.* [13] propose Scalable, Automated, Guided Execution (SAGE), a whitebox fuzzing tool that detects vulnerabilities on x86 Windows programs, more specifically on file-reading applications. Cadar *et al.* [14] proposed a new symbolic execution tool named KLEE, for detecting vulnerabilities in C (not PHP or web applications). 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. WAPTEC is a tool that checks the existence of vulnerabilities in web applications by using symbolic execution and dynamic analysis [37]. WAPTEC's objective is to identify the inputs that the client rejects but the server accepts. PHPQuickFix is a static analyser that checks the existence of simple bugs in HTML [38]. PHPRepair 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 [38]. Enderlin *et al.* [39] developed a constraint solver for arrays in Praspel, a language and a framework for contract-based testing in PHP, based on realistic domains. Driller [40] is an hybrid vulnerability searching tool, that uses fuzzing and symbolic execution.

VIII. Conclusion

The paper 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.

This paper aims to improve the detection of vulnerabilities in PHP code. For that purpose it presents a whitebox fuzzer called BaZINGA that combines concolic execution and static analysis with fuzzing. We provide experimental results of running BaZINGA with code samples and open source applications.

References

[1] J. Williams and D. Wichers, "OWASP Top 10 - 2017 rcl - the ten most critical web application security risks," OWASP Foundation, Tech. Rep., 2017.

[2] Imperva, "The state of web application vulnerabilities in 2017," Dec. 2017.

[3] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise alias analysis for static detection of web application vulnerabilities," in *Proceedings of the 2006 workshop on Programming languages and analysis for security*. ACM, 2006, pp. 27–36.

[4] I. Medeiros, N. F. Neves, and M. Correia, "Automatic detection and correction of web application vulnerabilities using data mining to predict false positives," in *Proceedings of the 23rd International Conference on World Wide Web*. ACM, 2014, pp. 63–74.

[5] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th International Conference on Software Engineering*. ACM, 2008, pp. 171–180.

[6] I. Medeiros, N. F. Neves, and M. Correia, "DEKANT: a static analysis tool that learns to detect web application vulnerabilities," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 1–11.

[7] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[8] P. Godefroid, M. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," pp. 1–20, 2012.

[9] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in *Proceedings of the 21st USENIX Conference on Security Symposium*, Aug 2012, pp. 26–26.

[10] F. Duchène, S. Rawat, J. Richier, and R. Groz, "Ligre: Reverse-engineering of control and data flow models for black-box XSS detection," in *Proceedings of the 20th Working Conference on Reverse Engineering*, Oct 2013, pp. 252–261.

[11] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Kameleonfuzz: evolutionary fuzzing for black-box xss detection," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. ACM, 2014, pp. 37–48.

[12] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy*, 2017, pp. 579–594.

[13] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Network and Distributed System Security*, vol. 8, 2008, pp. 151–166.

[14] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[15] S. W. Boyd and A. D. Keromytis, "SQLrand: Preventing SQL injection attacks," in *Proceedings of the 2nd Applied Cryptography and Network Security Conference*, 2004, pp. 292–302.

[16] 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*, Sep. 2005, pp. 106–113.

[17] W. Halfond and A. Orso, "AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2005, pp. 174–183.

[18] W. Masri and S. Sleiman, "SQLPIL: SQL injection prevention by input labeling," *Security and Communication Networks*, vol. 8, no. 15, pp. 2545–2560, 2015.

[19] I. Medeiros, M. Beatriz, N. F. Neves, and M. Correia, "Hacking the DBMS to prevent injection attacks," in *Proceedings of the 6th ACM on Conference on Data and Application Security and Privacy*, 2016, pp. 295–306.

[20] WASC, "Web Application Firewall Evaluation Criteria," Web Application Security Consortium, Tech. Rep., Jan. 2006.

[21] "OWASP Best Practices: Use of Web Application Firewalls, version 1.0.5," Mar. 2008.

[22] Imperva, "Hacker intelligence initiative, monthly trend report #8," Apr. 2012.

[23] K. Kuchcinski and R. Szymanek, "JaCoP Library User's Guide," available: http://jacopguide.osolpro.com/guideJaCoP.html. [Online; accessed 21-May-2017].

[24] K. Marriott, P. J. Stuckey, L. Koninck, and H. Samulowitz, "A minizinc tutorial," 2014, technical report, 2015. http://www. minizinc. org/downloads/doc-latest/minizinc-tute. pdf.

[25] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[26] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.5," Department of Computer Science, The University of Iowa, Tech. Rep., 2015, available at http://www.SMT-LIB.org.

[27] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "S3: A symbolic string solver for vulnerability detection in web applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1232–1243.
[28] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems*, vol. 1, no. 4, pp. 323–337, 1992.
[29] N. Surribas, "Wapiti," available: http://wapiti.sourceforge.net. [Online; accessed 23-March-2018].
[30] J. Boggiano, "Monolog," available: https://github.com/Seldaek/monolog. [Online; accessed 23-March-2018].
[31] "SAMATE - Software Assurance Metrics And Tool Evaluation," available at https://samate.nist.gov. [Online; accessed 21-Feb-2018].
[32] "DVWA - Damn Vulnerable Web Application," available at http://www.dvwa.co.uk/. [Online; accessed 21-Feb-2018].
[33] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, vol. 2, no. 6, pp. 76–79, 2004.
[34] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *USENIX Security Symposium*, 2012, pp. 445–458.
[35] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations." in *USENIX Security*, 2013, pp. 49–64.
[36] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow, "jäk: Using dynamic analysis to crawl and test modern web applications," in *International Workshop on Recent Advances in Intrusion Detection*, 2015, pp. 295–316.
[37] P. Bisht, T. Hinrichs, N. Skrupsky, and V. Venkatakrishnan, "Waptec: whitebox analysis of web applications for parameter tampering exploit construction," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011, pp. 575–586.
[38] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren, "Automated repair of html generation errors in php applications using string constraint solving," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 277–287.
[39] I. Enderlin, A. Giorgetti, and F. Bouquet, "A constraint solver for php arrays," in *2013 IEEE 6th International Conference on Software Testing, Verification and Validation Workshops*, 2013, pp. 218–223.
[40] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the Network and Distributed System Security Symposium*, 2016.