Athena: A framework to automatically generate security test oracle via extracting policies from source code and intended software behaviour

Hossein Homaei[a], Hamid Reza Shahriari[a,*]

[a]*Department of Computer Engineering and Information Technology, Amirkabir University of Technology, Tehran, Iran*

Abstract

Context: Software security testing aims to check the security behaviour of a program. To determine whether the program behaves securely on a particular execution, we need an oracle who knows the expected security behaviour. Security test oracle decides whether test cases violate the intended security policies of the program. Thus, it is necessary for the oracle to model the detailed security policies. Unfortunately, these policies are usually poorly documented. Even worse, in some cases, the source code is the only available document of the program.

Objective: We propose a method to automatically extract the intended security policies of the program under test from the source code and expected execution traces. We introduce a security test oracle, Athena, which utilises these policies to differentiate between the secure and potentially insecure behaviour of the program.

Method: We use a hybrid analysis approach to obtain the intended security policies. We investigate the program statements (gates) in which the software communicates with the environment. We analyse the transmitted messages in the gates and the control and data flow of the program to extract some security properties. Moreover, we specify the intended navigation paths of the program.

*Corresponding author

Email addresses: homayi@aut.ac.ir (Hossein Homaei), shahriari@aut.ac.ir (Hamid Reza Shahriari)

November 10, 2018

These properties and paths form the expected security policies. Athena utilises these policies to detect potential security breaches.

Results: Investigating common types of software vulnerabilities illustrates the flexibility of Athena in modelling various kinds of security policies. Moreover, we show the usefulness of the method by applying it to the real web applications and evaluating its capability to detect actual attacks.

Conclusions: Our proposed approach takes a step towards solving the test oracle automation problem in the domain of security testing.

Keywords: Software Security, Test Oracle, Vulnerability Analysis

---

1. Introduction

Software testing is defined as "a way to verify whether the system under test behaves in accordance with its specification through a controlled execution" [1]. It consists of selecting test cases, running the program under test, and examining the outputs [2]. The mechanism that examines the output and determines whether the system under test behaves correctly on a particular execution is called a test oracle [3, 4].

In contrast to software testing, "Security testing identifies whether the specified or intended security features are implemented correctly" [5]. Software security testing and software testing take the same steps, but they examine the program behaviour from two different perspectives. Security testing mechanisms examine the program to determine whether the system under test behaves securely (rather than correctly in the software test). For this purpose, analysers need a security test oracle. The security test oracle problem can be defined as the challenge of distinguishing between the desired secure behaviour from the potentially insecure behaviour of the program.

The main challenge of developing a complete test oracle, in general, is to generate expected behaviour automatically [6]. Complicated requirements of automated test oracle generation cause most analysers to provide expected values manually [7]. Thus, the test oracle problem inhibits greater progress in

automated testing methods and tools [8]. Although there have been some research efforts to overcome the odds, the problem of constructing automated or semi-automated oracles is open [1, 9, 10].

In this paper, we propose a framework to extract the expected security behaviour of programs automatically. This behaviour forms the basis of the security test oracle.

As stated by Avancini and Ceccato [11], a security test oracle checks whether security test cases actually expose application vulnerabilities. A software vulnerability is defined as an instance of a software mistake such that its execution can violate the explicit or implicit security policy [12]. Accordingly, security test oracle should determine whether test cases violate the intended security policies of the program. Therefore, it is necessary for the security test oracle to model the detailed security policies.

Unfortunately, security policies are often stated in general terms such as confidentiality, integrity, and availability [13]. Since the policies are specified in abstract terms, it is difficult in practice to investigate whether a security breach occurs.

Moreover, each program may have its specific security policies and requirements, which are not well documented. Specifically, it is common when we use Agile software development methodology or reuse an open source software. In this case, the source code may be the only available document to describe software properties. Therefore, it would be a valuable strategy to extract the intended security policies from source code instead of design models such as UML diagrams.

However, investigating the source code on its own is not sufficient to extract the expected security policies. When we reuse an open source component to utilise some of its specific features, some unnecessary features and codes may be inadvertently imported to our program. Thus, we need to execute our intended use case scenarios to ensure that the extracted policies are compatible with our real needs.

In this paper, we use a combination of static and dynamic analysis to ob-

tain the detailed security policies of programs. We identify the source code statements that are used as contact points between the program and environment. We call these points the gates. We employ the normal data-flow and the expected user privileges to extract the security properties of each gate. We also run the program using expected inputs and extract the normal execution paths. The security properties of the gates, in addition to the normal control flow of the program, represent the correct security behaviour of the program. This behaviour forms the basis of the security test oracle. Any test case which is not compliant with the oracle is considered exploitation because it violates the intended security policies.

We call our framework Athena (the goddess of wisdom who protects the Athena city) because it is a wise protector of the software city. Athena checks the gates and reports any abnormal security behaviour.

Briefly, our framework extracts the security policies automatically, then uses these policies to distinguish between secure and potentially insecure behaviour. Accordingly, we evaluate our approach in two ways: I) We demonstrate how the most common security policies can be extracted by Athena; II) We show the effectiveness of the implemented oracle by applying it to real applications and measuring its accuracy in detecting actual attacks.

The rest of this paper is organized as follows. Section two surveys related work. Section three defines the normal security behaviour and the security test oracle. In the fourth section, we propose a method to automatically extract security test oracle from source code and program execution traces. The fifth section illustrates the implementation. A simple running example is provided in section six. Section seven describes some empirical case studies to demonstrate how the framework can be applied to programs to model the most important security policies. We apply our method to the real applications and demonstrates the feasibility and usefulness of the proposed approach in section eight. Section nine discusses some issues of using security test oracle. Finally, section ten concludes the paper and describes our future work.

2. Related Work

While there are many methodologies for test case generation [14] and security test case generation [15, 16, 17, 18, 19, 20, 21], the test oracle problem is still an open issue [1]. In the last decade, techniques to automatically generate test oracles have attracted a lot of attention [22]. Although there exist some research efforts to automate the test oracle construction [6, 7, 23], less attention is paid to develop a security oracle [11].

Some software vulnerability analysis methods such as [24] use system crash as an implicit oracle. Further, the basic idea of fuzzing is to stress a program to find crashes which are interpreted as faults or errors [25]. Some other security testing methods check if the attack string used as an input is repeated in the output. For example, in [26], the oracle examines whether a web page contains the same JavaScript statement used in the corresponding test case. However, Li and Offutt's experiment [7] showed that testers should check more of the program state than just runtime exceptions. Furthermore, Staats et al. [27] claimed that considering internal state information in test oracles can improve the power of testing.

Automated dynamic web vulnerability scanners assess the application under test in three phases: crawling, generating specially-crafted inputs (attacks), and response analysis. Their analysis module analyses the HTTP responses returned by the web application in response to the attacks launched by the attacker module to detect possible vulnerabilities. For example, if the page returned in response to the SQL injection test vector contains a database error message, the analysis module infers the existence of this vulnerability. Obviously, black-box testing is not capable of precisely keeping track of the state of the application [28]. Moreover, dynamic scanners fail to detect application-specific vulnerabilities [28]. Furthermore, they may have trouble linking a later observation with the earlier injection event [29].

Srivastava and his colleagues' paper [30] is one of the first researches that introduces the concept of security test oracle. The key idea is to use any incon-

sistency between implementations of the same API as a security policy oracle. Their method takes an API and multiple implementations of it, definitions of security checks, and security-sensitive events as inputs. They investigate the required security checks before reaching security-sensitive events. If the given implementations vary in the security checks, their method will detect and report the inconsistency. This approach may produce false negatives if the same semantic bug occurs in all implementations of a system under test. Moreover, the assumption of the existence of multiple independent implementations of the same API, cannot be extended to all software components.

Avancini and Ceccato [11] have proposed a security oracle for Cross-Site Scripting vulnerabilities. They collect HTML pages in safe conditions and construct the safe model of the application under test as an oracle. An input is classified as an attack if the corresponding response page violates the safe model.

In [31], tree kernel methods have been used to train a classifier as a security oracle. In this approach, the expected parse trees of HTML response pages are constructed based on the results of a learning phase. The learning phase contains attacks and safe executions as negative and positive examples respectively. A web page is represented by the parse tree of the corresponding HTML code. Thus, if an input leads to an HTML response which is inconsistent with the expected parse tree, the attack will be detected.

In [32], a new tool called Circe has been introduced for XSS testing. The oracle construction phase of this tool encompasses three steps: test case perturbation, parse tree abstraction, and abstraction merge. A set of test cases is generated using mutation operators and concrete symbolic execution. The program under test is executed using these test cases. The response pages are collected and parsed. The details of parse trees are removed. Finally, The safe model is constructed by combining the abstract parse trees. In the testing phase, any new input that does not satisfy the safe model will be classified as code injection.

Bozic et al. [33] have proposed a method to generate test cases for exploiting XSS vulnerabilities. Their system includes an oracle that detects XSS attacks.

Obviously, their oracle cannot be generalized to detect other kinds of vulnerabilities. As a future work, they point out the oracle problem. They have also mentioned that a better oracle can lead to better test results.

Another approach for security analysis is to find vulnerabilities statically in the code, rather than testing the program dynamically. These methods specify wrong things rather than correct ones. They search for the predefined bug patterns in the code. This method has been applied to software bug detection [34] and security vulnerability analysis [35, 36]. This technique requires well-defined patterns that can be applied to various contexts. Otherwise, they are exposed to a high false positive rate. Moreover, these patterns do not capture context-specific semantics. Thus, they inherently suffer from the problem of missing vulnerabilities that violate a unique policy.

In conclusion, a few papers specifically deal with the security test oracle problem. Most of these researches consider only particular vulnerabilities. Moreover, there exist some security testing methods that capture program crashes to verify the test results. This approach is not sufficient to detect various security vulnerabilities because an attack may have other consequences rather than a system crash. Generally, the methods that do not consider the context-specific semantics of the program or internal program states will be incapable of detecting lots of vulnerabilities. In the next section, we propose a framework that models the expected security behaviour of programs considering these issues. This behaviour is extracted from source code and intended applications of the program. Thus, it is not required to access the design-level documents of the system. Moreover, we do not need to have multiple implementations of the system under test.

3. Athena Framework

Figure 1 provides an overview of the Athena framework. It extracts the expected normal security behaviour of a program using static and dynamic analysis. It also specifies the security behaviour of program on any particular

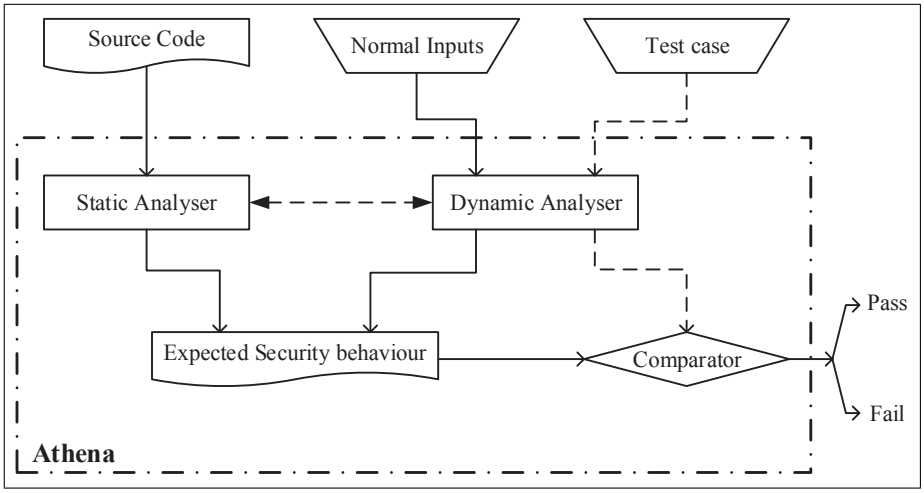

Figure 1: Athena framework

execution. This behaviour is compared with the expected one. Any deviation from normal behaviour is reported as a potential attack. The most important part of Athena framework is the model of security behaviour. In the remainder of this section, we define this model.

The program may have different behaviour when it is run by different users. Moreover, the environment in which the program is executed may affect the behavioural analysis. Therefore, we define the security behaviour of a program considering its environment and the user who runs it.

Definition 1. Security behaviour of program P utilised by user type u in environment e is denoted by P_u^e and defined as a 6-tuple $P_u^e =< G, p_0, p_f, A, S, M >$ where G is a finite set of all gates or channels used by program to communicate with the environment, $p_0 \in G$ and $p_f \in G$ are the initial and final gates (entry and exit points) of the program respectively, A is a set of arcs $A \subseteq G \times G$ that connects gates, entry point, and exit point to each other, S is a finite set of security properties, and M is a multivalued function $M : G \rightarrow S$ which declares security properties of gates.

If P_u^e is extracted from expected interactions between u and P, we call it normal security behaviour and denote it by NP_u^e.

In the above definition, gates are the points where the program communicates with the environment. More precisely, a program statement is a gate if and only if it reads from, writes to, or executes on an environmental resource. Each gate may have some security properties that specify how the program accesses the environment resources from that gate. We define the security property as follows:

Definition 2. A security property S is defined as a triple $S =< A,\ O,\ C >$ where

- A is an access type specified based on the resource usage;
- O is an object accessed by the program;
- C is a collection of conditions that should be met to gain access to the object.

Conditions can be divided into three categories: direct, indirect and local, which are called C_{dir}, C_{ind}, and C_{loc}, respectively. More formally, $C = C_{dir} \wedge C_{ind} \wedge C_{loc}$. These categories are described as follows:

- Direct conditions, C_{dir}, specify whether the communications through the current gate directly depend on the result of communications in other gates. For example, if an input, I, is used as an argument, arg, of an output gate, the condition $arg = I$ will be a direct condition for the output gate.
- Indirect conditions, C_{ind}, are the gate-dependent statements in the code that should be satisfied to reach the current gate. For example, if an input, I, is checked against a constant, C, in a conditional program statement before reaching a database gate, the condition $I = C$ will be an indirect condition for the database gate.
- Other conditions are classified as local conditions, C_{loc}. These conditions may vary, depending on the gate types. Access time and buffer length are

two examples of local conditions which are used in the security properties of database and memory gates respectively. These conditions may be specified manually or automatically. For example, analysers should specify the access time manually if it is required. However, buffer length can be automatically obtained from source code.

For example, assume that the program includes a gate (statement) for communicating with the file system. Here is a simple security property that may occur in the gate: reading (access type) from the file "X" (object) is permitted if the access is requested in the work hours and the user is authenticated (conditions).

Security test oracle is an entity who knows the secure behaviour of the program. Thus, we can define the security test oracle as the union of all normal security behaviour of the program.

Definition 3. The Security Test Oracle (STO) of program P in the environment E is denoted by STO_P^E and defined as follows:

$$STO_P^E = \bigcup_{\forall ut_i \in UT; \forall r \in E} NP_{ut_i}^r$$

Where UT (abbreviation of user type) encompasses all possible types of benign users including guest ones.

For each test case, the security test oracle should specify whether it is an exploit. We say that a program is exploitable if there exist some inputs so that executing the program using these inputs leads the program to behave insecurely. The insecure behaviour of a program is an anomaly or an attack. Based on definition 1, we can define attack as a violation of security properties of a gate or an unauthorized deviation from the normal flow of the program.

Athena traces the execution path of the program and checks the security properties of the gates to specify whether it is an exploit. Deviation from the normal flow of the program or violation of security properties of the gates will be reported as a potential attack.

4. Extracting Security Test Oracle

We use a hybrid approach, a combination of static and dynamic analysis, to extract security test oracle from the program. First, the analyser executes the intended use case scenarios to collect dynamic information. Next, the Interprocedural Control Flow Graph (ICFG) and Data Flow Graph (DFG) are extracted from the source code. The test oracle is constructed by combining these graphs with dynamic information. In this section, we explain these processes in more detail.

As mentioned above, the first step of constructing the oracle is collecting the dynamic information. This information includes the execution paths and the gate messages. We use coverage analysis tools to trace the paths. Moreover, we instrument the program to capture the messages transmitted via the gates.

A gate is a method by which the program communicates with the environment. For example, executeQuery is a database gate in Java programming language. Each gate is defined by the method signature and its parameters that should be captured. We store gate definitions in a text file. Athena uses these definitions to instrument the program. To log the gate messages, the instrumented program is executed using normal test cases.

Normal test cases are the inputs and execution conditions that form the benign (non-malicious) use of the program. These test cases can be specified by the analyser based on the expected use case scenarios. In this case, normal test cases are being used as a proxy for use case scenarios. There are some techniques [37] to extract test cases from use cases. There are also lots of test case generation methods surveyed in [14]. We do not investigate these methods because generating test cases is beyond the scope of the paper. However, we put forward the two following suggestions.

If you download an open source component or use a third-party application, we suggest using expected scenarios to construct the oracle. For example, assume that you download an open source software that handles the authen-

tication process using the RBAC[1] approach. The program specifies a default password for the super-user (in addition to the admin) role. Super-user can perform some critical tasks such as backing up the keys stored in the database. However, you never assign this role to the users. You do not even expect the program to carry out this task. Thus, there exist some paths in the program that are never used in your application but can be used by other companies and applications. In this case, we recommend that each company use its intended use case scenarios to construct the oracle. By contrast, if you want to extract the oracle from your self-developed program, it will be possible to use other test case generation methods, in addition to the previous one. In this case, the analyser can repeatedly execute the program by different test cases until the prime-path coverage criterion [38] is satisfied.

During program execution, the coverage analysis tool traces the paths and generates the coverage report. Simultaneously, the gate messages are logged in appropriate files. This information is passed to the following pseudo-code. The algorithm takes the source code, coverage report, logged messages, and manual local conditions and generates a graph that represents the normal security behaviour of the program.

```
Inputs: src_files, cov_rpt, gate_msgs, man_con
Output: normalModel
Function:
// 1. Extract parse tree from source code
ParseTree[] parseTrees = new ParseTree[src_files.length];
for (i = 0; i < src_files.length; i++)
 parseTrees[i]= Parser(src_files[i]);

// 2. Generate CFGs
// Visit all nodes in the parse tree and build CFG for all methods
ControlFlowGraph[] cfgs = new ControlFlowGraph[src_files.length];
for (i = 0; i < src_files.length; i++){
 ControlFlowVisitor cfgvisitor = new ControlFlowVisitor();
```

[1]Role Based Access Control

```
 cfgvisitor.visit(parseTrees[i]);
 cfgs[i] = cfgvisitor.getGraph();
}

// 3. Generate DFG
// Visit all nodes in the parse tree and find all def-use pairs
DFNode[] dfNodes = new DFNode[src_files.length];
for (i = 0; i < src_files.length; i++){
 DefUseVisitor duvisitor = new DefUseVisitor();
 duvisitor.visit(parseTrees[i]);
 dfNodes[i] = duvisitor.getDFNodes();
}
// Traverse cfgs in a depth-first manner and connect defs to uses
DataFlowGraph[] dfgs = new DataFlowGraph[src_files.length];
for (i = 0; i < src_files.length; i++){
 defuseTraverse(dfgs[i], cfgs[i], dfNodes[i]);

// 4. Generate ICFG
// Visit all nodes in the parse tree and find all function calls
ControlFlowGraph icfg = new ControlFlowGraph();
Map callMap = new Map<Node, Callees>;
for (i = 0; i < src_files.length; i++){
 ICFGVisitor icfgvisitor = new ICFGVisitor();
 icfgvisitor.visit(parseTrees[i]);
 callMap.add(icfgvisitor.getCallerCalleeMap());
}
// Add each cfg to the icfg graph
for (ControlFlowGraph cfg: cfgs)
 icfg.addGraph(cfg);
// Add function call edges to the icfg
for(CFGNode node: icfg)
 for(CFGNode callee: callMap(node)){
  icfg.addEdge(node, callee, label.CALLS);
  icfg.addEdge(callee.exitPoint(), node, label.RETURN);
 }

// 5. Generate normal model
// Generate a graph containing icfg and dfg information
```

```
oracleGraph normalModel = new oracleGraph(icfg, dfgs);
// Traverse the graph and remove uncovered nodes and edges
slicing(normalModel, cov_rpt);
// Traverse the graph, extract conditions, and add them to model
extractConditions(normalModel, gate_msgs, man_con);
```

The main steps of the algorithm are: Extracting the program parse tree from source code; Extracting the control and data flow graphs from the parse tree; Combining the dynamic analysis results with these graphs to form the oracle.

We use a parser to obtain the parse tree. Control and data flow graphs are directly extracted from the parse tree. We visit all statements in the source code and create appropriate nodes and edges of the graphs. Ammann and Offutt [38] have described this procedure in detail. In the remaining of this section, we explain the last step of the algorithm, generating the normal model.

As described in definition 1, program gates form the nodes in the P_u^e model. To determine which nodes should be connected to each other in the normal model, we utilise the coverage report. The ICFG nodes and edges which are visited during program execution are marked. Each visited node that contains a gate definition makes a node in the NP_u^e model. Every subpath that connects two visited gates in the ICFG is modelled as an arc in the normal program behaviour. If there are two gates in the same node of the ICFG, they will be connected by an ϵ arc. We also add two empty nodes p_0 and p_f to the model as mentioned in definition 1. All initial and final gates are connected to the initial and final places respectively.

After constructing the basic structure of the model, we should specify the security properties of the gates as described in definition 2. For each gate, the conditions are extracted as follows:

- Indirect conditions are the conditional statements that their variables depend on an environmental variable received from a prior gate. We name these statements effective conditional statements. All satisfied effective conditional statements, from the start node to the current gate, are cumulatively collected and saved in the last edge before reaching the gate.

In other words, we specify which effective conditions should be met to reach a gate.

To decide whether a conditional statement is an effective one, we benefit from the DFG. We follow the program execution path and register all variables that their values depend on the communications on the gates. If the variable used in the conditional statement depends on an environmental variable, the condition will be effective. Thus, it will be saved as an indirect condition in all the subsequent gates in the corresponding path of the ICFG except when there exists a new definition of the variable that changes the dependency relation in the rest of execution path. For example, suppose that v is an input variable of gate $g1$ which is used in a conditional statement C_1 before reaching gate $g2$. Statement C_1 is saved in C_{ind} of gate $g2$.

- direct conditions specify the data dependencies between variables used in the current gate and the variables used in the earlier gates. Similar to the method used to construct C_{ind}, we use DFG to form the C_{dir}. If an input variable which is received by the program in a gate affects some other gates, it should be stated in the security properties of those gates. Thus, all variables that are entered from the environment through the gates and are ancestors of the target variable in the graph form the direct condition set. This will help us to detect some kinds of taint flows.

- The local conditions are extracted from the gate message logs or specified by the analyser. As mentioned in section 3, security properties should be specified based on the gate type. For example, a simple security property of an output gate may exclude the <script> tags. By contrast, the security property of a database gate may describe authorized queries. Since there are different types of gates, we provide developers with various mechanisms to extract the local conditions. In the current implemented version of the framework, the available mechanisms are searching the message content, obtaining the message length, evaluating Javascript, and extracting the

parse tree of the strings that their ANTLR grammars are available. We will try to add more capabilities in the future implementation. More details about extracting different kinds of local conditions for various gate types will be presented in section 7.

These properties in addition to the gate connections construct normal security behaviour of the program under test.

5. Implementation

We implemented our proposed method to extract security test oracle from programs written in Java programming language. Figure 2 shows the whole process of oracle extraction. The third party tools that we use in our implementation are also demonstrated in the figure.

In the first step of the static analysis phase, we extract the parse tree from source code. We use ANTLR[2] [39] for this purpose. "ANTLR is a powerful parser generator for reading, processing, executing, or translating structured text or binary files[3]." Since we implement our framework for Java programs, we utilise the Java grammar[4] to generate the parser.

ANTLR automatically generates parse-tree walker in the form of visitor pattern. We develop a program which uses this walker to extract the ICFG and DFG from Abstract Syntax Tree (AST). Our program is designed to operate at the source code level rather than byte-code level. Thus, "the high-level abstractions are not compiled away during the translation to intermediate code" [40]. Consequently, the results would be more understandable.

In the dynamic analysis phase, we use OpenClover[5] code coverage tool to specify how the code statements are covered during program executions. We develop a Java program that parses the coverage reports and combines this

[2] ANother Tool for Language Recognition
[3] `http://www.antlr.org`
[4] `https://github.com/antlr/grammars-v4`
[5] `http://openclover.org`

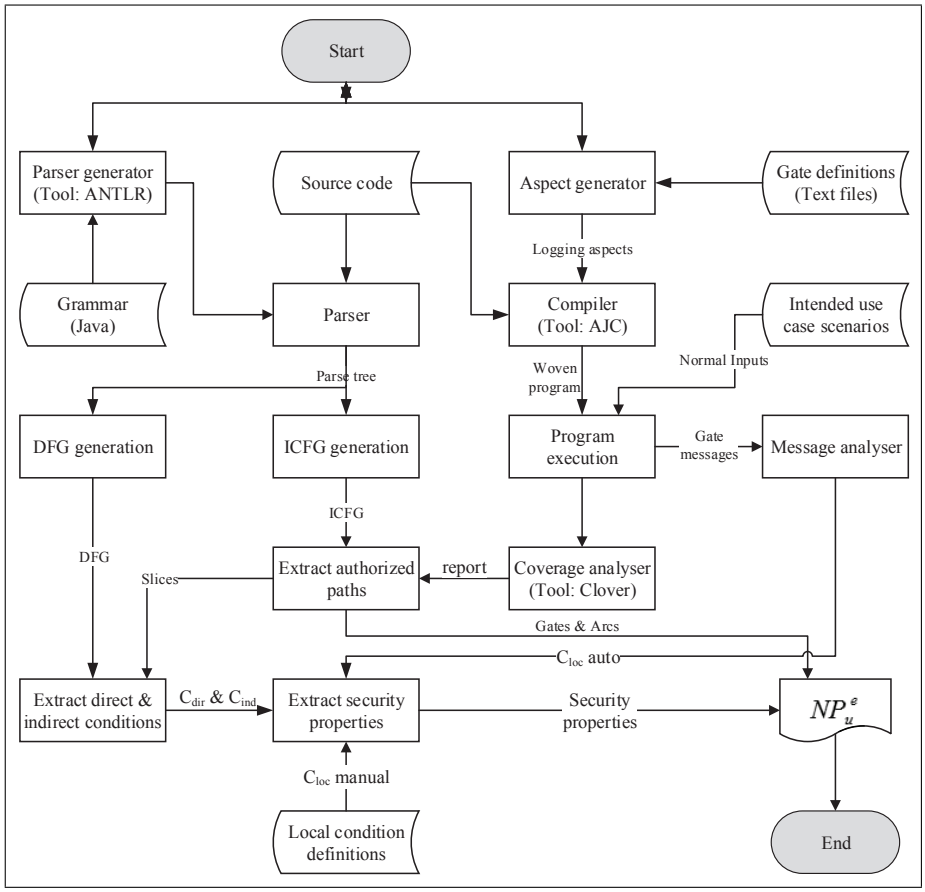

Figure 2: Extracting normal security behaviour of a program

information with the ICFG graph as mentioned in the previous section. Our implementation can also parse the reports of CodeCover[6] glass-box testing tool. In the future implementation we will try to support more dynamic tracing tools such as Cobertura[7] and JaCoCo[8].

In addition to the execution paths, we should log the gate messages. We instrument the program using logging aspects to collect these messages. Logging aspects "define the interception points needing logging and invoke the logging API upon the execution of those points" [41]. The program under test and the logging aspects are compiled by AspectJ. As a result, these aspects are woven into the program to specify where and how to log the messages.

Although it is possible to use a set of predefined logging aspects, it is not a rational approach to framework designing because I) the predefined set may be incomplete and therefore some required messages are missed; II) the analyser may want to investigate a set of specific gates and hence he does not need to capture other messages. For these reasons, we provide a mechanism that enables analysers to extend or customize the predefined list.

We develop a component, AspectGenerator, to automatically produce logging aspects. AspectGenerator uses gate definitions to produce the aspects. Each gate definition contains the method signature and the parameter position that should be captured during program execution. Gate definitions are introduced in the text files called gate lists. AspectGenerator reads the lists and automatically generates appropriate advice[9] for each gate to log the value of the specified parameter.

We have collected a list of commonly used Java gates and stored them in the appropriate text files such as DBGates and InputGates. The analysers can use the default lists or modify them to suit their needs. For example, the analyser

[6] `http://codecover.org/index.html`

[7] `http://cobertura.github.io/cobertura`

[8] `http://www.jacoco.org/jacoco`

[9] In the aspect-oriented programming, an advice is a code to be executed when a join point is reached in the application code [42].

can add a new method signature to the text file and specify the parameters that he wants to capture. AspectGenerator reads the specified gate declaration and automatically produces appropriate aspect. AspectJ weaves this aspect into the program. Thus, the messages transmitted via the gate will be captured during the program execution.

This strategy is invaluable to analysers, particularly when the system under test uses a third party library that has its own customized gates. Moreover, if a new vulnerability is reported in a specific gate or a new vulnerable library is used in the program, it will be possible to log its suspicious messages. In addition, the analyser can exclude the gates if he does not want to model all kinds of security policies.

The dynamic reports are combined with the static analysis graphs using the Java program developed by the authors. This program constructs the security test oracle as described in the previous section.

6. Running Example

In this section, we illustrate how the security test oracle is extracted from source code by a simple running example. Following code snippet is a Java method implemented in the 'CWE89 SQL Injection Environment execute 01' test case in Juliet [43] test suite for Java[10] with a few slight modifications:

```
1 public class CWE89_SQL_Injection__Environment_execute_01 extends
      AbstractTestCase{
2   public void bad() throws Throwable{
3       String data;
4       data = System.getenv("ADD");
5       Statement sqlStatement = null;
6       try{
7           dbConnection = IO.getDBConnection();
8           sqlStatement = dbConnection.createStatement();
```

[10] https://samate.nist.gov/SRD/testsuite.php

```
9           Boolean result = sqlStatement.execute("insert into users (
                status) values ('updated') where name='"+data+"'");
10          if(result)
11             IO.writeLine("Name, " + data + ", updated successfully"
                   );
12          else
13             IO.writeLine("Unable to update records for user: " +
                   data);
14       }
15       catch (SQLException exceptSql){
16          IO.logger.log(Level.WARNING, "Error getting database
                connection", exceptSql);
17       }
18    }
19 }
```

Athena extracts the control and data flow graphs of this method. These graphs are depicted in figure 3. Bold and dashed lines show the control and data flows respectively.

The source code has an input gate, 'System.getenv', and a database gate, 'Statement.execute'. For simplicity, assume that we do not consider other types of gates in this analysis. Thus, the nodes number 4 and 9 construct the basic structure of NP_u^e. We name the gates G1 and G2 respectively.

Athena also generates logging aspects. Since the mentioned gates have been declared in the default gate definition files, the corresponding aspects are automatically generated by AspectGenerator component. For example, 'System.getenv' is declared in InputGates.txt as follows:

```
String java.lang.System.getenv(String); -1
```

-1 in the mentioned declaration states that the return value of the method should be logged. AspectGenerator automatically generates the following advice for this gate:

```
after() returning (String argValue): call(String java.lang.System.
    getenv(String)) && !within(InputGates_Aspect){
```

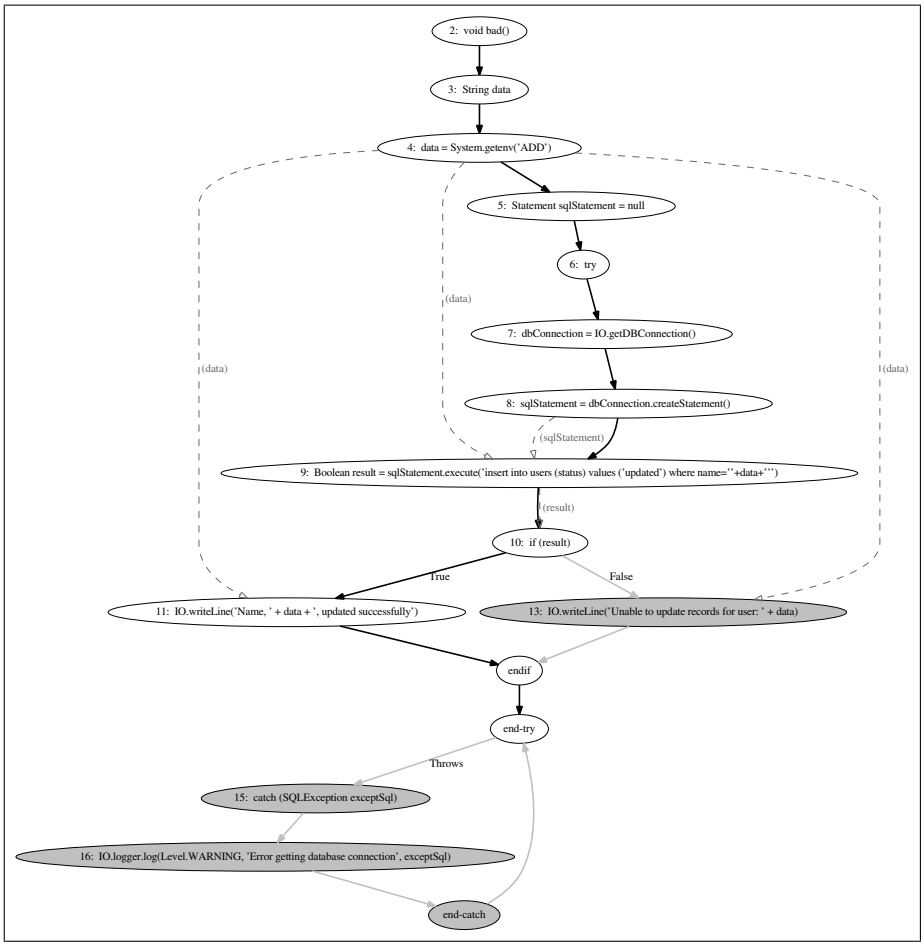

Figure 3: CFG and DFG of the running example

```
        String staticInfo = getStaticJoinPointInfo(
            thisJoinPointStaticPart);
        new File("./AspectsLoggedMessages").mkdirs();
        String filePathtoSaveMessageLog = "./AspectsLoggedMessages/"
            + staticInfo.substring(0, staticInfo.indexOf(";")).trim
            () + ".txt";
        try{
            FileWriter fw = new FileWriter(filePathtoSaveMessageLog,
                true);
            fw.write(staticInfo.substring(staticInfo.indexOf(";")+1)
                + ";" + argValue + "\n");
            fw.close();
        }
        catch(IOException ioe){
            System.err.println("IOException: " + ioe.getMessage());
        }
    }
```

The aspects are woven into the program by AJC compiler. The woven code is executed using normal intended test cases. For instance, in this example, we set $ADD = Homaei$ and $ADD = Hossein$ in the system environmental variables. These two configurations are used to represent intended scenarios for two user-types U_{local} and U_{guest} respectively. OpenClover traces the program execution and generates an HTML report. Athena parses this report and marks the covered nodes and edges. For example, when the program is run by U_{local}, the node number 13 will not be visited. In figure 3 we depict this scenario. The unvisited entities of the CFG are marked with grey colour.

The next step in the modelling process is to determine the security properties of the gates. This involves specifying C_{dir}, C_{ind}, and C_{loc}.

In the running example, the variable 'data' should be mentioned as an input dependent variable in the node number 9 because it depends on an input gate variable. Thus, the C_{dir} contains the condition $data = G1.data$. Athena traces the DFG to extract this property.

The conditional statement in the node number 10 contains variable `result`

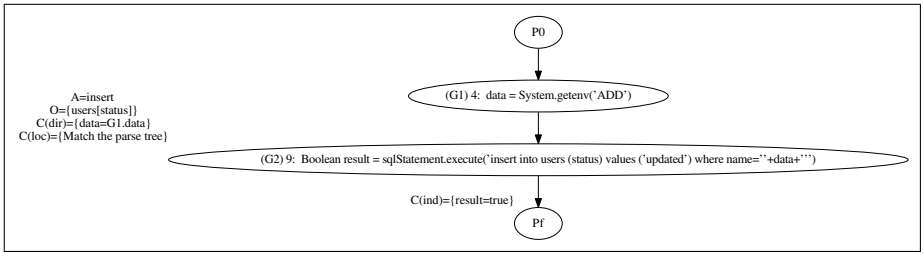

Figure 4: Security test oracle of the running example

which depends on the communications on the database gate. In other words, it is an effective conditional statement and therefore should be saved as an indirect condition in the following nodes. Thus, the C_{ind} of P_f is $result = true$.

Local conditions are extracted from the logged messages of the gates. Each gate may have some specific properties based on the gate type. The details of these properties will be explained in section 7. Since G2 is a DB gate, its security properties should specify the allowed queries. Athena reads the logged message of G2 and combines the obtained information with the DFG information to extract the symbolic mode of the executed query.

For the mentioned example, the logged message in G2 is "insert into users (status) values (updated) where name=Homaei". Athena knows that `data= G1.data` by exploring DFG flows. Moreover, she knows that `G1.data= Homaei` by investigating the logged message in G1. Thus, she replaces the concrete value `'Homaei'` with its symbolic value `data`. In other words, the symbolic query executed in G2 is "insert into users (status) values (updated) where name=data". The current version of Athena supports parsing SQL queries. Athena parses the obtained query and specifies the access type and the accessible objects. Figure 4 depicts the security test oracle of the running example. The security properties of the gates are represented in the left side of the corresponding nodes for better visualization.

Athena monitors the program execution and compares the real and expected properties. Using the oracle as an executable specification form, we can follow the same execution path in the oracle and report every violation. In other words,

we do not need a separate comparator. Instead, we follow the oracle paths based on the recorded actual paths and verify whether the test case is passed based on the security properties of the gates.

For example, assume that a user type U_{guest} attacks the program through passing the condition `if(result)` in the node number 10 of the CFG. In this case, the dynamic behaviour of the program violates the specified security policy, NP_u^e. The oracle forces the `result` variable to be equal to true at P_f. However, the actual value of this variable in the running example is equal to false. Thus, Athena detects the conflict and reports the violation. Moreover, SQL injections are detectable by Athena. If an injection occurs, the actual parse tree of the query will not match the expected one in G2. The details of detecting common types of attacks, including SQL injection, will be demonstrated in the next section.

7. Case Studies

An acceptable framework of security test oracle should be capable of modelling different types of security policies to detect various kinds of breaches. In this section, we demonstrate how Athena model the security policies that should be pursued to protect software against the most common software vulnerabilities [44]. We show that the security policies which are automatically extracted by Athena can be used to detect common types of attacks.

Note that although we introduce some detection mechanisms in this section, analysers can also develop their own methods using our framework. For example, one developer may prefer to use only the static information and develop a taint flow analysis whereas another developer may use a hybrid approach to alleviate the conservative static analysis problem.

7.1. SQL Attacks

7.1.1. SQL injection

SQL injections are malformed inputs that inject unexpected values into database gates. These inputs make it possible to get impermissible access to the

database. To detect SQL injections, the security property of each database gate should specify the allowed queries. Any access control violation in a database gate illustrates an attack.

Obviously, there may be lots of permitted queries in each gate. Since dynamic learning does not guarantee the completeness of the model, there is little likelihood that the normal behaviour encompasses all possible values that can be assigned to the gate variables and hence all possible normal queries and responses. Therefore, we should state this access control policy in abstract terms.

We replace the executed queries with their symbolic forms. We use the parse tree of these symbolic queries as the security property of the gates. For example, assume that the program under test, P, communicates with the table personalInfo in a relational database. Moreover, suppose that the symbolic query "select * from PersonalInfo where ID=id" is stated in a database gate of P. The parse tree of this query is depicted in figure 5. Thus, the formal representation of the security property is stated as follows:

$$S = < select,\ \{R_i \in PersonalInfo \mid R_i.ID = id\},\ C >$$

Athena detects SQL injections by comparing real query structure with the normal one in the same way as described in [45]. For instance, in the mentioned example, the query select * from PersonalInfo where ID='1' or '1'='1'; is reported as SQL injection because its parse tree differs from the expected one.

There are also some SQL attacks which can be detected straightforwardly by analysing the queries. For example, if a query such as 'drop table' is sent from a database gate which is permitted to send 'insert', the proposed method detects the attack. This kind of attack may be the result of parameter tampering or SQL injection. Regardless of the attack origin, we can detect it using its effects on the gate.

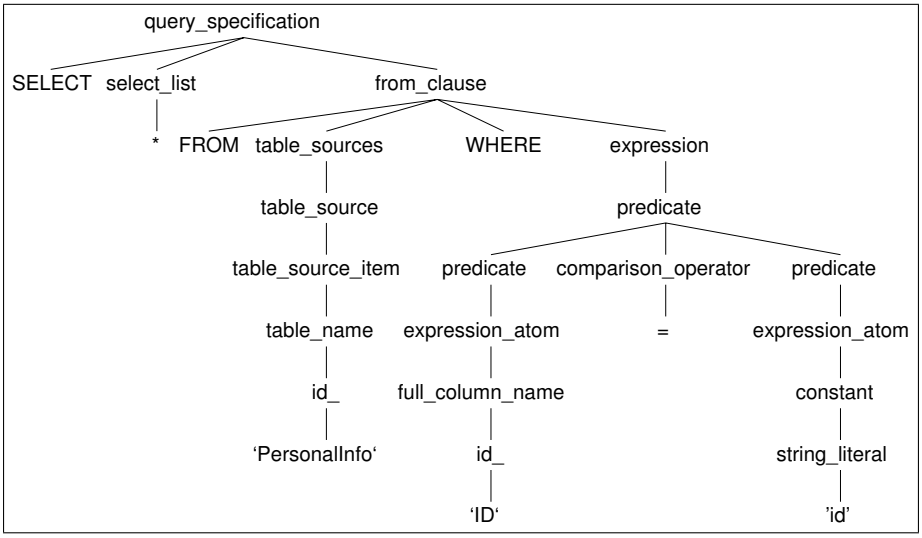

Figure 5: The parse tree of the query "SELECT * FROM PersonalInfo WHERE ID=id"

7.1.2. Query tampering

Another kind of injection attacks occurs when one of the query parameters is tampered by the attacker and replaced with an unauthorized but well-formed value. For example, suppose that fetching personal information from the database requires the identification number of the person who assumed to be logged in to the system. Replacing this number with a fake ID will cause confidential information leakage. In this kind of exploitation, the structure of malicious query is identical to the normal one. However, Athena can detect this attack by using the direct condition set.

For example, figure 6 shows the gates and their communications for a sample web application. The variable `'id'` in the query 'select * from PersonalInfo where ID=id' in gate `G3` depends on the database response in gate `G2`. This dependency is extracted from data flow analysis and is stated in the condition set C in the security property of gate `G3`. In other words, the security property of `G3` contains $C_{dir} = \{id = G2.id\}$. Now, suppose that an attacker replaces the original id with a fake one in the corresponding `HTTP GET` request and therefore accesses the personal information of another person. Athena will detect this attack by

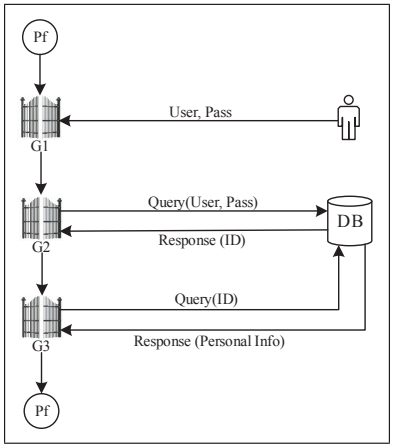

Figure 6: An example of SQL attack through parameter tampering

finding the difference between `G2.id` and `G3.id`.

7.2. XSS Attacks

XSS attacks are causing scripts to be executed on output gates. To detect this type of attack, we determine whether the user is permitted to run scripts in the output gate. Executing scripts in an output gate violates the security property if that gate is not permitted to run scripts. Notice that we consider only the input-dependent scripts in our XSS detection method. Thus, running a consistent script which does not depend on any external variable will not be reported as an exploit.

To check whether the gate is permitted to run scripts, we should investigate the normal strings transmitted via that gate. A simple approach for XSS detection is to search for <script> tags. This approach cannot detect complex XSS attacks. Thus, we have implemented a more accurate method using a javascript engine.

We parse the output strings and produce their parse trees using HtmlCleaner[11]. If the parse tree does not match with the normal one, the unmatched

[11] `http://htmlcleaner.sourceforge.net`

point is investigated to check whether it contains Javascript. We also check all attribute elements for Javascript. We use swt[12] library to examine these strings. The library contains a method that can detect whether a string is a Javascript command.

For example, assume that the string <img src="image.jpg">Image</img>, in which the image.jpg is a dependent string, is a normal string in an output gate. Obviously, in the normal behaviour, scripts are not allowed to execute in this gate. Suppose that the input javascript:alert('XSS') is entered into an input gate and leads the output gate to produce the string <img src="javascript:alert('XSS')">Image</img>. The parse tree of this string is normal because it does not contain any unmatched script element. However, the attribute value is a Javascript which is detected by the proposed method.

Generally, if the normal behaviour of a program does not allow users to run dependent scripts in an output gate, the XSS attacks will be detectable. However, it may be the case that the output gate is permitted to run some scripts. For example, suppose that the normal behaviour of a program utilised by the guest user allows him to run a specific script, X, in the output gate O. Since we model the security properties of output gates by specifying whether the user is permitted to run any arbitrary scripts, we cannot distinguish between benign and malicious scripts. In other words, we suppose that executing any script is allowed in gate O. Therefore, if a script rather than X is executed in gate O, our model will not classify it as an attack. If someone insists on detecting this specific kind of XSS attack, he should capture all normal scripts in the gates. However, we believe that this idea will increase the number of false positive alerts. Thus, we prefer to ignore detecting this specific rare type of XSS attack instead of being confronted with numerous false positives.

Notice that the proposed method is not intended for detecting DOM-based XSS attacks because (in contrast to stored and reflected types) they do not rely on the payload embedded by the server in the response page. In other words,

[12] `https://www.eclipse.org/swt`

"DOM-based XSS vulnerabilities can be executed without the server being able to determine what is actually being executed[13]."

7.3. Code Injections

If a program constructs a code segment using externally-influenced input, an attacker may inject her code into the program. Code injection attacks may have two consequences: deviation from the normal flow of the program and executing a command that is not intended.

The first case is detectable by our model and is interpreted as a control flow violation. The second case occurs when the attacker executes unexpected system commands. Since there is a limited number of intended commands in each gate, we suggest specifying all of them in the security property of the gate. In other words, we use a white-list approach to detect this type of attack, in contrast with XSS, where we used a black-list approach.

Following code snippet is a part of the 'CWE78 OS Command Injection console readLine 01' test case in Juliet test suite [43] for Java with a little modification to save article space:

```
/* read user input from console with readLine */
try
{
    readerInputStream = new InputStreamReader(System.in, "UTF-8");
    readerBuffered = new BufferedReader(readerInputStream);
    /* Input gate */
    data = readerBuffered.readLine();
}
...
String osCommand = "c:\\WINDOWS\\SYSTEM32\\cmd.exe /c dir ";
/* Execution gate */
Process process = Runtime.getRuntime().exec(osCommand + data);
```

[13] https://www.owasp.org/index.php/Testing_for_DOM-based_Cross_site_scripting_(OTG-CLIENT-001)

In this example, there is a straightforward path between the untrustworthy input gate, 'readLine', and the execution gate, 'exec'. This makes the program vulnerable to code injections. However, we use a more restrictive approach to decide whether a test case is an actual attack. We trace the normal behaviour of the program and save all expected commands in the security property of the execution gate. If a test case leads the program to execute a command that is not included in the specified white-list, we notify analyser of potential attack. Note that we exclude the constant parts of the executed string in the execution gate. For example, if the only permissible commands in the execution gate are 'type manual.txt' and 'dir', the security property of the gate will be stated as follows:

$$S = < exec,\ java.lang.Runtime.exec(arg0),$$
$$C_{loc} = \{arg_0 = type\ manual.txt \vee dir\} >$$

Since the dynamic analysis is intrinsically incomplete, some permissible commands may be omitted from the white-list. Thus, it is possible that Athena falsely detects a benign command as attack. However, since our framework is flexible, one can relax the restrictions using regex. If the command options or parameters are not important for the analyser, he can specify only the main command in the security property of the gate. For example, instead of considering 'type manual.txt' in the analysis, Athena can consider the regex 'type *' as the security property of the gate.

7.4. Access Control Violation

Since the security properties of gates describe how users can access the program resources, the access control violation is detectable by Athena. To model access control violations, it suffices to compare the expected user privileges with the real accesses to the resources during the test.

For example, assume that reading the secret file F is permitted just to the admin user from a specific gate of the program. The normal security property of the program in this gate can be modelled as a triple $< read, F, C >$. An

access control violation will be reported if any deviation from this property occurs. For instance, writing on F, or reading an impermissible file are some kinds of detectable attacks. Another example of access control deviation may be a privilege escalation of the guest user which can be modelled as a missed property in P^e_{guest}.

Since access control violation is a general category which encompasses various kinds of attacks, it is not possible to explain it by a single example. However, the following example demonstrates the overall picture of the detection mechanism. Following code snippet is a part of the 'CWE 470 Unsafe Reflection console readLine 01' test case in Juliet test suite [43] for Java with a little modification:

```
String data= "";
...
/* read user input from console with readLine */
readerInputStream= new InputStreamReader(System.in, "UTF-8");
readerBuffered= new BufferedReader(readerInputStream);
data= readerBuffered.readLine();
...
Class<?> tempClass= Class.forName(data);
Object tempClassObject= tempClass.newInstance();
```

Assume that the only class which is instantiated in the normal behaviour of the program is 'Testing.test'. The security property of the usage gate can be stated as follows:

$$S = < Instantiate,\ java.lang.Class.forName(arg_0),$$
$$C_{loc} = \{arg_0 = Testing.test\} >$$

Any attempts to instantiate other classes will be reported as an attack. As described in the previous subsection, we can use regex to relax the restrictions.

7.5. Buffer Overflow

A buffer overflow occurs when an allocated buffer in the memory is overwritten. To detect memory attacks such as buffer overflows, we should check

the buffer borders and investigate whether the boundaries are exceeded. We define memory gates as program statements that write into buffers on the stack. For example, the statement 'strcpy(dest, src)' in the C programming language writes the 'src' string on the 'dest' buffer in the stack and therefore should be considered as a memory gate. The security property of a memory gate should include the destination buffer length.

The general security policy on the buffer size is as follows: the length of the buffer where we want to write to should be larger than the size of reading data. For example, assume that the following code snippet is a part of the program under test written in C language[14]

```
char * dest;
char * src;
int dynamicVar;
//Get dynamicVar from stdin
...
dest = malloc(dynamicVar);
...
strcpy(dest, src); //Memory gate
```

We can declare the security property of the mentioned memory gate as follows:

$$S = < Write,\ strcpy(arg_0, arg_1),$$
$$C_{loc} = \{sizeof(arg_0) \geq sizeof(arg_1)\} >$$

We can extract the size of buffers 'dest' and 'src' using logger aspects and report the attack if the above security property is violated. Notice that we do not develop this mechanism in our implementation because Java programming language checks the bounds and hence is not vulnerable to this kind of attack. However, as mentioned above, it is theoretically possible to detect the attack.

[14]Since the Java has array bounds checking, we explain this subsection by C code.

7.6. Path Violation Attacks

Path violation attacks are the attacks in which the attacker deviates from normal execution or navigation path of the program. Forceful browsing, workflow bypass [46], and some kinds of logical attacks [47] are common forms of path violations. These attacks are detectable by Athena because the deviation from normal paths is interpreted as an attack. For example, suppose that the normal navigation of a website is as follows: user-login→ adding some products on shopping cart→ calculating the tax→ finalizing the purchase. If a test case bypasses the tax-calculation page, the logical attack will be reported.

Obviously, it is not possible to claim that all path violations are security breaches. For instance, in the mentioned example, if the test case is representative of the benign user who adds a product to his shopping cart and tries to buy it in his future login, it will be incorrectly reported as an attack. Athena is misled because the benign path, login→ open the shopping cart→ calculate the tax→ finalize, is not modelled in the training phase. Although this path is not modelled in the normal behaviour, it does not violate any security policies. The solution to mitigate this problem is to increase the precision of the training phase by learning all intended use case scenarios.

8. Practical Evaluation

In this section, we verify whether Athena can be used in real applications. We demonstrate that Athena is capable of modelling real applications and detecting actual attacks.

We use our method to extract security policies of four deliberately insecure web applications: Webgoat[15], Security Shepherd[16], Insecure Web App[17], and mngoat[18]. They have different levels of complexity and use various technologies such as AngularJS, H2 and MySQL databases, and the Spring framework.

[15] https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project
[16] https://www.owasp.org/index.php/OWASP_Security_Shepherd
[17] https://www.owasp.org/index.php/Category:OWASP_Insecure_Web_App_Project
[18] https://github.com/mcgyver5/mngoat

Table 1 shows the number of files and lines of code (LOC) in each application. Obviously, web applications include HTML, JSP, and XML files. However, we counted only the Java files because our current implementation does not support other languages.

Our test platform was a 2.20GHz Intel Core i7 machine running Windows 7 with 8GB RAM. We ran Athena on this platform and produced the normal model for each test application. To determine the maximum possible size (number of nodes and edges) of the extracted models, we assumed that all paths, except the exceptions, are normal. The last two columns of table 1 show the number of nodes and edges of the extracted models, respectively. Moreover, the fourth column of the table demonstrates how long it took Athena to produce the models.

Table 1: Summary of experimental evaluation

Web Application	#Files (Java)	LOC	Time (Sec.)	#Nodes	#Edges
Webgoat	189	25480	49.7	4583	5331
Security Shepherd	144	16071	37.1	2861	3825
Insecure Web App	14	629	8.4	167	149
mngoat	49	2126	5.3	461	344

To evaluate whether it is possible to use Athena for detecting real attacks, we examined the mentioned web applications using SQL injection and XSS attacks. We chose these types of attack because they have been the most common and the most common severe web application vulnerabilities based on the recent survey of Homaei and Shahriari [44]. XSS makes up 13 per cent of all vulnerabilities registered on the National Vulnerability Database (NVD). SQL injection encompasses 20 per cent of the most severe vulnerabilities and about 9 per cent of all vulnerabilities.

We used the proposed methods in subsections 7.1.1 and 7.2 to inspect the mentioned applications for SQL injection and XSS vulnerabilities. As specified in section 4, our implementation is capable of performing some analysis on the

logged messages. These include, but are not limited to parsing SQL queries and assessing whether HTML strings contain Javascript. Thus, our method is capable of examining gate messages for SQL injections and XSS attacks.

Since we examine deliberately insecure web applications, we know which pages are vulnerable to SQL injection and XSS attacks. Thus, in the training phase, we input normal strings into these pages. By normal string, we mean the one that does not lead to an attack, SQL injection or XSS in this context. We utilised Selenium [19] WebDriver and ngWebDriver [20] to automate the testing process. We tested the applications using the suspicious input strings introduced by wfuzz [21] web application fuzzer. This tool identifies two "wordlists" for SQL injection and XSS testing. These lists contain 125 and 39 suspicious strings respectively.

It should be noted that generating test cases is beyond the scope of the paper. The main point in choosing test data is to be unbiased. In other words, we should not intentionally choose the test cases that decrease the false rates or increase the true ones. Since we used an input list that is provided by a third party application, wfuzz, we hope that the input selection process would not bias the results of the study.

For each application, we repeated the testing process twice. First, we checked the application responses to specify which strings can really exploit the application. Secondly, we used Athena to perform the task automatically. The comparison between the results of the first and second tests demonstrates the capabilities of Athena in detecting attacks. Table 2 shows the testing results.

For example, the results show that ten strings, out of 125 ones, can exploit Webgoat SQL injection course. Athena identified all examined SQL injection attacks without any false alarm. In other words, Athena detected these attacks with the 100% detection rate and the 0% false negative and positive rates.

[19] `http://www.seleniumhq.org`

[20] `https://github.com/paul-hammant/ngWebDriver`

[21] `https://github.com/xmendez/wfuzz`

Table 2: Accuracy of attack detection mechanisms

Web Application	SQLi			XSS		
	#Attacks	#Detected	#False alarms	#Attacks	#Detected	#False alarms
WebGoat	10	10	0	9	9	0
Security Shepherd	8	8	0	9	9	0
Insecure Web App	11	11	0	15	14	0
mngoat	12	12	0	22	0	0

There are two points in table 2 need to be further discussed. First, there exists an XSS attack on Insecure Web App that was not detected by our tool. The string that causes the attack is '%3CIFRAME %20 SRC= javascript: alert(%2527XSS%2527)%3E%3C/IFRAME%3E. This string results in producing <IFRAME SRC= javascript: alert(%27XSS%27)></IFRAME> in the output gate. Although Athena detects the mismatch between the normal and the attack parse trees, she assumes that the javascript will not be validated by real browsers. The reason for this error is that the swt widget that we use to evaluate javascript cannot detect the string javascript:alert(%27XSS%27) as a valid javascript. Although it can handle the quotation marks, it cannot interpret the %27 character correctly. Briefly, the failure to detect the attack is due to the inaccuracy of the widget.

Secondly, our implementation cannot detect XSS attacks on mngoat program because there are no output gates in the Java files. The outputs are generated in .jsp files at the client side. Thus, our tool cannot capture them. Note that this is not the weakness of the framework. Instead, the reason for this issue is that the developed tool does not support .jsp analysis in its current version. As mentioned in section 5, our implemented tool analyses Java files. Thus, it cannot examine the programs that are developed in other languages.

9. Discussion

There are various approaches to construct test oracles. Barr et al. [8] have classified test oracle approaches into four categories: specified, derived, implicit,

and no test oracles. They have defined derived test oracle as an oracle that "distinguishes a system's correct from incorrect behaviour based on information derived from various artefacts (e.g. documentation, system executions) or properties of the system under test". Since our proposed method derives the oracle from source code and system execution, we can classify it under the derived test oracle category. This method would be an invaluable strategy, especially in the absence of well-documented security requirements and policies.

In section 7, we explained how Athena detects various kinds of attacks. Generally speaking, all type of attacks that violate some security properties or deviate from normal control flows would be detectable if sufficient details are available in the model driven from our framework. However, there are some kinds of attacks that Athena is unable to identify.

Attacks that simulate the exact normal behaviour of the program will not be detected by Athena. In other words, if the attacker actions and normal program behaviour are identical, it will be impossible to distinguish between normal and abnormal behaviour. For example, Cross-Site Request Forgery (CSRF) and password guessing are not detected by this method.

Moreover, Athena is incapable of testing the cryptographic properties. We assume that the random and secret variables are cryptographically secure. This will lead to the failure to detect some cryptographic attacks. For example, if the pseudo-random number generator function does not work correctly, a non-random number may be generated and used as a nonce in the communication.

Note that the aim of the framework is not to detect all kinds of attacks, but rather the software attacks that occur because of poor implementation. If some security policies are misunderstood in the software design phase and, in consequence, an attack occurs, Athena cannot detect it. For example, if the least privilege policy is not developed correctly in the design phase, Athena cannot identify the relevant exploits in the implemented software. For instance, if the software designer falsely believes that a user needs to have an access privilege on a resource, the test team will also authorise this access. In other words, Athena will infer an unrequired right during the training phase. It leads

to learn imprecise security properties and therefore the security test oracle may fail to classify test cases correctly. Obviously, it is not the disadvantage of Athena, but rather is the result of poor software design.

In addition to the undetectable attacks, we should discuss the accuracy of identifying detectable attacks. The exploit detection rate may depend on the accuracy of the training data, in some cases. The dynamic analysis is intrinsically incomplete. Since we use dynamic analysis to extract some parts of the security test oracle, the derived model may be incomplete. In other words, we cannot ensure that all normal behaviour of the program is modelled. For example, since the exceptions are not represented in the normal operation of the program, Athena considers them attacks. Thus, there may exist some cases in which the test results fail while the system under test is secure.

To achieve better results, it is also possible to give feedback to the analyser. For example, we can identify the statements that are partially covered or not covered in the learning phase. The analyst may classify these statements as security or regular checks. If the conditional statement is classified as a regular (non-security) check, all of its outgoing edges and also all of its successors that have single output will be labelled covered. However, we do not implement this interactive approach because the analyst may not have enough information about all of the statements and conditions in the code and thus he may incorrectly classify the node.

Finally, we should discuss the modelling cost. To construct the security test oracle for a program, we should examine the gates. The modelling cost depends on the number and type of the gates considered in the analysis. The fewer gates the analyser chooses, the fewer program slices Athena investigates. Furthermore, to detect different attacks, different kinds of information are needed to be gathered from the source code and the graphs. Thus, the cost of creating STO will differ based on the analysis purpose. For example, detecting all kinds of memory errors may need taking expensive stack snapshots. However, to identify directory traversal attack, it suffices to specify whether the file access is permitted based on the training data.

10. Conclusions and Future Work

There are several research challenges in software testing including test oracle automation and non-functional property testing [9]. This article tried to take a step towards solving these two research challenges. We proposed a security test oracle framework, Athena, which is capable of specifying various security policies and identify software vulnerabilities. We suggested a method to extract the oracle from source code and normal interactions between program and environment. Thus, it is not required to access the design-level documents of the software. We implemented the proposed framework and provided analysers with a set of capabilities that can be used to specify different security policies. Investigating common types of attacks demonstrated the flexibility of Athena. We applied the framework to detect various attacks as diverse as SQL injection and path violation. Moreover, we illustrated the usefulness of Athena by analysing actual applications and identifying real attacks.

As future work, we plan to translate security properties and normal navigation paths to a checkable form such as code or assertions and check them during execution. As a result, we will be able to embed a tailored software firewall in the program and protect the application from exploitation. We also suggest combining Athena with a monitoring system to detect attacks. Moreover, we intend to use the test oracle to conduct test case generation. It would open a new potential direction in security vulnerability analysis, automatic exploit generation, and automated software repair.

References

[1] R. A. P. Oliveira, U. Kanewala, P. A. Nardi, Chapter Three - Automated Test Oracles: State of the Art, Taxonomies, and Trends, in: A. Memon (Ed.), Advances in Computers, Vol. 95, Elsevier, 2014, pp. 113–199.

[2] E. J. Weyuker, On Testing Non-Testable Programs, The Computer Journal 25 (4) (1982) 465–470.

[3] A. Memon, I. Banerjee, A. Nagarajan, What test oracle should I use for effective GUI testing?, in: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, IEEE, 2003, pp. 164–173.

[4] L. Baresi, M. Young, Test oracles, Tech. Rep. CIS-TR01-02, University of Oregon (2001).

[5] M. Felderer, P. Zech, R. Breu, M. Büchler, A. Pretschner, Model-based security testing: a taxonomy and systematic classification, Software Testing, Verification and Reliability 26 (2) (2016) 119–148.

[6] Vineeta, A. Singhal, A. Bansal, A study of various automated test oracle methods, in: Proceedings of the 5th International Conference of The Next Generation Information Technology Summit (Confluence), IEEE, 2014, pp. 753–760.

[7] N. Li, J. Offutt, An Empirical Analysis of Test Oracle Strategies for Model-Based Testing, in: Proceedings of the IEEE Seventh International Conference on Software Testing, Verification and Validation, IEEE, 2014, pp. 363–372.

[8] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, S. Yoo, The Oracle Problem in Software Testing: A Survey, IEEE Transactions on Software Engineering 41 (5) (2015) 507–525.

[9] A. Orso, G. Rothermel, Software testing: A research travelogue (2000–2014), in: Proceedings of the on Future of Software Engineering, ACM, 2014, pp. 117–132.

[10] M. Pezz, Towards Cost-effective Oracles, in: Proceedings of the 10th International Workshop on Automation of Software Test (AST '15), IEEE, 2015, pp. 1–2.

[11] A. Avancini, M. Ceccato, Grammar based oracle for security testing of web applications, in: Proceedings of the 7th International Workshop on Automation of Software Test (AST), IEEE, 2012, pp. 15–21.

[12] A. Ozment, Improving vulnerability discovery models, in: Proceedings of the 2007 ACM workshop on Quality of protection (QoP '07), ACM, 2007, pp. 6–11.

[13] M. Bishop, Computer Security: Art and Science, Addison-Wesley Professional, Boston, MA, USA, 2003.

[14] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, J. Jenny Li, H. Zhu, An orchestrated survey of methodologies for automated software test case generation, Journal of Systems and Software 86 (8) (2013) 1978–2001.

[15] W. Du, A. P. Mathur, Testing for software vulnerability using environment perturbation, Quality and Reliability Engineering International 18 (3) (2002) 261–272.

[16] S. Sparks, S. Embleton, R. Cunningham, C. Zou, Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting, in: Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC), IEEE, 2007, pp. 477–486.

[17] M. Sutton, A. Greene, P. Amini, Fuzzing: brute force vulnerabilty discovery, Addison-Wesley Professional, 2007.

[18] V. Ganesh, T. Leek, M. Rinard, Taint-based directed whitebox fuzzing, in: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), IEEE, 2009, pp. 474–484.

[19] T. Wang, T. Wei, G. Gu, W. Zou, Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection, in: Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P '10), 2010, pp. 497–512.

[20] W. G. J. Halfond, S. R. Choudhary, A. Orso, Improving penetration testing through static and dynamic analysis, Software Testing, Verification and Reliability 21 (3) (2011) 195–214.

[21] D. Zhang, D. Liu, Y. Lei, D. Kung, C. Csallner, N. Nystrom, W. Wang, SimFuzz: Test case similarity directed deep fuzzing, Journal of Systems and Software 85 (1) (2012) 102–111.

[22] M. Pezz, C. Zhang, Chapter One - Automated Test Oracles: A Survey, in: A. Memon (Ed.), Advances in Computers, Vol. 95, Elsevier, 2014, pp. 1–48.

[23] S. Shahamiri, W. Kadir, S. Mohd-Hashim, A Comparative Study on Automated Software Test Oracle Methods, in: Proceedings of the Fourth International Conference on Software Engineering Advances (ICSEA '09), IEEE, 2009, pp. 140–145.

[24] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, D. R. Engler, EXE: Automatically Generating Inputs of Death, ACM Transactions on Information and System Security (TISSEC) 12 (2) (2008) 10:1–10:38.

[25] B. P. Miller, L. Fredriksen, B. So, An Empirical Study of the Reliability of UNIX Utilities, Communications of the ACM 33 (12) (1990) 32–44.

[26] S. Kals, E. Kirda, C. Kruegel, N. Jovanovic, SecuBat: a web vulnerability scanner, in: Proceedings of the 15th international conference on World Wide Web (WWW 2006), ACM, Edinburgh, UK, 2006, pp. 247–256.

[27] M. Staats, M. W. Whalen, M. P. E. Heimdahl, Programs, tests, and oracles: the foundations of testing revisited, in: Proceedings of the 33rd International Conference on Software Engineering (ICSE), IEEE, 2011, pp. 391–400.

[28] A. Doupé, M. Cova, G. Vigna, Why johnny can't pentest: An analysis of black-box web vulnerability scanners, in: Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'10), Springer-Verlag, 2010, pp. 111–131.

[29] J. Bau, E. Bursztein, D. Gupta, J. Mitchell, State of the art: Automated black-box web application vulnerability testing, in: Proceedings of the 31st

IEEE Symposium on Security and Privacy (S&P '10), IEEE Computer Society, 2010, pp. 332–345.

[30] V. Srivastava, M. D. Bond, K. S. McKinley, V. Shmatikov, A Security Policy Oracle: Detecting Security Holes Using Multiple API Implementations, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11), ACM, 2011, pp. 343–354.

[31] A. Avancini, M. Ceccato, Security Oracle Based on Tree Kernel Methods, in: A. Moschitti, B. Plank (Eds.), Trustworthy Eternal Systems via Evolving Software, Data and Knowledge, no. 379 in Communications in Computer and Information Science, Springer Berlin Heidelberg, 2012, pp. 30–43.

[32] A. Avancini, M. Ceccato, Circe: A grammar-based oracle for testing Cross-site scripting in web applications, in: Proceedings of the 20th Working Conference on Reverse Engineering (WCRE), IEEE, 2013, pp. 262–271.

[33] J. Bozic, B. Garn, I. Kapsalis, D. Simos, S. Winkler, F. Wotawa, Attack pattern-based combinatorial testing with constraints for web security testing, in: Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE, 2015, pp. 207–212.

[34] D. Hovemeyer, W. Pugh, Finding Bugs is Easy, ACM SIGPLAN Notices 39 (12) (2004) 92–106.

[35] M. Martin, B. Livshits, M. S. Lam, Finding Application Errors and Security Flaws Using PQL: A Program Query Language, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05), ACM, 2005, pp. 365–383.

[36] L. Tan, X. Zhang, X. Ma, W. Xiong, Y. Zhou, AutoISES: Automatically Inferring Security Specifications and Detecting Violations, in: Proceedings

of the 17th Conference on USENIX Security Symposium, USENIX Association, 2008, pp. 379–394.

[37] C. Nebut, F. Fleurey, Y. Le Traon, J.-M. Jezequel, Automatic test generation: A use case driven approach, IEEE Transactions on Software Engineering 32 (3) (2006) 140–155.

[38] P. Ammann, J. Offutt, Introduction to Software Testing, 2nd Edition, Cambridge University Press, New York, NY, USA, 2016.

[39] T. Parr, The definitive ANTLR 4 reference, Pragmatic Bookshelf, Raleigh, NC, USA, 2013.

[40] E. Sderberg, T. Ekman, G. Hedin, E. Magnusson, Extensible intraprocedural flow analysis at the abstract syntax tree level, Science of Computer Programming 78 (10) (2013) 1809 – 1827.

[41] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications Co., Greenwich, CT, USA, 2003.

[42] J. D. Gradecki, N. Lesiecki, Mastering AspectJ: aspect-oriented programming in Java, John Wiley & Sons, Inc., New York, NY, USA, 2003.

[43] T. Boland, P. E. Black, Juliet 1.1 c/c++ and java test suite, Computer 45 (10) (2012) 88–90.

[44] H. Homaei, H. R. Shahriari, Seven years of software vulnerabilities: The ebb and flow, IEEE Security & Privacy 15 (1) (2017) 58–65.

[45] W. G. J. Halfond, A. Orso, AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks, in: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05), ACM, 2005, pp. 174–183.

[46] G. Deepa, P. S. Thilagam, Securing web applications from injection and logic vulnerabilities: Approaches and challenges, Information and Software Technology 74 (2016) 160–180.

[47] X. Li, Y. Xue, Logicscope: Automatic discovery of logic vulnerabilities within web applications, in: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS '13), ACM, 2013, pp. 481–486.