Supporting Automated Vulnerability Analysis using Formalized Vulnerability Signatures

Mohemed Almorsy, John Grundy and Amani S. Ibrahim
Computer Science & Software Engineering, Faculty of Information & Communication Technologies
Swinburne University of Technology, Hawthorn, Victoria, Australia
[malmorsy, jgrundy, aibrahim]@swin.edu.au

ABSTRACT

Adopting publicly accessible platforms such as cloud computing model to host IT systems has become a leading trend. Although this helps to minimize cost and increase availability and reachability of applications, it has serious implications on applications' security. Hackers can easily exploit vulnerabilities in such publically accessible services. In addition to, 75% of the total reported application vulnerabilities are web application specific. Identifying such known vulnerabilities as well as newly discovered vulnerabilities is a key challenging security requirement. However, existing vulnerability analysis tools cover no more than 47% of the known vulnerabilities. We introduce a new solution that supports automated vulnerability analysis using formalized vulnerability signatures. Instead of depending on formal methods to locate vulnerability instances where analyzers have to be developed to locate specific vulnerabilities, our approach incorporates a formal vulnerability signature described using OCL. Using this formal signature, we perform program analysis of the target system to locate signature matches (i.e. signs of possible vulnerabilities). A newly–discovered vulnerability can be easily identified in a target program provided that a formal signature for it exists. We have developed a prototype static vulnerability analysis tool based on our formalized vulnerability signatures specification approach. We have validated our approach in capturing signatures of the OWSAP Top10 vulnerabilities and applied these signatures in analyzing a set of seven benchmark applications.

Categories and Subject Descriptors

F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Semantics of Programming Languages - *Program analysis*; K.6.5 **[Security and Protection]**: Verification.

General Terms

Algorithms, Design, Security, Languages, Verification

Keywords

Software security; Vulnerability analysis; Formal vulnerability specification; Common weaknesses enumeration (CWE)

1. INTRODUCTION

New computational paradigms such as Cloud Computing and Service-Oriented Architectures (SOA) depend on outsourcing IT systems for hosting on third-party platforms outside of the enterprise network perimeter (usually on the internet). This increases the possibility of hackers attacking and exploiting vulnerabilities in such applications. In addition, the number of newly discovered vulnerabilities is increasing rapidly. Delays in discovering and mitigating such vulnerabilities increase the probability of successful application attacks and security breach.

Web applications have become the prominent application delivery model used in such platforms as they do not require client deployment or configuration, and can be centrally updated and managed. However, web application vulnerabilities continue to make up the largest percentage of the total reported vulnerabilities in software applications. Web applications vulnerabilities constitute 75% on average of the total reported vulnerabilities over the last three years[1]. Of these reported vulnerabilities, well-known vulnerabilities such as Cross site scripting (XSS) represents 28%, while SQL Injection (SQLI) vulnerabilities represent 20%. Reported vulnerabilities are usually recorded in commonly available vulnerability databases such as NVD or CVEdetails.com. Vulnerabilities/weaknesses definitions are maintained in the Common Weaknesses Enumeration (CWE) database. This database is used as a reference framework by application developers, deployment engineers and security engineers to help identifying possible weaknesses to attack in software applications. However, a key problem with CWE is that recorded vulnerabilities are almost specified informally. Thus, each security vendor develops their security analysis tools based on their own understanding of such vulnerabilities.

Commercial vulnerability scanners such as AppScan, Web inspect, Cenzic, McAfee focus on black-box vulnerability analysis to avoid being limited to specific programming languages or platforms. However, none of these scanners cover all known vulnerability types [1]. Moreover, they are limited in discovering stored forms of XSS and SQLI vulnerability. To achieve good results with vulnerability analysis, multiple scanners should be applied [1].

Existing research efforts [2-7] focus on specific vulnerability types. Most focus on SQLI [8, 9], XSS [8, 10, 11], or input sanitization [12, 13]. These efforts use static analysis with many variations [2, 14], dynamic analysis [8], or hybrid of static and dynamic techniques [15, 16]. However, they focus on specific vulnerabilities only. Thus, new vulnerabilities cannot be incorporated for checking unless we have new algorithms.

A key problem with both industrial and academic efforts is that they are not comprehensive enough to cover known vulnerabilities or extensible to incorporate new vulnerabilities. Many tools depend on their own encoded representations of vulnerabilities, which are suitable for their own analysis approaches and algorithms. From our investigation in these efforts, we reached a

[1] www2.cenzic.com/downloads/Cenzic_AppSecTrends_Q1-Q2-2010.pdf

conclusion that the key problem really lies in the vulnerability definitions themselves and not in introducing new analysis techniques (most of the existing approaches use similar techniques with various combinations). Moreover, the various existing vulnerabilities databases, while useful, are not directly utilized by vulnerability analysis tools due to their informality; however, we figured out that different security analysis tasks including vulnerability analysis, attack analysis, and threat analysis can be facilitated if we have a formal vulnerability definition. Our analysis of the vulnerability analysis domain leads us to following research questions:

- What details do we need to capture to fully describe a given vulnerability?
- How can we formalize the signatures of possible security vulnerabilities?
- How can we effectively use such formal vulnerability specifications in automating the vulnerability analysis process?

In this paper, we introduce a new, comprehensive vulnerability specification schema. This schema captures formal rich details of a given application vulnerability/weakness including categories, preconditions, consequences, signatures, etc. A key entry that we focus on in this schema is the vulnerability signature. This signature specifies a set of invariants, when matched; it means that the given vulnerability exists. We adopt Object Constraint Language (OCL) in capturing such signatures. OCL is a declarative and formal language [17] based on first order logic and set theory. Vulnerability signatures are validated against a comprehensive system description model that covers most program entities including classes, methods, statements, inputs, sources, outputs, targets, etc. Furthermore, it helps in developing more abstract signatures not coupled with specific programming language or platform.

As an initial step in validating our vulnerability schema and signature specification approach, we have developed an OCL-based static application Vulnerability Analysis tool. This tool uses a new static vulnerability detection approach that performs program analysis looking for matches for vulnerability signatures, defined in OCL, in a given program source code. Then, it produces an overall vulnerability assessment report for the target application. This vulnerability analyzer will be extended to support dynamic analysis as well using vulnerability analysis workflow engine. A Key difference between our static vulnerability analysis tool and existing tools is that it uses our formal vulnerability specifications to detect source code vulnerabilities, while working on an abstract system representation. Moreover, it analyzes programs for any (new) vulnerability that has defined signature(s). This is compared to existing efforts that have specific (built-in) algorithms to discover certain vulnerability types only. We have developed a prototype tool supporting our approach and evaluated it in capturing the well-known TOP10 vulnerabilities reported by OWSAP. We have validated our toolset in locating these vulnerabilities in a set of open source web applications.

In section 2, we analyze the existing security vulnerabilities and map this analysis on the Top10 OWSAP vulnerabilities. Section 3 describes our vulnerability definition schema, the vulnerability signature specification, and our OCL-based static vulnerability analysis tool. Section 4 describes our prototype implementation details. In section 5, we discuss our experimental evaluation and results. Section 6 discusses the implications of our work and key directions for further research. Section 7 reviews related work.

2. BACKGROUND

To understand the root causes of security vulnerabilities we analyzed different system structures, components, and deployment models. We applied this analysis on the Top10 vulnerabilities reported by OWSAP. We summarize our conclusions as follows:

2.1 Analysis of Security Vulnerabilities

A given software system, whether desktop, web, or even embedded is based on a hosting service – e.g. web server, operating system, virtual server, etc. (Figure 1). A hosting service provides a set of APIs that the hosted system can use to read inputs from possible input sources (users, files, memory, database, etc.) or write outputs to possible output targets. Any vulnerability in the hosting service implies that an attacker can control inputs and/or outputs of the target system. The hosting media is a place where the hosted system runs – e.g. a process in case of web server, or memory in case of OS. If the hosting media breached, it may be used to control the hosted system inputs, outputs, or even processing (overriding kernel data using buffer overflow). However, these entities are out of the software system control.

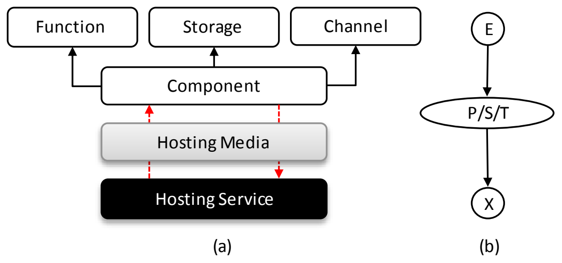

Figure 1. An overview of the host-system-component relations

Any target system is composed of a set of components. These components may be subsystems, composite components, or simple components. System components may be hosted on the same hosting service instance or different instances (different servers). In the latter case, they have to communicate through communication channels, which may be unsecure (an attacker may eavesdrop, or intercept messages). A system component may be an ***active component***, a component that can take actions or perform operations such as system functions. Active components are able to secure themselves and their processed data – e.g. authenticating users, authorizing users, encrypting data, etc. Or alternatively, a ***passive component,*** a component that cannot take actions to change data it maintains, such as storage components (databases/files) or transmission components (communication channels). Passive components cannot secure themselves e.g. a file or table cannot enforce security access on its contents by itself. They depend on other components, such as the hosting service (OS, DBMS), or active system components to manage/secure such components. This is a big open issue in data security area - e.g. data leakage protection - where confidentiality of data moving between different applications with different security levels may be threatened. Both active and passive components may be breached by the hosting service e.g. read data in memory, files, or on communication channels.

Each component, regardless of its type, has a set of entry points (E) and set of exit, output points (X), and is used in processing (P), as a storage (S), or as a communication channel (T). These entry and exit points can be compromised by an attacker who has control on the hosting service to read/write/modify/delete the data. Usually the number of entry points and exit points – the "attack surface" -

is used as a security metric when assessing systems security [18]. Furthermore, an active component may have vulnerabilities related to ***inputs*** (input validation - input coming from a user passing by the hosting service), ***outputs*** (output validation and exceptions – outputs may depend on malicious/modified inputs or passed through a vulnerable hosting service), or **processing** (logical errors – e.g. race conditions, malicious data corruption, service overloading). We use this analysis in categorizing vulnerabilities according to the source of vulnerability, such as input validation, output validation, processing, and hosting service vulnerabilities. This helps in deciding which types of vulnerabilities can be identified by static analysis, dynamic analysis, etc. Moreover, it helps in deciding the mitigation actions that can be applied to block such vulnerabilities.

2.2 OWSAP Top10 Security Vulnerabilities

Before we discuss how we formalize software system vulnerability definitions, we give an overview of the OWSAP Top10 web application vulnerabilities. OWSAP (Open Web Security Application Project) is a community effort to define and share knowledge about web application security approaches. We discuss these Top 10 vulnerabilities and signatures we deduce from the vulnerabilities recorded in NVD and CWE. These signatures are used by our vulnerability analysis tool; however, they can be further revised by experts to get more accurate results.

Injection Flaws: This type of vulnerabilities includes several well-known attacks intended to compromise application inputs in order to gain control or modify data, such as SQLI, OS command injection, LDAP query injection, and XPath query injection. All arise from input validation problems. "All external inputs are untrusted" is a well-known security principal that should be realize in securing systems. These vulnerabilities occur whenever a system trusts an input from the user – *first order injection* – or from a repository – *stored or second order injection* – and uses it to build dynamic queries that run OS or database commands without sufficient input sanitization or validation. An attacker can use this type of vulnerabilities to execute malicious commands or gain privileged access to the system under attack. Figure 2 shows code vulnerable to SQLI. For example, a password argument of the form "' OR (1=1) OR ''='" allows access to any specified username e.g. 'admin' or 'root'. The signature of these vulnerabilities is a dynamic query statement that uses external inputs without proper sanitization.

```
Public bool LogUser(string username, string password) {
    string query = "SELECT username FROM Users WHERE
    UserID ='" username " ' AND Password = '" +  password + "'";
```

Figure 2. A code snippet vulnerable to SQLI attack

Cross-Site Scripting Flaws: This is a two-step vulnerability. First, an attacker uses the application to store malicious data. Whenever a victim sends a request to resource X, the web server responds with data containing "malicious code" without being encoded. This malicious code executes on the victim browser causing disclosure of her confidential information to the attacker. This vulnerability type may be from stored data (e.g. from a database) or reflected (from user input). This is very common attack in applications that use user inputs for search or discussions. The signature of these vulnerabilities is to call output functions using external or stored inputs without sanitization or encoding.

Broken Authentication and Session Management Flaws: This is a common problem with security authentication. It includes attacks such as: authentication bypassing via external inputs (depend on external input to bypass authenticating the current requester); authentication checking not included in critical functions; using hard-coded credentials; using an easy to guess password; or session timeouts not set or checked. This enables unauthenticated users to maliciously access and use system resources. Figure 3 shows a code snippet vulnerable to improper authentication attack, where a user can modify their cookie to bypass the authentication check. The signature of these vulnerabilities is that every publicly accessible function should not trust external inputs to bypass (by conditional statement) triggering the authentication function.

```
if( Request.Cookies["Loggedin"] != true ) {
        if(  !AuthenticateUser(Request.Params["username"],
                                        Request.Params["password"] ) )
                throw new Exception("Invalid user");
}
DoAdministrativeTask();
```

Figure 3. A code snippet vulnerable to authentication Bypass

Insecure Direct Object Reference Flaws: authenticated users can send malicious inputs to access unauthorized data. Figure 4 shows an example where attacker sends custID = XYZ instead of custID = ABC. This enables the attacker to access other customers' data. The signature of these vulnerabilities is that user inputs are not authorized before used in business functions.

```
if( !AuthenticateUser( Request.Params["username"],
                            Request.Params["password"] ) )
            throw new Exception("Invalid user");
updateCustomerBalance(Request.QueryString["custID"], nBalance);
```

Figure 4. A code snippet vulnerable to improper authz

Cross-Site Request Forgery (CSRF) Flaws: an attacker deceives an authorized user by sending a forged request to the user's application to perform malicious actions. This attack requires the victim to have a valid session or cookie with the application (already authorized). The signature of these vulnerabilities is that requests' origins are not validated or that responses are usually predictable or have fixed URL format. It is usually difficult to identify CSRF using static analysis techniques because it is usually managed by the web server.

Security Misconfiguration Flaws: the system is not securely configured. This includes exposing information through exceptions; system executing with higher privileges than required; system files are accessible to unauthenticated users; or resources have misconfigured permissions. Some of these vulnerabilities can be discovered from the exception handlers whether they expose system details or not. Others need to be examined by application responses for unauthorized actions using dynamic analysis.

Unvalidated Redirect and Forward Flaws: the application redirects requests to a target URL that is concatenated from user inputs "Response.Redirect(userInput)". This type of vulnerability is similar to the injection vulnerabilities where web redirect functions use external inputs to build the redirect URL.

Failure to Restrict URL Access Flaws: an application does not perform access control on resources or URLs. These vulnerabilities can be easily examined by checking webpage methods for authorization function calls. Dynamic analysis is required to check application responses for unauthorized URLs.

Insufficient Transport Layer Protection Flaws: sensitive data including credentials and customer data are transmitted in plain text. The signature of these vulnerabilities is that output data are transmitted without passing by encryption functions. Dynamic analysis is required to examine application responses (if the protection is done on the transport layer).

From this analysis, we deduced two points: **(i)** Top10 vulnerabilities reflect categorization we introduce in Section 2.2 – i.e. input validation such as SQLI, URL redirection, CSRF; output validation such as XSS, information exposure; and hosting service such as security misconfiguration and insufficient transport layer protection; and **(ii)** many of these Top10 vulnerabilities can be discovered using static source code analysis (vulnerabilities related to the program itself), while other require dynamic analysis (vulnerabilities related to the hosting service or configurations).

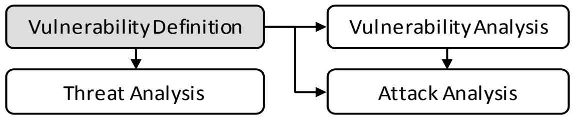

Figure 5. Vulnerability definition and security analysis tasks

2.3 Why We Need a Formalized Vulnerability Definition?

The security analysis for a given system includes different tasks that are usually performed at different stages of the system implementation. ***Threat analysis*** is conducted at early stages of the system development usually during the system design phase. Here the software development team work together to identify possible problems that may arise from using specific platforms, architectures, languages, and the expected deployment model. A formal vulnerability definition, as shown in Figure 5, facilitates identifying possible weaknesses in a given platform or language. ***Vulnerability analysis*** is applied during system development or after development has been completed. It targets identifying security-compromising errors in the system implementation. A formal vulnerability definition helps in automating vulnerability analysis as we will show later in this paper. ***Attack analysis*** is applied after the system has been deployed or when a detailed deployment model becomes available. It focuses on identifying possible attack vectors on system resources given the networked system. A formal definition helps identifying preconditions and consequences for each vulnerability instance found in the program.

3. OUR APPROACH

We base our security analysis approach on (***i***) a formal vulnerability definition schema that captures every detail related to a given vulnerability. This helps in every security analysis task, as discussed above; (***ii***) a formal vulnerability signature specification approach that can capture security vulnerability signatures; and (***iii***) an extensible vulnerability analysis tool that perform signature-based program analysis. Here, we introduce a static analysis component only. We are working on an integrated vulnerability analyzer that performs static and dynamic analysis.

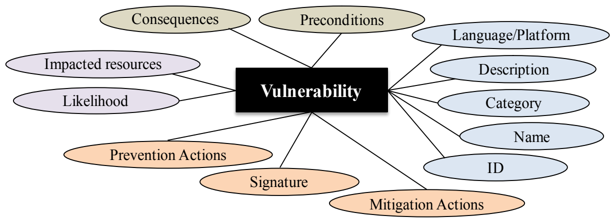

Figure 6. Weakness definition schema

3.1 Vulnerability Definition Schema

We studied the various security analysis tasks (vulnerability, attack and threat analysis) to identify the key items required in these tasks that should be included in a vulnerability definition schema, shown in Figure 6. These vulnerabilities' definitions should be managed by security experts (may be used as extension of CWE database).

Vulnerability ID: Every discovered vulnerability instance, as in the NVD database, should have a reference to its parent weakness or vulnerability definition. This helps retrieving vulnerability details e.g. preconditions, consequences.

Category: Many categorization-schemas for software vulnerabilities do exist. Each categorization schema helps understanding weaknesses from a specific point of view e.g. developers or researchers. A categorization based on the root cause or source of the weakness, shown in Figure 1, helps in vulnerability analysis, mitigation, and even avoidance. Thus, we propose to categorize vulnerabilities as input validation, processing logic, output validation, hosting service, hosting media, communication channel, storage, and security control vulnerabilities.

Language/platform: specifies the language(s) that a given vulnerability applies to - i.e. many languages have language-specific vulnerabilities such as C, C++, C#, Java, etc. We also use this to describe the technology or architecture paradigm inherent with the vulnerability - e.g. client-server, web-based, service-oriented, or multi-tier, along with the underlying environment e.g. web server, client, application server, database server. This helps in threat analysis to identify possible vulnerabilities that may exist and start taking precautions to avoid such vulnerabilities.

Preconditions: This attribute aids both vulnerability analysis and attack analysis. Preconditions are a list of the capabilities that an attacker should possess, or the list of system configurations that need to be present in order to exploit this vulnerability e.g. to exploit a specific vulnerability, an attacker might have to have root access, user access, remote root access, public access, etc.

Consequences: if a given vulnerability exploited, what will be the benefits achieved by the attackers e.g. disclosure of system information, invalid processing, invalid results, execute an unauthorized function, elevate permission, bypass security, crash, or Denial-of-Service - DOS. This can be used in planned attacks e.g. using vulnerability V1 will help the attacker to obtain a set of privileges. These privileges may be preconditions of vulnerability V2. The consequence of V2 may be the actual goal of the attacker.

Impacted resources: this specifies the resources that will be impacted if the given vulnerability exploited including memory, configuration files, registry, customer data, credentials, cryptography keys.

Likelihood: The probability that the given vulnerability is exploited by an attacker may be low, medium, or high. This depends on the complexity of the given vulnerability and attacker capabilities as defined in the vulnerability preconditions.

Vulnerability signature: A vulnerability signature describes patterns that when matched in a target program mean it is likely to have the given vulnerability. This may be signature of code snippets, or signature of system response for requests with specific signatures. Every single vulnerability may have different signatures that capture different forms (scenarios), or that are applicable with different vulnerability analysis techniques.

Prevention: a list of precautions to be followed or checked during code review. These might be rules to check during system development or deployment; combinations of architectures; languages and platforms to use or not to use.

Mitigations: Indicates how we can modify the vulnerable system entities to block a discovered vulnerability. This may require modification of the vulnerable code parts; changing system configurations; or even changing system architecture.

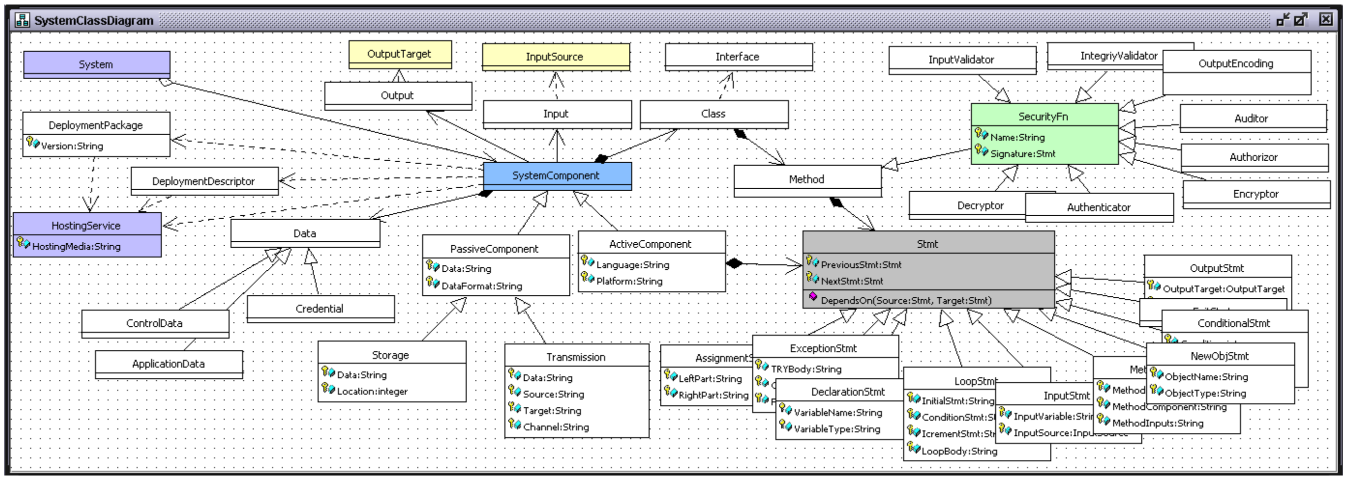

Figure 7. Our system description class diagram used by our OCL-based vulnerability signature approach

3.2 Vulnerability Signature Specification

Existing software security weakness, or vulnerability definitions, in the Common Weakness Enumeration (CWE) database help in understanding the nature of a given vulnerability. However they do not directly help in locating such vulnerabilities in target systems. Formalizing these descriptions helps vulnerability analysis tools in automating the vulnerability analysis process. Ideally a formal vulnerability signature, specified by security experts, should be specified on an abstract level far from the source code and programming language details, enabling locating possible vulnerability instances in different programs written in different programming languages.

We use OCL as a well-known, extensible, and formal language to specify semantic rather than syntactical signatures of security weaknesses. To support specifying and validating OCL-based signatures, we have developed a system-description model, shown in Figure 7. This model is inspired from our analysis of security vulnerabilities (Section 2). It captures main entities in an object-oriented program including components, classes, instances, inputs, input sources, output, output targets, methods, method body, method statements e.g. if-else statements, loops, new objects, etc. Moreover, it captures security concepts such as authentication, authorization, audit, etc. and other system details such as hosting service, deployment descriptors, etc. Each entity has a set of attributes such as method name, accessibility, variable name, variable type, method call name, etc. This enables specifying OCL-based vulnerability signatures on different system entities other than source code entities (classes, methods, code-blocks) such as deployment descriptors (configuration files), hosting services (web server), storage, output targets, or input sources. Of course, this requires developing different parsers other than code parsers that can read such entities. Moreover, this requires a comprehensive vulnerability analyzer that supports locating signatures in such entities as well as source code.

The vulnerability analysis tool should have different profiles for different languages and platforms (ASP.Net, PHP, C#, Java, etc.). Thus vulnerabilities with signatures containing input source or output target security authentication, authorization, sanitization and other functions can be interpreted differently based on the program platform or programming language used. If the system uses custom sanitization or security functions, developers have to mark their security functions in the resulting system model.

Table1 shows some vulnerability signatures specified in OCL using our system description model (Figure 7), For example:

Table1. Examples of OCL-specified vulnerability signatures

Vul.	Vulnerability Signature
SQLI	Method.Contains(S : MethodCall | S.FnName = "ExecuteQuery" AND S.Arguments.Contains(X : IdentifierExpression | X.Contains(InputSource)))
XSS	Method.Contains(S : AssignmentStatement | S.RightPart.Contains(InputSource) AND S.LeftPart.Contains(OutputTarget))
Improper Authn.	Method.IsPublic == true AND Method.Contains(S : MethodCall | S.IsAuhenitcationFn == true AND S.Parent == IFElseStmt AND S.Parent.Condition.Contains(InputSource))
Improper Authz.	Method.IsPublic == true AND Method.Contains(S : Expression | S.Contains(X: InputSource | X.IsSanitized == False OR X.IsAuthorized == False)

SQLI Signature: any method that has method call statement *"S"* where the callee function is *"ExecuteQuery"* and one of the *parameters* passed to it is previous assigned to untrusted identifier coming from one of the input sources. This initial signature can be revised to incorporate taint analysis checking. Taint analysis can be defined as an OCL function that adds every variable assigned to a user input parameter to a suspected list. In this case we update the vulnerability signature to use "*Method.SuspectedList().Contains(X)*" instead of *X.Contains(InputSource)*" as in Table1.

XSS Signature: any method statement *"S"* of type assignment statement where left part is of type "*output target*" e.g. text, label, grid, etc. and right part uses input from the tainted input sources.

Improper Authentication Signature: any public method that has statement "*S*" of type "*method call*" where the callee method is marked as Authentication function while this method call can be skipped using user input as part of the bypassing condition.

Improper Authorization Signature: any public method that has statement "S" of type "expression" – i.e. any statement - where "S" uses data X without being sanitized, authorized, or simply taint data (Method.SuspectedList().Contains(X) == true).

A key problem with these signatures is that we do not consider security solutions applied beyond the system source code either using proxies to filter SQL queries or using security controls deployed on the web server as an http handler. These can be handled by appending a dynamic signature forming a sequence of OCL constraints to be checked on system responses to malicious requests. Another issue is that we may have different signatures with different complexities for the same vulnerability. We expect security experts to develop strong and complete signatures. Weak signatures mean more false positives, which may annoy developers, or more false negatives, which may harm customers.

3.3 OCL-based Static Vulnerability Analyzer

Given that vulnerability signatures are now formalized (in OCL), the static vulnerability analysis component becomes a program analysis tool that traverses the given program looking for code snippets that match the given vulnerability signatures. Figure 8 describes the architecture of our static vulnerability analyzer based on the formalized vulnerability signature concept.

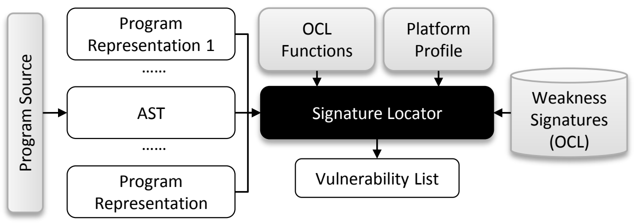

Figure 8. OCL-based static vulnerability analysis tool

Program source code: the program to be analysed can be either a source code or even program binaries (dlls, exes). In the latter case, we use de-compilation techniques to reverse engineer the source code of the given program.

Abstract Program Representation: to avoid being specific to programs written in a specific programming language or with a specific coding style, we transform the given system code into an abstract syntax tree (AST) representation. The program AST abstracts most of the source code details away from specific language constructs. Extracting source code AST requires using different language parsers (we currently support C++, VB.Net and C#). Then, we perform more abstraction by transforming this AST to our system description model, shown in Figure 7. We support specifying signatures on other system aspects including features, architecture, etc. For example one may check for vulnerability signatures of code that realize specific features. This also helps in combining static analysis and dynamic analysis where results of the static analysis used to drive black-box testing scenarios.

Signature locator: This is the main component in our vulnerability analysis tool. It receives the abstract system representation and generates a list of possible vulnerabilities in the given system along with their locations in code. At analysis time, it loads the platform profile based on the details of the program under analysis. Then, it loads the defined weaknesses in the weaknesses' signatures database (specified in OCL), based on the target program platform/language. The signature locator transforms these signatures into constraints and checks on program entities - i.e. code snippets that match the specified signatures. The OCL functions represent a library of predefined functions that can be used in specifying vulnerability signatures and in identifying possible matches. This includes control flow analysis, data flow analysis, string analysis, taint-analysis, etc. The developed Weaknesses' signatures are compiled using OCL compiler and validated against our system description model before getting stored in the weaknesses' signatures database.

To locate vulnerability matches, the signature locator translates every vulnerability OCL-signature in a *visitor* class, as in Figure 9, which has a handler (method) for every concept used in the OCL-signature – e.g. if the signature checks that the method is public, then the visitor class will have a handler for system entities of type method definition. This handler contains a set of checks based on the given OCL-signature. The visitor class traverses the target program entities. If a visited node has a handler, this handler is triggered – e.g. a visitor for SQLI signature (Figure9), has handlers for "method definition" and "method call" nodes. In the method call handler, it will have a condition to check the called method. If it is "ExecuteQuery", it marks this entity and continues to visit its arguments. Otherwise, it skips for another system entity. The signature locator generates a list of discovered vulnerabilities along with code locations thought to have these vulnerabilities. We use Application Vulnerability Description Language [2] - AVDL - to represent the identified vulnerabilities in XML format to support interoperability with existing vulnerability databases such as NVD.

4. IMPLEMENTATION

We briefly describe some implementation details of our formal static vulnerability analysis tool. *First*, we developed a UI component to assist security experts in capturing vulnerability signatures' in OCL. This provides vulnerability specification and signature editing including checking validity of OCL statements and testing of specifications on sample source code. We use an existing OCL parser [22] to parse and validate signatures against our system description model (Figure 7). Once validated, the vulnerability signature is stored in the signatures database.

Next, to parse the given program source code and generate a system abstract model, we use an existing .Net parser ***NReFactory*** Library, which supports VB.Net and C#. Moreover we have used a C parser written in python called *pycparser*. Thus we now support locating vulnerabilities in C#, VB.Net, C, and C++. We are working on parsers for PhP and Java. For a system with binaries only available - we use an existing de-compilation tool ***ILSPY*** to generate code from binaries. This is currently supported for C# and VB.Net only. *Third,* we developed a class library to transform the generated AST into a more abstract (summarized) representation as specified in our system description model. This reduces its size and complexity to reflect only necessary details required in signatures' matching, reduce complexity and make our technique more scalable than if a full AST was used. Other system models such as system features, architecture, etc. can be specified by the system provider and added to our AST model. *Fourth,* our signature locator has an OCL translator that translates a given OCL signature into a corresponding *visitor* class. This visitor class is used to traverse system representation entities. For each entity, it performs customized checks as determined in the OCL signature.

```
public class SQLIVisitor : AbstractAstTransformer {
   public override object VisitMethodCall(InvocationExpression S)  {
       if(S.FnName == "ExecuteQuery")  {
                      foreach (Statement X in S.Arguments) {
                          if(X.AcceptVisitor(this) != null)  {
                               count++;
                               list.Items.Add(S.StartLocation + S.EndLocation);
                          }  …
     public override VisitIdentifierExpression(IdentifierExpression X) {
                      if( OCLLibrary.IsTainted(X.Identifier) == true )
                              return true;
                      return null; …
```

Figure 9. Sample of the SQL injection Visitor class

Figure 9 shows a sample visitor class generated from the simple SQL injection signature specified in Table 1. The SQLIVisitor class implements a set of predefined functions based on each part in the SQL injection signature e.g. the VisitMethodCall function is related to the condition "Method.Contains(**S : MethodCall**)", etc.

[2] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=avdl

```
context Method:: SuspectedList(): Collection(Identifier)
   Let userInputs: Collection(Identifier) = Method. Parameters
   Post: result = Method.Body->select(stmt:AssignmentStmt |
           RightPart.Contains(userInputs)->select(id: IdentifierExp))
```

Figure 10. Sample of the tainted-data analysis function

Our OCL functions library has a set of functions required during the analysis phase. This includes control-flow analysis (CFA), data-flow analysis (DFA), Tainted-data analysis, etc. These functions are defined in OCL and can be extended with further static analysis functions based on future vulnerability analysis needs. An OCL to C# transformer performs a transformation for these functions as well as new OCL signatures once defined. Program slicing and taint analysis techniques (core techniques in program and security analysis area) can be easily captured in OCL. Figure 10 shows a sample tainted-data analysis function defined in OCL. This can be extended to filter sanitized variables (variables processed by sanitization functions).

```
<Profile platform="ASP.Net">
    <InputSources>
        <Source> Web.HttpRequest.get_QueryString</Source>
       <Source>Web.HttpRequest.get_Cookies</Source> …
  <OutputTargets>
       <Target>System.Web.HttpResponse.Write</Target>
       <Target> UI.WebControls.TextBox.set_Text</Target>
      <Target> WebControls.HyperLink.set_NavigateUrl</Target> …
```

Figure 11. Sample of the platform profile for ASP.Net

Our vulnerability analyzer depends on platform profiles to set the analysis context. Platform profile is an XML document that contains information about a specific platform. It is used to set the context of the signature locator according to the target system implementation platform. Figure 11 shows an example of a platform profile for ASP.Net. This is different from Java or PHP profiles. These functions are used by the signature locator as values for the abstract concepts (input sources, output targets, etc.).

5. EVALUATION

In this section we summarize our experimental evaluation we have performed to assess the capabilities of our approach in capturing as well as identified security vulnerabilities. We apply the OCL-based vulnerability signatures illustrated in Section 3.

Table 2. Summary of benchmark applications statistics

Benchmark	Downloads	KLOC	Files	Classes	Method	AST
Galactic	-	16.2	99	101	473	187
SplendidCRM	>400	245	816	6177	6107	765
KOOBOO	>2,000	112	1178	7851	5083	78
BlogEngine	>46,000	25.7	151	258	616	163
BugTracer	>500	10	19	298	223	93
NopCommerce	>10 Rel.	442	3781	5127	9110	484
Webgoat	-	15	105	125	165	150

5.1 Benchmark Applications

We have selected a set of seven web-based, open source web applications developed ASP.NET as a benchmark to evaluate our approach. These applications cover a wide business spectrum including: Galactic is an ERP system developed internally in our group for testing purposes. SplendidCRM is an open source CRM that is developed with the same capabilities of the well-known open source SugarCRM. It has a commercial and community versions. KOOBOO is an open source Enterprise CMS used in developing websites. BlogEngine is an open source ASP.NET 4.0 blogging engine. BugTracer is an open-source, web-based bug tracking and general purpose issue tracking application. NopCommerce is an open-source eCommerce solution with more than 10 releases. Webgoat is developed by OWSAP for security testing purposes. Except for Galactic, we did not have any experience with these applications security. Table2 summarizes statistics of these applications including: known No. download, size, KLOC, files, classes, methods, and AST build time (msec).

$$Precision = \frac{Valid\ Discovered\ Vulnerabilities\ (TP)}{Total\ Discovered\ Vulnerabilities\ (TP+FP)} \qquad \text{Eq. 1}$$

$$Recall \quad = \frac{Valid\ Discovered\ Vulns\ (TP)}{Total\ vulns\ in\ a\ given\ system\ (TP+FN)} \qquad \text{Eq. 2}$$

$$F-Measure \ = 2\ \frac{Precision*Recall}{Precision+Recall} \qquad \text{Eq. 3}$$

5.2 Metrics used

To assess the effectiveness of our approach in discovering security vulnerabilities using static program analysis, we use a set of metrics to measure the soundness and completeness of the analysis technique. These metrics are precision rate, recall rate, and F-measure. The precision metric is used to assess the soundness of the approach. A high precision means that the approach returns more valid results (true positive - TP) than invalid results (false positive - FP). Thus the maximum precision is achieved when no false positives (see Equation 1 below). The recall metric is used to assess the completeness. A high recall means that the approach returns most of the valid results (true positive - TP) than missed valid results (false negative - FN), see Equation 2. The F-measure metric combines both precision and recall. It is used to measure the overall effectiveness of the approach (weighted harmonic mean). This metric depends on the importance of the recall rate and the precision rate e.g. if we are interested in high precision (more valid vulnerabilities) then we will give precision factor high weight, and vice-versa. In our evaluation, we assume that the importance of the precision rate and recall rate is equal, see Equation 3.

Table 3. Experimental results of applying OCL-based vulnerability analysis tool on benchmark applications. (D) no. of discovered vulnerability, (FP) no. of false positives, and (FN) no. of false negatives. Columns represent IDs of the benchmark applications: [1] Galactic, [2] Splendid, [3] KOOBOO, [4] BlogEngine, [5] BugTracer, [6] NopCommerce, and [7] Webgoat.

Vulnerability		[1]	[2]	[3]	[4]	[5]	[6]	[7]	Total
SQLI	**D**	2	12	14	3	9	19	8	67
	FP	0	2	2	0	0	2	0	6
	FN	0	2	2	1	3	1	1	10
Authn. Bypass	**D**	2	2	1	0	0	0	2	7
	FP	0	0	0	0	0	0	0	0
	FN	0	0	0	0	1	0	1	2
Authz. Bypass	**D**	2	3	11	4	0	0	3	23
	FP	1	0	2	0	0	0	0	3
	FN	0	0	2	0	2	3	0	7
XSS	**D**	3	5	10	2	0	4	5	29
	FP	0	1	1	0	0	0	1	3
	FN	1	2	2	1	2	1	0	9
CSRF	**D**	5	6	13	10	0	12	3	49
	FP	1	0	1	0	0	1	0	3
	FN	0	1	2	0	4	1	0	8
Info. Expo.	**D**	3	0	0	10	0	0	3	16
	FP	0	0	0	0	0	0	0	0
	FN	0	0	0	0	2	1	0	3
URL Redirect	**D**	1	0	2	8	0	6	0	17
	FP	0	0	0	0	0	0	0	0
	FN	0	0	0	0	3	1	0	4
Total	**D**	**18**	**28**	**51**	**37**	**9**	**41**	**24**	**208**
	FP	**2**	**3**	**6**	**0**	**0**	**3**	**1**	**15**
	FN	**1**	**5**	**8**	**2**	**17**	**8**	**2**	**43**

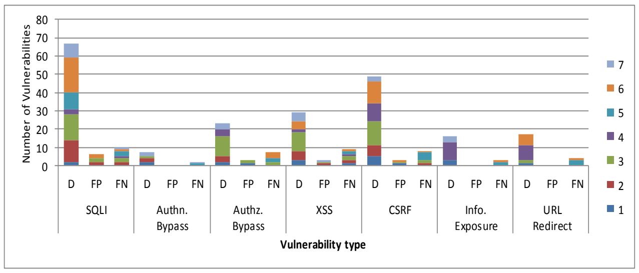

Chart 1. Discovered vulnerabilities per vulnerability type

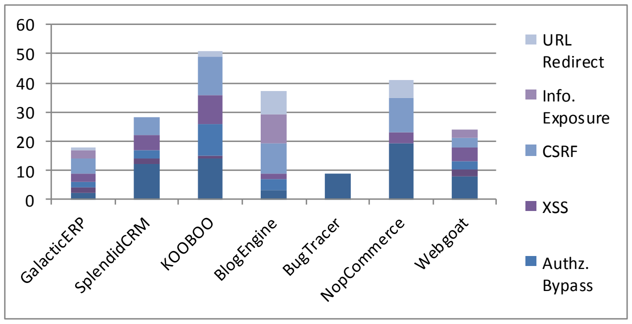

Chart 2. Discovered vulnerabilities per benchmark application

5.3 Experimental Results

Table 3 summarizes results of our experiments. We used our approach to analyse applications in the benchmark suite to identify seven of the Top10 web applications vulnerabilities (from the OWSAP2010 report). Other vulnerabilities could not specify static signatures (use static program analysis). However, specifying dynamic signatures for these vulnerabilities is easy. Table 3 summarizes, for each application and each vulnerability analysed, the total time taken, number of vulnerabilities in the code base found, false positives (analyser thought vulnerability but there isn't on manual analysis), and false negatives (manual code analysis indicates a vulnerability but our analysis tool did not discover it at this code location).

Chart 1 shows the number of discovered vulnerabilities grouped by vulnerability type. The SQLI represents the most frequent vulnerability in all applications, then cross site reference forgery (CSRF) vulnerability. After that, cross site scripting (XSS) and authorization bypassing vulnerabilities are relatively equal. This is mostly conforming to the ranking reported by OWSAP2010.

Chart 2 shows the number of vulnerabilities identified in every application. It is clear that nopCommerce and KOOBOO are the most vulnerable applications. However, if we consider the application size factor, we see that the ratio of vulnerabilities discovered per compared to application size is about equal. Moreover, some applications such as BlogEngine use Microsoft membership for access control, which eliminates the authentication bypassing vulnerabilities.

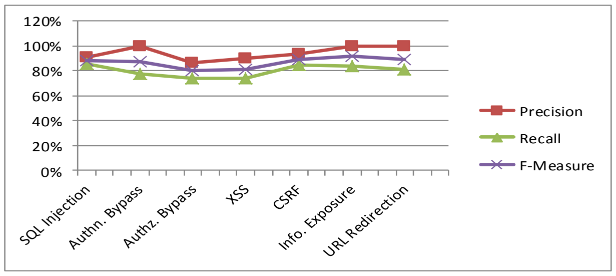

Chart 3. Our achieved precision, recall, and F-measure rates

Chart 3 shows the precision, recall, and F-measure rates for each vulnerability type. This chart shows that we achieve a high precision rate for most of the vulnerability types. The precision metric is on average 93%. This means that for each identified 100 vulnerabilities we have 7 false positives. This chart also shows a good recall rate, although it is relatively lower than precision rate we achieved. The recall metric is on average 82%. This means that in every 100 vulnerability instances, we can correctly identify 82 and we miss 18 instances. This value could be improved if we use a hybrid dynamic and static analysis approach. The overall effectiveness of the approach (F-measure) is around 87%. A key result from this chart is that the recall metric is higher in SQLI, XSS, Information disclosure, and URL redirection than in the other vulnerabilities. This justifies our initial supposition that although we succeeded in developing a static signature for these signatures (CSRF, authorization and authentication bypass), it is difficult to achieve a high correct detection rate without dynamic analysis.

5.4 Performance Evaluation

Chart 4 shows the time (in sec) required to analyse the benchmark applications to locate the existing vulnerabilities' instances for the given set of vulnerability signatures. It is clear that the SQLI vulnerability takes much more time to identify than XSS and authorization bypassing. The authentication bypass takes the lowest time. The time required to identify a given vulnerability depends on the number and complexity of the specified OCL signatures.

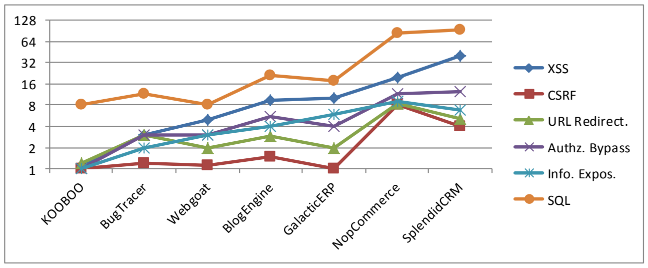

Chart 4. Performance of approach per vulnerability and application

6. DISCUSSION

In this research we introduce a formal vulnerability definition schema; signature specification approach based on OCL; and static vulnerability analyser. Vulnerability definition schema covers most of the details required in security analysis tasks (attack, threat and vulnerability analysis tasks). Vulnerability signature is specified on an abstract system representation. This allows applying the same signature on different systems developed with different languages. Use of OCL allows formalizing and easing of signatures validation and testing. Moreover, a new vulnerability can be easily located in the target system as far as we have a formal signature for it.

We succeeded in producing a vulnerability analysis tool that can work online without a need for new algorithms, modules, or patches. The current static analyser achieves a precision rate of 93% and recall rate of 82%. Although we are usually interested in high recall rate which implies less false negatives i.e. less number of vulnerabilities that could not be detected by the analyser, in the current tool we focused on high precision rate where number of reported vulnerabilities that are false positives i.e. not a real vulnerability, are less. The reason behind this decision was that static analysers are usually used by system developers who are interested in getting less false positives to mitigate. Another reason is related to the nature of security vulnerabilities. From our experiments, we determined that not all vulnerabilities can be captured using static analysis, and the same applies using dynamic analysis. Vulnerabilities related to source code such as SQLI and XSS can be described and located using static analysis. Vulnerabilities such as CSRF are difficult to use static signatures.

From our experience in developing signatures of the TOP10 vulnerabilities and our experiments we determined that: **(i)** it is better to use dynamic analysis tools with certain vulnerabilities, such as CSRF, because these vulnerabilities can be handled by the web server. This means that we have a high false positive if we use static analysis tool to locate these vulnerabilities; **(ii)** some vulnerabilities can be easily identified and located by static analysis such as SQL Injection and XSS vulnerabilities; **(iii)** some vulnerabilities such as DOM-based SQL and XSS vulnerabilities need a collaborating static and dynamic analysis to locate them. We believe that combining static and dynamic analysis is needed to increase the precision and recall rates.

A key problem with static analysis tools is the use of aspect-oriented security techniques, where security is weaved within the system at runtime. In this case we will have a high false positive rate because we report vulnerabilities that are already mitigated by the aspect-based security. The same will occur if external security controls are used, such as in database engines to filter SQL queries, using DB proxies to filter queries, using web server's deployed security controls such as encryptions, authentication, and authorization, or even provided by the platform through configurations such as ASP.NET membership or other anti-CSRF/ anti-XSS security controls. These can be discovered using dynamic vulnerability analysis extensions.

The lack of system engineers' annotations of the system security functions may lead to high false positive. However, this problem can be solved by employing dynamic vulnerability analysis. Dynamic vulnerability analysis approaches cannot help in locating specific code snippets where vulnerabilities exist. Moreover, they cannot help testing code coverage. Thus, a hybrid approach of static and dynamic analysis is required. We are extending our analyser to support both. We use a workflow engine to define the analysis sequence, using different approaches, to locate a given vulnerability. This increases the recall rate of the overall approach. Moreover, we plan to include confidence level with reported vulnerabilities. This helps developers to prioritize based on criticality and importance.

Our OCL-based signatures and vulnerability analysis tool can be used in different program analysis problems such as aspect mining, refactoring – locating "bad-smells", or reengineering "impact analysis". In these cases system engineers have to specify signatures they want to locate in their programs.

7. RELATED WORK

Existing efforts in vulnerability analysis can be categorized into static analysis, dynamic analysis, and hybrid analysis based approaches. Most of these efforts designed for specific vulnerability types mainly SQLI, XSS. Jimenez *et al.* [19] review various software vulnerability prevention and detection techniques. Broadly, static program analysis techniques work on the source code level. This includes pattern matching that searches for a given string inside source code, tokens extracted from source code, or system byte code e.g. calls to specific functions. Data flow and taint analysis identify data coming from untrusted sources to mark as tainted i.e. should not be used before being sanitized or filtered. Model checking to detect vulnerabilities depends on extracting a system model from the system source code and developing a set of constraints on the model that should not occur. An issue is that model checking approaches often suffer from a state explosion problem and generate only a counterexample. Dynamic analysis techniques analyse a system as a black box, avoiding being overwhelmed with system details. Fuzzy testing provides random data as input to the application in order to determine if the application can handle it correctly or not. Dynamic techniques are however limited in code coverage.

Static analysis approaches: NIST [20] has been conducting a security analysis tools assessment project (SAMATE). A part of this project is to specify a set of weaknesses that any source code security analysis approach should support including SQL injection, XSS, OS command injection, etc. They have also developed a set of test cases that help in assessing the capabilities of a security analysis tool in discovering such vulnerabilities. Halfond *et al.* [9] introduce a new SQL injection vulnerability identification technique base on positive tainting. They identify "trusted" strings in an application and only these trusted strings to be used to create certain parts of an SQL query, such as keywords or operators. Lei *et al.* [21] trace the memory size of buffer-related variables and instrument the code with corresponding constraint assertions before the potential vulnerable points by constraint based analysis. They used model checking to test for the reachability of the injected constraints. Dasgupta *et al.* [5] introduce a framework for analysing database application binaries to automatically identify security, correctness and performance problems especially SQLI vulnerabilities. They adopt data and control flow analysis techniques as well as identifying SQL statements, parameters, tables and conditions and finally analyse such details to identify SQLI vulnerabilities. Martin et al [6, 7] introduce a program query language PQL that can be used to capture definition of program queries that are capable to identify security errors or vulnerabilities. PQL query is a pattern to be matched on execution traces. They focus on Java-based applications and define signatures in terms of code snippets. This limits their capabilities in locating vulnerabilities' instances that matches semantically but not syntactically. Wassermann *et al.* [11] introduce an approach to finding XSS vulnerabilities based on formalizing security policies based on W3C recommendation. They conduct a string-taint analysis using context free grammars to represent sets of possible string values. They then enforce a security policy that the generated web pages include no untrusted scripts. Jovanovic *et al.* [4] introduce a static analysis tool for detecting web application vulnerabilities. They adopt flow-sensitive, inter-procedural and context-sensitive data flow analysis. They target identifying XSS vulnerabilities only. Ganesh et al [8, 14] introduce a string constraint solver to check if a given string can have a substring with a given set of constraints. They use this to conduct white box and dynamic testing to verify if a given system is vulnerable to SQLI attacks.

Dynamic analysis approaches: Bau et al [1] perform an analysis of black box web vulnerability scanners. They conducted an evaluation of a set of eight leading commercial tools to assess the supported classes of vulnerabilities and their effectiveness against these target vulnerabilities. A key conclusion of their analysis is that all these tools have low detection rates of advanced and second-order XSS and SQLI. The average percentage of discovered vulnerabilities equals 53%. The analysis shows that these tools achieve 87% in session management vulnerabilities and 45% in the cross site scripting vulnerabilities. Kals et al [2] introduce a vulnerability scanner that uses a black-box approach to scan web sites for the presence of exploitable SQLI and XSS vulnerabilities. They do not depend on a vulnerability signature database, but they require attacks to be implemented as classes that satisfy certain interfaces. Weinberger et al [10, 12] introduce an analysis of a set of 14 frameworks that provide XSS sanitization techniques. They identify limitations including lack of context-sensitive sanitization that result in developing custom sanitizer that need to be validated for their correctness, and supporting client-side code "DOM-based XSS". Felmetsger et al [3] use an approach for automated logic vulnerabilities detection in web applications. They depend on inferring system specifications of a web application's logic by analysing system execution traces. They then use model checking to identify specification violations. A key limitation of this approach is the extraction of properties specifications to be validated. They assume that collected traces represent correct system behaviour.

Hybrid analysis approaches: Monga et al [15] introduce a hybrid analysis framework that blends static and dynamic approaches to detect vulnerabilities in web applications. The application code is translated into an intermediate form. The resulting static model is filtered to focus only on dangerous statements. This reduces model size where dynamic analysis will be conducted, mitigating the performance overhead of the dynamic taint analysis approach. This approach, as most taint analysis approaches (either static or dynamic), targets only injection-related vulnerabilities. Balzarotti et al [13] introduce composition of static and dynamic analysis approaches "Saner" to help validating sanitization functions in web applications. The static analysis is used to identify sensitive sources/sinks methods. Dynamic analysis used to analyse the identified suspected paths.

Compared to existing efforts, our approach achieves scalable, extensible and powerful signature-based vulnerability analysis not coupled to specific vulnerability, analysis technique, or language/platform. Our approach is based on formalizing vulnerability definition including the vulnerability signature part.

8. SUMMARY

We introduce a new automated formal vulnerability analysis approach. Our approach is based on formalized vulnerability definition schema. A part of this schema is the formal vulnerability signature. This signature specifies a set of invariants that confirm the existence of a given vulnerability in the target program. We adopt OCL in specifying vulnerability signatures. We developed a static vulnerability analysis tool that uses our formally specified vulnerabilities signatures to locate possible matches in the target system. A new vulnerability can be easily identified provided that it has a formal signature. We validated our approach on a set of seven open source applications from different domains, different sizes and different development models. Our experimental results show that our OCL-based static analysis tool achieves (93%) precision rate and (82%) recall rate. This means that we achieve a good FP rate (7%) and a fair FN rate (18%). Moreover, these rates can be improved using a dynamic analysis extension, based on our formal signatures approach, which we are currently working on.

ACKNOWLEDGEMENTS

The authors are grateful to Swinburne University of Technology and the FRST SPPI project for support for this research.

REFERENCES

[1] J. Bau, E. Bursztein, D. Gupta, et al, "State of the Art: Automated Black-Box Web Application Vulnerability Testing," in Proc. *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 332-345.

[2] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "SecuBat: a web vulnerability scanner," presented at the Proc. of 15th Int. Conf. on World Wide Web, Edinburgh, Scotland, 2006.

[3] V. Felmetsger, L. Cavedon, C. Kruegel, et al, "Toward automated detection of logic vulnerabilities in web applications," in *Proc. 19th USENIX Conf. on Security*, Washington, DC, 2010, pp. 10–10.

[4] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting Web application vulnerabilities," in *Proc. of 2006 IEEE Symposium on Security and Privacy*, 2006, pp. 258-263.

[5] A. Dasgupta, V. Narasayya, and M. Syamala, "A Static Analysis Framework for Database Applications," in *Proc. IEEE Int. Conf. on Data Engineering*, 2009, pp. 1403-1414.

[6] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: a program query language," in *Proc. 20th annual ACM Conf. on Object-oriented programming, systems, languages, and applications* CA, USA, 2005, pp. 365-383.

[7] M. S. Lam, M. Martin, et al, "Securing web applications with static and dynamic information flow tracking," in *Proc. 2008 ACM symposium on Partial evaluation and semantics-based program manipulation*, California, USA, 2008, pp. 3-12.

[8] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL Injection and cross-site scripting attacks," in *Proc. of 31st Int. Conf. on Software Engineering*, 2009, pp. 199-209.

[9] W. G. J. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," in *Proc. of 14th ACM Int. symposium on Foundations of software engineering*, Oregon, USA, 2006, pp. 175-185.

[10] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A systematic analysis of XSS sanitization in web application frameworks," in *Proc. of 16th European Conf. on Research in computer security*, Leuven, Belgium, 2011, pp. 150-171.

[11] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proc. of 30th Int. Conf. on Software engineering*, Leipzig, Germany, 2008, pp. 171-180.

[12] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and precise sanitizer analysis with BEK," in Proc. 20th USENIX Conf. on Security, San Francisco, CA, 2011.

[13] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications," in *Proc. of 2008 IEEE Symposium on Security and Privacy*, 2008, pp. 387-401.

[14] V. Ganesh, A. Kieżun, et al, "HAMPI: a string solver for testing, analysis and vulnerability detection," in *Proc. of 23rd Int. Conf. on Computer aided verification*, Snowbird, UT, 2011, pp. 1-19.

[15] M. Monga, R. Paleari, and E. Passerini, "A hybrid analysis framework for detecting web application vulnerabilities," in Proc. 2009 ICSE Workshop S/W Engineering for Secure Systems, 2009.

[16] R. Zhang, S. Huang, Z. Qi, et al, "Static program analysis assisted dynamic taint tracking for software vulnerability discovery," *Computer&Mathmatics Application,*vol. 63, pp. 469-480, 2012.

[17] M. Cengarle and A. Knapp, "OCL 1.4/5 vs. 2.0 Expressions Formal semantics and expressiveness," *Software and Systems Modeling,* vol. 3, pp. 9-30, 2004.

[18] P. K. Manadhata and J. M. Wing, "An Attack Surface Metric," *IEEE Transactions on Software Engineering,* vol. 37, pp. 371-386, 2011.

[19] A. Jimenez, and A. Cavalli "Software Vulnarabilities, Prevention and Detection Methods: A Reviw," in *Proc. European Workshop on Security in Model Driven Architecture*, Netherlands, 2009, p. 6-13.

[20] NIST, "Source Code Security Analysis Tool Functional Specification Version 1.1," May 2007, Accessed 2011.

[21] W. Lei, Z. Qiang, and Z. Peng Chao, "Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking," in *8th IEEE Int. Working Conf. on Source Code Analysis and Manipulation*, 2008, pp. 165-173.

[22] T. Vajk, G. Mezei, and T. Levendovszky, "An Incremental OCL Compiler for Modeling Environments," Electronic Communications of the EASST, vol. Volume 15: OCL Concepts and Tools, 2008.