Static Code Analysis for Software Security Verification: Problems and Approaches

Zeineb Zhioua
SAP Labs France
zeineb.zhioua@sap.com

Stuart Short
SAP Labs France
stuart.short@sap.com

Yves Roudier
EURECOM
yves.roudier@eurecom.fr

***Abstract*—Developing and deploying secure software is a difficult task, one that is even harder when the developer has to be conscious of adhering to specific company security requirements. In order to facilitate this, different approaches have been elaborated over the years to varying degrees of success. To better understand the underlying issues, this paper describes and evaluates a number of static code analysis techniques and tools based on an example that illustrates prevalent software security challenges. The latter can be addressed by considering an approach that allows for the detection of security properties and their transformation into security policies that can be validated against security requirements. This would help the developer throughout the software development lifecycle and to insure the compliance with security specifications.**

***Index Terms*—static analysis, code analysis tools, security properties, program modeling**

I. Introduction

Flaws can be introduced into software throughout its development lifecyle, that result from design or development errors. Undetected flaws can turn into security vulnerabilities at run-time, and can be exploited by intruders to introduce serious damages to the software critical resources. Undetected flaws ultimately entail maintenance costs, in addition to losses incurred by potential attacks. The use of security testing techniques is now well established to find out such flaws.

Static code analysis tools to avoid such issues and to help produce safe and secure software are slowly catching up yet still fail to capture all the requirements expected in terms of secure software engineering. The derivation and validation of security properties in software indeed present a challenging problem that different researchers have tried to tackle based on a variety of security analysis tools and models. These tools make use of different techniques, such as lexical analysis, syntactic and semantic analysis. Each of these tools is focused on specific security properties and requires human intervention to different degrees.

Dynamic analysis' strength lies in the possibility to take action in the presence of dynamic inputs. This type of analysis generally considers the program to analyze as an atomic artifact. It resorts to different approaches in order to analyze the run-time behavior of the program, often without any access to the source code. The latter approach requires a sufficient number of test cases and is quite time-consuming, yet it cannot ensure an automatic verification or a complete coverage of the test cases space of the analyzed program. Dynamic analysis reports failures at the instant they occur, and provides details allowing the developer/tester to make the required corrections.

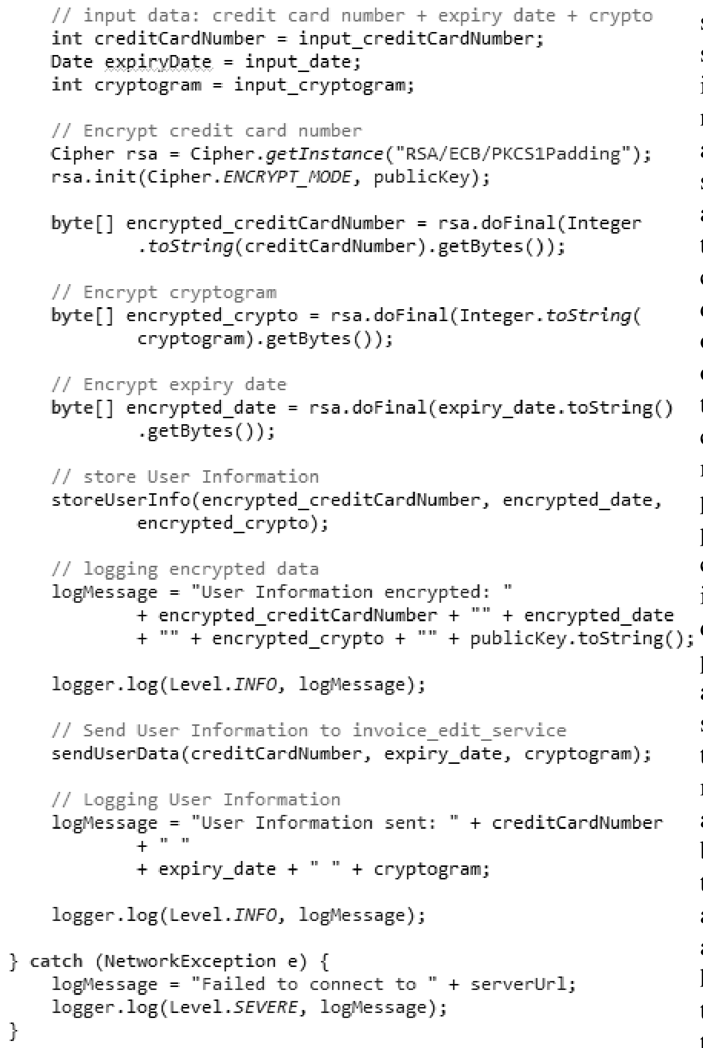
In comparison, static analysis in all its forms ensures a complete coverage of the program branches [4], used APIs, program dependencies, or configuration files explored. Static analysis refers to different methodologies, including model checking and model provers, to verify the execution paths of a program without actually executing it [2]. Unlike manual review, which relies on the tedious examination of sequences of the concrete or symbolic execution of a program, static code analyzers can capture comprehensive and accurate models of the software, like for instance an abstract representation of all the execution paths, which test-case execution falls short to cover. We discuss in this paper different static analysis approaches to securing software and what are the challenges ahead in order to assess different types of security properties, notably complex ones defined at the application level in close relationship with the design.

This paper is organized as follows: Section 2 highlights common security issues based on a motivating example. Section 3 discusses different definitions of software security properties that can be found in the literature. Section 4 then briefly surveys different static code analysis techniques that are used today, while Section 5 provides further details on a selection of four static code analysis tools representative of current uses. Section 6 discusses the applicability of such tools, as well as experimental results with respect to the motivating example. Finally, section 7 concludes the paper and discusses future work.

II. Motivating Example

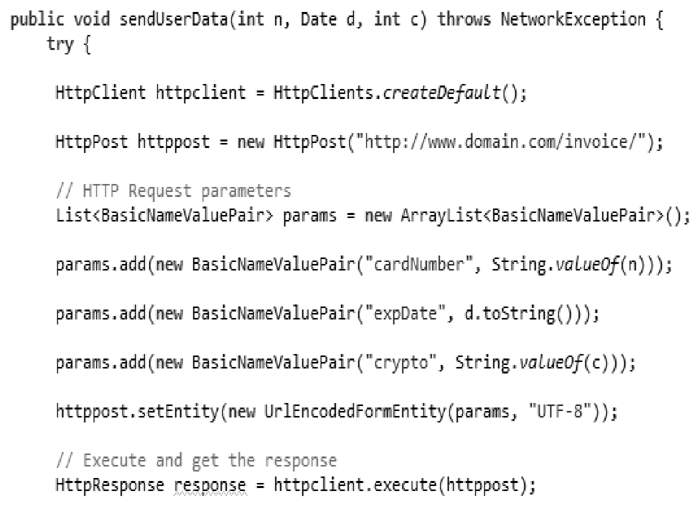
To illustrate further the concepts discussed in the previous section, and to motivate the idea we are pushing to achieve, we present a sample code (fig.1 and fig.2) in which we have injected security flaws and introduced the notion of security properties.

The payment information of the user are provided as input. The credit card number, the 3-digit cryptogram and the expiration date are then encrypted using an encryption library. The encrypted data is afterwords stored in the database for different reasons, such as the payment made. This may provide assurance about the confidentiality of these critical data in



```
    // input data: credit card number + expiry date + crypto
    int creditCardNumber = input_creditCardNumber;
    Date expiryDate = input_date;
    int cryptogram = input_cryptogram;

    // Encrypt credit card number
    Cipher rsa = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    rsa.init(Cipher.ENCRYPT_MODE, publicKey);

    byte[] encrypted_creditCardNumber = rsa.doFinal(Integer
            .toString(creditCardNumber).getBytes());

    // Encrypt cryptogram
    byte[] encrypted_crypto = rsa.doFinal(Integer.toString(
            cryptogram).getBytes());

    // Encrypt expiry date
    byte[] encrypted_date = rsa.doFinal(expiry_date.toString()
            .getBytes());

    // store User Information
    storeUserInfo(encrypted_creditCardNumber, encrypted_date,
            encrypted_crypto);

    // logging encrypted data
    logMessage = "User Information encrypted: "
            + encrypted_creditCardNumber + "" + encrypted_date
            + "" + encrypted_crypto + "" + publicKey.toString();

    logger.log(Level.INFO, logMessage);

    // Send User Information to invoice_edit_service
    sendUserData(creditCardNumber, expiry_date, cryptogram);

    // Logging User Information
    logMessage = "User Information sent: " + creditCardNumber
            + " "
            + expiry_date + " " + cryptogram;

    logger.log(Level.INFO, logMessage);

} catch (NetworkException e) {
    logMessage = "Failed to connect to " + serverUrl;
    logger.log(Level.SEVERE, logMessage);
}
```

Fig. 1. Sample code



```
public void sendUserData(int n, Date d, int c) throws NetworkException {
    try {

     HttpClient httpclient = HttpClients.createDefault();

     HttpPost httppost = new HttpPost("http://www.domain.com/invoice/");

     // HTTP Request parameters
     List<BasicNameValuePair> params = new ArrayList<BasicNameValuePair>();

     params.add(new BasicNameValuePair("cardNumber", String.valueOf(n)));

     params.add(new BasicNameValuePair("expDate", d.toString()));

     params.add(new BasicNameValuePair("crypto", String.valueOf(c)));

     httppost.setEntity(new UrlEncodedFormEntity(params, "UTF-8"));

     // Execute and get the response
     HttpResponse response = httpclient.execute(httppost);
```

Fig. 2. SendUserData method

storage; here we mean that these critical data have to be kept secret when stored and persisted in the database. However, the invocation of the method (sendUserData) sends the credit card number, the cryptogram and the expiry date in plain text to an external source. This is a security breach that automated source code vulnerability detection tools cannot recognize automatically using string-matching, and independently from the application expected security properties. Even though the critical data (the assets) are encrypted, the program cannot be deemed as secure or, in this case, ensuring the confidentiality of the payment information. On the other hand, the action of sending critical data in plain text is a severe vulnerability, that violates the security property confidentiality. Integrity and confidentiality of data are classically guaranteed by the implementation of access control mechanisms. If an unauthorized party gets access to the sensitive payment information, the property confidentiality is then breached. A typical use case consists in using logging functionalities for analysis and auditing purposes. As we can see in the sample code, the encrypted data are logged with the encryption key (publicKey), and the payment information are logged in plain text; this represents a severe security threat, that will only be detected after the software is released to the customer, or even worse, after the flaw is exploited by intruders. The encryption mechanism may insure the data is kept secret, but can't provide assurance about where and how the data will propagate, where it will be stored, or where it will be sent or processed. This entails the need for controlling information flow using static code analysis. This same idea is emphasized by Andrei Sabelfeld, and Andrew C. Myers [30], who deem necessary to analyze how the information flows through the program. According to the authors, a system is deemed to be secure regarding the property confidentiality, if the system as a whole ensures this property. If we had a security policy that expresses the "confidentiality of user's payment information" requirement, current static code analysis tools will not be able to concretize it or to relate it the concrete user information variables in the source code. Hence, the code analysis tools will not afford to verify the compliance of the developed program with this security policy. If a known vulnerability is identified, static code analysis tools will only detect its exact location, that is, in which line of the source code, but they don't provide the means to back-track the source of vulnerability and identify the source that led to it.

The main issues we raised in this sample code are related to capabilities of static code analysis tools to:

- define the assets to be protected: we mean by assets the critical resources/variables in the source code
- represent/concretize abstract security properties with respect to the code: in other words, how to map between the abstract security policy and the assets to be protected
- detect the presence of the "confidentiality in storage" of the assets
- detect the bad programming practice "send critical data in plain text"

- back-track the source of the vulnerability: we need to backtrack the security vulnerability and identify its sources
- establish the mapping between the vulnerability and the violated security properties: security vulnerability can be perceived as violation of security property
- detect the bad programming practice "storage/logging of the encryption key with the encrypted data in the same table"
- translate the detected properties into a security policy

III. Security Properties

Several studies have suggested the use of static code analysis in order to establish the satisfaction of security properties in software implementations. The objectives of such analysis vary widely as security properties are understood very diversely as well.

Many authors take it for granted that security properties can be defined universally. In contrast, some authors contend that security properties are highly dependent on the level of abstraction considered and on the application developed. For instance, [19], Antonio Maña and Gimena Pujol [1] classify security properties along several dimensions in emerging open and distributed environments: Abstract Security Properties (ASP) represent security properties considered over the initial draft of the software architecture during application requirements engineering, and can be formalized. Concrete Security Properties (CSP) map ASPs to security mechanisms or algorithms implemented into software. The same ASP can be proven by different CSPs. ASPs and CSPs can be connected by logical relationships, for instance through the logical implication, and which they term Semantic Security Properties (SSP). In contrast, Domain Security Properties (DSP) are specifications of security properties generic to a given domain.

ASPs are for instance often considered in relationship with complex access control or usage control model policies introduced into software, especially through language based security approaches. For instance, the JIF framework [42] relies on the static analysis of the information flows of a security policy and the verification of the conformance of the flows implemented with those specified in that policy. However, application level security concerns have not really made their way into mainstream static code analysis techniques, which focus mostly on concrete security properties, and which is sometimes even restricted to code safety analysis.

Code safety can be considered as the most concrete level of abstraction in software for what regards security properties. Practitioners generally look for the absence of exploitable safety vulnerabilities in software. Though this is most generally addressed through security testing, static validation may help. The dual modeling problem introduced by John Wilander et al. [9], [8] for instance sheds some light on the correlation between security properties and bad programming practices that may result in safety issues. Wilander et al. introduce a static analysis methodology based on the detection of security and insecurity patterns. The analysis starts by ruling out the presence of an insecurity pattern in the code. If one such pattern is detected, the analysis then proceeds to check whether faulty behaviors are prevented by a security pattern encompassing the insecurity pattern scope. This means that potential vulnerabilities are foiled by the presence of appropriate security mechanisms. The analysis relies on the mapping of functionalities over code snippets. This mapping may somewhat restrict the expressivity of the analysis in that only simple functions are likely to be automatically recognized in a complex source code. The insecurity and security patterns, together with the mapping between both types of patterns can be regarded as a security policy. A number of researchers similarly regards security properties as classes of good vs. bad programming practices with respect to a given security policy. For instance, Aris Zakinthinos and E.S. Lee [18] define security properties as the instantiation of a security policy that can be satisfied by more than a single property. The authors distinguish between security properties regarding high-level (trusted) and low-level (untrusted) users, in such a way that low-level users are not able to make deductions about events generated by the high-level users. Another definition [17] considers security properties following a classical IT security policy: such a policy ensures that software assets or resources have CIA (confidentiality, integrity, availability) properties, and therefore that their confidentiality cannot be violated, and that they cannot be corrupted, or made unavailable, if so specified. The network security properties of authorization, message confidentiality and integrity, non-repudiation of sending or receipt are also used in distributed software, or in software involving several principals. In the following section, we will explore more the static code analysis and the capabilities it offers in the aim of identifying and validating security properties in source code.

IV. Static Code Analysis

The previous section has given an overview about the different definitions attributed to security properties in the literature, and most of them were considered in the modeling and representation of security properties that source code analysis tools made use of.

A. Static Code Analysis in the Practice of Software Development

Delivering and deploying secure software has always been a challenging issue that software vendors try to achieve. Creating more secure software can help reduce security related maintenance and update costs, given the fact that it helps reduce the sources requiring security corrections.
Static code analysis can be used according to Fabian van den Broek [14] in two situations; the first is during testing, and the second during development and before testing. Many developers rely on testing to ensure the safety of their programs; but this doesn't guarantee the security of the developed software, given the fact that it aims at verifying that the it only meets the functional requirements [28]. Hence, functional testing falls

short in covering the security aspect of the software under inspection. In addition, it is applied only to executable and not to source code or byte-code, and takes place rather late in the software development process. Unlike testing, static code analysis can be applied to single files or to entire program code, and doesn't require the development to be complete. In the earlier stages of the development process, developers may make mistakes and programming errors that can be detected by compilers; in many cases, compilers provide corrections to the detected flaws, and the development process is still ongoing. However, this principle is not applicable to most security vulnerabilities that can remain undetected. The more a flaw is undiscovered, the greater it costs to fix [5]. Some organizations try to overcome this lack of focus on security by performing Penetration tests [2]. Another approach consists in detecting the security flaws in the source code of the developed software before it is released or even tested. This can be manual, meaning that tests are carried out by human analyst/developer, or automatic, using a code analysis tool. Note that manual code review can be much more time-consuming than automatic code review, specially when the entire source code is to be analyzed, or the code to inspect is large. [32]
From this perspective, automatic static code analysis can be integrated and applied regularly to the Software Development Life cycle. These tools are to be used to complement the manual code review, and not to totally replace it as precised by Jernej et al. [33]. Having all this is mind, we can emphasize on the importance and necessity of the static code analysis in reducing the sources of security issues in the early stages of the software development life cycle, and before it is released to customers.

B. Techniques for Static Code Analysis

Static analyzers are used for different purposes, such as security vulnerabilities and bugs detection, verification of security properties as well as for program understanding [6].

Formal static analysis makes use of different approaches in the objective of analyzing the program without executing it. Control-flow and Data-flow are two of the commonly adopted formal methods for program representations, and are used in static code analysis.

1) Model Checking: Model checking is one of the formal approaches that was first introduced by Steffen and Schmidt, and is applicable to programs having finite states, or that can be reduced to finite state; it is a technique that allows the automatic verification of properties on finite-state systems. This approach requires the model construction and the properties specification as well. Model checking supposes first the construction of a model that is the transformation of the system into a formalism (such as the Kripke structure [11]) accepted by a model checking tool. The modeling may also require a certain level of abstraction in order to eliminate irrelevant details. Model checking requires as a second step the definition and the specification of the properties to be met by the software model subject of the analysis, and is usually given using logical formalism, like for instance the temporal logic. However, once the specification of the requirements is achieved, no human intervention can be performed on the input specification. Most of the Model Checking methods are focused on the Temporal Logic, and were introduced by a number of researches, among them [21], who proposed a Model Checker allowing to verify the compliance of finite state systems to a Temporal Logic specification. The verification is performed by exploring the state space in order to determine whether the specified properties are satisfied or not.

2) Control-flow analysis: Control-flow analysis is one of the common used techniques for static code analysis. The program control-flow is modeled as a directed Control Flow Graph (CFG), and was first introduced by Frances E. Allen [29]. CFG is directed graph that is used to represent blocs of code in the the form of nodes, the control dependencies in the form of directed edges, starting with an entry node and concluding with the end point of the program. The CFG construction can be carried out based on an abstract syntax graph representation such as AST (Abstract Syntax Tree) to which control flow information are introduced [12] [13]. The main focus of this technique is to determine how the procedures in a program call each other, as well as to determine which functions are effectively called.

3) Data-flow analysis: Data-flow analysis, on the other hand, is based on the abstract representation of the analyzed program semantics, and is focused on the extraction of the possible values of data. It aims at representing data dependencies in the source code, and allows to track the effect of input data. It aims also at mapping program's statements with the data-flow. The latter gathers information about the possible set of values [24], and is often performed on the Control Flow Graph. Data-flow analysis has for objective to statically predict the the dynamic behavior of the analyzed program.

4) Symbolic analysis: Symbolic analysis consists in considering the program variables. According to Wolfgang Wogerer [25], this approach can be seen as a compiler that translates the program being analyzed to an intermediate language, consisting of symbolic expressions and recurrences. This technique is supported by computer algebra systems, that adopt simplification methods to ensure the quality of the output results. The analyzed program consists of three parts: the state, the state condition and the path condition. As for the state, it is composed of a (variable, value) pair. State condition is a logic formula that describes assumptions about the variables values. The path condition, on the other hand, is a logic formula that defines the condition for which the program point is reached. Symbolic analysis is deemed to be useful in transforming unpredictable loops to predictable sequences, and is mainly used for code optimization, performed by compilers.

5) Information-flow analysis: Information flow is mainly analyzed using dynamic analysis approach, however, static code analysis can be used to approximate the information flows and ensure their security, according to Pistoia et al. [4] In the information-flow terminology, we distinguish between direct and indirect informatiom flow. The direct arises from direct data flows, and the indirect is induced indirectly by branching control flow [40].

V. Static Code Analysis Tools

In order to have concrete information, we conducted an analysis on source code analysis tools. The performed research doesn't aim at classifying or ranking the tools, but instead, to allow a deeper understanding of the analysis approach followed by these tools, as well as the security properties they detect. Static analysis tools are used mainly in the objective of detecting security vulnerabilities in the code, so that the developer makes the needed corrections on the identified security flaws before the software is released to customer.

We investigated a number of static analysis tools with the aim of identifying their analysis methodologies, and comparing their accuracy and performance. The precision of the analysis performed by the studied tools is estimated regarding the amount of false positives they report. The main objective behind this analysis is to shed the light on areas where innovations can be proposed. Some of the security analysis tools generate an important number of false positives (false alarm), which reduces the efficiency of the considered tool [7]. On the other hand, a tool that reduces both false negatives (flaws that the tool doesn't report) and false positives (reported flaws that the program doesn't contain) is deemed to be more accurate. Automated tools usually use string regular expressions they match against source code statements in order to identify security vulnerabilities [43]. Other elements are to be taken into consideration when making an investigation about static analysis tools, namely the considered security properties, the required human intervention amount, as well as the output visual aspect and interpretation complexity.

This section presents a research on a number of static analysis tools. We would consider ones that allow the detection and evaluation of security properties, as well as tools that are used to detect security vulnerabilities in the source code. We will justify the second alternative based on the fact that security vulnerabilities are violations of security properties. We carried out experiments on the sample code (fig.1, fig.2) using the presented tools. For some of the considered tools, we had to translate the sample code to the C programming language, as these tools only support C.

A. MOPS

MOPS (MOdel checking Program for Security Properties) [3] makes use of the model checking technique to check for violation of security rules, that are defined as temporal safety properties. It is based on a formal modeling approach for both the program and the security properties, and proceeds by the analysis of the implemented models.

As for program representation, MOPS models the program in the form of Push Down Automaton (PDA), that contains all the feasible execution paths. Push Down Automata are used as tools to analyze procedural sequential programs, and more specifically those having recursive procedures [16]. As for automata, they are according to Schneider [15] used in the objective of specifying security policies that can be enforced by mechanisms. MOPS makes use of this approach to model security properties in the form of Finite State Automata (FSA), that dictate the order of security-relevant operations sequence. The modularity of security properties was also proposed by MOPS; this approach allows the decomposition of complex security properties into simpler and reusable basic security properties, that are easy to model and to extend (such as role based access). MOPS verifies that the security properties are properly respected in all the execution paths of the analyzed program, making use of the model checking on the PDA, and checks if risky states are reachable within the PDA.

B. SPlint

Secure Programming LINT (SPlint) is an annotation-based data-flow static code analyzer for C for security vulnerabilities and programming flaw detection. It makes use of annotations (semantic comments) entered by the developer. The annotations serve as specification of the constraints (properties) about a library, a variable, a function or a type. In other words, annotations serve as properties specification. SPlint execution is an iterative process, that helps the developer/analyst to detect vulnerabilities locations, and to eliminate the warnings by adjusting annotations or modifying the code. SPlint parses the source code of the program subject to the analysis, and generates the Abstract Syntax Tree (AST) based on the formal semantics of the program's programming language. The annotations entered by the developer serve as specification of the high-level security properties about an asset. SPlint generates constraints from the annotations entered by the developer and adds them to the Abstract Syntax Tree (AST) of the program to analyze. As for annotations format, they are similar to comments in C: /* @notnull@ */ and they are syntactically associated to functions parameters, return values, variables, etc. Annotations allow expressing intra-procedural pre-conditions and post-conditions on assets. The developer/analyst can customize the security properties, and add security patterns to be detected in the AST according to his needs, which makes SPlint an extensible tool. If the property expressed by the annotation is violated, SPlint reports a warning for any return path that fails to satisfy the property.

C. GraphMatch

GraphMatch is a code analysis tool/prototype for security policy violation detection [9]. For the program modeling, GraphMatch makes use of a widely used tool called

CodeSurfer[1] [26] that generates the System Dependence Graph (SDG) from the source code provided as input. SDG [20] is an inter-procedural dependence graph representation, and is an extension to the Program Dependence Graph (PDG). SDGs were first proposed by the authors of *"Interprocedural Slicing Using Dependence Graph"* [31], and have proved to be useful in performing deep analysis of programs [27]. They have been developed over the last two decennies, and consist now a basis to perform the code analysis, based on different approaches, such as slicing [31] [39] [41] or Model Checking [34]
As for the PDG, it is a directed graph whose nodes are predicates (variable declarations, assignments, control predicates) and edges are data and control dependence representation; both types are computed using respectively control-flow and data-flow analysis. PDG is the intra-procedural representation of a program, and considers the control and data flow dependencies within a procedure. On the other hand, SDG modeling considers the inter-procedural calls, that is, the control and data dependencies between procedures in a program. Given the fact that the generated SDG is in a proprietary file format, the authors have deemed necessary to transform the generated SDG into the GraphMatch's file format [10]. GraphMatch allows the users to customize the positive and negative security patterns that the program will be analyzed against, as well as to define the relationships between positive and negative security properties. John Wilander [8] has considered examples of security properties covering both positive and negative ones, that according to the author meet good and bad programming practices.[8]. GraphMatch traverses the generated SDG with the objective of finding security pattern matching. For a security property violation to be raised, GraphMatch proceeds in two steps: verification of the negative pattern matching first, followed by a verification of the embedding positive security pattern. If negative security patterns are found, the tool doesn't report a violation immediately, but proceeds to the verification of the embedding security property. If it corresponds to a positive security pattern, then GraphMatch doesn't raise a warning. However, if the embedding security pattern match is not found, GraphMatch raises a warning and reports a security property violation.

D. Fortify

Fortify is a static analysis tool that processes the source code in a way similar to a code compiler. It has the ability to detect and fix vulnerabilities in the source code and to be run on multiple environments (Windows, Linux, Mac). Fortify takes as input the source code of a single file or an entire application composed of many files, and conducts a semantic analysis approach; it represents semantically the control flow and the data flow of the code. The tool is able to map the execution and the data flow and can for example recognize that input data are left untested or invalidated before being passed to a function or component. Fortify performs an inter-procedural analysis in the objective of making the analysis as accurate as possible. [2] The tool detects four types of issues: Semantic, Data Flow, Control Flow, Configuration and Structural. The Semantic Analyzer detects potentially dangerous uses of functions and APIs at the intra-procedural level. Basically a smart *GREP* [3]. The Data Flow analyzer detects potential vulnerabilities that involve tainted data (user-controlled input) put to potentially dangerous use. The data flow analyzer uses global, inter-procedural taint propagation analysis to detect the flow of data between a source (site of user input) and a sink (tainted data, or dangerous function call or operation) The Structural Analyzer detects flaws in the structure or the definition of the program. As for the Configuration Analyzer, it looks for dangerous flaws in the application deployment configuration files. The Control Flow Analyzer detects potentially dangerous sequences of operations. By analyzing control flow paths in a program, the control flow analyzer determines whether a set of operations are executed in a certain order.

The generated file is then processed by the Audit Workbench Tool that presents the results in a user-friendly format [22]. The Audit Workbench tool is customizable and enables the user to configure the custom rules (from the rulespacks) for audit; the user selects the types of issues he wants to be warned about. The Workbench tool flags the detected vulnerabilities, provides the problem description and how it might be fixed.

VI. Evaluation and Discussion

In this section, we will reflect more upon the outcome of the static code analysis tools investigation, and will illustrate the results in a summary table, containing the main points of our interest. We focus on the program modeling approach that exploits the source code properties, and represents the program in a faithful model.

Different criteria are to be taken into account for evaluating a static code analysis tool, such as the security properties representation model, and which analysis approach is used to validate these security properties. The abilty of the tool in modeling and detecting specified security properties is highly dependent on the abstraction level of the program modeling.
Another criterion is the soundness of the analysis tool. It can be evaluated taking into account the type of security vulnerabilities the tool is able to detect, as well as the amount of false positives and false negatives it produces. Reporting an important amount of false alarms can be misleading to the users, who will get discouraged of using that tool [35]. The soundness and accuracy of the tool's performed analysis is highly dependent on the precision of the code representation [36]; false alarms can emanate from an incorrect modeling of the system.

We are also interested in the mapping between detected vulnerabilities and the violated security properties, which is not covered by most of the studied static analysis tools.

[1]http://hiper.cis.udel.edu/lp/lib/exe/fetch.php/courses/cisc879/codesurfer-demo.pdf

[2]http://stackoverflow.com/questions/13051974/how-does-fortify-software-work

[3]http://blog.linuxacademy.com/linux/grep-tutorial-searching-file-contents/

The tools usability is one of our points of interest. A tool is deemed to be usable if its generated results are understandable to an average developer, meaning not having advanced knowledge in security. The usability can also emanate from the quality of its output and how it is presented to the user.

We focus also on the ability of the tool to allow users define their own rules against which the program will be evaluated. We consider the latter of paramount importance when dealing with static code analysis; most of the analyzed tools have their set of predefined rules. From this perspective, the tool will not detect a flaw if the corresponding pattern is not predefined.

We consider also the extensibility of the tool, that is, the integration possibilities it offers. A good tool has to be efficient and scalable enough to perform the analysis on complex or large programs, taking into account the dependencies between components without impacting the speed of its execution.

TABLE I
EVALUATION OF STATIC CODE ANALYSIS TOOLS.

	MOPS	SPlint	GraphMatch	Fortify
Program model	PDA	CFG	SDG	semantic DFG and CFG
Security properties model	FSA	Constraint-based	positive security pattern (PDG)	NA
Security vulnerabilities model	NA	NA	PDG	Signature-based patterns
Static analysis method	Intra-procedural Control Flow, Inter-procedural Control Flow and Model-checking	Intra-procedural Control Flow, Intra-procedural Data Flow	Security pattern matching	Inter-procedural semantic, Data Flow, Control Flow configuration and structural analysis
Customizable security properties	yes	yes	yes	yes [4]
Extensibility	NA	NA	No public API	Commercial tool
Output	Text	Text	Text	HTML or XML file [5]
Supported programming languages	C	C	C	C#, ABAP, C, C++, COBOL, Java, PHP, Python, Visual Basic, JavaScript, VB Script, etc. [23]
Analysis result	no violation reported	no violation reported	NA	Privacy violation

SPlint is able to detect not only security-related vulnerabilities, but also coding errors that may affect the quality of the code [6]. However, SPlint doesn't handle multiple programming languages, and is only limited to C programs. Regarding its accuracy, SPlint produces an important number of false positives that lead to confusion when interpreting the results. In addition, it performs only intra-procedural data flow analysis; the control-flow and the inter-procedural data-flow analysis it performs are very limited. SPlint relies on annotations added by developers in their source code, in other words, a number of vulnerabilities will remain undetected if specific annotations are not added.

MOPS is control-flow sensitive, and doesnt consider data-flow dependence, that according to the others limits its scalability. Other shortcomings of MOPS are its incapability to analyze multi-threaded programs or dynamic methods invocation. The experiment on our sample code translated in C programming language, detected no violation for both SPlint and MOPS.

As for GraphMatch tool, it proceeds by model checking using the dependence graph, which combines both the program data and control dependencies, and proceeds by positive and negative security patterns. The tool doesn't scale to the analysis of distributed systems, but only considers a single source code file. Another issue with this tool is its non-scalability [9]. In addition, the graph matching performed by this tool has high complexity, which can impact the performance of the analysis. The tool is more focused on the liveness and safety properties, such as integer input validation and the double *free()* flaw where the *free()* method is called twice attempting to free the same memory allocated using the *malloc()* method. GraphMatch is mainly focused on the order and sequence of instructions, but doesn't cover high level security properties such as confidentiality. We couldn't carry out the experiment on our sample code, as GraphMatch was not available.

As for Fortify, this tool is scalable, and doesn't have restrictions on size of the program to analyze. The performed analysis on our sample code produced as output a critical issue "Privacy violation", accompanied with explanation about the vulnerability (private user information enters the program, the data is written to an external location), its location, and recommendations on how to avoid it. However, it produces false positives. [6]

As shown in the motivating example presented in section 2, static analysis tools allow only the detection of security vulnerabilities, but are unable to identify the security properties that might be affected by this vulnerability. Such identification can be realized by exploiting the vulnerabilities knowledge base (such as NVD). On the other hand, where no security vulnerability is detected, can one deem the analyzed program as secure? Going beyond the detection of security vulnerabilities, and tackling the problem of retrieving security properties using static code analysis is not trivial. Besides, the used encryption mechanism doesn't provide assurance about the security property "confidentiality", that is highly dependent on the flow of the critical data through the program model, which has to be accurate, and has to exploit the source code properties, meaning the control and data dependencies. Besides, the model needs to have a certain level of abstraction and exploit as much as possible the program properties. PDGs are considered a standard tool that allows the modeling of information flow through a program [37], and their strength consists in considering the order of sequences in the program. Hence, they provide an over-approximation of the analyzed program possible behaviors at run-time [38]. SDG modeling, which is an extension to the PDG, considers the inter-procedural calls, that is, the control and data dependencies between procedures in a program. From this perspective, we foresee the use of SDG as a program modeling approach, for its accuracy and its capability in modeling information-flow through a program.

VII. Conclusion and Future Work

In this paper, an assessment of selected static code analysis approaches and tools was carried out. Static code analysis here consists in mapping security vulnerabilities to a program representation. We presented how verification of simple security properties in the source code took place with different such representations.

In relation to the paper's motivating example we have also showed that the static code analysis tools discussed are not able to cover all the outlined issues. We plan to extend one of these tools in order to accommodate more complex security properties that may be derived from the code, even when no security vulnerability or threat is detected (e.g., confidentiality in storage). In order to overcome these challenges, we plan to address a few fundamental problems in coming work, namely:

- One such key question will be how to translate detected security properties enforced by mechanisms into a security policy that can be afterwards validated against the security requirements expressed in the specification of the program? Making the result of such an analysis easy to understand and to interpret for a developer will also be critical.
- A second critical question related to the determination of the scope of a vulnerability. How to detect the source of a vulnerability must be addressed in relationship with the type of static analysis selected. Running a code analysis tool allows the detection of a vulnerability in the exact code location, but sometimes, programmers and code reviewers need to identify the source of that vulnerability [38], and hence to back-track the analysis that led to such an incorrect state[39]. A related concern is how to determine which security property is under threat when a vulnerability is detected.

Acknowledgment

This work was partly supported by the EU-funded project OPTET [grant no. 317631].

References

[1] Antonio Maña and Gimena Pujol, *Towards Formal Specification of Abstract Security Properties*, 0-7695-3102-4/08, 2008 IEEE
[2] Brian CHESS, and Gary MCGRAW, *Static Analysis for Security*
[3] Hao Chen, and David Wagner. *MOPS: An Infrastructure for Examining Security Properties of Software*, Copyright 2002 ACM 1-58113-612-9/02/0011
[4] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav, *A Survey of Static Analysis Methods for Identifying Security Vulnerabilities in Software Systems*, (n.d.).
[5] Brian Chess, and Jacob West *Secure Programming with Static Analysis, Software Security Series edition.*
[6] Patrik Hellstrm, *Tools for Static Code Analysis: A Survey*, n.d.
[7] C.C. Michael, and Steven Lavenhar, *Source Code Analysis Tools - Overview.*, Cigital, Inc. 2005-2007 (n.d.).
[8] John Wilander, *Modeling and Visualizing Security Properties of Code Using Dependence Graphs*, (n.d.).
[9] John Wilander and Pia Fak, *Pattern Matching Security Properties of Code using Dependence Graphs*
[10] John Wilander, *Contributions to Specification, Implementation, and Execution of Secure Software*, n.d.
[11] Michael Huth. Model Checking Modal Transition Systems Using Kripke Structures, n.d. http://pubs.doc.ic.ac.uk/fairness-abstraction-3-valued/fairness-abstraction-3-valued.pdf.
[12] Emma Soderberg, Gorel Hedin, Eva Magnusson, and Torbjorn Ekman, *Extensible Intraprocedural Flow Analysis at the Abstract Syntax Tree Level*
[13] R.M. Smelik, *Specication and Construction of Control Flow Semantics*
[14] Fabian van den Broek, Static Code Analysis in Java.
[15] FRED B. SCHNEIDER, *Enforceable Security Policies.*, ACM Transactions on Information and System Security, Vol. 3, No. 1, February 2000, Pages 3050. (n.d.).
[16] O. Burkart and B. Ste?en. Composition, Decomposition and Model Checking of Pushdown Processes. Nordic Journal of Computing, 2:89125,1995.
[17] *ARM Security Technology*, (n.d.).
[18] Aris Zakinthinos, and E.S. Lee. *A General Theory of Security Properties*, (n.d.). 1081-6011J97 $10.00 0 1997 IEEE.
[19] Marco Anisetti, Claudio A. Ardagna, Ernesto Damiani, Fulvio Frati, and Hausi A. M uller, *Web Service Assurance: The Notion and the Issues*, (n.d.). http://www.mdpi.com/1999-5903/4/1/92.
[20] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel, *System-Dependence-Graph-Based Slicing of Programs With Arbitrary Interprocedural Control Flow*, (n.d.).
[21] Edmund M. Clarke, and E Allen Emerson, *Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic* (n.d.).
[22] Hyunji Kim, *Fortify Source Code Analaysis Tools*, n.d.
[23] *HP Fortify Software Security Center v3.60, System Requirements*
[24] Aarthi Ganesan and Aparna Boddupalli, *Static Analysis by Abstract Interpretations: For detection of security* vulnerabilities.
[25] Wolfgang Wogerer, A Survey of Static Program Analysis Techniques.
[26] http://www.grammatech.com/research/technologies/codesurfer
[27] Code Surfer, *Dependence Graphs and Program Slicing*
[28] Dejan Baca, *Automated static code analysis - A tool for early vulnerability detection*
[29] Frances E. Allen (July 1970). "Control flow analysis"
[30] Andrei Sabelfeld, and Andrew C. Myers, *Language-Based Information-Flow Security*, IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, VOL. 21, NO. 1, JANUARY 2003, n.d
[31] SUSAN HORWITZ, THOMAS REPS, and DAVID BINKLEY, *Interprocedural Slicing Using Dependence Graphs*, ACM Transactions on Programming Languages and Systems, Vol. 12, No. 1, January 1990, Pages 26-60. (n.d.).
[32] B. Boehm, *Software Engineering Economics. New York: Prentice-Hall*, 1981
[33] Jernej Novak, Andrej Krajnc, and Rok ontar, *Taxonomy of Static Code Analysis Tools*
[34] Masahiro Matsubara, Kohei Sakurai, and Fumio Narisawa, *Model Checking with Program Slicing Based on Variable Dependence Graphs*
[35] Ivo Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo Moreira, *An Overview on the Static Code Analysis Approach in Software Development*
[36] Deguang Kong, Quan Zheng, Chao Chen, Jianmei Shuai, and Ming Zhu, *ISA: A Source Code Static Vulnerability Detection System Based on Data Fusion*
[37] Christian Hammer, Jens Krinke, and Gregor Snelting, *Information Flow Control for Java Based on Path Conditions in Dependence Graphs*
[38] Fang Deng, and James A. Jones, *Weighted System Dependence Graph*, 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation
[39] Mark Weiser, *Programmers Use Slices When Debugging*
[40] Paritosh Shroff, Scott F. Smith, and Mark Thober. Dynamic Dependency Monitoring to Secure Information Flow, n.d.
[41] MARK WEISER, *Program Slicing*
[42] Boniface Hicks, Sandra Rueda, Trent Jaeger, Patrick Drew McDaniel: From Trusted to Secure: Building and Executing Applications That Enforce System Security. USENIX Annual Technical Conference 2007: 205-218.
[43] Justin Clarke, Rodrigo Marcos Alvarez, Dave Hartley, Joseph Hemler, Alexander Kornbrust, Haroon Meer, Gary OLeary-Steele, Alberto Revelli, Marco Slaviero, and Dafydd Stuttard, *SQL Injection Attacks and Defense, n.d. http://adrem.ua.ac.be/sites/adrem.ua.ac.be/files/sqlinjbook.pdf*