A New View on Classification of Software Vulnerability Mitigation Methods

Maryam Mouzarani
Dept. of Computer Engineering
& Information Technology,
Amirkabir University of Technology,
Tehran, Iran.
Email: mouzarani@aut.ac.ir

Babak Sadeghiyan
Dept. of Computer Engineering
& Information Technology,
Amirkabir University of Technology,
Tehran, Iran.
Email: basadegh@aut.ac.ir

***Abstract*—Software vulnerability mitigation is a well-known research area, and many methods have been proposed for it. Some papers try to classify these methods from different specific points of views. In this paper, we aggregate all proposed classifications and present a comprehensive classification of vulnerability mitigation methods. We define software vulnerability as a kind of software fault, and correspond the classes of software vulnerability mitigation methods accordingly. In this paper, the software vulnerability mitigation methods are classified into vulnerability prevention, vulnerability tolerance, vulnerability removal and vulnerability forecasting. We define each vulnerability mitigation method in our new point of view and indicate some methods for each class. Our general point of view helps to consider all of the proposed methods in this review. We also identify the fault mitigation methods that might be effective in mitigating the software vulnerabilities but are not yet applied in this area. Based on that, new directions are suggested for the future research.**

I. Introduction

Software is an important part of a computer system. Being complex or created by incompetent developers, faults might be introduced to the software. There are faults that cause violating the system security. These faults are called vulnerability. There has been much research on preventing, detecting and analyzing software vulnerabilities.

By the time of writing this paper there is a number of surveys on the methods of mitigating vulnerabilities, i.e. [1], [2], [3] and [4]. Among them, [4] surveys the static analysis vulnerability detection methods that are applied in three areas that are associated with sources of vulnerabilities, i.e., access-control, information-flow and application-programming-conformance. It reviews around 88 papers. The studied methods, however, do not cover all the software vulnerability classes. Static analysis methods are also surveyed in [3]. It reviews 23 papers and classifies their methods with a different point of view. In [1] static and dynamic analysis methods are classified and 18 papers are briefly reviewed. The classification for static analysis methods presented in that paper is similar to the one in [3]. The most comprehensive survey is presented in [2] by Shahriar et al. in 2012. They review 173 papers and classify their methods in four classes, i.e., static analysis, dynamic analysis, monitoring and hybrid analysis.

In this paper, we present a new definition for software vulnerability. Based on this definition, vulnerability mitigation methods are classified and reviewed with a new point of view. We use the general classification of fault mitigation methods as a base and extend it to a detailed classification of software vulnerability mitigation methods.

Our comprehensive classification aggregates many of the classification presented in the previous surveys, i.e., [1], [3], [2] and [5]. Also, the general perspective applied in our survey helps to identify the fault mitigation methods that are not yet used in mitigating software vulnerabilities. Since we consider the software vulnerability as a type of fault, these methods may be helpful in mitigating software vulnerabilities. We suggest new directions for the future researches based on our analysis during the review of the proposed vulnerability mitigation methods.

In this paper, our definition of software vulnerability is presented in section II. Based on this definition, software vulnerability mitigation methods are classified in section III. In this section, each class is described in details and some examples are reviewed. Section IV concludes the paper and presents some future directions.

II. Defining software vulnerability

To review vulnerability mitigation methods, a precise definition of software vulnerability is required. Different researchers have suggested definitions for this term which are nearly analogous but have differences. Matt Bishop et al. define software vulnerability by modeling the software as a state machine in [6], [7], [8] and [9]. In this model, a vulnerable state is the state that let unauthorized reads, changes or accessibility modifications to a source. They define vulnerability as a property in the system that let it enter into a vulnerable state. In [8] Bishop defines vulnerability as a weakness that makes it possible for a threat to occur, where a threat is a potential violation of security policy. Amoroso defines vulnerability as an unfortunate characteristic that allows a threat to potentially occur [10]. There are other definitions of software vulnerability in relation with security policy, e.g. [11] and [1]. Most of them define it as a property, characteristic or weakness that may cause compromising the security policy.

In order to clarify the terms property and security policy compromise, we redefine ”software vulnerability”. We use the precise definitions for the concepts in software security and reliability that are presented in [12] and construct our definition of software vulnerability. The taxonomy in [12] is presented in 2004 for the concepts of software security and reliability, such as fault, error, failure, vulnerability and attack. The authors define fault as the cause of error, while error is a state of the system that is probable to failure. Failure -or service failure- is an event in which the delivered service is deviated from the correct service. In fact, a fault may become active and produce an error. Also the error may propagate inside the system and produce more errors. If the propagated error reaches system boundaries and affects the services, it becomes a failure.

A service is defined in [12] as the behavior perceived by users in system boundaries. Correct services are determined by the system specification. Some parts of the system behavior are specified by the security policy, which is a partial system specification. Thus when a system deviates from the security policy, a security failure occurs. This means that compromising security policy causes a security failure.

Faults are classified in [12] based on eight criteria, such as the phase of creation or occurrence, the objective, the phenomenological cause, the system boundary and the dimension. All combinations of the eight elementary fault classes would result in 256 different combined classes. The authors, however, believe that not all combinations are possible. For example, there is no malicious non-deliberate faults, or all the natural faults are non-malicious.

An attack is defined in [12] as a malicious external fault. An attack may be either an external hardware malicious fault, such as heating the RAM with a hairdryer to cause memory errors, or an external software malicious fault, such as a Trojan horse [12]. The term vulnerability is also defined in [12] as an internal fault that enables an external fault to harm the computer system, although harming the computer system is not clearly defined.

According to the previous definitions, we consider software vulnerabilities as:

Definition 2.1 Software vulnerabilities are internal faults that may cause a security failure.

We have concluded this definition, out of the definitions in [12], [8], [10], [11] and [1], since looking a vulnerability as a fault, instead of a property, better clarifies the concept of vulnerability by considering its relation to error, security failure and thus security policy. Like faults, a vulnerability may be dormant and never be activated. It also may be activated and propagated in the system. The activated vulnerability might never reach the boundaries. As an example, suppose that a buffer overflow occurs and the value of a return address in the stack changes as a result. But using a monitoring procedure, the unauthorized change is detected and the program halts. Thus, the security policy is not violated. Monitoring the program, as a vulnerability detection method, is explained in section III-B. When an active vulnerability reaches the system boundaries, it causes a security failure. For example, an attacker may activate the format string vulnerability in a program and make it print some confidential data from the memory [13]. Since the active vulnerability has reached the system boundaries, it has made a security failure.

III.Vulnerability mitigation methods

Since vulnerability mitigation is a well-known research area, a structured approach is required to review the previous related works. In this paper, we review vulnerability mitigation methods using a new point of view. We classify and review these methods based on how we define software vulnerability. In the previous section, software vulnerability is defined as an internal software fault. Since we considered vulnerability as a type of fault, the classifications of fault mitigation methods can be used as a base for classifying vulnerability mitigation methods. Avizienis et al. present a classification for the means of mitigating the faults to achieve a secure and dependable system in [12]. We use this general classification as a base and extend it into a detailed classification of vulnerability mitigation methods. Our classification is illustrated in figure 1. The vulnerability mitigation classes that are shown in figure 1 are described in more details in the following sections. This figure presents a comprehensive view of the previous efforts in mitigating software vulnerabilities. Our classification also aggregates the classifications presented in the previous surveys, such as the ones presented in [1], [3], [2]. Moreover, this classification helps to identify the fault mitigation methods that can be applied to improve current software vulnerability mitigation methods. This helps to suggest new directions for the future research.

A. *Vulnerability prevention*

Generally, fault prevention means avoiding the fault introduction and occurrence in the application during the development. A fault may be introduced during any of the development phases: requirement analysis, design and implementation. To prevent the occurrence of software vulnerabilities during these phases, software security is emerged. Software security is the process of designing, building and testing software for security [14]. It aims at designing and implementing a secure software and educating developers, architects and users to build security in the software [14]. There are various secure software development methods presented by now, such as Microsoft Security Development Lifecycle (SDL) [15], Security Quality Requirement Engineering (SQUARE) [16] and McGraw's secure development method [14]. Also, there are secure coding best practices that are suggested for different programming languages. These best practices educate the programmers to prevent introduction of well-known vulnerabilities during the coding phase, such as [17] for .NET framework, [18] for C/C++ and [19] for Java.

The programmers' lack of security knowledge is an important reason for the introduction of vulnerabilities. Transferring the related information to the developers is an issue in vulnerability prevention. The SHIELDS project was an example of the attempts in this area [20]. The goal in this project was to create a database of security related information for programmers that can be used automatically. A unified modeling language was proposed in SHIELDS for representing this information [21]. Using this language, it is possible to specify a vulnerability class and its relations to the well-known attacks. It also helps to define the methods of preventing a vulnerability class. Thus, it helps the developers to learn how to prevent vulnerabilities in order to achieve the security goals of the application. Some tools were also developed based on this language in that project, such as GOAT [20] and TestInv-Code [22].

B. *Vulnerability Tolerance*

In spite of vulnerability prevention efforts, vulnerabilities are created. Thus, vulnerability tolerance is required. Generally, fault tolerance methods accept the existence of faults

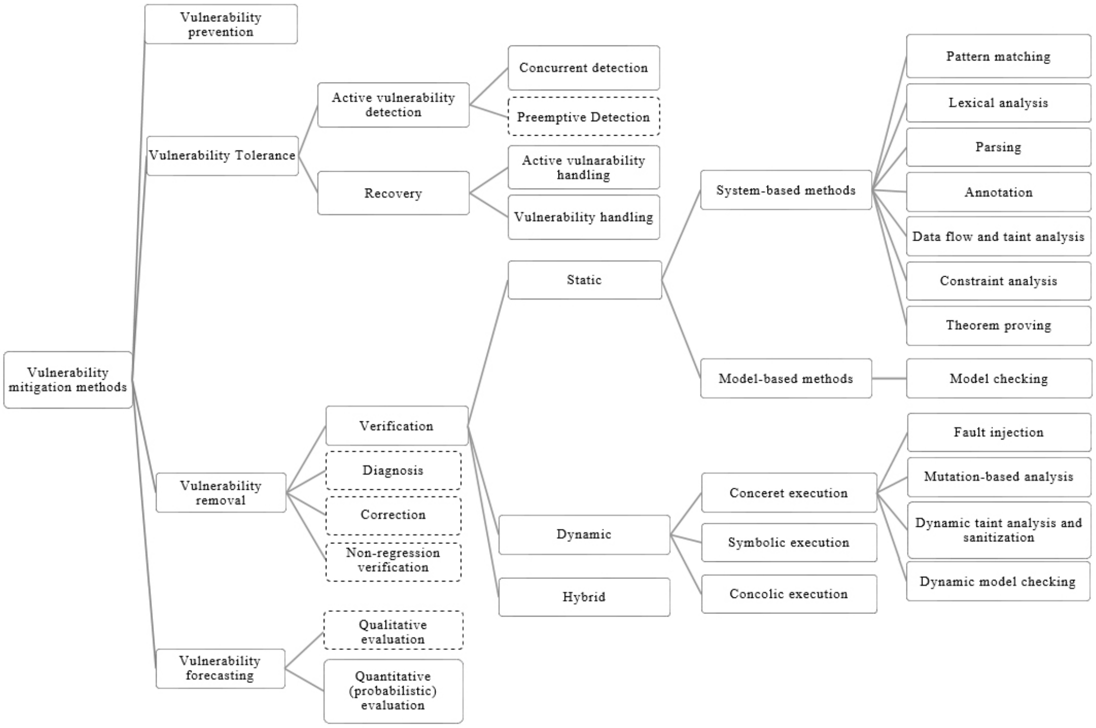

Fig. 1. Our classification of vulnerability mitigation methods according to the classification of fault mitigation methods in [12]. The boxes with dashed borders show the methods that have not been used in mitigating software vulnerabilities yet.

and focus on preventing the activated faults from reaching the system boundaries and causing a failure. Fault tolerance is usually performed in two steps: error detection and recovery [12]. Therefore, we study monitoring methods based on three aspects: the applied error detection, error handling and fault handling techniques. Please note that since we look a vulnerability as a fault, we consider error as an active vulnerability. Thus, the mentioned three aspects are also named as active vulnerability detection, active vulnerability handling and vulnerability handling techniques respectively.

Error detection (active vulnerability detection)

There are vulnerability mitigation methods that control the execution of a program and detects active vulnerabilities at run-time. These methods are also called monitoring methods [2]. Various active vulnerability detection techniques have been used in the proposed monitoring methods. Some examples are monitoring the memory and validating its integrity [23], [24], [25], controlling the flow of user provided data (taint analysis) [26], [27], [28], [29] and validating the arguments of specific functions [30], [31], [32].

For example, the return addresses of functions in the stack memory of the program are monitored in [23], [24] and [25] to detect stack overflows at run-time. If any unauthorized changes of the return addresses is detected, it is concluded that a buffer overflow vulnerability has become active in the program.

Some monitoring methods track the flow of user provided un-trusted data at run-time and react appropriately if the un-trusted data reach sensitive statements in the program, such as [26], [27], [28], [29]. This method is used to tolerate various vulnerabilities, such as DOM-based XSS [29], SQL injection [26], [27], [28], buffer overflow [26], [27], [28] and format string [26], [27], [28]. Some monitoring methods locate specific functions in the program and control their arguments during the program execution, such as [30], [31] and [32]. For example, in [31] the program code is analyzed statically and the query strings, that are used as the arguments of SQL functions, are parsed to extract the ASTs of legitimate queries. In this method, the code is instrumented to control the values of SQL queries before executing the relevant functions. Before executing a query with un-trusted data, the monitoring procedure extracts the AST of the query. It then compares the extracted AST with the AST of the legitimate queries. Any inconsistency between the two ASTs might reveal a malicious query. Thus, an appropriate reaction is taken by the monitoring procedure to prevent security failures.

Detecting the errors may be performed during the normal service delivery (concurrent detection). Also, it may be performed in specific times in which the application does not deliver services (preemptive detection). The latter is usually applied to eliminate the negative effects of software aging. All the studied monitoring methods detect active vulnerabilities during the normal service delivery. However, preemptive error detection can be used to detect the activation of vulnerabilities that makes the program overuse the system resources, like the memory leakage vulnerability.

Error handling (active vulnerability handling)

After an error is detected, it is handled in one of three ways: rollback, roll-forward and compensation. Using the rollback method, the system is restored to a previously stored error-free state. Then, the program continues normal execution from the restored state. In some applications, such as real-time applications, there is no time to rollback. Thus, roll-forwarding is performed to change the system state into a degraded new state that contains no errors. Then, the program executes normally from the degraded state. Roll-forwarding is applicable for predictable errors. Another error handling method is compensation. In this method, the redundancy in the current state is used to mask the error and let the program continues the execution. Many of the monitoring methods halt the program and generate an error message when they detect an active vulnerability, e.g. [32], [27], [33], [24]. In other words, many of the monitoring methods do not perform error handling. Some monitoring methods call an exception handler and take the program to a pre-defined state [34], [26], [28]. Most of the monitoring methods that are used for web applications ignore the requests that result in errors and continue normal execution [30], [31], [35], [29]. Calling exception handlers and ignoring the malicious requests can be considered as simple roll-forwarding actions, since the erroneous state is changed into an error-free state and the program continues normal execution. However, more intelligent reactions can be performed after detecting active vulnerabilities. For example, in [36] the stack content and return addresses are stored to compensate for buffer overflow errors. When a buffer overflow error is detected, the monitoring procedure uses the stored data to help the program continue execution securely.

Many of the presented monitoring methods focus on detecting active vulnerabilities, but less attention is paid to handling the active vulnerabilities. It seems that more effort is required on designing appropriate handling methods for active vulnerabilities. Although halting the program and throwing an exception prevents a successful attack, they violate the availability of the software to the legitimate users. Thus, it may result in deniable of service. Therefore, more intelligent active vulnerability handling techniques should be designed for the monitoring methods. Since the rollback technique is usually used for the transient faults and software vulnerability is a permanent fault, this technique cannot be applied in the monitoring methods. Thus, the roll-forward and compensation techniques can be used to design more intelligent active vulnerability handling methods.

Fault handling (vulnerability handling)

After handling the error, sometimes fault handling is performed to remove the fault and prevent the similar errors in the future. Of course, sometimes the fault is handled immediately after error detection. Fault handling is performed by first recognizing causes of the error. Usually, the faulty component is isolated to prevent the future activation of the fault. A spare fault-free component is then replaced by the faulty one. The system is reconfigured based on the new structure. We are not aware of any monitoring method that consists of a vulnerability handling procedure. However, there are some specific methods for automatically patching the software vulnerabilities, such as [37], [38], [39], [40] and [41]. These methods might be usable in the proposed vulnerability tolerance methods to handle the vulnerabilities. The automatic patching methods analyze the malicious data that is used in an attack and modify the program to filter similar data in the future. These methods can be combined with preemptive active vulnerability detection techniques to generate a complete vulnerability tolerance solution.

Table I summarizes the presented vulnerability tolerance methods so that the reader can review them easier. To sum up, there are various monitoring methods with enhanced error detection mechanisms presented by now. These methods pay more attention to detecting the errors. This might be due to the difference between software vulnerability and the other faults. Usually, software vulnerability is activated by malicious external faults. Therefore, detecting an active vulnerability reveals an ongoing attack. The software should resist the attack as soon as possible to prevent further damages. Thus, the quick detection of the active vulnerability is very important. Halting the program is the fastest low-risk response to the attack. However, it makes the program unavailable to the legitimate users as well. Thus, more intelligent error handling and vulnerability handling techniques should be added to the monitoring methods. To do so, a good starting point is inspiring by the current fault handling and error handling techniques and designing software vulnerability handling techniques.

C. Vulnerability removal

Vulnerability removal is performed to detect and remove the vulnerabilities that are created in software despite the vulnerability prevention efforts. Based on figure 1, the fault removal process consists of four steps: verification, diagnosis, correction and non-regression verification. During the verification step, it is verified if the system adheres to the specification. If not, the reason (fault) is diagnosed and corrected. After removing the fault, the verification is repeated to check if the removal was effective. The verification at this step is called non-regression verification.

Most of the vulnerability removal methods focus on the verification step and don't suggest any diagnosis or correction methods for the detected vulnerabilities. There are, however, special vulnerability diagnosis methods that diagnose the vulnerabilities that are exploited by malicious users. For example, in [42] exploitation of memory corruption vulnerabilities is detected and then the exploited vulnerability is automatically diagnosed. The result of diagnosis consists of the instruction that are exploited by an attacker to corrupt critical program data, the stack trace at the time of memory corruption and the history that the corrupted data are propagated after the initial corruption. This information helps the developers to remove the diagnosed vulnerabilities. We could not find any vulnerability diagnosis or correction procedure that is used after the verification step of a vulnerability removal method. We need vulnerability diagnosis and correction procedures that can be used after the verification step, not after detecting an attack. In other words, these procedures should not be based on the attack information, but based on the information achieved during the verification step.

Some vulnerability detection methods perform the verification step by checking if the software adheres to the security specification, while some of them verify if specific vulnerabilities exist in the software. Figure 1 illustrated our classification of vulnerability verification methods. We divide the verification methods into three main classes: static, dynamic and hybrid methods.

1) Static analysis

Static analysis methods do not execute the program. Instead, they examine the program code and study its possible behaviors. Therefore, the result of static analysis is true regardless of the input data and static methods are usually sound and conservative [43]. A sound method is able to detect any specified vulnerability in the program. In other words, if a vulnerability is defined for the static analyzer and exists in a program, the analyzer will surely find it. In order to

Reviewed Paper	Active vulnerability detection	Active vulnerability handling	Vulnerability handling	Vulnerability
[23]	Detect unauthorized changes of return addresses. (preemptive)	Halts or restarts the program. (rollback)	None	Buffer overflow
[24]	Detect unauthorized changes of return addresses. (preemptive)	Halts the program.	None	Buffer overflow
[25]	Detect unauthorized changes of return addresses. (preemptive)	Halts the program.	None	Buffer overflow
[26]	Monitor the flow of un-trusted data. (preemptive)	Invokes exception handlers. (roll-forward)	None	Any vulnerability that is exploitable by malicious input data.
[27]	Monitor the flow of un-trusted data. (preemptive)	Halts the program.	None	Any vulnerability that is exploitable by malicious input data.
[30]	Monitor the argument of SQL-related functions. (preemptive)	Ignores the request. (roll-forward)	None	SQL injection
[31]	Monitor the argument of SQL-related functions. (preemptive)	Ignores the request. (roll-forward)	None	SQL injection
[32]	Monitor the format argument of printing functions. (preemptive)	Halts the program.	None	Format string
[28]	Monitor the flow of un-trusted data. (preemptive)	Invokes exception handlers. (roll-forward)	None	SQL injection Buffer overflow Format string
[36]	Detect unauthorized changes of return addresses. (preemptive)	Recovers the stack. (compensation)	None	Buffer overflow
[29]	Monitor the flow of un-trusted data. (preemptive)	Ignores the request. (roll-forward)	None	DOM-based XSS

TABLE I: Examples of vulnerability tolerance methods.

be sound, the analyzer produces conservative results that are weaker than the actual ones and may not be very useful [43]. In fact, static analysis is appropriate in proving the absence of a specific vulnerability. Usually static analyzers create many false alarms, hence they cannot be very useful in proving the existence of a specific vulnerability. Static analysis may be performed on the program or on the behavior model of the program [12]. Thus, static analysis methods are divided into two main classes: program-based and model-based methods.

a) Program-based methods

As figure 1 shows, these methods are classified into seven subclasses. Each class is explained as follows.

Pattern Matching

The most basic static analysis method is pattern matching. A pattern matcher considers the program as a text file. It may not even distinguish between the code and the comments. The pattern matcher searches for vulnerable functions or patterns in the text of the program code. Thus, this method can be implemented using any pattern matching utility, such as grep. Such a tool needs a database of the vulnerability patterns. As an example, Flawfinder [44] scans C/C++ programs to detect buffer overflow or format string in them. This tool ignores the text inside the comments and strings. However, it does not recognize the type of function parameters and control flow or data flow of the program. This lack of knowledge results in many false decisions. Thus, it makes many false positive and false negative alarms.

Lexical analysis

In this method, source code of the program is tokenized in order to recognize the variables and function arguments. Thus, the results of a lexical analyzer can be more accurate than the results of a pattern matcher. As an example, the tool ITS4 applies lexical analysis to detect buffer overflow, format string and race condition vulnerabilities in C or C++ programs [45]. ITS4 scans the source code statically and breaks it into series of lexical tokens. These tokens are compared with the token streams that are defined in a vulnerability database. The vulnerability database contains several handlers for well-known vulnerable functions in C/C++.

Parsing

In this method, source code of the program is parsed and represented in Abstract Syntax Trees (AST). The ASTs are then used to analyze the program syntactically and semantically. For example, Lint uses this method to detect vulnerabilities in programs written in C [46]. As another example, in [47] the ASTs of the source code are extracted and compared to the ASTs of different vulnerable codes. The main idea in [47] is that different vulnerabilities in software may be related to the same flawed programming pattern. Thus, the suggested method uses the ASTs of known vulnerable codes and searches for similar patterns in the target program. When a similar pattern is found in the program, it may reveal an unknown vulnerability.

Data flow and taint analysis

In this method, the flow of data among the instructions is analyzed to determine possible values that a variable holds during the run time. Two well-known program representations are used in this method: control flow and data flow graphs. In a control flow graph, each node represents an instruction and a directed edge between two nodes represents their execution dependency. In other words, a directed edge from node A to node B means that node A is executed after the execution of node *B*. The data flow graph is a modified version of the control flow graph in which new directed edges are added to show the data dependency among the instructions. In this graph, the node *A* has data dependency to the node *B*, if the data that is used in node *A* is already manipulated in node *B*. As an example, the vulnerability detection method presented in [48] extracts the control flow and data flow graphs from the source code. It then compares extracted graphs with some patterns of known vulnerabilities. In this method, known vulnerabilities are specified as simple patterns of vulnerable functions or more complex flow-based rules.

A subclass of data flow analysis is called taint analysis. A taint analyzer only tracks the flow of data that come from un-trusted resources. The un-trusted resources include the network protocols, keyboard, touchpad, webcam, files, etc. Since most of the vulnerabilities are exploited by un-trusted input data, this method pays attention to the flow of un-trusted input data in the program. If such data reach sensitive statements in the program, a vulnerability may be reported by the taint analyzer. The sensitive statements, called sinks, are defined according to the specified vulnerabilities. For example, the functions that execute SQL queries are usually defined as the sinks for SQL injection vulnerability. The propagation of tainted data among the instructions is determined based on some predefined rules. For example, if the data in a tainted variable is assigned to an un-tainted variable, the un-tainted variable will get tainted too.

Taint analysis is used in many of the proposed vulnerability detection solutions, e.g. [49], [50], [51], [52] and [53], to detect various vulnerability classes. Since this method focuses on the flow of tainted data, it does not consider the execution paths in the program that are not affected by malicious data. This feature reduces the time of analysis and number of produced false positives. However, there are vulnerability classes that cannot be specified in such a source-sink structure, e.g. logic vulnerabilities. Although an attacker exploits logic vulnerabilities with malicious data, the sinks cannot be easily specified for this class of vulnerability. For example, the sinks for SQL injection vulnerability are the query execution statements. But a sink for logic vulnerabilities may be any statement that manipulates the input data.

Annotation-based methods

Annotation is a comment that the programmer makes in the code about the desired behavior of a function or an instruction. It may be defined as a set of pre- and post-conditions or as simple pre-execution conditions. An annotation-based analysis algorithm reads the annotations, analyzes the code statically and verifies if the conditions are met in the program. There are plenty of annotation languages presented so far, such as SPLINT [54], MECA [55], Sparse [56], SAL [57] and aComment [58].

Since there is a huge number of statements and functions in the programs, manual annotation is usually very time consuming and fault prone [58]. There are annotation languages that provide some facilities to annotate the program more easily, such as MECA [55] and aComment [58]. Among them, aComment is designed to help in detecting concurrency faults in the operating systems and allows the programmers to define the pre- and post-conditions that are related to the interrupts in each function. It also infers the annotation of some functions automatically to reduce the programmers' workload. In this way, the programmers are not supposed to annotate all the functions manually.

Although some of these languages help in reducing the required time and effort for annotating the programs, they usually have a different syntax and semantics than the applied programming languages. Therefore, the programmers and verifiers have to make extra efforts to learn another language in order to use this method. Also, the programmers should be familiar with the security requirements of the programs and the vulnerability classes to annotate the program appropriately. Therefore, the success of this method depends on the programmers' knowledge of software security. Moreover, this method is not helpful in analyzing the COTS[1] software and third party components since their source code is not available.

Constraint analysis

In this method, the program is analyzed statically and some constraints are calculated for specific objects in it. The constraints are defined according to specific vulnerabilities and are solved to verify if the program suffers from those vulnerabilities. Constraint analysis was first proposed by Wagner et al. in [50]. The resulted tool, called BOON, considers the strings in a C program as an abstract data type. There are also predefined functions that manipulate this data type, such as $strcpy()$, $strcat()$, etc. BOON summarizes the state of each string by two integer values: the allocated size for the string and its current length. For each string in the buffer, it analyzes the string manipulating statements in the program to verify if the length of the string exceeds its allocated size. If such condition is inferred, the program might contains buffer overflow vulnerabilities.

It is important to note that the constraints are determined by the analyzer in this method, not by the programmer. This makes the constraint analysis method different from the annotation-based analysis method. Moreover, constraint analysis does not increase the programmer's workload since generation of the constraints is performed automatically and does not involve the programmer. Of course, it cannot profit the programmers' knowledge of the code to do a more efficient analysis.

Theorem proving

In this method, the software and its specification are expressed as some formulas of logics or algebraic systems. Also, the security requirements of software are expressed as some theorems. Proving these theorems demonstrates the satisfaction of the security requirements. Otherwise, there is a fault (vulnerability) in the program. As an example, in [59] the source code of target program is statically analyzed and some first-order formulas are generated that assert the absence of certain faults and vulnerabilities, such as out-of-bounds array access. If the generated asserts are proved, the program does not contain such faults and vulnerabilities.

Although the results of analysis are accurate in the theorem proving methods, they demand expertise and enough experience. In fact, theorem proving is difficult to be achieved automatically and requires high-quality staff to apply this method, which is very time-consuming. So it is generally used to verify correct design rather than the actual code [60].

[1] Commercial off-the-shelf

Analysis Method	Description	Advantages	shortcomings	Examples
Pattern Matching	Considers the program as a text file and searches for vulnerability patterns in the text.	Simple, fast.	Does not have any idea about the types of function parameters and control or data flow of the program and so generates many false alarms.	[44].
Lexical Analysis	Tokenizes the code to recognize variables and function arguments.	Variables and function arguments are recognized. More accurate than the pattern matching method.	Lack of knowledge about the syntax and semantics of the code causes false alarms. Requires the high level source code.	[45].
Parsing	Parses the code and represents it in Abstract Syntax Trees (AST) to be analyzed syntactically and semantically.	Understands the code syntactically and semantically, less false alarms in comparison with the above two methods.	Requires the high level source code.	[47], [46], [30], [31], [66].
Annotation-based methods	Comments that the programmer makes about the desired behavior of the code. The code is then analyzed statically to verify if the conditions are met.	Profits the programmers' knowledge to do a focused analysis.	The programmer must learn an additional language to do the annotation.	[54], [55], [56], [57], [58].
Theorem proving	The security requirements of software are expressed as some theorems. Proving these theorems, demonstrates the satisfaction of the security requirements or existence of vulnerabilities.	Accuracy.	Difficult to be achieved automatically and requires high-quality staff to apply this method	[59].
Data flow analysis (Taint analysis)	Tracks the flow of the data that comes from un-trusted resources and warns if the data reaches sensitive program points.	Reduces the analysis time and number of false positives by not considering the execution paths in the program that are not affected by un-trusted data.	Cannot detect vulnerabilities that are not defined specifically in a source-sink structure.	[51], [52], [53], [67], [50], [68], [69], [70].
Constraint analysis	Analyzes the program, associates constraints with some objects in the code and solves them to verify if the program is vulnerable.	Constraints are generated automatically and do not increase the programmer's workload.	Does not profit the programmer's knowledge of the code (in comparison with the annotation method).	[50], [51], [52], [53].
Model checking	Models the program and then checks the model to verify if it satisfies specified requirements.	Only the modeling and requirement specification is performed manually by the human analyzer, rest of the analysis is done automatically.	Modeling the program and specifying its security requirements- if done manually- is time consuming and fault prone. State-explosion problem when the number of program states is large.	[61], [62], [63], [71], [64].

TABLE II: Static analysis methods: a comparison.

b) Model-based methods (model checking)

In this method, the program is modeled and then analyzed to verify if it complies with its specifications, e.g. [61], [62], [63] and [64]. If a specific requirement is not satisfied in the software, this method provides some counter examples. Model checking helps the human analyzers by automating a noticeable part of the analysis. Although modeling and specifying the requirements may be done manually, analyzing all possible states of the program and verifying the requirements are done automatically. This is a great help in analyzing large programs. A well-known example of using this method for detecting vulnerabilities is MOPS [63]. Using MOPS, the program is modeled as a push-down automaton[2]. Also, the requirements are defined through safety properties. A safety property is represented as a finite state automaton. It defines the ordering constraints on security related operations. After modeling the program and defining its constraints, MOPS searches exhaustively through possible program states to check if a reachable state violates the safety properties.

Abstracting the program in a model is a challenging task in this method. The model should be expressive enough to have a precise analysis. Thus, some model checking methods also help the analyzer to model the target program. For example, MOPS uses the control flow of the program to build its automata. Also, in [64] the GCC compiler is used to automatically model and verify the programs that are written in any language supported by this compiler, i.e. C, C++, Java, etc. This is done by employing an intermediate language of GCC, called GIMPIL, that is common to all the supported languages. The model is extracted from the intermediate representation of the program and is checked against the defined specification by the use of Moped. Moped is a model checking tool for push down systems[3] [65].

There are, however, some shortcomings in the model checking method. Although the modeling phase is performed automatically in some model checking methods, the analyzer should manually specify the security requirements in this method. This is again time consuming and may cause errors in the results. Also, the method suffers from state-explosion problem for large programs.

Table II summarizes the reviewed static analysis methods. Note that all these methods inherit the general advantages and shortcomings of static analysis.

[2] A push down automaton is a type of computational model. It is similar to NFAs except that it uses an additional component called a stack. In this model, state transitions are chosen based on three components; input signal, current state and what is at the top of the stack. Thus, the stack plays the role of an additional memory for it.

[3] Push down systems are similar to push down automata except that they don't have any input alphabet. They consist of a finite set of states, a finite stack alphabet and a finite set of transition rules. Therefore, they are not used as language acceptors and are mainly used to model programs with recursion.

2) *Dynamic Analysis*

By executing the program with actual data, dynamic analysis studies the exact run-time behavior of the program. Dynamic analysis can be as fast as the execution of the program, whereas static analysis generally requires more computation time to obtain accurate results [43]. The main challenge in dynamic analysis methods is executing all the possible execution paths in the program and activating all vulnerabilities in those paths. In fact, acquiring an appropriate test data set, that make the program behave more diversely, is an issue in these methods. The most important shortcoming of dynamic analysis methods is that they are unable to guarantee the analysis of all feasible execution paths. Therefore, the dynamic analysis is not sound and is mostly used to prove the existence of specific vulnerabilities in the programs. The dynamic methods are classified into two main classes in [12]: methods that use symbolic input values and methods that use actual (concrete) input values to test the program. Based on the recent advances in dynamic analysis methods, we classify these methods in three classes based on the type of applied input values: concrete execution, symbolic execution and concolic (concrete + symbolic) execution methods. The following subsections describe each class in more details.

a) Concrete execution

In this method, the program is executed with actual data and its behavior is analyzed to detect vulnerabilities. There are four dynamic analysis methods that use actual data to execute the program during the analysis: fault injection, mutation-based analysis, dynamic taint analysis and dynamic model checking.

Fault injection

In this method, the external faults are injected to the program to examine its behavior. According to our definition in section II, the external faults abuse the internal faults and cause unauthorized behaviors in the program. In other words, internal faults are activated by the external fault and are propagated to reach the program boundaries. Therefore, inability to handle external faults may reveal a vulnerability in the program.

The external faults may be injected by corrupting input data to verify if the program is able to handle them. Most of the black-box vulnerability scanners corrupt input data and analyze the reaction of the program, such as [72] and [73]. The black-box scanners have access to the inputs and outputs of the program. They might also have very little knowledge about the program internal structure [74]. They usually create the corrupted data based on known attack patterns to study if the program can resist these attacks or suffers from the relevant vulnerabilities. Another group of dynamic vulnerability detectors that inject corrupted input data to the programs are *fuzzers*. Takanen et al. introduced fuzzing for detecting vulnerabilities for the first time. They suggested injecting unexpected random input data to the program and studying its behavior [74]. The difference between fuzzers and black-box vulnerability scanners is that fuzzers don't corrupt input data exactly based on a list of attack patterns. In fact, they generate numerous random faulty data hoping that some data make the program crash. The main advantages of this method were simplicity and independence from the analyzed program. Thus, the method could be used easily to detect vulnerabilities in different programs. However, fuzzers were not intelligent enough to corrupt input data effectively and cover most of the execution paths. In order to have better program coverage, new fuzzers focus on producing well-formed corrupted data [75], satisfying data validation checks in the program like checksums [76], being aware of the state of the program during the fuzzing [77] and producing consistent input data with the path conditions to make the program execute all the branches [78], [79], [80], [81]. All these enhancements made fuzzers play an effective role in detecting vulnerabilities during the recent years [82]. Injecting faults into the program can be done randomly or intelligently. By the word random, we mean that faulty data are generated semi-randomly based on predefined patterns. For example, in order to detect buffer overflow, random input data with different lengths are generated. Here the predefined pattern determines the length of input data and the other properties are set randomly. Takanen et al. consider random fuzzers as the ones that make small random changes into the valid data. For example, a FTP fuzzer may randomly add valid/invalid commands to the test data or chose the arguments of the commands randomly [74]. Random fuzzers sometimes use evolutionary algorithms to guide random choices and extend the program coverage, e.g. [83], [84]. Random corruption of data is simple and independent from the logic and structure of the programs. Moreover, randomness helps to reveal a wide range of behaviors of the programs while the designed test-cases by the human analyzer may not. This is because the designed test-cases are prepared by a human analyzer who may not think of all possible behaviors of the program.

Corrupting the data intelligently is performed based on a previous analysis of the program. Although it requires more analysis efforts, it helps in extending the program coverage. For example, imagine a program that compares one of the input values with an integer value and exits if they are not equal. Using the random method, the possibility of passing this constraint is one out of 2^{32}. By analyzing the code before injecting faulty data, the analyzer is able to extract the constraint and generate the data in a way that complies with the constraint. This helps the intelligent corruption method to have more reliable program coverage [74].

Mutation-based analysis

As mentioned before, acquiring appropriate test data is an issue in dynamic analysis. When the program behaves normally during the test process, it means that either there is no vulnerability in the program or the test data don't reveal the vulnerabilities in the program. In the latter case, the data set is not diverse enough to activate the vulnerabilities. Mutation is a method that is concerned with enhancing the data set during the dynamic analysis. In this method, specific vulnerabilities are injected into the program code intentionally. If the current data set does not detect the injected vulnerability, it will not detect similar vulnerabilities in the original version of the program. Thus, the analyzer augments the data set so that it can detect the vulnerability. A version of a program in which a specific vulnerability is created, is called a mutant. For example, in a mutant the function $strncpy()$ is replaced with $strcpy()$ to make it buffer overflow vulnerable. A good test data set distinguishes the mutants from the original version of the program and kills them. If no test-case kills the mutants, the data set must be augmented [85].

This method is effective in detecting software vulnerabilities [85], though it requires considerable amount of time and effort. If the changed statements in a mutant are executed by the test data, the mutant would be effective. Otherwise, the result of analysis does not reveal the difference between the mutant and the original version of the program. Therefore, some computations are required to generate appropriate test-cases that make the program execute the intended path which contains the vulnerability.

Also, automatic creation of mutants for complex vulnerabilities is a challenge. As an example, the $strncpy()$ functions are automatically changed to $strcpy()$ for creating mutants to detect buffer overflow in [85]. There are, however, more complicated buffer overflow scenarios like copying an array in a loop that causes overflow. Moreover, creating mutants for logic vulnerabilities requires a deep understanding of the logic of the program. Thus, automatic generation of mutants may not be feasible.

Dynamic model checking

This method, which is also called execution-based model checking [86], [87], is a model checking method that executes the program exhaustively and checks if it satisfies the specifications. For example, the tools VeriSoft [88], JavaPathFinder [89], CMC [90], Bogor [91] and DART [92] apply this method in their analysis. Random execution in dynamic model checking is mostly the result of two factors: program inputs and scheduling choices of a scheduler [87]. For each random input and schedule choice, the resulted behavior of the program is analyzed by monitoring the process and its environment, e.g. registers and the stack. Here, each state consists of the entire machine state. When the execution reaches a state, in which the specification is compromised, the related input value and schedule choice are presented as a counter-example.

An advantage of dynamic model checking is that by executing the program, the machine handles the semantics of the instructions. In other words, there is no need to formally represent the semantics of the programming language and the machine instructions [87]. However, there is a time-state-soundness tradeoff in this method. Since the states represent the entire machine state, they contain many details and require more storage space. Thus, storing all the states might be infeasible for large programs. At the same time, exploring the states without a history of visited ones may cause visiting similar states again and again. When no state is recorded, the model checker spends too much time to make sure it has traversed all possible states. Storing the states reduces the verification time by making sure that no state is revisited. Yet, it requires too much space [87].

Dynamic taint analysis

This method is similar to static taint analysis as it tracks the flow of information from un-trusted sources to the sinks. However, it tracks the flow of tainted data during the execution of the program, some examples are [93], [94], [95] and [96]. Schwartz et al. describe this method precisely in [93]. They introduce a language, named SIMPIL, that formally defines the algorithms of dynamic taint analysis. Before the execution, all the variables are considered untainted. While executing the program, variables may get tainted according to a predefined policy. This policy defines how the taint data propagate from a variable to other variables. For example, when tainted data are used in an argument of an arithmetic operation, the policy defines that the result of this operation should be considered tainted. If a tainted value reaches a sink, the analyzer reports a vulnerability.

The basic taint analysis methods limit taint propagation to the direct assignments. This might make the results of the analysis inaccurate [97]. Sarwar et al. present some scenarios in [97] to show how basic taint analysis can be ineffective. An example scenario is that the tainted data are used in a conditional statement (without any direct assignment to other variables) and affect on the control flow of the program. Also, tainted data might be used to define the number of an iterative action or as the index of an un-tainted array. The taint analysis method should pay attention to these indirect effects of the tainted data in calculating the taint propagation. Considering such effects is not always easy. For example, the tainted variable might cause information leakage through a side channel. To detect such vulnerability, the analyzer should taint a large amount of variables that results in many false alarms [97].

Table III summarizes and compares the advantages and disadvantages of the concrete execution methods. Each method inherits the advantages and shortcomings of dynamic analysis.

b) Symbolic execution

Using the symbolic execution method, the program is executed with symbolic input values instead of concrete data values [98], [99]. Thus, the values of program variables are represented as symbolic expressions over the symbolic input. During the symbolic execution, the state of the program and the conditions of the current path are calculated symbolically. The path conditions are updated any time a branch instruction is executed. At the end of an executed path, the path conditions are solved using a constraint solver. There are various constraint solvers presented by now, such as STP [100] and Z3 [101] that solve the constraints on binary vectors and Hampi [102] and S3 [103] that solve the constraints on string variables. If the constraint solver solves the path conditions, it generates some concrete input data that are used to execute the intended path in the program.

There are several challenges with the symbolic execution method. For example path explosion, the overhead of constraint solving for complicated paths, non-determinism of concurrent programs and the trade-off between precision and scalability of modeling the memory are some of the challenges in applying symbolic execution [104]. Cadar and Sen present the challenges of this method and mention some solutions for them [104].

To overcome these challenges, a solution is combining symbolic execution with concrete execution. The result is a new method that is called concolic execution. This method is described in the next section.

c) Concolic execution

A problem with pure symbolic execution is that the constraints of complex loops and recursive functions may get very complicated and cannot be resolved in an acceptable time [105]. Concrete execution applies real data to execute the program. There is a little chance to traverse all the feasible paths in this method. Using the combined method, concolic + symbolic execution, the concrete data is used to simplify the complex constraints that are generated by the symbolic execution. This method was first presented by Godefroid et al. in [92]. Concolic execution is performed by changing some symbols in the complex constraints into the concrete values. This helps to achieve better program coverage with much less computation overhead.

Concolic execution is used in many of the recent fuzzers to extend their knowledge about the program, such as KLEE

Analysis Method	Description	Advantages	shortcomings	Examples
Fault injection	Faults are generated semi-randomly. (random corruption) Fault injection is based on some previous analysis of the program. (intelligent corruption)	Simplicity and independence in random corruption of inputs. (random corruption) More reliable code coverage, less false negatives. (intelligent corruption)	Cannot detect logic vulnerability. Less reliable code coverage. (random corruption) More effort is required for testing each single program. (intelligent corruption)	[83],[84], [75], [76], [78],[79], [80].
Mutation-based Analysis	Injects vulnerability into the program code. If the current data set does not reflect the injected vulnerability, it would not detect similar vulnerabilities in the original version of the program.	Reduces false negatives by enriching test data.	Expensive in time and computation. Automatic mutation of complicated vulnerabilities is a challenge.	[85].
Dynamic taint analysis and sanitization	Tracks the flow of information from input sources to the sinks during the run-time.	Reduces the analysis time and number of false positives by not considering the paths in the program that are not affected by malicious data.	Cannot detect vulnerabilities that are not defined in specific source-sink structure.	[93],[96].
Dynamic Model checking	A model checking method in which the program is executed with concrete input values exhaustively.	No need to formally represent the semantics of the programming languages and machine instructions.	Time-state-soundness trade-off.	[88],[89], [90],[91], [92].

TABLE III: Concrete execution methods: a comparison

[78], EXE [80], Simfuzz [75], CUTE [106], SAGE [79], Taintscope [105] and [107]. For example, CUTE combines symbolic execution with concrete execution to create input data traverse deeper paths in the program. It first executes the program with concrete input data. During the execution, it calculates symbolically the constraints of the executed path. The calculated constraints are then negated one by one, from the last to the first. After each negation, the resulted constraints are queried from a constraint solver. If the constraint solver solves the new constraints, the result is used to generate new test data that traverse other execution paths in the program.

Concolic execution is also used in other dynamic vulnerability detection methods. For example in [108] a dynamic model checking method is applied that uses concolic execution for state-space exploration of the analyzed application. In [108], concolic execution helps to model the application as a finite-state automata and to guide further state-space exploration.

3) *Hybrid analysis*

The previous sections described static and dynamic analysis methods and their advantages and shortcomings. The idea of combining static and dynamic analysis was first proposed by Ernst in [43]. He suggested that hybrid analysis can combine the static and dynamic analysis methods to generate a new analysis method that profits a great amount of soundness and accuracy advantages of each method with little sacrifices. From then, many researchers have combined these methods, in different manners, to make up for each other's shortcomings.

For example, Monga et al. combine static and dynamic analysis to detect XSS and SQL injection vulnerabilities in PHP applications in [109]. The suggested method first analyzes the code statically and extracts the control flow graph of the functions in it. These graphs are then connected together to obtain an inter-procedural control flow graph (iCFG). The iCFG is analyzed to extract the possible paths from the tainted sources to the sinks in it. For each sink, backward slicing is used to detect the statements that affect the tainted argument. These statements are monitored at run time. When a tainted value is used in a sink, the monitoring procedure passes it to an oracle to verify if it can exploit a vulnerability. The oracle have a database of well-known attack patterns that are used to exploit different vulnerabilities. For example, the implemented oracle for mysql_query() performs a limited syntactically analysis on the SQL queries and searches for the tainted characters in unsafe positions. In this method, the sanitizing procedures are assumed to be perfect.

Monitoring and static analysis methods are also combined in [110] to detect SQL injection errors and prevent the successful attacks. In the static analysis phase the hotspots, that are statements in the program that execute a SQL query, are identified. Also the control flow of the program is extracted. Then, the query strings in the hotspots are parsed. Considering the control flow of the program, a FSA for each hotspot is created to model the legitimate queries. During the monitoring phase, the queries are checked against the relative FSA to prevent execution of malicious queries.

As the last example, hybrid analysis is used in [111] to detect logic vulnerabilities in web applications. The logic vulnerabilities are usually related to the intended functionality of an application. Thus, there is no general specification for them that can be used in different applications. For example, consider an online store that allows the users to use coupons for having discount on specific items. It has a policy which determines that each coupon should be used only once. A logic vulnerability, however, allows the users to reuse a coupon and reduce the cost to zero. Since logic vulnerabilities are created based on the functionality of the application, the vulnerability detection method requires the specification of the program. The proposed method in [111] consists of two steps. First, the web application is executed with normal input data. The executed traces are then analyzed to infer the specification of the application. This is based on the intuition that normal behavior of the program reflects the properties that are intended by the programmer. In fact, because the specification of the program is not always available, this method uses dynamic analysis to obtain it. The inferred specification is presented in the form of likely invariants. In the second step, model checking is used to analyze the web application based on the inferred specification.

D. Vulnerability forecasting

Generally, fault forecasting is used to predict the quality or quantity of the faults that are left in the system and will be activated in the future. It is mainly concerned with estimating the current reliability of the system and predicting its future reliability. This prediction may be qualitative or quantitative (usually probabilistic). The qualitative forecasting identifies and ranks the future failure modes. Also, the event combinations that lead to the failures are identified.

The quantitative forecasting is performed by modeling or operational testing. These methods are complementary, since the results of operational tests are usually used to model the system more accurately. Software Reliability Growth Models (SRGM) are used generally for fault forecasting. In fact, SRGMs model the testing process [112]. In most of these models, the rate of fault detection gradually reduces and the cumulative number of faults eventually approaches a fixed value. These models help to predict the number of left faults in the software and determine when the software is ready to be released. They are also used to estimate the required efforts for future maintenance.

There are probabilistic models for predicting the rate of vulnerability detection, named Vulnerability Detection Models (VDM). Alhazmi and Malaiya proposed a specific model, named AML[4], for vulnerability detection in [113]. In this model the rate of vulnerability detection depends on two factors: one of these factors reduces as the number of remaining undetected vulnerabilities declines. The second factor increases with the time. In this way, the rate of vulnerability detection is modeled in a S-shaped form. In fact, AML is created based on the observation that the detectors[5] pay little attention to the newly published software. Gradually people become familiar with the software and the detectors pay more attention to it. Thus, the rate of vulnerability detection increases by time and peaks at some period. By the introduction of newer versions of the program, the detectors' interest becomes lower and the rate of vulnerability detection decreases. Alhazmi and Malaiya examined the applicability of this model to various operating systems in [113] and [114]. The results demonstrated that AML fits the data of several operating systems.

All the mentioned models are time-based. It means that they determine the detection rate based on the calendar time. An effort-based model, named AME [2], is proposed by Alhazmi and Malaiya in [113]. They believe that time-based models do not consider the changes that occur in the environment during the lifetime of the system. Thus, they consider the number of installations as an important environmental factor that affects the rate of vulnerability detection. It is based on the observation that the detectors are more interested in the software that is installed in many computers. Therefore, the rate of vulnerability detection is modeled in AME based on the number of installations. Sung-whan Woo et al. explore the applicability of AML and AME to some HTTP servers, i.e., IIS and Apache. The results indicate that these models are applicable to the HTTP servers in addition to the operating systems [112].

A problem with the studied VDMs is that they are parametric models that should be fitted to real vulnerability data [115]. To model the vulnerability detection rate in a specific application, a large amount of historical vulnerability data is required. Therefore, it is necessary that many of the vulnerabilities be discovered already. Hence, these models cannot be applied to predict the detection rate for newly released software. Also, it is shown in [116] that the precision of VDMs depend on the number of known vulnerabilities. The precision of the VDMs are usually very low at the early stages in the lifecycle of the program. It seems that the problem is because the models don't consider the features of each application in their predictions. Thus, they need a history of detected vulnerabilities to estimate the security level of the program. There are many features in each application and its environment that affect the rate of vulnerability detection. Rahimi and Zargham present a VDM in [115] based on two effective factors: code complexity and code quality. The code complexity is defined based on the cyclomatic complexity. Also, the code quality determines its compliance with secure coding practices. They believe that more vulnerabilities are detected in the applications with lower code quality. Also, the possibility of detecting vulnerabilities is less in the applications with complicated codes. Thus, the source code of the application is statically analyzed to compute the two factors. The computed data are then used to model the vulnerability detection rate. Since this model does not need a database of detected vulnerabilities, it can be used for newly released applications. The authors analyze four applications to study the impact of these factors on the vulnerability detection trend. The analysis results show that the proposed method can predict vulnerabilities even in early stages of the application's lifecycle.

Of course, this method does not consider many of the effective factors on the detection trend. For example, it only calculates the cyclomatic complexity to estimate the code complexity. There are other complexity metrics that can be considered in addition to this one. The environmental parameters can also be considered for a good prediction. As an example, even the seasonal changes affect the rate of vulnerability discovery. It is shown in [117] that more vulnerabilities are reported during the mid-end and year-end months. Also, the presented method in [115] is based on analyzing the source code of the application. So it cannot be helpful when the source code is not available.

There are some qualitative methods for estimating the current security level of the application. For example OWASP ASVS consists of several check lists that helps to determine the security level of a web application [118]. It classifies the check lists in thirteen classes, such as authentication, access control, session management, etc. In each class the check lists are grouped into three security levels. If an application passes all the check lists of a group, it is achieves the respective security level. These methods only estimate the current security level. Based on the current level, it is possible to predict the future failure modes. However, we could not find any qualitative method that predicts and ranks the future security failures based on the current state.

IV. Conclusions

During the past decades various methods have been presented for mitigating software vulnerabilities. A comprehensive classification of the proposed methods helps to achieve a general understanding of this research area. In this paper, we

[4] Alhazmi-Malaiya Logistic model

[5] People that analyze applications in order to detect vulnerabilities in them and report them to the vendors of applications or use them maliciously.

[2] Alhazmi-Malaiya Effort-based model

defined software vulnerability as an internal fault. By considering software vulnerability as a type of fault, we classified the vulnerability mitigation methods based on the general classification of the fault mitigation methods. We extended the general classification of fault mitigation methods, represented it in the context of software vulnerability and added more detailed subclasses into it. We divided vulnerability mitigation methods into four main classes: vulnerability prevention, vulnerability tolerance, vulnerability removal and vulnerability forecasting.
The vulnerability prevention methods attempt to prevent the occurrence of software vulnerability. Software security and the secure coding best practices are examples of these efforts. The question is why, despite the vulnerability prevention efforts, vulnerabilities are still created. Oliveira et al. believe that educating the developers is not enough for preventing the vulnerabilities [119], because security is not an issue for the developers. They believe that the human's memory is limited and can only keep a limited number of mental elements available at a time. The programmers are also supposed to create applications with correct functionality and acceptable performance. Under the time pressure, an ordinary situation in software programming, the programmers usually chose the simplest solutions for developing the software and pay little attention to the security concerns. Oliveira et al. suggest developing assistant tools that remind the educated programmers the security concerns during the development.

Besides educating the programmers, intelligent assistant tools are required to notify the security concerns at specific statements or functions. Thus, in the future we should work on designing and implementing intelligent assistant tools that help the programmers to avoid generating vulnerabilities during the design and implementation of the applications. These tools should be intelligent enough not to bother the developers with many false alarms. They may use the enhanced static analysis methods to analyze the code during the coding phase and warn the programmers at sensitive situations. This will help the programmers to use their security knowledge more effectively in preventing the vulnerabilities.

Vulnerability tolerance methods accept the existence of vulnerabilities in the programs and prevent the active vulnerabilities from making security failures. In this paper, the vulnerability tolerance methods were studied based on three aspects: the applied active vulnerability detection technique, active vulnerability handling technique and vulnerability handling technique.
All the reviewed vulnerability mitigation methods detect active vulnerabilities during the normal execution of the program (concurrently). However, there are active vulnerabilities that overuse system resources and make the resources unavailable to legitimate users after a period of time, such as the memory leakage vulnerability. The security failure as a result of these vulnerabilities can be prevented by checking the system resources periodically. Thus, preemptive error detection can be applied to detect if such vulnerabilities are active.

Most of the vulnerability tolerance methods focus on detecting the active vulnerabilities. However, less attention is paid to handling the (active) vulnerability. In the proposed methods, active vulnerabilities are handled by halting the program, restarting the program or invoking an exception handler. Although these mechanisms limit the negative effects of the active vulnerability, they violate the availability of the application to the legitimate users. Thus, more intelligent vulnerability handling techniques are required for the current vulnerability tolerance methods. A good starting point is inspiring by the current fault tolerance methods. As an example, software diversity is a fault tolerance method that is used to make the programs reliable. In this method, various versions of software with the same specification but different design or implementation- process the same request and the correct result is achieved by voting the results of the different versions. The result of such system is more reliable, since it is less possible that all the versions of software suffer from the same fault and so a request does not cause errors in all versions. This method can be used in software security to tolerate malicious requests. For example, recently software diversity has been used in [120] to tolerate active vulnerabilities in web browsers. In the proposed method, different browsers are used to process the user's requests. Since the browsers are designed and implemented differently, it is less probable that all the applied browsers contain similar vulnerabilities. Thus, malicious data cannot compromise all the browsers. The correct response to the client's request is achieved by voting the responses of the browsers. Also, some protection mechanisms, such as ASLR that protects system against memory corruption vulnerabilities [121], are inspired by the idea of using diversity to make the program unpredictable for the attackers. A new direction for the future research could be adopting the current fault tolerance methods to handle different software vulnerabilities. As there is no vulnerability handling mechanism in the current proposed methods, we should work on designing complete vulnerability tolerance methods that contain appropriate active vulnerability handling and vulnerability handling mechanisms.

Vulnerability removal is performed to detect and remove the vulnerabilities in the software. The focus of most of the current vulnerability removal methods is on verifying the vulnerabilities. In fact, less attention is paid on designing appropriate methods for diagnosis, correction and regression verification of software vulnerabilities. There are some vulnerability diagnosis and correction methods that are used after detecting the exploitation of a vulnerability. But specific methods are required for diagnosis and correction of the vulnerabilities that are detected during the verification of the program. Currently, automatic patching methods analyze an attack and generate software patches based on the pattern of malicious data that are used in the attack. These methods can be modified to automatically generate patches based on the results of analyzing the program and according to the mechanism of detected vulnerabilities.

There are numerous vulnerability detection methods presented by now. Most of the recent vulnerability detection methods tend to combine the previous methods in order to profit their advantages at the same time. For example, the concolic execution method combines the concrete and symbolic execution methods to reduce the complexity of pure symbolic execution and increase the program coverage. As another example, static taint analysis is used with the constraint analysis method to limit the overhead of program analysis and compute the constraints only on the tainted data [51]. Also, the control and data flow of the program are extracted in [122], [62] and [63] to model the program automatically and detect vulnerabilities by performing the model checking. There are more possible combinations that are not applied yet and might be effective in detecting the vulnerabilities more accurately. For example, the annotation can be used in model checking to profit the programmers' knowledge for modeling the program.

Also, most of the vulnerability removal and vulnerability tolerance methods consider a specific vulnerability class based on their own definition of the relevant software vulnerability. Therefore, an imprecise definition of the intended vulnerability would cause inaccurate results in the proposed method. In addition, some methods only consider a limited number of vulnerability classes. To handle new vulnerability classes, the algorithm of these methods has to be changed. Many of the presented methods or tools are not able to detect all of the vulnerability classes [123], [124]. By now, the researchers' focus was mainly on designing more accurate methods.

A new research trend is making the vulnerability detection methods extendable. In this way, an accurate method for detecting a specific vulnerability can also be used to detect other vulnerabilities. To make a vulnerability detection method extendable, we suggest designing vulnerability detection algorithms that are independent from the sought vulnerability classes. Such algorithms are able to detect any specified vulnerabilities in the program. Designing such methods requires a general model for specifying the vulnerabilities that encompasses any vulnerability classes, even the future ones. Based on this model, various vulnerabilities are specified for the detection algorithms to be detected automatically in the programs.

There are a few extendable vulnerability removal methods, such as [125], [126] and [124]. However, these methods are limited to specific programming languages. Also, some of them are not expressive enough to specify any vulnerability classes. For example, in [127] an extendable vulnerability method is presented for detecting the vulnerabilities in web applications that are written in Java. The specification method of [127] does not support some data types, e.g. integer, float and character. Therefore, it is not possible to define certain operations, such as mathematical operations or comparing the characters, in specifying a vulnerability. This is not a limitation for specifying vulnerabilities in object-oriented programs, such as Java. Because they encapsulate these operations in certain methods for each data type. It is, however, a limitation for specifying vulnerabilities in other languages, such as C. For example, it is not possible to specify integer overflow vulnerabilities in C programs with this method.

The vulnerability forecasting methods predict the number of left vulnerabilities in the software and determine when the software is ready to be released. They are also used to estimate the required efforts for the future maintenance. Some vulnerability forecasting methods use the vulnerability detection models to predict the rate of vulnerability detection during the lifecycle of the software. The first vulnerability detection models were in fact software reliability growth models that were applied for predicting the vulnerability detection rate. The next models were designed especially for the vulnerability detection rate. These models consider effective parameters on the detection of vulnerabilities, such as time and the number of installations. Since these models do not consider characteristics of the software in their predictions, they need a history of the detected vulnerabilities to predict the vulnerability detection rate in the future. These models are not accurate especially at the early stages in the lifecycle of programs. New vulnerability detection models consider the characteristics of the software to achieve more accurate predictions. In this paper, we reviewed a vulnerability detection model that is based on two characteristics of the program: cyclomatic complexity of the source code and the level of compliance with secure coding practices. There are, however, other characteristics that affect on the rate of vulnerability detection, such as software support, version of the program, availability of the source code or usage of third-party components. For example, the rate of vulnerability detection decreases in time for a program with effective support that periodically presents patches and resolves problems in the program. Also, it might be more difficult to detect vulnerabilities in the higher versions of a program than in its lower versions. The future models can use other characteristics of software to model the vulnerability prediction rate more accurately. Also, they can combine software characteristics with the effective environmental factors, such as time and number of installations, to generate more accurate models.

The possibility of analyzing the program and the program analysis method become important when the vulnerability detection models consider the characteristics of software. For example, a model may be based on some characteristics in the source code of the program. Thus, it is not possible to use such model when the source code of the program is not available. Also, vulnerability forecasts based on inaccurate software analysis are not reliable. In the future, static and dynamic analysis methods that are proposed for detecting the vulnerabilities can be used to better analyze the current characteristics of a program and predict the future rate of vulnerability detection accurately.

References

[1] W. Jimenez, A. Mammar, A. Cavalli, R. Fourier, Software vulnerabilities, prevention and detection methods: a review, in: Proceeding of the first International Workshop on Security in Model Driven Architecture (SEC-MDA)., 2009.

[2] H. Shahriar, M. Zulkernine, Mitigating program security vulnerabilities: Approaches and challenges, ACM Computing Surveys (CSUR) 44 (3) (2012) 11.

[3] K. Zafar, A. Ali, Static techniques for vulnerability detection, Linkoping University, Sweden.

[4] M. Pistoia, S. Chandra, S. J. Fink, E. Yahav, A survey of static analysis methods for identifying security vulnerabilities in software systems, IBM Systems Journal 46 (2) (2007) 265–288.

[5] B. Liu, L. Shi, Z. Cai, M. Li, Software vulnerability discovery techniques: A survey, in: Proceeding of the Fourth International Conference on Multimedia Information Networking and Security (MINES), IEEE, 2012, pp. 152–156.

[6] M. Bishop, A taxonomy of unix system and network vulnerabilities, Tech. rep., Technical Report CSE-95-10, Department of Computer Science, University of California at Davis (1995).

[7] M. Bishop, D. Bailey, A critical analysis of vulnerability taxonomies, Tech. rep., Technical Report CSE-96-11, Department of Computer Science, University of California at Davis. (1996).

[8] M. Bishop, Computer security: art and science, Addison-Wesley, 2002.

[9] H. R. Shahriari, R. Jalili, M. Bishop, A general framework for categorizing vulnerabilities regarding their impact on security policy, Computers and Security.

[10] I. V. Krsul, Software vulnerability analysis, Ph.D. thesis, Purdue University (1998).

[11] R. C. Seacord, A. D. Householder, A structured approach to classifying security vulnerabilities, Tech. rep., Technical report CMU/SEI-2005-TN-003. Carnegie-mellon univ pittsburgh pa software engineering inst. (2005).

[12] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, Dependable and Secure Computing, IEEE Transactions on 1 (1) (2004) 11–33.

[13] M. F. Ringenburg, D. Grossman, Preventing format-string attacks via automatic and efficient dynamic checking, in: Proceedings of the 12th ACM conference on Computer and communications security, ACM, 2005, pp. 354–363.

[14] G. McGraw, Software security: building security in, Vol. 1, Addison-Wesley Professional, 2006.

[15] M. Howard, How do they do it? a look inside the security development lifecycle at microsoft, MSDN Magazine (2005) 107–114.

[16] N. R. Mead, T. Stehney, Security quality requirements engineering (SQUARE) methodology, Vol. 30, ACM, 2005.

[17] Secure Coding Guidelines, https://msdn.microsoft.com/en-us/library/d55zzx87%28v=vs.90%29.aspx, [Online; accessed 2016-10-14].

[18] R. C. Seacord, Secure Coding in C and C++, Pearson Education, 2005.

[19] F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, D. Svoboda, The CERT Oracle Secure Coding Standard for Java, Addison-Wesley Professional, 2011.

[20] The Shields Project, (2012), http://www.shields-project.eu, [Online; accessed 2016-10-23].

[21] D. Byers, N. Shahmehri, Unified modeling of attacks, vulnerabilities and security activities, in: Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems, ACM, 2010, pp. 36–42.

[22] N. Shahmehri, A. Mammar, E. Montes de Oca, D. Byers, A. Cavalli, S. Ardi, W. Jimenez, An advanced approach for modeling and detecting software vulnerabilities, Information and Software Technology 54 (9) (2012) 997–1013.

[23] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, H. Hinton, Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks., in: Usenix Security, Vol. 98, 1998, pp. 63–78.

[24] T.-c. Chiueh, F.-H. Hsu, Rad: A compile-time solution to buffer overflow attacks, in: Proceeding of the 21st International Conference on Distributed Computing Systems, 2001., IEEE, 2001, pp. 409–417.

[25] B. B. Madan, S. Phoha, K. S. Trivedi, Stackoffence: a technique for defending against buffer overflow attacks, in: Proceeding of International Conference on Information Technology: Coding and Computing. ITCC 2005., Vol. 1, IEEE, 2005, pp. 656–661.

[26] M. Dalton, H. Kannan, C. Kozyrakis, Raksha: a flexible information flow architecture for software security, in: ACM SIGARCH Computer Architecture News, Vol. 35, ACM, 2007, pp. 482–493.

[27] G. E. Suh, J. W. Lee, D. Zhang, S. Devadas, Secure program execution via dynamic information flow tracking, in: ACM SIGOPS Operating Systems Review, Vol. 38, ACM, 2004, pp. 85–96.

[28] J. Clause, W. Li, A. Orso, Dytan: a generic dynamic taint analysis framework, in: Proceedings of the 2007 international symposium on Software testing and analysis, ACM, 2007, pp. 196–206.

[29] B. Stock, S. Lekies, T. Mueller, P. Spiegel, M. Johns, Precise client-side protection against dom-based cross-site scripting, in: Proceedings of the 23rd USENIX security symposium, 2014, pp. 655–670.

[30] S. Manmadhan, T. Manesh, A method of detecting sql injection attack to secure web applications, International Journal of Distributed and Parallel Systems 3 (6) (2012) 1.

[31] Z. Su, G. Wassermann, The essence of command injection attacks in web applications, in: ACM SIGPLAN Notices, Vol. 41, ACM, 2006, pp. 372–382.

[32] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, J. Lokier, Formatguard: Automatic protection from printf format string vulnerabilities., in: USENIX Security Symposium, Vol. 91, Washington, DC, 2001.

[33] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, M. Franz, Multi-variant program execution: Using multi-core systems to defuse buffer-overflow vulnerabilities, in: proceedings of International Conference on Complex, Intelligent and Software Intensive Systems. CISIS 2008., IEEE, 2008, pp. 843–848.

[34] G. S. Kc, A. D. Keromytis, V. Prevelakis, Countering code-injection attacks with instruction-set randomization, in: Proceedings of the 10th ACM conference on Computer and communications security, ACM, 2003, pp. 272–280.

[35] G. Iha, H. Doi, An implementation of the binding mechanism in the web browser for preventing xss attacks: introducing the bind-value headers, in: Availability, Reliability and Security, 2009. ARES'09. International Conference on, IEEE, 2009, pp. 966–971.

[36] S. Gupta, P. Pratap, H. Saran, S. Arun-Kumar, Dynamic code instrumentation to detect and recover from return address corruption, in: Proceedings of the 2006 international workshop on Dynamic systems analysis, ACM, 2006, pp. 65–72.

[37] M. Zhang, H. Yin, Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications, in: Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS 2014), 2014.

[38] W. Cui, M. Peinado, H. J. Wang, M. E. Locasto, Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing, in: Security and Privacy, 2007. SP'07. IEEE Symposium on, IEEE, 2007, pp. 252–266.

[39] T. Wang, C. Song, W. Lee, Diagnosis and emergency patch generation for integer overflow exploits, in: Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2014, pp. 255–275.

[40] A. Smirnov, T.-c. Chiueh, Automatic patch generation for buffer overflow attacks, in: Information Assurance and Security, 2007. IAS 2007. Third International Symposium on, IEEE, 2007, pp. 165–170.

[41] Z. Liang, R. Sekar, D. C. DuVarney, Automatic synthesis of filters to discard buffer overflow attacks: A step towards realizing self-healing systems., in: USENIX Annual Technical Conference, General Track, 2005, pp. 375–378.

[42] J. Xu, P. Ning, C. Kil, Y. Zhai, C. Bookholt, Automatic diagnosis and response to memory corruption vulnerabilities, in: Proceedings of the 12th ACM conference on Computer and communications security, ACM, 2005, pp. 223–234.

[43] M. D. Ernst, Static and dynamic analysis: Synergy and duality, in: WODA 2003: ICSE Workshop on Dynamic Analysis, 2003, pp. 24–27.

[44] D. A. Wheeler, Flawfinder, http://www.dwheeler.com/flawfinder, [Online; accessed 2016-10-23] (2001).

[45] J. Viega, J.-T. Bloch, Y. Kohno, G. McGraw, Its4: A static vulnerability scanner for c and c++ code, in: Proceedings of the 16th Annual Computer Security Applications Conference, ACSAC'00, IEEE, 2000, pp. 257–267.

[46] S. C. Johnson, Lint, a C program checker, Bell Telephone Laboratories, 1977.

[47] F. Yamaguchi, M. Lottmann, K. Rieck, Generalized vulnerability extrapolation using abstract syntax trees, in: Proceedings of the 28th Annual Computer Security Applications Conference, ACM, 2012, pp. 359–368.

[48] H. Kim, T.-H. Choi, S.-C. Jung, H.-C. Kim, O. Lee, K.-G. Doh, Applying dataflow analysis to detecting software vulnerability, in: Proceeding of the 10th International Conference on Advanced Communication Technology. ICACT 2008., Vol. 1, IEEE, 2008, pp. 255–258.

[49] N. Jovanovic, C. Kruegel, E. Kirda, Pixy: A static analysis tool for detecting web application vulnerabilities, in: Proceeding of the 2006 IEEE Symposium on Security and Privacy, IEEE, 2006, pp. 6–pp.

[50] D. Wagner, J. S. Foster, E. A. Brewer, A. Aiken, A first step towards automated detection of buffer overrun vulnerabilities., in: NDSS, 2000, pp. 2000–02.

[51] V. Ganapathy, S. Jha, D. Chandler, D. Melski, D. Vitek, Buffer overrun detection using linear programming and static analysis, in: Proceedings of the 10th ACM conference on Computer and communications security, ACM, 2003, pp. 345–354.

[52] Y. Xia, J. Luo, M. Zhang, Detecting memory access errors with flow-sensitive conditional range analysis, in: Embedded Software and Systems, Springer, 2005, pp. 320–331.

[53] F. Yu, T. Bultan, O. H. Ibarra, Symbolic string verification: Combining string analysis and size analysis, in: Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2009, pp. 322–336.

[54] D. Evans, D. Larochelle, Improving security using extensible lightweight static analysis, software, IEEE 19 (1) (2002) 42–51.

[55] J. Yang, T. Kremenek, Y. Xie, D. Engler, Meca: an extensible, expressive system and language for statically checking security prop-

erties, in: Proceedings of the 10th ACM conference on Computer and communications security, ACM, 2003, pp. 321–334.

[56] Y. Tsuruoka, J. Tsujii, S. Ananiadou, Accelerating the annotation of sparse named entities by dynamic sentence selection, BMC bioinformatics 9 (Suppl 11) (2008) S8.

[57] M. Howard, A brief introduction to the standard annotation language (SAL), http://blogs.msdn.com/b/michael_howard/archive/2006/05/19/602077.aspx, [Online; accessed 2016-10-22] (2006).

[58] L. Tan, Y. Zhou, Y. Padioleau, acomment: mining annotations from comments and code to detect interrupt related concurrency bugs, in: Proceedings of the 33rd international conference on software engineering, ACM, 2011, pp. 11–20.

[59] D. Detlefs, G. Nelson, J. B. Saxe, Simplify: a theorem prover for program checking, Journal of the ACM (JACM) 52 (3) (2005) 365–473.

[60] G. Tian-yang, S. Yin-sheng, F. You-yuan, Research on software security testing, World Academy of science, engineering and Technology 70 (2010) 647–651.

[61] S. Chaki, S. Hissam, Precise buffer overflow detection via model checking (2005).

[62] W.-S. Rödiger, Merging static analysis and model checking for improved security vulnerability detection, Ph.D. thesis, Master thesis, Dept. of Com. Sc. Augsburg University (2011).

[63] H. Chen, D. Wagner, Mops: an infrastructure for examining security properties of software, in: Proceedings of the 9th ACM conference on Computer and communications security, ACM, 2002, pp. 235–244.

[64] R. Hadjidj, X. Yang, S. Tlili, M. Debbabi, Model-checking for software vulnerabilities detection with multi-language support, in: Proceeding of the sixth Annual Conference on Privacy, Security and Trust, PST'08., IEEE, 2008, pp. 133–142.

[65] J. Esparza, D. Hansel, P. Rossmanith, S. Schwoon, Efficient algorithms for model checking pushdown systems, in: Computer Aided Verification, Springer, 2000, pp. 232–247.

[66] J. Ren, B. Cai, H. HE, C. HU, A method for detecting software vulnerabilities based on clustering and model analyzing, Journal of Computational Information Systems 7 (4) (2011) 1065–1073.

[67] V. B. Livshits, M. S. Lam, Finding security vulnerabilities in java applications with static analysis, in: Proceedings of the 14th conference on USENIX Security Symposium, Vol. 14, 2005.

[68] G. Wassermann, Z. Su, Static detection of cross-site scripting vulnerabilities, in: Proceeding of the ACM/IEEE 30th International Conference on Software Engineering. ICSE'08., IEEE, 2008, pp. 171–180.

[69] J. Newsome, D. Song, Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, in: Proceedings of the 12th Network and Distributed System Security Symposium (NDSS05), 2005.

[70] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, M. Veanes, Fast and precise sanitizer analysis with bek, in: Proceedings of the 20th USENIX conference on Security, USENIX Association, 2011, pp. 1–1.

[71] L. Wang, Q. Zhang, P. Zhao, Automated detection of code vulnerabilities based on program analysis and model checking, in: Proceeding of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE, 2008, pp. 165–173.

[72] G. Pellegrino, D. Balzarotti, Toward black-box detection of logic flaws in web applications, in: Proceedings of the Network and Distributed System Security (NDSS) Symposium, 2014.

[73] S. Kals, E. Kirda, C. Kruegel, N. Jovanovic, Secubat: a web vulnerability scanner, in: Proceedings of the 15th international conference on World Wide Web, ACM, 2006, pp. 247–256.

[74] A. Takanen, J. D. Demott, C. Miller, Fuzzing for software security testing and quality assurance, Artech House, 2008.

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

[76] T. Wang, T. Wei, G. Gu, W. Zou, Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection, in: Proceeding of 2010 IEEE Symposium on Security and Privacy (SP), IEEE, 2010, pp. 497–512.

[77] A. Doupé, L. Cavedon, C. Kruegel, G. Vigna, Enemy of the state: A state-aware black-box web vulnerability scanner., in: USENIX Security Symposium, 2012, pp. 523–538.

[78] C. Cadar, D. Dunbar, D. R. Engler, Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs., in: OSDI, Vol. 8, 2008, pp. 209–224.

[79] P. Godefroid, M. Y. Levin, D. A. Molnar, Automated whitebox fuzz testing., in: NDSS, Vol. 8, 2008, pp. 151–166.

[80] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, D. R. Engler, Exe: automatically generating inputs of death, ACM Transactions on Information and System Security (TISSEC) 12 (2) (2008) 10.

[81] I. Haller, A. Slowinska, M. Neugschwandtner, H. Bos, Dowsing for overflows: A guided fuzzer to find buffer boundary violations., in: Usenix Security, 2013, pp. 49–64.

[82] S. Heelan, Vulnerability detection systems: Think cyborg, not robot, IEEE Security and Privacy 9 (3) (2011) 74–77.

[83] S. Sparks, S. Embleton, R. Cunningham, C. Zou, Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting, in: Proceeding of Twenty-Third Annual Computer Security Applications Conference. ACSAC 2007., IEEE, 2007, pp. 477–486.

[84] J. DeMott, R. Enbody, W. F. Punch, Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing, BlackHat and Defcon.

[85] H. Shahriar, M. Zulkernine, Music: Mutation-based sql injection vulnerability checking, in: Proceeding of the Eighth International Conference on Quality Software. QSIC'08., IEEE, 2008, pp. 77–86.

[86] A. Groce, R. Joshi, Extending model checking with dynamic analysis, in: Verification, Model Checking, and Abstract Interpretation, Springer, 2008, pp. 142–156.

[87] R. Jhala, R. Majumdar, Software model checking, ACM Computing Surveys (CSUR) 41 (4) (2009) 21.

[88] P. Godefroid, Model checking for programming languages using verisoft, in: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1997, pp. 174–186.

[89] K. Havelund, T. Pressburger, Model checking java programs using java pathfinder, International Journal on Software Tools for Technology Transfer 2 (4) (2000) 366–381.

[90] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, D. L. Dill, Cmc: A pragmatic approach to model checking real code, ACM SIGOPS Operating Systems Review 36 (SI) (2002) 75–88.

[91] M. B. Dwyer, J. Hatcliff, Bogor: an extensible and highly-modular software model checking framework, in: ACM SIGSOFT Software Engineering Notes, Vol. 28, ACM, 2003, pp. 267–276.

[92] P. Godefroid, N. Klarlund, K. Sen, Dart: directed automated random testing, in: ACM Sigplan Notices, Vol. 40, ACM, 2005, pp. 213–223.

[93] E. J. Schwartz, T. Avgerinos, D. Brumley, All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask), in: Proceeding of the 2010 IEEE Symposium on Security and Privacy (SP), IEEE, 2010, pp. 317–331.

[94] S. Lekies, B. Stock, M. Johns, 25 million flows later: large-scale detection of dom-based xss, in: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, ACM, 2013, pp. 1193–1204.

[95] M. G. Kang, S. McCamant, P. Poosankam, D. Song, Dta++: Dynamic taint analysis with targeted control-flow propagation., in: NDSS, 2011.

[96] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, G. Vigna, Saner: Composing static and dynamic analysis to validate sanitization in web applications, in: Proceeding of IEEE Symposium on Security and Privacy. SP 2008., IEEE, 2008, pp. 387–401.

[97] G. Sarwar, O. Mehani, R. Boreli, D. Kaafar, On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices, in: Proceeding of the 10th International Conference on Security and Cryptography (SECRYPT), 2013.

[98] L. A. Clarke, A program testing system, in: Proceedings of the 1976 annual conference, ACM, 1976, pp. 488–491.

[99] D. Davidson, B. Moench, T. Ristenpart, S. Jha, Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution., in: USENIX Security, 2013, pp. 463–478.

[100] V. Ganesh, D. L. Dill, A decision procedure for bit-vectors and arrays, in: Computer Aided Verification, Springer, 2007, pp. 519–531.

[101] L. De Moura, N. Bjørner, Z3: An efficient smt solver, in: Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2008, pp. 337–340.

[102] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, M. D. Ernst, Hampi: a solver for string constraints, in: Proceedings of the eighteenth international symposium on Software testing and analysis, ACM, 2009, pp. 105–116.

[103] M.-T. Trinh, D.-H. Chu, J. Jaffar, S3: A symbolic string solver for vulnerability detection in web applications, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2014, pp. 1232–1243.

[104] C. Cadar, K. Sen, Symbolic execution for software testing: three decades later, Communications of the ACM 56 (2) (2013) 82–90.

[105] Z. Wang, J. Ming, C. Jia, D. Gao, Linear obfuscation to combat symbolic execution, in: Computer Security–ESORICS 2011, Springer, 2011, pp. 210–226.

[106] K. Sen, D. Marinov, G. Agha, CUTE: a concolic unit testing engine for C, Vol. 30, ACM, 2005.

[107] D. Molnar, X. C. Li, D. A. Wagner, Dynamic test generation to find integer bugs in x86 binary linux programs, in: Proceedings of the 18th conference on USENIX security symposium, USENIX Association, 2009, pp. 67–82.

[108] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, E. X. Wu, D. Song, Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery., in: USENIX Security Symposium, 2011, pp. 139–154.

[109] M. Monga, R. Paleari, E. Passerini, A hybrid analysis framework for detecting web application vulnerabilities, in: Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems, IEEE Computer Society, 2009, pp. 25–32.

[110] W. G. Halfond, A. Orso, Combining static analysis and runtime monitoring to counter sql-injection attacks, in: ACM SIGSOFT Software Engineering Notes, Vol. 30, ACM, 2005, pp. 1–7.

[111] V. Felmetsger, L. Cavedon, C. Kruegel, G. Vigna, Toward automated detection of logic vulnerabilities in web applications, in: USENIX Security Symposium, 2010, pp. 143–160.

[112] S.-W. Woo, H. Joh, O. H. Alhazmi, Y. K. Malaiya, Modeling vulnerability discovery process in apache and iis http servers, Computers & Security 30 (1) (2011) 50–62.

[113] O. H. Alhazmi, Y. K. Malaiya, Quantitative vulnerability assessment of systems software, in: Proceedings of annual reliability and maintainability symposium, 2005, pp. 615–620.

[114] O. Alhazmi, Y. Malaiya, I. Ray, Security vulnerabilities in software systems: A quantitative perspective, in: Data and Applications Security XIX, Springer, 2005, pp. 281–294.

[115] S. Rahimi, M. Zargham, Vulnerability scrying method for software vulnerability discovery prediction without a vulnerability database, Reliability, IEEE Transactions on 62 (2) (2013) 395–407.

[116] O. H. Alhazmi, Y. K. Malaiya, Application of vulnerability discovery models to major operating systems, Reliability, IEEE Transactions on 57 (1) (2008) 14–22.

[117] H. Joh, Y. K. Malaiya, Seasonal variation in the vulnerability discovery process, in: Proceedings of ICST'09 International Conference on Software Testing Verification and Validation, 2009., IEEE, 2009, pp. 191–200.

[118] Application Security Verification Standard (ASVS), `https://www.owasp.org/images/5/58/OWASP_ ASVS_ Version_ 2.pdf`, [Online; accessed 2016-10-8].

[119] D. Oliveira, M. Rosenthal, N. Morin, K.-C. Yeh, J. Cappos, Y. Zhuang, It's the psychology stupid: how heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots, in: Proceedings of the 30th Annual Computer Security Applications Conference, ACM, 2014, pp. 296–305.

[120] H. Xue, N. Dautenhahn, S. T. King, Using replicated execution for a more secure and reliable web browser., in: NDSS, 2012.

[121] L. Szekeres, M. Payer, T. Wei, D. Song, Eternal war in memory, in: IEEE Symposium on Security and Privacy, 2013.

[122] D. Balzarotti, M. Cova, V. V. Felmetsger, G. Vigna, Multi-module vulnerability analysis of web-based applications, in: Proceedings of the 14th ACM conference on Computer and communications security, ACM, 2007, pp. 25–35.

[123] J. Bau, E. Bursztein, D. Gupta, J. Mitchell, State of the art: Automated black-box web application vulnerability testing, in: Proceeding of the 2010 IEEE Symposium on Security and Privacy (SP), IEEE, 2010, pp. 332–345.

[124] M. Almorsy, J. Grundy, A. S. Ibrahim, Supporting automated vulnerability analysis using formalized vulnerability signatures, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2012, pp. 100–109.

[125] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and discovering vulnerabilities with code property graphs, in: Proceedings of 2014 IEEE Symposium on Security and Privacy (SP), IEEE, 2014, pp. 590–604.

[126] W. Mallouli, A. Mammar, A. Cavalli, W. Jimenez, Vdc-based dynamic code analysis: Application to c programs, Journal of Internet Services and Information Security 1 (2/3) (2011) 4–20.

[127] B. Livshits, Improving software security with precise static and runtime analysis, Ph.D. thesis, Stanford University (2006).

[128] U. Shankar, K. Talwar, J. S. Foster, D. Wagner, Detecting format string vulnerabilities with type qualifiers., in: USENIX Security Symposium, 2001, pp. 201–220.

[129] L. C. Paulson, Proving properties of security protocols by induction, in: Proceedings of the 10th workshop on Computer Security Foundations, IEEE, 1997, pp. 70–83.

[130] M. Burrows, M. Abadi, R. M. Needham, A logic of authentication, in: Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, Vol. 426, The Royal Society, 1989, pp. 233–271.

[131] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL: a proof assistant for higher-order logic, Vol. 2283, Springer Science & Business Media, 2002.

[132] D. Larochelle, D. Evans, Statically detecting likely buffer overflow vulnerabilities., in: USENIX Security Symposium, Vol. 32, Washington DC, 2001.

[133] C. E. Landwehr, A. R. Bull, J. P. McDermott, W. S. Choi, A taxonomy of computer program security flaws, ACM Computing Surveys (CSUR) 26 (3) (1994) 211–254.

[134] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, D. A. Webb, Security analysis and enhancements of computer operating systems, Tech. rep., Technical report NBSIR-76-1041. National bureau of standards Washington inst for computer sciences and technology. (1976).

[135] R. Bisbey, D. Hollingworth, Protection analysis: Final report (1978).

[136] P. Anderson, Codesurfer/path inspector, Proceeding of the 20th IEEE International Conference on Software Maintenance, 2004. Proceedings, 2004.

[137] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Software verification with blast, in: Model Checking Software, Springer, 2003, pp. 235–239.

[138] S. Rawat, D. Ceara, L. Mounier, M.-L. Potet, Combining static and dynamic analysis for vulnerability detection, arXiv preprint arXiv:1305.3883.

[139] R. Anderson, Security in open versus closed system the dance of boltzmann, coase and moore, Open Source Software Economics.

[140] E. Rescorla, Is finding security holes a good idea?, Security & Privacy, IEEE 3 (1) (2005) 14–19.

[141] J. D. Musa, K. Okumoto, A logarithmic poisson execution time model for software reliability measurement, in: Proceedings of the 7th international conference on Software engineering, IEEE Press, 1984, pp. 230–238.

[142] J. Dahse, T. Holz, Simulation of built-in php features for precise static code analysis, in: Symposium on Network and Distributed System Security (NDSS), 2014.