A DETECTION AND CORRECTION APPROACH FOR OVERFLOW VULNERABILITIES IN GRAPHICAL USER INTERFACES

A Thesis Submitted to
The Graduate School of Engineering and Sciences of
İzmir Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Can Arda MÜFTÜOĞLU

December 2009
İZMİR

We approve the thesis of **Can Arda MÜFTÜOĞLU**

Assist. Prof. Dr. Tuğkan TUĞLULAR
Supervisor

Assist. Prof. Dr. Tolga AYAV
Committee Member

Assist. Prof. Dr. Gökhan Dalkılıç
Committee Member

18 December 2009

Prof. Dr. Sıtkı AYTAÇ
Head of the Department of Computer Engineering

Assoc. Prof. Dr. Talat YALÇIN
Dean of the Graduate School of Engineering and Sciences

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Assist. Prof. Dr. Tuğkan TUĞLULAR, for his guidance and encouragement. His enthusiasm, inspiration, and great efforts during this research made this work valuable.

Furthermore, I had the pleasure of working with Prof. Dr. Fevzi Belli and Michael Linschulte throughout my research.

I am also grateful to my friends; Barış Yıldız, Fatih Tekbacak, Orhan Dağdeviren, Dilek Tapucu, Emre Gözütok, Serdal Okur and Özgür Kaya for their encouragement and support.

Finally, I should thank my parents who always supported me throughout my whole education as well as in my graduate study.

ABSTRACT

A DETECTION AND CORRECTION APPROACH FOR OVERFLOW VULNERABILITIES IN GRAPHICAL USER INTERFACES

The objective of this thesis is to propose an approach for detecting overflow vulnerabilities such as buffer and boundary overflows by using static analysis and correcting these vulnerabilities by applying a correction mechanism which uses static code insertion. GUI is tested by specifying user interface requirements and converting this specification into an event-sequence model. Decision table notion is used for modeling the dependencies and boundary restrictions on input data and generating test cases. The test cases are applied to the GUI as inputs manually in real environment. The faults are observed. Then, the overflow vulnerability analysis tool is used to analyze the source code of the program. The deficiencies related to overflow vulnerabilities are found by static analysis. After that, the correction mechanism is applied to the deficient parts of the source code. The software is tested in real environment again. The proposed approach is observed to be successful for detecting and correcting overflow vulnerabilities in GUIs.

ÖZET

GRAFİK KULLANICI ARAYÜZLERİNDEKİ TAŞMA ZAYIFLIKLARI İÇİN BİR BELİRLEME VE DÜZELTME YAKLAŞIMI

Bu çalışmada, arabellek ve sınır değeri gibi taşma zayıflıklarının statik analiz yöntemiyle belirlenmesi ve bu zayıflıkların statik kod ekleme yöntemini kullanan bir düzeltme mekanizmasıyla düzeltilmesine yönelik bir yaklaşım önerilmektedir. Grafik kullanıcı arayüzleri, kullanıcı arayüzü gereksinimlerinin belirlenmesi ve bu gereksinimlerin bir etkinlik dizisi modeline dönüştürülmesi ile sınanmaktadır. Karar tablosu kavramı, girdi verisi üzerindeki sınır değeri kısıtlamaları ve bağımlılıklarını modellemek ve test durumlarını oluşturmak için kullanılmaktadır. Oluşturulan test durumları, reel bir ortamda grafik kullanıcı arayüzüne girdi olarak uygulanmaktadır. Hatalar yakalanmaktadır. Sonrasında, geliştirilen taşma zayıflığı analiz aracı programın kaynak kodunu analiz etmektedir. Taşma zayıflıklarının yarattığı eksiklikler, statik analiz yöntemiyle bulunmuştur. Daha sonra, önerilen düzeltme mekanizması, kaynak kodunun eksikliği olan kısımlarına uygulanmıştır. Yazılım reel ortamda tekrar sınanmıştır. Öne sürülen yaklaşımın, taşma zayıflıklarının belirlenmesi ve düzeltilmesinde başarılı bir yaklaşım olduğu görülmüştür.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Graphical user interfaces (GUIs) add up to half or more of the source code in software [1]. They have become popular and common in computer based systems. Testing GUIs is a challenging task for various reasons. First of all, great number of combinations of inputs and events that occur as system outputs are possessed by the input space. Second, an enormous number of states are possessed by even simple GUIs and also many complex dependencies may hold between the states of the GUI system and its inputs [2].

GUIs may allow intruders to gain control over a system or access to its stored data by intentionally caused overflows. Overflow vulnerabilities are caused by various deficiencies, but in this thesis only buffer overflow and boundary overflow vulnerabilities are examined which are related to the input that is taken from GUI elements. Buffer overflow is a type of vulnerability where a process which in our case is the input entered from the GUI element stores data in a buffer outside the memory that the developer allocated for. Also, a boundary overflow is an input error and occurs when values are entered that violate the range of values [3]. Such entries exceed the implicitly or explicitly specified but not implemented boundary values; thus both overflows are consequences of deficient control mechanism in the software concerning system constraints.

The approach in this thesis suggests testing the GUI of software by specifying user interface requirements and converting this specification into a model from which valid and invalid test cases can be generated [2]. For specification of user-system interactions, an event-based formal model, where the inputs and events are merged and assigned to the vertices of an event transition diagram, called event sequence graph (ESG) is chosen. The arcs in ESG visualize the sequence relation of the events. An ESG is a simple albeit powerful formalism for capturing the behavior of interactive systems [2]. However, modeling complex boundary restrictions on input data as well as dependencies among them inflates the ESG model of a system under consideration

(SUC). The nodes of the underlying ESG by decision tables (DTs) are refined to overcome this problem, which visualize Boolean algebraic constraints on input data [4]. Based on the rules of the DT, which are the constraints on the input data, test data are generated. Equivalence class partitioning and boundary value analysis approaches support the test case generation process [5-6].

The overflow vulnerability detection and correction approach proposed here is composed of two phases: (1) testing the SUC with the test cases generated by DT augmented ESGs and (2) detecting deficiencies in the source code related to overflow vulnerabilities and correcting these deficiencies if they are found during the test phase. After generating the test cases, the SUC is tested manually in real environment by entering the generated test cases to its user interface. The faults are obtained and extracted manually to a file. A detection and correction mechanism is created to eliminate the faults that are obtained after applying the test cases. Algorithms for detection and correction of buffer overflow and boundary overflow vulnerabilities through static analysis are developed. The idea is to detect the deficiencies that are caused by the unchecked input which are entered to the system through the GUI elements, by analyzing the source code statically and correcting these deficiencies by static code insertion. To achieve this, an overflow vulnerability analysis tool is implemented and coded in Java programming language. The overflow vulnerability analysis tool uses the detection and correction algorithms that are developed. Applying the proposed detection and correction tool, the new corrected version of the SUC is tested in real environment again. The faults before and after applying our method are compared. The summary of the approach is shown as flowchart in Figure 1.1. To show and check the usefulness, reliability and validity of the tool, a case study has been performed. Port scanner software which is coded in C++ programming language is used as the example software in the case study section.

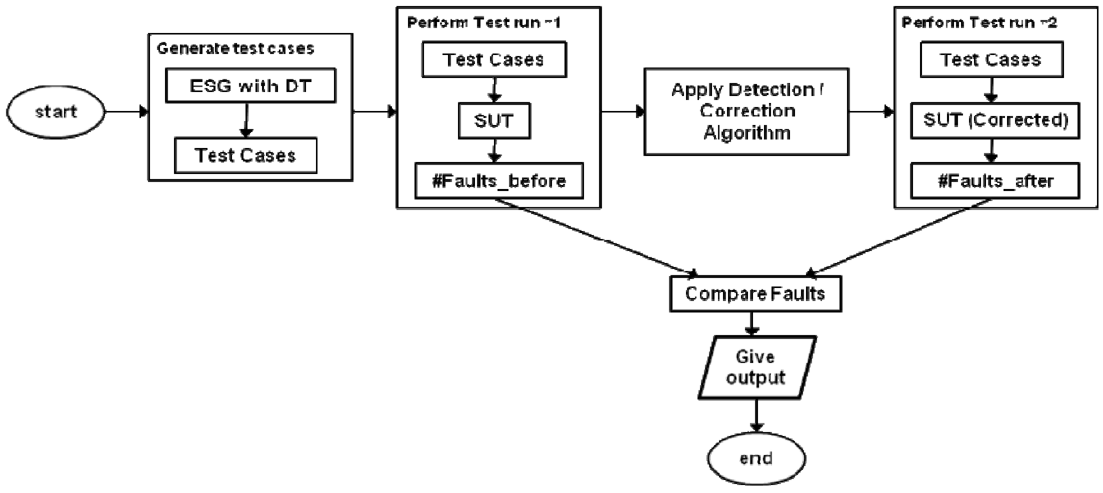


Figure 1.1. Summary of the approach

This thesis is organized as follows. Chapter 2 describes the type of GUI-based overflow vulnerabilities that we examine, the concepts and techniques that are used in this thesis for GUI testing, general description of static analysis and the related work done so far about these concepts. Chapter 3 discusses how to model a GUI by DT augmented ESGs, and to generate test cases by considering the rules of DTs. Chapter 4 mentions about detection of overflow vulnerabilities by static analysis, correction by using static code insertion into the source code and shows the implementation of the detection and correction algorithms as overflow vulnerability analysis tool. Chapter 5 demonstrates a case study by considering the two type of overflow vulnerabilities described above. Finally, Chapter 6 gives the conclusion of this thesis work.

CHAPTER 2

BACKGROUND

2.1. GUI Based Overflow Vulnerabilities

In this thesis, two types of GUI-based overflow vulnerabilities that are examined are: (1) buffer overflow vulnerability which in here is considered as a type of GUI-based overflow vulnerability when the input being received from the system exceeds the maximum length of the variable type that is assigned to and (2) boundary overflow vulnerability which occurs when the input being received from the system exceeds a predefined boundary value. These vulnerabilities both result in unexpected and undesired behavior of the system.

2.1.1. Buffer Overflow

Buffer overflows are the most common form of security threat in software systems, and vulnerabilities attributed to buffer overflows have dominated CERT advisories [7]. Buffer overflow vulnerabilities occur when a string of characters or numbers of unchecked length is entered into a program. In this thesis, we consider the inputs that are entered to the program from the GUI elements. The inputs that are entered to the system should be checked according to the maximum length of the variable type that it is assigned to and if the size is exceeded by the given input, the program should give an error and abort the current process, otherwise buffer overflow vulnerability occurs. This allows user supplied input to overwrite other variables, thereby changing their values. Such attacks can change the value of a return address from a function call and cause control to jump to malicious code that was also entered via the buffer overflow [8].

Furthermore, unchecked input can cause the system to work in an undesired way. Consider a program that includes a GUI with some elements such as text field, text area, checkbox, button etc. When a numerical input which will be assigned to a variable with a type of unsigned integer is entered to an element of the GUI, if the numerical input is higher than the maximum length that is allowed for the unsigned integer variable type, then the value of the input is assigned to the variable erroneously. Hence, it may lead to malfunctions of the entire system which may result to vulnerabilities for attacks.

According to the work in [9], the taxonomy of buffer overflows that was developed by Zitser [10] that contained 13 attributes was modified and expanded to address the problems encountered with its application with the new taxonomy that consists of 22 attributes. In this thesis, the third attribute in the new taxonomy "Data Type" which indicates the type of data stored in the buffer (character, integer, floating point, wide character, pointer, unsigned character, and unsigned integer) is considered.

2.1.2. Boundary Overflow

Boundary overflow vulnerability occurs when the input being received by a system causes the system to exceed an assumed boundary resulting in vulnerability [3]. These boundaries aren't related to the maximum length of the variables as in Buffer Overflow; however they are predefined values that are related to the semantics of the GUI elements. For example, consider a port scanner where the start and end port values are entered to the program to scan the range of these values. Even if the input values that are entered from the GUI elements for the start and end port values don't exceed the maximum length of the unsigned integer type, there are semantically predefined boundary values. The start port value can't be lower than 0 and the end port value can't be higher than 65535. Checking these input values with respect to the predefined boundaries is critical for the program to behave as intended. Otherwise, boundary overflow occurs and the program works in an unexpected way.

2.2. GUI Testing

The most popular form of user interfaces are direct-manipulation interfaces called GUIs [11]. As GUIs become more popular, they are increasingly being used in various systems [12] and testing them is necessary to avoid undesired situations [11]. Difficulty occurs in many reasons in testing the correctness of GUI. There are a great number of possible interactions with a GUI. Each sequence can result in a different state of the system. Thus, the GUI is needed to be tested in a large number of states [13].

GUI testing involves several steps. Firstly, a set of test cases must be generated. Because of the difficulties, especially due to the huge number of set of possible test cases, this is challenging for GUI testing. After the test cases are constructed, they must be executed where this is the actual testing part. The aim is to check whether the GUI is performing as intended. An incorrect GUI state can lead to unexpected events, and also can make the further execution of the test cases useless. Consequently, the execution of the test case must be terminated when an error is detected [14].

In [14], the design of Planning Assisted Tester for Graphical User Interface Systems, which is called PATHS, is presented. The goal of the work is to facilitate the automation of GUI testing by using a GUI testing technique based on user event interaction sequences. There are also many previous works on automated GUI testing. Brooks et. al. [15] presented a new technique for testing of GUI applications. Information on the actual usage of the application is used to ensure that a new version of the application will function correctly. Usage profiles, sequences of events that end-users execute on a GUI, are used to develop a probabilistic usage model of the application which an algorithm that is developed in [15] uses the model to generate test cases that represent events the user is most likely to execute. In another previous work of automated GUI testing [16], a test oracle technique is developed to determine if a GUI behaves as expected for a given test case, where the oracle uses a formal model of a GUI, expressed as sets of objects, object properties, and actions. The oracle automatically derives the expected state for every action in the test case when the formal model of a GUI is given.

Also, some interesting work have been done on regression testing of GUIs. In regression testing, when the structure of a GUI is modified, test cases from the original GUI are either reusable or unusable on the modified GUI. Since GUI test case generation is expensive, the goal of the work in [17] is to make the unusable test cases usable. A novel GUI regression testing technique that first automatically determines the usable and unusable test cases from a test suite after a GUI modification, then determines which of the unusable test cases can be repaired so they can execute on the modified GUI, and lastly repairs the test cases is presented. An extended version of the previous work has also been published [18], where repairing transformations is used to repair the test cases and an empirical study for four open-source applications has also been performed.

Knowing that it is still not clear how to define GUI test cases and how many actions should be comprised of a GUI test case, the work in [19] proposes an approach that defines GUI test cases as a sequence of primitive GUI actions and treats GUI test suites as an inner hierarchy of formal language. In [20], coverage criteria for GUIs are being focused, which are important rules that provide an objective measure of test quality. A new coverage criteria is presented to help determine whether a GUI has been adequately tested. These coverage criteria use events and event sequences to specify a measure of test adequacy.

The characteristics of GUIs present special challenges when verifying a GUI's behavior [21-23]. Many of these challenges stem from the fact that GUIs are event-based systems. A test case usually consists of a single set of inputs, and the expected result is the output that results from processing that input. With GUIs, the input is an entire action sequence, where the effect of each action most probably depends upon the effects of its previous actions. Instead of a specific output, each action affects the state of the GUI. Furthermore, comparison of the expected and actual GUI states cannot wait until the entire action sequence has been executed. It is necessary to verify the state of the GUI after the execution of each action. If not, incorrect GUI behavior for one action may result in a state in which future actions in the sequence cannot be executed at all [16]. While testing a system, a model of the system helps to predict and control its behavior. Modeling a system acquires the understanding of its abstraction, and in the case of testing GUIs, there is the need of a formal specification tool distinguishing between legal and illegal situations. These requirements are fulfilled by ESGs.

In this thesis, dynamic input validation is combined with static analysis for evaluating given constraints. Input validation checks the syntax and, partly, semantics of information provided by user via user interface (UI), mostly realized as a GUI [24]. Because UI errors may lead to malfunctions of the entire system which, in turn, may lead to vulnerabilities for attacks [25], various specification and implementation based test techniques exist to validate UI [26]. Among them, code coverage rate helps to detect possible input errors and, by using correction algorithms, the illegal inputs can be enforced to set them in defined boundaries. ESGs [2] can be used for analysis and validation of UI requirements prior to implementation and testing of the code [27]. An ESG is a simple albeit powerful formalism for capturing the behavior of a variety of interactive systems that include real-time, embedded systems and graphical user interfaces [2]. In [4], decision tables are introduced to refine a node of the ESG where the test cases are generated according to the rules of the decision table.

2.3. Event Sequence Graphs

Apart from the notion of finite state automata (FSA), in ESG, the simplification by merging the inputs and states helps the test engineer to easily understand and check the external behavior of the system, hence the "inputs" and "states" are turned into "events".

Definition 1: An *event sequence graph* $ESG = (V, E, \Xi, \Gamma)$ is a directed graph where $V \neq \varnothing$ is a finite set of vertices (nodes), $E \subseteq V \times V$ is a finite set of arcs (edges), $\Xi, \Gamma \subseteq V$ are finite sets of distinguished vertices with $\xi \in \Xi$, and $\gamma \in \Gamma$, called entry nodes and exit nodes, respectively, wherein $\forall v \in V$ there is at least one sequence of vertices $\langle \xi, v_0, \ldots, v_k \rangle$ from each $\xi \in \Xi$ to $v_k = v$ and one sequence of vertices $\langle v_0, \ldots, v_k, \gamma \rangle$ from $v_0 = v$ to each $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for i = 0,…,k-1 and $v \neq \xi, \gamma$ [28].

Ξ (*ESG*), Γ (*ESG*) represent the entry nodes and exit nodes of a given ESG, respectively. To mark the entry and exit of an ESG, all $\xi \in \Xi$ are preceded by a pseudo vertex '[' $\notin$ V and all $\gamma \in \Gamma$ are followed by another pseudo vertex ']' $\notin V$. The semantics of an ESG is as follows: Any $v \in V$ represents an event. For two events v, v' $\in V$, the event v' must be enabled after the execution of v iff $(v, v') \in E$. The operations on identifiable components of the GUI are controlled and/or perceived by input/output devices, i.e., elements of windows, buttons, lists, checkboxes, etc. Thus, an event can be a user input or a system response; both of them are elements of V and lead interactively to a succession of user inputs and expected desirable system outputs.

Definition 2: Let V, E be defined as in Definition 1. Then any sequence of vertices $\langle v_0, \ldots, v_k \rangle$ is called an *event sequence* (*ES*) iff $(v_i, v_{i+1}) \in E$, for i=0,…,k-1.

Moreover, an ES is *complete* (or, it is called a *complete event sequence, CES*), iff $v_0 \in \Xi$ and $v_k \in \Gamma$ [28].

Note that the pseudo vertices '[', ']' are not included in ESs. An ES = $\langle v_i, v_k \rangle$ of length 2 is called an *event pair* (EP). A CES may invoke no interim system responses during user-system interaction, i.e., it may consist of consecutive user inputs and a final system response.

The approach proposed in this thesis assumes that upon a faulty user input the system has to inform the user, and, wherever possible, point him or her properly in the right direction in order to reach the desirable final or interim situation. Due to this requirement, a complementary view is necessary to consider potential user errors in the modeling of the system. Graphically speaking, missing edges of the ESG represent undesirable user-system interactions, i.e., *faulty event pairs (FEP)*. FEPs can systematically be constructed by either (1) adding arcs in the opposite direction wherever only one-way arcs exist, or (2) adding two-way arcs between vertices wherever no arcs connect them, or finally, (3) adding self-loops to vertices wherever none exist.

Definition 3: Let $ES = \langle v_0, \ldots, v_k \rangle$ be an event sequence of length k+1 of an ESG and FEP = $\langle v_k, v_m \rangle$ a faulty event pair. The concatenation of the ES and FEP then forms a *faulty event sequence FES* = $\langle v_0, \ldots, v_k, v_m \rangle$. FES is *complete* (or, it is called a *faulty complete event sequence, FCES*) iff $v_0 \in \Xi$. The ES as part of a FCES is called a *starter* [28].

CES and FCES form test cases to the SUC. The SUC is supposed to accept test inputs described by CESs in the specified order whereas test inputs described by FCESs should result in a warning.

2.4. Decision Table Augmented ESGs

Modeling input data, especially concerning causal dependencies between each other as additional nodes, inflates the ESG model. To avoid this, decision tables are introduced to refine a node of the ESG. Such refined nodes are double-circled.

Definition 4: A Decision Table $DT = \{C, A, R\}$ represents actions that depend on certain constraints where:

- $C \neq \varnothing$ is the set of constraints
- $A \neq \varnothing$ is the set of actions
- $R \neq \varnothing$ is the set of rules that describe executable actions depending on a certain combination of constraints [28].

Decision tables [29] are popular in information processing and are also used for testing, e.g., in cause and effect graphs. A decision table logically links conditions (”if”) with actions (”then”) that are to be triggered, depending on combinations of conditions (”rules”) [4].

Definition 5: Let R be defined as in Definition 4. Then a *rule* $R_i \in R$ is defined as $R_i = (C_{True}, C_{False}, A_x)$ where:

- $C_{True} \subseteq C$ is the set of constraints that have to be resolved to true
- $C_{False} = C \backslash C_{True}$ is the set of constraints that have to be resolved to false
- $A_x \subseteq A$ is the set of actions that should be executable if all constraints $t \in C_{True}$ are resolved to true and all constraints $f \in C_{False}$ are resolved to false [28].

Note that $C_{True} \cup C_{False} = C$ and $C_{True} \cap C_{False} = \varnothing$ under regular circumstances. In certain cases it is inevitable to remark conditions with a *don't care* (symbolized with a '-' in DT), i.e., such a condition is not considered in a rule and $C_{True} \cup C_{False} \subset C$. A DT is used to refine data input of GUI’s.

2.5. Static Analysis

Static analysis performs the analysis of the source code of a program without executing it. The analysis that it performs can be preformed based on source code, binary format or both. The software developers are able to perform thorough and quick inspection of the source code to locate different types of issues that they are concerned about [24]. For example, to ensure that a source code ensures a specific exception handling mechanism, a static analysis tool that is designed to check some predefined deficiencies is required to run and then change the source code to meet the requirements.

Many of the static analysis tools produce false positives (false alarms) or false negatives (ignoring an important issue), or both. Static analysis tools that are focused on software security usually try to minimize the number of false negatives in order to avoid leaving out any possibly important security problems. Thereby, they tend to generate more false positives.

Static analysis tools provide the developers to improve reliability, security and overall quality of software by detecting predefined programming errors early in development process. Hence, the programmers have an opportunity to correct these deficiencies before the deployment phase of the software development process and also before a malicious user has a possibility to exploit the vulnerability of the program [24].

Static analysis techniques are used to handle buffer overflow problems, which are one of the common security issues as they may lead to vulnerabilities like system crash, corruption of data or undesired system access. They occur when a programmer implements incorrect bound checks on buffer size or even fails to do bounds checking where data is written into a fixed length buffer [3]. By definition buffer overflow is similar to boundary overflow, as mentioned in Section 2.1.2., which is an input error and occurs when values are entered that violate the range of values. Such entries exceed the implicitly or explicitly specified but not implemented boundary values. Therefore, research results on buffer overflows can be applied to boundary overflow problems.

There are various static analysis tools which are capable of detecting buffer overflows. Flawfinder [30] is a pattern matching vulnerability detection tool which contains a large database of vulnerable patterns in C and C++ programs. During the analysis of the source code, it produces a list of potential security flaws, each with an associated risk level. The list of flaws reported by the tool is sorted by their risk level showing the riskiest first. ITS4 [31] is another pattern matching tool which has the ability to analyze multiple languages in addition to C. ITS4 also has special handlers for some rules in its ruleset, which perform additional processing to determine if a match is really a vulnerability. Cqual [32] is a type-based analysis tool which extends the C type system with custom type qualifiers and propagates them through type inference. Some vulnerabilities can be expressed as type inconsistencies and detected using Cqual. SPLINT [33] is a tool for checking C programs for security vulnerabilities and programming mistakes which uses a lightweight data-flow analysis to verify assertions about the program. The tool can be used to detects problems such as NULL pointer dereferences, unused variables, memory leaks and buffer overruns. Additional checks can also be extended in a special extension language. MOPS [34] is a tool for verifying the conformance of C programs to rules that can be expressed as temporal safety properties. It represents these properties as finite state automata and uses a model checking approach to find if any insecure state is reachable in the program. BOON [35] is a tool for detecting buffer overrun vulnerabilities in C code which generates integer range constraints based on string operations in the source code and solves them, producing warnings about potential buffer overflows. The range analysis is flow-insensitive and generates a very large number of false alarms. UNO, another static analysis tool accepts user-defined properties of application specific requirements to overcome specific problems [36]. In [37], identification of buffer overrun vulnerabilities by statically analyzing C source code is performed. A scalable analysis based on modeling C string manipulations as a linear program has been demonstrated. A prototype has been developed and used it to identify several vulnerabilities in popular security critical applications.

In [38], taint propagation is defined as a technique which is used by static analysis tools to find software vulnerabilities caused by failed or missing input validation. In taint propagation, the tool tracks the tainted data, including also the parts of the program where the tainted data has effect on. A taint analysis is performed to find the places where data is read from an untrusted source [39], e.g., by using Patterson's value range propagation algorithm for calculating the range of possible values for each variable [40].

Static analysis for securing web applications is quite a popular topic. There have been many researches on this field. Many verification tools, as described in the previous paragraph are discovering previously unknown vulnerabilities in C programs, raising hopes that the same success can be achieved with Web applications. In [41], a sound and holistic approach to ensuring Web application security is described. A lattice-based static analysis algorithm has been created by viewing Web application vulnerabilities as a secure information flow problem. A tool named WebSSARI (Web application Security by Static Analysis and Runtime Inspection) has also been created to test the developed algorithm, and used it to verify many open-source Web application projects. In [42], the problem of vulnerable Web applications by means of static source code analysis has been addressed. In the mentioned work, flow-sensitive, interprocedural and context-sensitive data flow analysis are used to discover vulnerable points in a program and the open source prototype implementation of the concepts described in this work is created as a tool named Pixy which is targeted at detecting cross-site scripting vulnerabilities.

The work in [43] presents two sets of observations relating static and dynamic analysis. The first observation concerned synergies between static and dynamic analysis. Wherever one is utilized, the other may also be applied, often in a complementary way, and existing analyses should inspire different approaches to the same problem. Furthermore, it was claimed that existing static and dynamic analyses often have had very similar structure and technical approaches. The second observation was that some static and dynamic approaches are similar in that each considers, and generalizes from, a subset of all possible executions. So that, researchers need to develop new analyses that complement existing ones. In the same work [43], it is suggested that researchers erase the boundaries between static and dynamic analysis and create unified analyses that can operate in either mode, or in a mode that blends the strengths of both approaches. Also, in [44], the merits of static analysis versus software model checking for finding bugs in system software are compared. Comparison has been performed on three different projects which the first two projects both used model checking and static analysis while the third project used only model checking. Considering that, we both apply model checking and static analysis to test software, the comparison of static analysis versus model checking approach have become useful during the work in this thesis.

CHAPTER 3

ESG-BASED TESTING OF GUI

3.1. Modeling GUI Events by ESG

Deterministic finite-state automata are commonly used for user interface specification and testing [45, 46]. FSA are extensively accepted for the design and specification of sequential systems. They have very good recognition and confirmation capabilities to distinguish between correct and faulty events. An event is an externally observable phenomenon, such as an environmental or a user stimulus, or a system response [4].

FSA can be represented by [2]:

- Set of inputs,
- Set of outputs,
- Set of states,
- Output function that maps pairs of inputs and states to outputs,
- Next-state function that maps pairs of inputs and states to next states.

To represent GUI, the elements of the FSA are clarified as follows:

- Input set: Identifiable GUI elements that can be controlled by input/output devices (i.e. button, textfield, checkbox etc.)
- Output sets:
    - Desired outputs: The outputs that the user is likely to have (i.e. correct results, expected responses)
    - Undesired outputs: The outputs that the user is not likely to have (i.e. faulty results, unexpected responses)

Since the user is interested in external behavior of the system, the notions of “state” and “input” are used on one side and “state”, “system response” and “output” on the other side synonymously. The focus is primarily on the expectations of the user.

In ESG, simplification from FSA is done by merging the inputs and states to events, so that the test engineer can easily understand and check the external behavior of the system. In GUI-based testing, the need of a formal specification tool distinguishing between legal and illegal events are fulfilled by ESGs. Any chains of edges from one vertex to another one, materialized by sequences of user inputs-states-triggered outputs define an interaction sequence (IS) which traverses the ESG from one vertex to another [2].

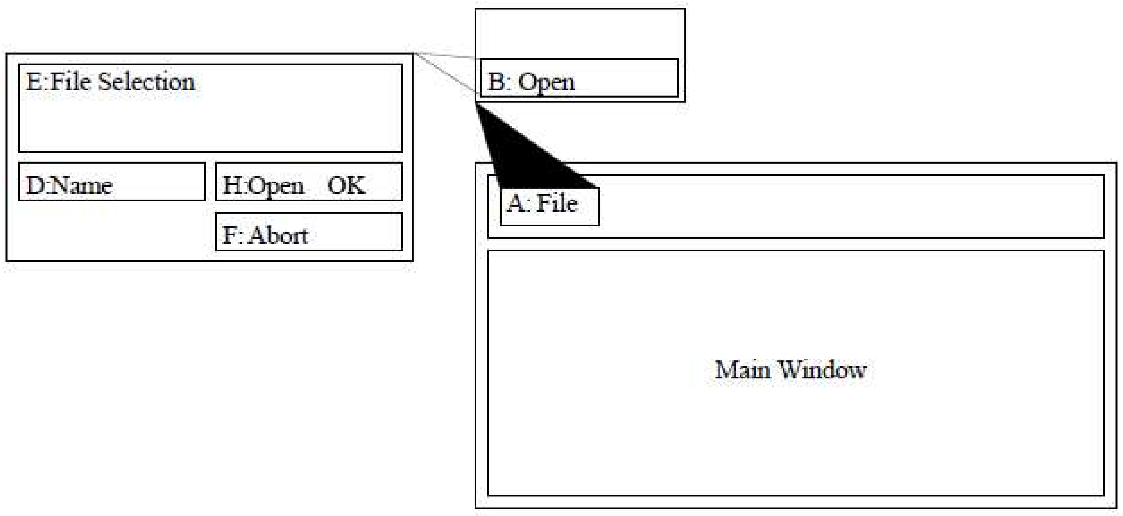


Figure 3.1. File menu of a text editor

Figure 3.1 represents abstractly the file menu of a text editor that enables a user to open a file. The black triangle symbolizes a click on File that opens the menu, the white triangle a click on Open that opens a dialog box to select a file. Clicking File several times shows no differences. The dialog box has to be closed in order to get access to the main window. To select a file in the dialog box, a user can click on a file or enter a filename in field Name. The input field Name has no default-entry. Finally, a user can click Open OK or Abort.

Figure 3.2 presents the GUI described in the Figure 3.1 as an ESG. The terms event, state, and situation will be used here synonymously. The ESG presents inputs which interact with the system that leads to events as system responses that are desired situations in compliance with the user's expectation. Based on the shown graph, the interaction sequences can be generated.

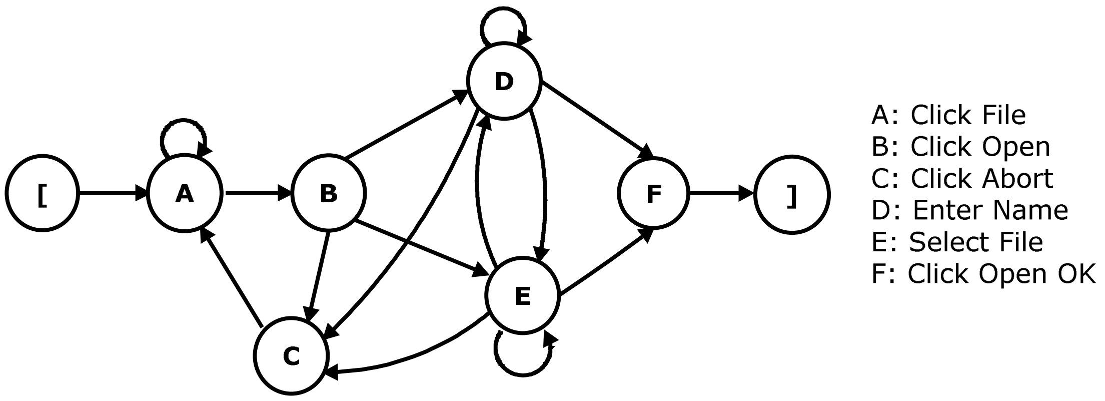


Figure 3.2. Figure 3.1 presented as an ESG

The conversion of the Figure 3.1 into Figure 3.2 (that is the formal representation) must be done manually which requires both theoretical skill and practical experiences in GUIs. However, the most of the following work, including generation of test cases, detection and correction of overflow vulnerabilities are carried out automatically with the algorithms that are described in the following sections.

After constructing the ESG, all legal sequences of user-system interactions can be identified. These sequences may be complete or incomplete, depending on whether they do or do not lead to a well-defined system response that the user expects. Also, legal interaction pairs (IP) of inputs as the edges of the ESG can be identified. The CISs and IPs are generated based on the ESG. Table 3.1 shows the IPs that are deducted from the ESG in Figure 3.2.

Table 3 1. The set of IPs of the ESG

Graph	IPs
Open File	AA, AB, BC, BD, BE, CA, DC, DD, DE, DF, EC, ED, EE, EF

Besides the generation of the well-defined system responses that the user expects, in contrast, fault modeling of the system can be done using ESGs. Faults are caused mostly from the specification errors when the expected behavior of the system is wrongly specified or the implementation errors when the implementation is not in compliance with the specification.

When modeling of the faults of a system, the system must detect all inputs that cannot lead to a desired event, inform the user, and navigate the user properly to reach a desired situation. So that, a view that is complementary to the modeling of the system is needed. To this end, the notion of Faulty Interaction Pairs (FIP) is introduced in [2] which consist of inputs that are not legal to the specification.

Figure 3.3 generates the completed ESG of Figure 3.2 by adding the FIPs as the edges in opposite direction wherever only way exists, adding loops wherever none exists and adding bi-directional edges wherever none exists.

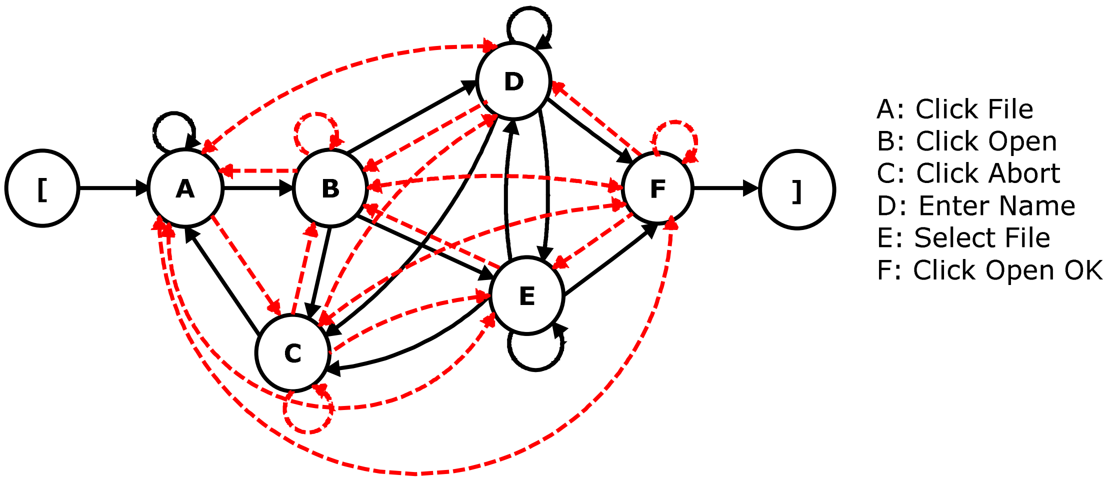


Figure 3.3. Completed ESG

With the completed ESG, it is now possible to construct all potential interaction faults by building all illegal actions that are not in compliance with the specification (Table 3.2). Extension through an IS can be started from the entry to the first node of FIP which brings the system to a faulty situation that is called a faulty complete interaction sequence (FCIS) (Table 3.3). If the system is enforced into a faulty state, the system must make the illegal event undone and be conducted into a legal state by a recovery mechanism.

Table 3.2. The set of FIPs of the ESG

Graph	FIPs
Open File	AD, AE, AC, AF, BA, BB, BF, DA, DB, EA, EB, CB, CD, CE, CC, CF, FA, FB, FD, FE, FC, FF

Table 3.3. The set of FCISs of the ESG

Graph	FCISs
Open File	AD, AE, AC, AF, ABA, ABB, ABF, ABDA, ABDB, ABEA, ABEB, ABCB, ABCD, ABCE, ABCC, ABCF, AB(E + D)FA, AB(E + D)FB, AB(E + D)FD, AB(E + D)FE, AB(E + D)FC, AB(E + D)FF

After generating IPs, CISs, FIPs and FCISs, normally the test process is done such that (a) generating the test cases as CISs and FICSs, (b) inputting the test cases to transduce the system into a legal or illegal state, respectively and (c) observe the system output whether the output leads to a desired response or an undesired response which provides a good exception handling mechanism. In this thesis, the testing approach uses the steps (b) and (c), but for step (a), the test cases are generated by using DTs.

3.2. Modeling User Inputs by Decision Tables

Modeling user input also requires the understanding of the causal dependencies between the nodes of the ESG. This requirement expands the ESG model. Hence, for the nodes that need to be refined in ESG, DTs are used. As mentioned in Section 2.2.2, DTs are used for testing when a situation has to be modeled with cause and effects. It links conditions with actions depending on its rules which are represented as True (T) or False (F). Figure 3.4 shows the updated version of the ESG in Figure 3.2 where the refined nodes are displayed in double-circled shape.

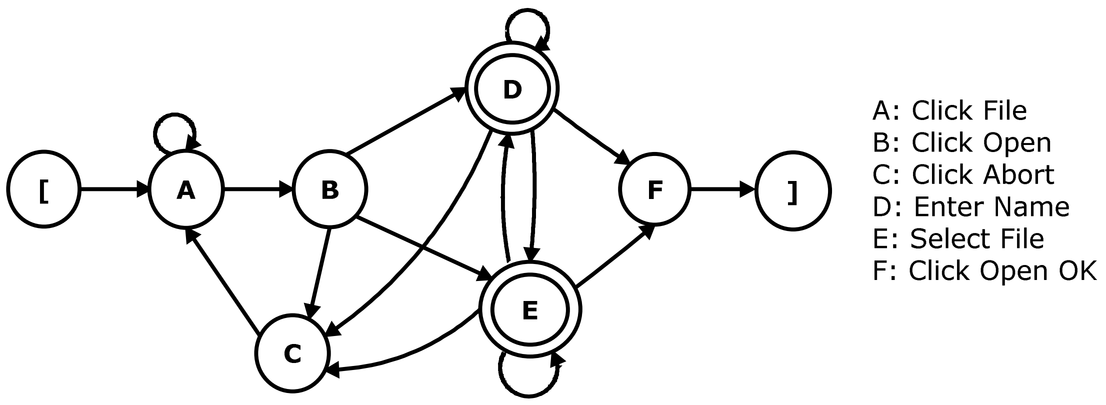


Figure 3.4. Updated ESG by double circling the refined nodes

Looking at the application in Figure 3.1 again, Name and File Selection fields that are represented with D and E in Figure 3.4 respectively have preconditions that have to be fulfilled before these processes can be done. For instance, it can be specified that the text editor accepts only the files with the ".txt" and ".rtf" extension and the maximum length that a file path can have is 255. Table 3.4 structures the corresponding decision process as a DT. The node "Enter Name" and "Select File" of Figure 3.4 is refined by the DT in Table 1.

Table 3.4. Decision Table for nodes "D" and "E" of Figure 3.4

Filename data	**R1**	**R2**	**R3**	**R4**
Filename extension is ".txt" or ".rtf"	T	T	F	F
File path length <= 255	T	F	T	F
Actions				
Open file	X	e	e	e

3.3. Test Case Generation Using Decision Tables

Nodes of an ESG represent either events or other ESGs or DTs. An ESG visualizes sequences of events and therefore allows detection of discrepancies in the sequential execution of user-system-interaction [4]. DTs augment the ESG given to support analysis of causal dependencies and restrictions of events. Especially, considering the pre-conditions, it could be possible to avoid vulnerabilities introduced by invalid inputs. Pre-conditions will ensure that the inputs taken from GUI are valid. For input validation, only pre-condition rules are entered into the DT. Hence, DT augmented ESGs are reduced and simplified by considering only the pre-conditions of the GUI inputs.

Equivalence class testing supplemented with boundary value analysis [5, 6] is used to generate test cases. Equivalence class testing partitions the input space into equivalence classes according to the input conditions. Test cases are designed by selecting at least one condition from each equivalence class. Boundary value analysis complements the equivalence partitioning by selecting test cases at the edges of a class [6]. Equivalence class testing is strengthened by the cause-effect testing approach which uses DTs to generate test cases where the input conditions represent the causes and actions represent the effects. A test case generation algorithm is developed to generate test case values from DTs by considering three validation types; namely isolated validation, interdependency validation, and service-specific validation [47], depicted in Figure 3.5.

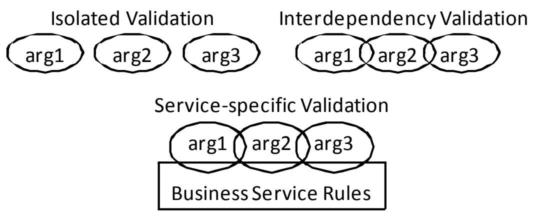


Figure 3.5. Input validation types [47]

Isolated validation checks boundary conditions (restrictions) and interdependency validation checks relations between the variables (dependencies). Service-specific validation considers conditions related to business or service. Consider the start and end port values in a port scanner as an example. For isolated validation, the considered variables should be between the port ranges (0-65535). The restriction that the start port value should be lower than the end port value is associated with interdependency validation. The dynamic and/or private ports are from 49152 through 65535 [48]. No ports can be registered in the dynamic range and it is commonly used by operating system kernels. The port allocations are only valid for the duration of the session of the connection. For service-specific validation, the port values between the dynamic ranges are not valid when the session is closed, although the values are inside the port ranges and the dependency requirement holds for the port values. Hence, input validation aspect supports all three types. Test case generation algorithm is represented below:

```
Input: Decision Table
Output: Test Cases
for each rule
        generate a test case by considering each variable in isolation
        modify the test case by considering the interdependencies between the variables
        modify the test case by considering the service-specific conditions
end for
```

For all the rules in the decision table, a specific test value for each variable is generated by regarding all its conditions and test values for all variables constitute the test case for that rule. First four columns in the decision table are for holding the type of the validation, variable, operator and boundary, which can be a value or a variable depending on the type of the validation, respectively.

The decision table that is used to generate the test cases using the test case generation algorithm in the case study section is shown in Table 5.1 and Table 5.4 for boundary overflow and buffer overflow respectively, and the test cases are displayed in Table 5.2 and Table 5.5 for boundary overflow and buffer overflow respectively. The algorithm creates a list for each constraint containing the conditions of the variables and generates the test values according to the following policy: First, test values are generated according to the boundary conditions for each variable in isolation. The generated values are alternating around the boundary values. Second, the relationships and dependencies for the variables are considered and the test values are altered according to the relation if needed. Finally, the test values are modified by considering the service-specific conditions.

CHAPTER 4

DETECTION AND CORRECTION OF OVERFLOW VULNERABILITIES IN GUI

4.1. Detecting and Correcting Vulnerabilities by Static Analysis

For vulnerability detection, static source analysis aims to find specific vulnerable code patterns, or detect a possible control-flow path on which the program reaches a vulnerable state. Unlike dynamic analysis, that is able to detect vulnerabilities in code that is reached during the normal operation of a program. Static source analysis also has its limitations. Static analysis tools rely on a static rule set of predetermined code patterns and conditions that they should detect, report and sometimes correct. These predetermined rules are developed by security experts and allow users with less security expertise to find bugs they would not otherwise be able to recognize [49]. However, a tool will only be able to report vulnerabilities that exist in its rule set. If a tool finds no bugs in a program, we cannot say that no bugs exist.

A static source analysis tool may not find all instances of a rule that has been written for a specific vulnerability. Any sufficiently complex analysis of a program is equivalent to solving the halting problem [50] that is provably undecidable [51]. Any static analysis that must return a result and do it within a reasonable amount of time will have to make approximations. As mentioned in the background section, this leads to false positives and false negatives in the output of the tool. One approach to minimizing the number of false negatives is to make the analysis more conservative as potentially vulnerable any code that is not provably safe [49]. A tool that guarantees the safety of a low percentage of your source, but reports the remaining high percentage (for instance, 99%) as vulnerable is meaningless. Making the analysis less conservative will lower the number of false positives but some vulnerability will remain undetected. The design of static analysis tools is influenced by finding as much vulnerability as possible, and at the same time, keeping the number of false positives low [49].

In this thesis, a detection algorithm which is based on the notion of static analysis is proposed to check the error handling mechanism of the SUC related to overflow vulnerabilities. As mentioned in the background section, the overflow vulnerabilities which we emphasized on are buffer and boundary overflow vulnerabilities. The detection algorithm which we proposed here is capable of finding both of the vulnerabilities, and also any type of vulnerability which can be represented as Boolean expressions.

The algorithm scans the source code statically, detects the points that may cause problems (possible violation of buffers and boundaries) and checks the error handling mechanism of the SUC against overflow vulnerabilities. The deficient parts of the error handling mechanism related to overflow vulnerabilities are identified and marked first. Once detected, a mechanism is required to correct the deficiencies. The correction algorithm provides an error handling mechanism through extension of source code by adding the pre-conditions where control for the overflow vulnerability does not exist.

Overflow detection algorithm consists of four steps. In step 1, pre-conditions, which are defined in the DT, are uploaded. Pre-conditions include Boolean expressions related to both buffer and boundary conditions. In the same DT, all of the conditions can be present. Pre-conditions are formed of:

- type: specifies the type of the condition (as mentioned in Section 3.3, the validation types are namely isolated validation (IS), interdependency validation (IN), and service-specific validation (SS)),
- variable: specifies the name of the variable to be matched to the variable that is defined throughout the source code of the program
- operator: is the Boolean operator
- boundary: specifies the predefined boundary of the defined variable. Boundary can be a numerical value or also another variable specifying that another variable is the boundary of the defined variable

In step 2, the variable definitions along with their specified types are tracked from the source code and displayed in a table. It is possible to limit the type of variables, since displaying all the variables in a program makes no sense if only some specified types are needed. For example, we can only limit and display the variables which have integer and unsigned integer types.

In step 3, the variables shown in the table are matched with the uploaded pre-conditions. The matched conditions are added to the condition list of the matched variable and from now on that variable will be traced according to its conditions that reside in its condition list.

Step 4 traces the variables from the source code and finds the first line that the variable is used. Then, the pre-conditions are compared with the conditions of the variable written in the source code. The conditions of the variables which hold the pre-conditions are marked with "checked" and the ones which don't hold are marked with "not checked".

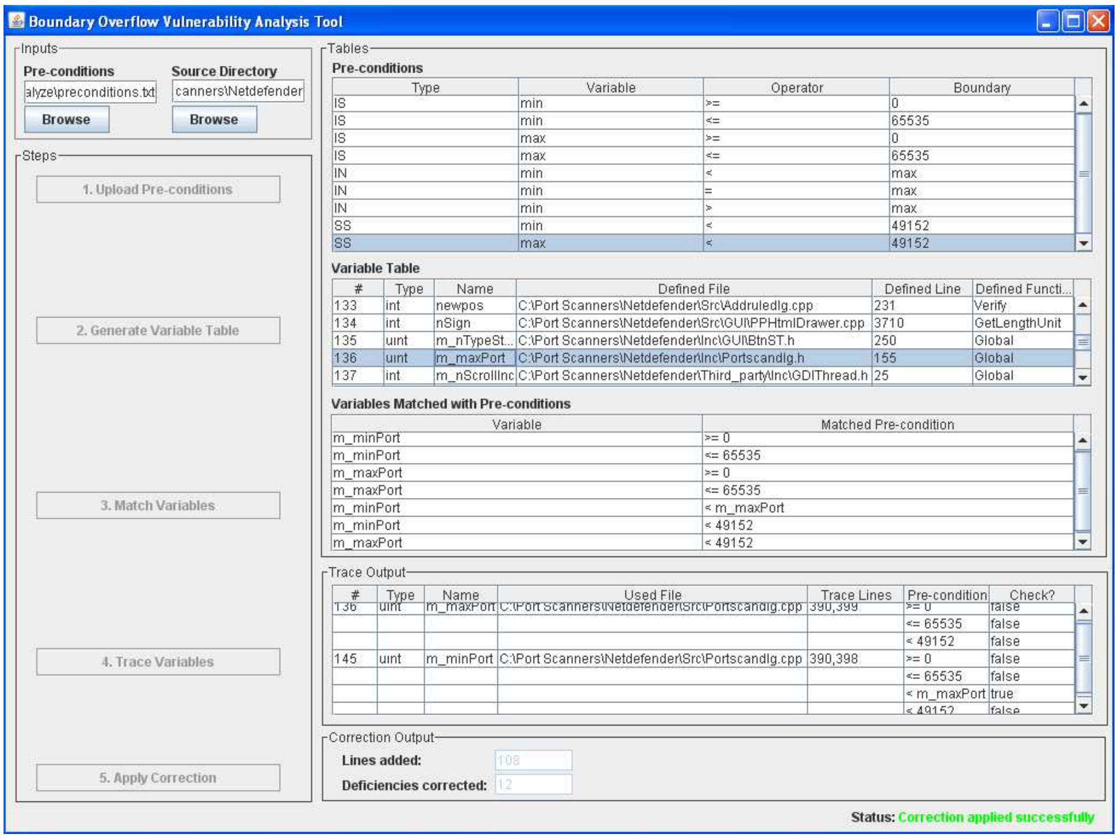
After detection algorithm is completed, correction algorithm may be executed. The correction mechanism extends the source code by inserting "Require" function (the function for implementing pre-conditions) as in [52]. The correction algorithm is applied to the variables which have conditions unchecked throughout the code, meaning that there is no control mechanism for that condition of the variable. The mechanism is applied after the trace line of the variable where the condition check for the variable does not exist.

4.2. Implementation and Tool Support

For the implementation of the overflow detection and correction approach as introduced in the previous subsection, an overflow vulnerability analysis tool is developed in Java. The tool runs in Microsoft Windows environment and works on software developed in C++. As a static analysis tool, it analyzes the source code of SUC, finds the deficient parts that may cause buffer and boundary overflow vulnerabilities and extends the source code by inserting pre-condition functions to ensure that the specified conditions hold before the inputs are processed. The class "Assertions" [52] that provides the functions required for emulating pre-conditions and post-conditions is used for exception handling, where in our case, only the pre-conditions are considered. Its "Require" function is used by our tool to be inserted where the deficiency of a control mechanism exists for overflow vulnerability.

The overflow vulnerability analysis tool takes two inputs: (1) the directory of the software to be analyzed and (2) DT for the pre-conditions that are related to GUI input elements. Our implementation requires a manual matching of the listed variables with the pre-conditions from DT augmented ESG model of the GUI. The tool outputs the the conditions of the variables as well as whether or not condition checks exist in the source code related to buffer or boundary overflow vulnerability. The correction mechanism is applied by informing the user about the insertion of the exception handling code where the pre-condition checks do not exist. Figure 4.1 shows GUI of the tool that enables to input source directory of the software to be checked, shows the detection steps, suggests corrections and displays the outputs.

As can be seen in the GUI of the overflow vulnerability analysis tool, the tool takes the source directory of the program and pre-conditions file as input and uploads the pre-conditions to the JTable element with type, variable, operator and boundary tabs. Then, by analyzing the source code, the variables are displayed in the JTable element by the type, name, defined file, defined line and defined function of the variable. After that, the variables are matched with the pre-conditions in the pre-conditions table. At least one matching operation has to be performed in order to enable the button of step 4; otherwise there is no point to trace the variables without any conditions. Thereafter, the variables are traced throughout the source code and a table is displayed that shows the result of the detection process by listing the type and name of the variable, the filename that the variable is used, its pre-conditions and the status if the check occurs in the source code for the given pre-condition or not. After displaying the detection output, the correction mechanism can be applied by pressing the step 5 button. The correction is performed and as an output, the number of lines added and deficiencies corrected are printed on the bottom part of the tool.



Boundary Overflow Vulnerability Analysis Tool

Inputs

Pre-conditions: alyze\preconditions.txt

Source Directory: canners\Netdefender

Steps

1. Upload Pre-conditions
2. Generate Variable Table
3. Match Variables
4. Trace Variables
5. Apply Correction

Tables

Pre-conditions

Type	Variable	Operator	Boundary
IS	min	>=	0
IS	min	<=	65535
IS	max	>=	0
IS	max	<=	65535
IN	min	<	max
IN	min	=	max
IN	min	>	max
SS	min	<	49152
SS	max	<	49152

Variable Table

#	Type	Name	Defined File	Defined Line	Defined Functi...
133	int	newpos	C:\Port Scanners\Netdefender\Src\Addruledlg.cpp	231	Verify
134	int	nSign	C:\Port Scanners\Netdefender\Src\GUI\PPHtmlDrawer.cpp	3710	GetLengthUnit
135	uint	m_nTypeSt...	C:\Port Scanners\Netdefender\Inc\GUI\BtnST.h	250	Global
136	uint	m_maxPort	C:\Port Scanners\Netdefender\Inc\Portscandlg.h	155	Global
137	int	m_nScrollInc	C:\Port Scanners\Netdefender\Third_party\Inc\GDIThread.h	25	Global

Variables Matched with Pre-conditions

Variable	Matched Pre-condition
m_minPort	>= 0
m_minPort	<= 65535
m_maxPort	>= 0
m_maxPort	<= 65535
m_minPort	< m_maxPort
m_minPort	< 49152
m_maxPort	< 49152

Trace Output

#	Type	Name	Used File	Trace Lines	Pre-condition	Check?
136	uint	m_maxPort	C:\Port Scanners\Netdefender\Src\Portscandlg.cpp	390,399	>= 0	false
					<= 65535	false
					< 49152	false
145	uint	m_minPort	C:\Port Scanners\Netdefender\Src\Portscandlg.cpp	390,398	>= 0	false
					<= 65535	false
					< m_maxPort	true
					< 49152	false

Correction Output

Lines added: 108

Deficiencies corrected: 12

Status: Correction applied successfully

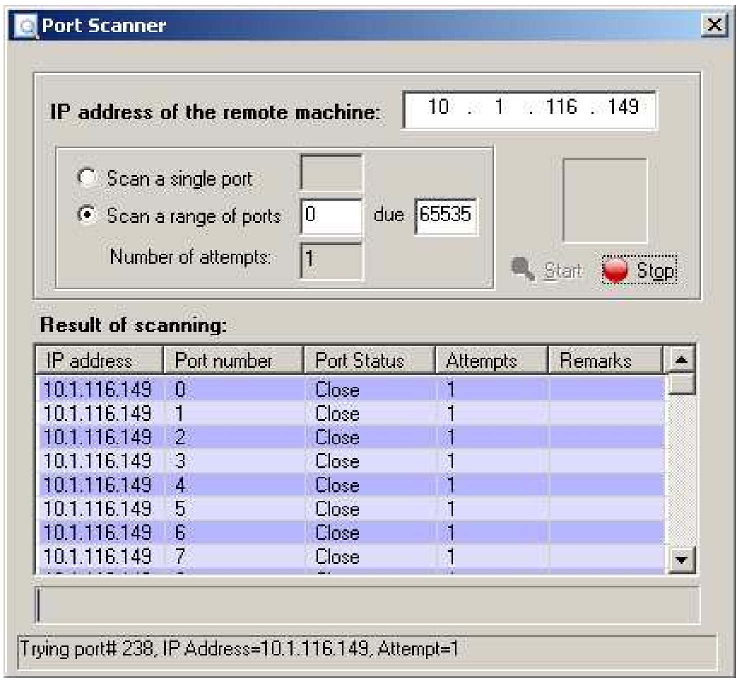
Figure 4.1. Overflow vulnerability analysis tool - GUI screen [28]

CHAPTER 5

CASE STUDY

The approach and the tool introduced in Section 4.2 are evaluated in this section by using a port scanner software as the SUC. A port scan function scans a single port or a range of ports, i.e., ports between a given start and end, to check whether they are open or not. The user interface behavior of the port scan function is modeled by using DT augmented ESG. Test cases are generated for start and end port from the decision table using test case generation algorithm presented in Section 3.3. Test of the port scan function is evaluated in a real network environment and faults have been recorded. As a next step, overflow vulnerability analysis tool analyzed the source directory to detect and correct the vulnerabilities related to boundary and buffer overflow. Finally, the faults detected before and after applying the overflow detection algorithm are compared.

The case study is performed on the basis of the port scanner part of open source firewall software, i.e., Netdefender Firewall (version 1.5) [53]. Its GUI is shown in Figure 5.1. The ESG model of the port scanner is given in Figure 11. The DTs given by Table 5.1 and Table 5.4 refines the related nodes of the ESG for boundary and buffer overflow vulnerabilities respectively, which are double-circled [4], e.g., the node labeled "Enter start&end ports" of Figure 5.2 is refined by Table 5.1 and Table 5.4.



Port Scanner

IP address of the remote machine: 10 . 1 . 116 . 149

Scan a single port

Scan a range of ports 0 due 65535

Number of attempts: 1

Start Stop

Result of scanning:

IP address	Port number	Port Status	Attempts	Remarks
10.1.116.149	0	Close	1	
10.1.116.149	1	Close	1	
10.1.116.149	2	Close	1	
10.1.116.149	3	Close	1	
10.1.116.149	4	Close	1	
10.1.116.149	5	Close	1	
10.1.116.149	6	Close	1	
10.1.116.149	7	Close	1	

Trying port# 238, IP Address=10.1.116.149, Attempt=1

Figure 5.1. Netdefender Port Scanner

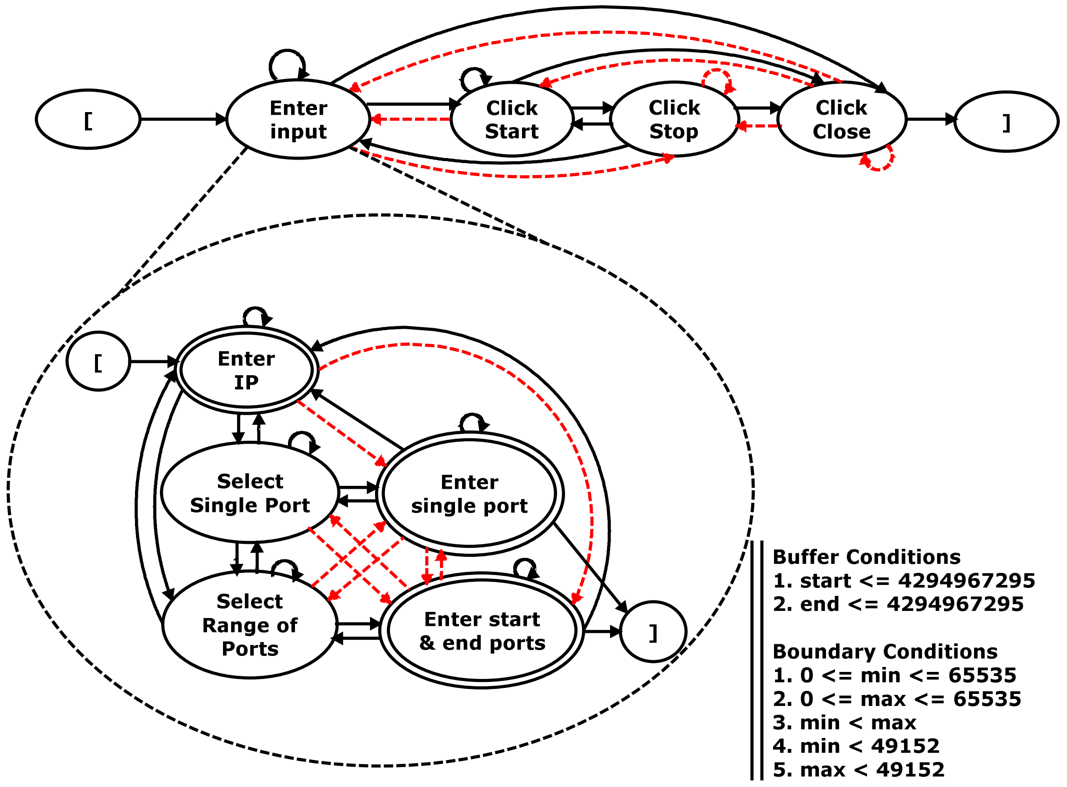


Figure 5.2. ESG model of the port scanner showing legal and illegal interaction pairs [28]

5.1. Boundary Overflow

In this section, the evaluation is performed by considering the conditions that are related to boundary overflow vulnerability. As mentioned in Section 2, boundary overflow vulnerabilities occur when the input being received by a system causes the system to exceed an assumed boundary resulting in vulnerability. Table 5.1 structures the decision process by modeling possible actions for boundary conditions. The DT is built to generate test data for the start and end port values of the port scanner according to the rules.

Test case generation algorithm is applied to generate test data according to the rules of the DT. For each rule, a test pair is generated based on equivalence class testing and boundary value approach. The constraints in the first part (rows 1-4) of the decision table indicate the boundary conditions. Meanwhile, the constraints in the rows 5-7 indicate the relations of the variables with each other. Furthermore, the constraints in the rows 8-9 are included due to service-specific constraints.

Table 5.1. Decision Table for "Enter start&end ports" of Figure 5.2 for boundary conditions [28]

	Conditions	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**	**R11**	**R12**	**R13**	**R14**	**R15**	**R16**	**R17**	**R18**	**R19**	**R20**	**R21**	**R22**	**R23**	**R24**
1	start >= 0	F	F	F	F	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
2	start <= 65535	T	T	T	T	T	T	F	F	F	F	F	F	T	T	T	T	T	T	T	T	T	T	T	T
3	end >= 0	F	F	F	T	T	T	F	T	T	T	T	T	F	F	T	T	T	T	T	T	T	T	T	T
4	end <= 65535	T	T	T	F	T	T	T	F	F	F	T	T	T	T	F	F	T	T	T	T	T	T	T	T
5	start < max	F	F	T	T	T	T	T	F	F	T	F	F	F	F	T	T	F	F	F	F	F	T	T	T
6	start = max	F	T	F	F	F	F	F	F	T	F	F	F	F	F	F	F	F	F	F	T	T	F	F	F
7	start > max	T	F	F	F	F	F	F	T	F	F	T	T	T	T	F	F	T	T	T	F	F	F	F	F
8	start < 49152	T	T	T	T	T	T	F	F	F	F	F	F	F	T	F	T	F	F	T	F	T	F	T	T
9	end < 49152	T	T	T	F	F	T	T	F	F	F	F	T	T	T	F	F	F	T	T	F	T	F	F	T
Actions																									
A1	Exception 1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X								
A2	Exception 2	X	X						X	X		X	X	X	X			X	X	X	X	X			
A3	Exception 3				X	X		X	X	X	X	X	X	X		X	X	X	X		X		X	X	
A4	Accept input																								X

The algorithm created a list for each constraint containing the conditions of the variables and generates the start and end port value test case pairs. Table 5.2 shows the generated test values as the output of the test data generation algorithm.

Table 5.2. Test Cases generated from rules of the Decision Table related to boundary conditions [28]

Rule	Test Values	Rule	Test Values
R1	(-1,-2)	**R13**	(49152,-1)
R2	(-1,-1)	**R14**	(0,-1)
R3	(-2,-1)	**R15**	(49152,65536)
R4	(-1,65536)	**R16**	(0,65536)
R5	(-1,49152)	**R17**	(49153,49152)
R6	(-1,0)	**R18**	(49152,0)
R7	(65536,-1)	**R19**	(1,0)
R8	(65537,65536)	**R20**	(49152,49152)
R9	(65536,65536)	**R21**	(0,0)
R10	(65536,65537)	**R22**	(49152,49153)
R11	(65536,49152)	**R23**	(0,49152)
R12	(65536,0)	**R24**	(0,1)

The port scanner is evaluated in a LAN and the generated test values are applied as inputs to the GUI of the port scanner. The user interface outputs are obtained and the network packet outputs are captured. The outputs are extracted to a spreadsheet document. Table 5.3 shows a sample view of the spreadsheet document. The document displays the test values as input pair, GUI and network packet outputs, state of the case (erroneous or not), and the error message.

Table 5.3. Outputs of the test cases #1 [28]

#	Input Pair	GUI Output	Network Packet	Erroneous?	Error Message?
1	(-1,-2)	No output	No packet	Yes	Message 1
2	(-1,-1)	(-1,0,1,2,3,...∞)	65535,1,2,...65535…	Yes	No
3	(-2,-1)	(-2,-1,0,1,2,3,...∞)	65534,65535,1,2,...65535…	Yes	No
4	(-1,65536)	No output	No packet	Yes	Message 1
5	(-1,49152)	No output	No packet	Yes	Message 1
6	(-1,0)	No output	No packet	Yes	Message 1
7	(65536,-1)	(65536...∞)	1,2,...65535,1,2,....65535…	Yes	No
8	(65537,65536)	No output	No packet	Yes	Message 1
9	(65536,65536)	(65536)	No packet	Yes	No
10	(65536,65537)	(65536,65537)	1	Yes	No
11	(65536,49152)	No output	No packet	Yes	Message 1
12	(65536,0)	No output	No packet	Yes	Message 1
13	(49152,-1)	(49152...∞)	49152,49153,...65535…	Yes	No
14	(0,-1)	(0...∞)	1,2,...65535,1,2,...65535…	Yes	No
15	(49152,65536)	(49152...65536)	49152,49153,…65535	Yes	No
16	(0,65536)	(0...65536)	1,2,…65535	Yes	No
17	(49153,49152)	No output	No packet	Yes	Message 1
18	(49152,0)	No output	No packet	Yes	Message 1
19	(1,0)	No output	No packet	Yes	Message 1
20	(49152,49152)	(49152)	49152	No	
21	(0,0)	(0)	No packet	No	
22	(49152,49153)	(49152,49153)	49152,49153	No	
23	(0,49152)	(0…49152)	1,2,…49152	No	
24	(0,1)	(0,1)	1	No	

Message 1: “The maximum range cannot be less than the minimum one”.

After the evaluation, it is observed that the control mechanism of the port scanner that is related to out of boundary values is deficient for the port scanner of Netdefender firewall. Hence, the overflow vulnerability analysis tool inserted control statements to fulfill the deficiencies of the software. After the insertion of control statements related to boundary constraints in the port scanner of Netdefender firewall, the software is evaluated in LAN again and the generated test cases are applied as inputs to the GUI of the port scanner. The outputs considerably differ from the ones in Table 5.3. In erroneous cases (1-19), the software outputs the right error message and aborts sending the packets.

5.2. Buffer Overflow

In this section, the evaluation is performed by considering the conditions that are related to buffer overflow vulnerability which is a type boundary overflow vulnerability. As mentioned in Section 2, buffer overflow vulnerabilities occur when a string of characters or numbers of unchecked length is entered into a program. The conditions that the start and end port values can have are related to the maximum length that the variable type can have (in the case of port scanner, the type of the variable is “unsigned integer” and its maximum length is 4,294,967,295) which the inputs from the GUI as start and end port values will be assigned.

Table 5.4. Decision Table for “Enter start&end ports” of Figure 5.2 for buffer conditions

Conditions	**R1**	**R2**	**R3**	**R4**
start <= 4,294,967,295	F	F	T	T
end <= 4,294,967,295	F	T	F	T
Actions				
A1: Exception	X	X	X	
A2: Accept input				X

Table 5.4 structures the decision process by modeling possible actions for related conditions. The decision table is built to generate test data for the start and end port values of the port scanner according to the rules related to buffer overflow vulnerability. As the previous subsection, Test case generation algorithm is applied to

generate test data according to the rules of the decision table. The algorithm created a list for each constraint containing the conditions of the variables and generated the start and end port value test case pairs. Table 5.5 shows the generated test values as the output of the test data generation algorithm.

Table 5.5. Test Cases generated from rules of the Decision Table related to buffer conditions

Rule	Test Values
R1	(4294967296, 4294967296)
R2	(4294967296, 4294967295)
R3	(4294967295, 4294967296)
R4	(0, 4294967295)

The port scanner is evaluated in a LAN and the generated test values are applied as inputs to the GUI of the port scanner as in the previous subsection. The user interface outputs are obtained and the network packet outputs are captured. The outputs are extracted to a spreadsheet document which is shown in Table 5.6 as the test values as input pair, GUI and network packet outputs, state of the case (erroneous or not), and the error message.

Table 5.6. Outputs of the test cases #2

#	Input Pair	GUI Output	Network Packet	Erroneous?	Error Message?
1	(4294967296, 4294967296)	No output	No packet	Yes	No
2	(4294967296, 4294967295)	No output	No packet	Yes	No
3	(4294967295, 4294967296)	No output	No packet	Yes	No
4	(0, 4294967295)	(0…∞)	1,2,...65535,1,2,....65535…	Yes	No

The case study shows that the overflow vulnerability analysis tool has successfully carried out detection and correction operations. Analysis of the evaluation results encourages the generalization that boundary and buffer overflow vulnerabilities are not considered and thus countermeasure actions are neglected during software development. Therefore, the tool that is introduced in this thesis might be useful to prevent likely failures or undesirable situations that may occur as a consequence of deficiency control mechanism in the software.

CHAPTER 6

CONCLUSION

In this thesis, a solution for the GUI-based overflow vulnerability problem is proposed and implemented. GUI-based overflow vulnerabilities that are examined in this thesis are buffer overflow and boundary overflow vulnerabilities. Buffer overflow is a type of vulnerability where a process which in our case is the input entered from the GUI element stores data in a buffer outside the memory that the developer allocated for. A boundary overflow is an input error and occurs when values are entered that violate the range of values where in our case, the buffer overflow vulnerability can be recognized as a subset of boundary overflow vulnerability.

First, the GUI elements which may cause the overflow vulnerability is modeled by DT augmented ESGs. Once the elements have been modeled, the rules of the DT which are combinations of Boolean expressions are used to generate the test cases. Equivalence class partitioning and boundary value analysis approaches supported the test case generation process. Then, the SUC is tested manually in real environment by entering the test case values to its user interface. The faults are obtained and extracted manually to a file. A detection and correction algorithm is introduced to validate the error handling mechanism of SUC related to overflow vulnerabilities and provide it with necessary exception handling mechanism where none exists. The deficiencies that are caused by the unchecked input which are entered to the system through the GUI elements are detected by analyzing the source code statically and corrected by static code insertion into the source code of the program. For the implementation of the approach, an overflow vulnerability analysis tool is implemented and developed in Java programming language. After correcting the deficiencies in the source code related to overflow vulnerabilities, the software is tested manually in real environment again with the same test cases as in the previous test run. Port scanner software which is coded in C++ has been tested for the evaluation of the overflow vulnerability analysis tool.

Results of tests showed that the approach is very effective for finding deficiencies in the error handling mechanism of SUC concerning boundary and buffer overflow vulnerabilities. The overflow vulnerability analysis tool has successfully carried out detection and correction operations in the source code related to overflow vulnerabilities. It is observed from the analysis that the overflow vulnerabilities like boundary and buffer overflow are not considered and neglected throughout the software development. Hence, the overflow vulnerability analysis tool that is introduced in this thesis might be the solution to prevent the undesirable situations that may occur as a result of the deficiencies in the software related to overflow vulnerabilities.

Main contributions of this thesis are:

- The testing approach introduced includes an algorithm to effectively generate test cases from a seed DT constructed from pre-conditions and applying equivalence class partitioning and boundary value techniques.
- Detection algorithm is introduced and implemented to check the error handling mechanism of the SUC related to overflow vulnerabilities.
- Correction algorithm extends the source code of the SUC with the error handling mechanism for the variables which have conditions unchecked throughout the code, meaning that there is no control mechanism for that condition of the variable.

Future work aims at further reduction of the manual test effort and a better self adaptability of the model due to changes in applications. Furthermore, making experiments with various types of network software and different types of programming language as the source code for SUC, the efficiency of the approach can be determined more specifically.

REFERENCES

[1] P. Amman, J. Offutt, “Introduction to software testing”, *Cambridge University Press*, 2008.

[2] F. Belli, “Finite state testing and analysis of graphical user interfaces”, in *Proceedings of the 12th International Symposium on Software Reliability Engineering*, Washington, DC, USA, 2001, pp. 34-43, IEEE.

[3] P. Mell, M. C. Tracy, “Procedures for handling security patches”, *NIST Special Publication 800-40*, 2002.

[4] F. Belli, M. Linschulte, “On Negative Tests of Web Applications”, *Annals of Mathematics, Computing & Teleinformatics*, vol. 1, no. 5, 2007, pp. 44-56.

[5] T. Tuglular, “Test Case Generation for Firewall Implementation Testing using Software Testing Techniques”, in *Int. Conf. on Sec. of Inf. and Networks*, N. Cyprus, 2007.

[6] H. Liu, H. B. Kuan Tan, “Covering code behavior on input validation in functional testing”, *Information and Software Technology*, vol. 51, no. 2, 2009, pp. 546-553.

[7] CERT/CC. (2004). Advisories. [Online]. Avaliable: http://www.cert.org/advisories

[8] J. J. Tevis, J. A. Hamilton, “Methods for the prevention, detection and removal of software security vulnerabilities”, in *Proceedings of the 42nd Annual Southeast Regional Conference,* ACM, New York, NY, 2004, pp. 197-202.

[9] K. Kratkiewicz, R. Lippmann, “A Taxonomy of Buffer Overflows for Evaluating Static and Dynamic Software Testing Tools”, in *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*, NIST Special Publication 500-265, National Institute of Standards and Technology, 2005, pp. 44–51.

[10] ISO 5806, “Specification of single-hit decision tables”, *Information processing*, 1984.

[11] J. H. Hayes, J. Offutt, “Input validation analysis and testing”, *Empirical Software Engineering*, vol. 11, no. 4, 2006, pp. 493-522.

[12] MSDN. (2009, Dec.). Design guidelines for secure web application. [Online]. Avaliable: http://msdn.microsoft.com/library/default.asp?url=/library/enus/ secmod/html/secmod77.asp

[13] P. Jorgensen, “Software testing: a craftman’s approach”, *CRC Press*, 2002, pp. 359.

[14] F. Belli, A. Hollmann, N. Nissanke, “Modeling, analysis and testing of safety issues – an event-based approach and case study”, in *Proceedings of the 26th Int. Conf. Computer Safety, Reliability, and Security*, Springer, 2007, pp. 276-282.

[15] P.A. Brooks, A. M. Memon, “Automated gui testing guided by usage profiles”, in *Proceedings of the Twenty-Second IEEE/ACM international Conference on Automated Software Engineering,* ACM, New York, NY, 2007, pp. 333-342.

[16] A. M. Memon, M. E. Pollack, M. L. Soffa, “Automated test oracles for GUIs”, in *Proceedings of the 8th ACM SIGSOFT international Symposium on Foundations of Software Engineering: Twenty-First Century Applications* (San Diego, California, United States, November 06 - 10, 2000), ACM, New York, NY, 2000, pp. 30-39.

[17] A. M. Memon, M. L. Soffa, “Regression testing of GUIs”, in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT international Symposium on Foundations of Software Engineering*, ACM, New York, NY, 2003, pp. 118-127.

[18] A. M. Memon, “Automatically repairing event sequence-based GUI test suites for regression testing”, *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 2, 2008, pp. 1-36.

[19] K. Cai, L. Zhao, H. Hu, C. Jiang, “On the Test Case Definition for GUI Testing”, in *Proceedings of the Fifth international Conference on Quality Software*, IEEE Computer Society, Washington, DC, 2005, pp. 19-28.

[20] A. M. Memon, M. L. Soffa, M. E. Pollack, “Coverage criteria for GUI testing”, *SIGSOFT Softw. Eng. Notes,* vol. 26, no. 5, 2001, pp. 256-267.

[21] B. A. Myers, D. R. Olsen, Jr., J. G. Bonar, “User interface tools”, In *Proceedings of ACM INTERCHI ‘93 Conference on Human Factors in Computing Systems – Adjunct Proceedings, Tutorials, 1993,* p. 239.

[22] B. A. Myers, “Why are human-computer interfaces difficult to design and implement?”, *Technical Report CS-93-183*, Carnegie Mellon University, School of Computer Science, July 1993.

[23] W. I. Wittel, Jr., T. G. Lewis, “Integrating the MVC paradigm into an object-oriented framework to accelerate GUI application development”, *Technical Report 91-60-06*, Department of Computer Science, Oregon State University, 1991.

[24] B. A. Myers, J. D. Hollan, I. F. Cruz, “Strategic Directions in Human Computer Interaction”, *ACM Computing Surveys*, vol. 28, no. 4, 1996, pp. 794-809.

[25] D. T. Wick, N. M. Shehad, A. R. Hajare, “Testing the Human Computer Interface for the Telerobotic Assembly of the Space Station", in *Proceedings of the Fifth International Conference on Human-Computer Interaction*, vol. 1 of *II. Special Applications*, 1993, pp. 213-218.

[26] L. White, "Regression Testing of GUI Event Interactions", in *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society Press, Washington, 1996, pp. 350-358.

[27] A. M. Memon, M. E. Pollack, M. L. Soffa, "A planning-based approach to GUI testing", in *Proceedings of The 13th International Software/Internet Quality Week*, May 2000.

[28] T. Tuglular, C. A. Muftuoglu, F. Belli, and M. Linschulte, "Event-Based Input Validation Using Design-by-Contract Patterns", *The 20th annual International Symposium on Software Reliability Engineering (ISSRE 2009)*, Mysuru, India, 2009.

[29] M. Zitser, "Securing Software: An Evaluation of Static Source Code Analyzers", *Master's Thesis*, Massachusetts Institute of Technology, Cambridge, MA, 130 pages, 2003.

[30] D. A. Wheeler. (2009, Nov.). Flawfinder. [Online]. Avaliable: http://www.dwheeler.com/flawfinder

[31] J. Viega, J. T. Bloch, T. Kohno, G. McGraw, "ITS4: A static vulnerability scanner for C and C++ code", *ACM Transactions on Information and System Security*, vol. 5, no. 2, 2002.

[32] J Foster, "Type qualifiers: Lightweight specifications to improve software quality", *Ph.D. thesis*, University of California, Berkeley, 2002.

[33] D. Evans (2009, Oct.). Splint. [Online]. Avaliable: http://www.splint.org

[34] H. Chen, D. Wagner, "MOPS: an infrastructure for examining security properties of software", *Tech. Rep. UCB//CSD-02-1197*, University of California, Berkeley, 2002.

[35] B. Chess, G. McGraw, "Static Analysis for Security", *IEEE Security and Privacy*, vol. 2, no. 6, 2004, pp. 76-79.

[36] G. Holzmann, "UNO: Static source code checking for user-defined properties", *Bell Labs Technical Report*, Bell Laboratories, Murray Hill, NJ, 27 pages.

[37] 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 (Washington D.C., USA, October 27 - 30, 2003), CCS '03*, ACM, New York, NY, 2003, pp. 345-354.

[38] L. Kolmonen, "Securing Network Software using Static Analysis", *Seminar on Network Security*, Helsinki University of Technology, 2007.

[39] A. I. Sotirov, “Automatic Vulnerability Detection using Static Source Code Analysis”, *MS Thesis*, University of Alabama, 2005.

[40] J. R. C. Patterson, “Accurate static branch prediction by value range propagation”, in *Proc. of SIGPLAN Conf. on prog. language design and implementation*, 1995, pp. 67–78.

[41] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, S. Kuo, “Securing web application code by static analysis and runtime protection”, in *Proceedings of the 13th international Conference on World Wide Web (New York, NY, USA, May 17 - 20, 2004), WWW '04*, ACM, New York, NY, 2004, pp. 40-52.

[42] N. Jovanovic, C. Kruegel, E. Kirda, “Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)”, in *Proceedings of the 2006 IEEE Symposium on Security and Privacy (May 21 - 24, 2006), SP*, IEEE Computer Society, Washington, DC, 2006, pp. 258-263.

[43] M. D. Ernst, "Static and dynamic analysis: synergy and duality", in *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 07-08 June 2004.

[44] D. Engler, “Static analysis versus model checking for bug finding”, in *CONCUR 2005 - Concurrency theory*, Lecture Notes In Computer Science, vol. 653. Springer-Verlag, London, 2005, pp. 1-1.

[45] R. David, P. Thevenod-Fosse, "Detecting Transition Sequences: Application to Random Testing of Sequential Circuits", in *Proc. Int. Symp. Fault-Tolerant Computing FTCS-9*, 1979, pp. 121-124.

[46] S. Naito, M. Tsunoyama, “Fault Detection for Sequential Machines by Transition Tours”, *in Proc. of FTCS*, 1981, pp. 238-243.

[47] A. Stevenson, “Aspect-Oriented Smart Proxies in Java RMI”, *Thesis for Master of Mathematics in Computer Science*, University of Waterloo, Ontario, Canada, 2008.

[48] IANA. (2009, Feb.). Port Numbers. [Online]. Available: http://www.iana.org/assignments/port-numbers

[49] A. I. Sotirov, "Automatic Vulnerability Detection Using Static Source Code Analysis", *Master Thesis*, Department of Computer Science, University of Alabama, Alabama, USA, 2005.

[50] H. Rice, “Classes of recursively enumerable sets and their decision problems”, *Transactions of the American Mathematical Society 83*, 1953.

[51] A. Turing, “On computable numbers, with an application to the Entscheidungsproblem”, in *Proceedings of the London Mathematical Society*, vol. 42, 1937, pp. 230–265.

[52] P. Guerreiro, “Simple Support for Design by Contract in C++”, in *Proc. of the 39th Int. Conf. and Exhibition on Technology of Object-Oriented Languages and Systems*, IEEE, Washington, DC, 2001.

[53] Netdefender. (2009, March). Netdefender firewall version 1.5. [Online]. Available: http://www.codeplex.com/netdefender