Two complementary approaches to detecting vulnerabilities in C programs

Willy Ronald Jimenez Freitez

DOCTORAT EN CO-ACCREDITATION
TELECOM SUDPARIS ET L'UNIVERSITE EVRY VAL D'ESSONNE

Spécialité : Informatique

Ecole Doctorale : Sciences et Ingénierie

Présentée par : Willy JIMENEZ

Pour obtenir le grade de

DOCTEUR DE TELECOM SUDPARIS

Two Complementary Approaches to Detecting Vulnerabilities in C Programs

---

Soutenue le 04 octobre 2013 devant le jury composé de :

Directrice de thèse :	
Mme. Ana CAVALLI	Professeur, Telecom SudParis
Rapporteurs :	
Mr. Manuel NUÑEZ	Professeur, Université Complutense de Madrid
Mme. Hélène WAESELYNCK	Chargée de Recherche (HDR), LAAS-CNRS Toulouse
Encadrant :	
Mme. Amel MAMMAR	Maître de Conférences (HDR), Telecom SudParis
Examinateur :	
Mr. Sylvain PEYRONNET	Professeur, Université de Caen Basse-Normandie

Thèse n° 2013TELE0017

Contents

List of Figures

List of Tables

Résumé

De manière générale, en informatique, les vulnérabilités logicielles sont définies comme des cas particuliers de fonctionnements non attendus du système menant à la dégradation des propriétés de sécurité ou à la violation de la politique de sécurité. Ces vulnérabilités peuvent être exploitées par des utilisateurs malveillants comme brèches de sécurité. Comme la documentation sur les vulnérabilités n'est pas toujours disponible pour les développeurs et que les outils qu'ils utilisent ne leur permettent pas de les détecter et les éviter, l'industrie du logiciel continue à être paralysée par des failles de sécurité. C'est pourquoi, la détection de ces vulnérabilités dans le logiciel est devenue une préoccupation majeure.

Les vulnérabilités logicielles proviennent des failles potentielles dans la conception des programmes mais également des erreurs commises dans leurs mises en œuvre comme l'utilisation abusive des aspects dangereux et source d'erreurs du langage de programmation. Pour le langage C par exemple, les vulnérabilités sont dues essentiellement à l'arithmétique des pointeurs, le manque d'un type de base pour les chaînes de caractères ou aussi l'absence de vérification des bornes des tableaux.

Nos travaux de recherche s'inscrivent dans le cadre du projet Européen SHIELDS [1] et portent plus particulièrement sur les techniques de modélisation et de détection formelles de vulnérabilités. Dans ce domaine, les approches existantes sont peu nombreuses et ne se basent pas toujours sur une modélisation formelle précise des vulnérabilités qu'elles traitent [15, 18, 26]. De plus, les outils de détection sous-jacents produisent un nombre conséquent de faux positifs/négatifs [2]. Notons également qu'il est assez difficile pour un développeur de savoir quelles vulnérabilités sont détectées par chaque outil vu que ces derniers sont très peu documentés.

En résumé, les contributions réalisées dans le cadre de cette thèse sont les suivantes:

1. Définition d'un formalisme tabulaire de description de vulnérabilités appelé *template*.

1. http://er-projects.gf.liu.se/
2. Un faux positif (resp. négatif) est la détection (resp. non détection) à tord d'une vulnérabilité

Ce formalisme a l'intérêt de faciliter la communication entre les différents acteurs de l'équipe du développement.

2. Définition d'un langage formel, appelé *Condition de Détection de Vulnérabilité* (VDC), qui permet de modéliser avec précision l'occurrence d'une vulnérabilité. Une approche de génération de VDCs à partir des templates a été également définie. Enfin, la génération de VDCs permet une détection automatique de vulnérabilités par le test passif sur les traces d'execution du programme.
3. Définition d'une deuxième approche de détection de vulnérabilités combinant le model checking et l'injection de fautes.
4. Evaluation des deux approches sur des études de cas.

Abstract

In general, computer software vulnerabilities are defined as special cases where an unexpected behavior of the system leads to the degradation of security properties or the violation of security policies. These vulnerabilities can be exploited by malicious users or systems impacting the security and/or operation of the attacked system. Since the literature on vulnerabilities is not always available to developers and the used tools do not allow detecting and avoiding them; the software industry continues to be affected by security breaches. Therefore, the detection of vulnerabilities in software has become a major concern and research area. Software vulnerabilities result from potential flaws in the design of programs and also from errors in their implementation as the misuse of dangerous aspects and error-prone programming languages. In the case of the C programming language, for example, vulnerabilities are primarily due to pointer arithmetic, the lack of a basic type for strings and the lack of bounds checking.

Our research was done under the scope of the SHIELDS [3] European project and focuses specifically on modeling techniques and formal detection of vulnerabilities. In this area, existing approaches are limited and do not always rely on a precise formal modeling of the vulnerabilities they target [15, 18, 26]. Additionally detection tools produce a significant number of false positives/negatives. Note also that it is quite difficult for a developer to know what vulnerabilities are detected by each tool because they are not well documented.

Under this context the contributions made in this thesis are:

1. Definition of a formalism called template, created to capture the description of vulnerabilities given in natural language or by a vulnerability model. This formalism has the advantage of facilitating the communication between the different actors of the development team.
2. Definition of a formal language, called Vulnerability Detection Condition (VDC), which can accurately model the occurrence of a vulnerability. Also a method to

3. http://er-projects.gf.liu.se/

generate VDCs from templates has been defined.

3. Defining a second approach for detecting vulnerabilities which combines model checking and fault injection techniques.
4. Experiments: both approaches were evaluated with particular case studies, results showed that the use of VDCs for vulnerability detection is promising due to the use of a repository to store the vulnerability representation and/or the instantiation used by the detection tool.

Chapter 1

Introduction

1.1 Context and Motivations

In computer science, software vulnerabilities are generally defined as specific instances of not intended functionality in a certain software/system that might lead to degradation of security properties or the violation of the security policy. Considering that most of the actual systems are interconnected through Internet or mobile devices, interacting with other systems or users, then the possibility that vulnerabilities can be exploited by malicious code or misuse of the system is a major concern.

Software vulnerabilities arise from deficiencies in the design of computer programs or mistakes in their implementation. An example of a design flaw was the Solaris sadmin service, which allowed any unprivileged user to forge their security credentials and execute arbitrary commands as root compromising the whole system. In order to solve this kind of problem a redesign and reinforce of the use of a stronger authentication mechanism was needed. Vulnerabilities of this kind are harder and costly to fix, but fortunately they are rare. In practice, most software vulnerabilities are a result of programming mistakes, in particular the misuse of unsafe and error-prone features of the programming language, in the case of C language: pointer arithmetic, lack of a native string type and lack of array bounds checking.

Though the causes of software vulnerabilities are not much different from the causes of software defects in general, their impact is more severe. A user might be willing to

save his work more often in case a program crashes, but there is little they can do to lessen the consequences of a security compromise. This makes the problem of detecting existing vulnerabilities and preventing new ones an important task for software developers. Although efforts are being made to reduce security vulnerabilities in software, we note in published statistics that the number of vulnerabilities and the number of computer security incidents resulting from exploiting vulnerabilities are growing [10]. One of the reasons for this is that information on known vulnerabilities is not easily available to software developers or integrated into the tools they use.

Under this context, European project SHIELDS [1] was launched with the purpose of increasing software security knowledge by reducing the gap between software security experts and practitioners. The idea is to help software developers to effectively prevent the occurrences of known vulnerabilities when building software. To do that, a Vulnerability Repository Service (SVRS) is available online to software developer in order to facilitate the dissemination of vulnerability knowledge; including for instance formalisms for representing security information that improve the vulnerability comprehension and/or some tools that can be use for detection.

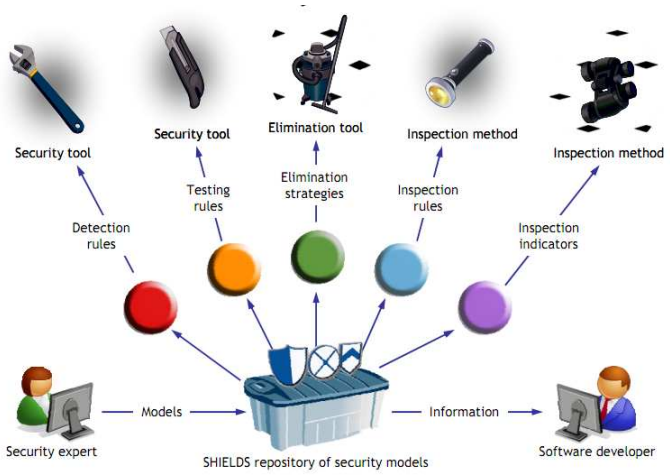


Figure 1.1: SHIELDS Repository Schema

SHIELDS project's addressed some of the problems encountered in the design and development of secure software like:

1. The lack of a common vulnerabilities database that can be shared between security

experts and software developers.

2. The lack of powerful integrated development environment that permits to prevent occurrences of known vulnerabilities when building software.
3. The lack of detection methods and tools that detect sufficiently many (or even all) known vulnerabilities, yet do not generate too many false positives.
4. The lack of rich formalisms tackle the problem of vulnerability detection that combines tool based automation with human skills.

1.2 Contributions

The present thesis, achieved in the context of mentioned European SHIELDS project contributes with:

- Definition of a graphical representation of a software vulnerability, called *template*, that facilitates the communication with software developers. Roughly speaking, this template describes how the vulnerability occurs and under which conditions.
- Definition of a formal language, called *Vulnerability Detection Conditions* (VDCs), to describe the presence of a vulnerability without any ambiguity. A systematic approach to produce VDCs from templates describing a vulnerability is also provided. The generated VDCs are then used to an effective detection of vulnerabilities following a dynamic code analysis based on the passive testing technique that check the presence of vulnerabilities on its execution traces. A graphical tool to help the software users elaborate their VDC has been developed in collaboration with the Montimage SME.
- Definition of a model-checking based approach, using SPIN [21], to detect vulnerabilities. This approach consists in extracting the information, relevant for vulnerability detection, from the C code to verify and mapping it into PROMELA [21], the input language of SPIN, in order to verify the presence/absence of vulnerabilities. Each counterexample returned by SPIN denotes a potential vulnerability that should be confirmed by applying the active testing technique and injecting the related values into the C code and observing how it behaves.

– Evaluation of the developed approaches on some case studies.

1.3 Publications

– Nahid Shahmehri, Amel Mammar, Edgardo Montes de Oca, David Byers, Ana Cavalli, Shanai Ardi and Willy Jimenez, "An Advanced Approach for Modeling and Detecting Software Vulnerabilities", Journal Information and Software Technology, vol 54, issue 9, September 2012.
– Amel Mammar, Ana Cavalli, Willy Jimenez, Wissam Mallouli and Edgardo Montes de Oca. "Using testing techniques for vulnerability detection in C programs", 23th IFIP Int. Conference on Testing Software and Systems (ICTSS) 2011, November 7-10, Paris, France. Best Paper Award.
– Wissam Mallouli, Amel Mammar, Ana Cavalli and Willy Jimenez. "VDC-Based Dynamic Code Analysis: Application to C Programs", Published in the international Journal of Internet Services and Information Security (JISIS). Volume 1, Issue 2/3, pages 4-20, August 2011.
– Amel Mammar, Ana Cavalli, Natalia Kushik, Willy Jimenez, Nina Yevtushenko and Edgardo Montes de Oca. "A SPIN-based Approach for Detecting Vulnerabilities in C Programs", Second Workshop on Program Semantics, Specification and Verification: Theory and Applications (PSSV 2011) ST-Petersburg, Russia, July 12-13, 2011.
– Willy Jimenez , Amel Mammar, Ana Cavalli. "Software Vulnerabilities, Prevention and Detection Methods: A Review", SEC-MDA workshop, 24 June 2009, Enschede, The Netherlands.

1.4 Organization of the Thesis

This thesis is organized as follows:

– The first chapter describes the common vulnerabilities that may occur in software. How they occur, how they can be modeled and how to detect them. Static detection techniques works directly on the source code without executing it, while dynamic ones require to run software to perform the detection. Each of these techniques has

its advantages and limits. An emphasis is done on the models covered by SHIELDS project.

- Second chapter presents the formal language, *Vulnerability Detection Condition* (VDC), which was defined to describe the occurrence of a vulnerability without any ambiguity. A systematic approach is defined in order to derive the VDCs from a graphical modeling language called *Vulnerability Cause Graph* (VCG). Then, a dynamic detection method using these VDCs and based on passive testing is presented.
- Fourth chapter presents an exploration of a model checking-based approach to detect vulnerability in a program developed with C programming language. The approach translates the original C code into Promela language; vulnerabilities are describe as assertions in Promela also, and then SPIN model checker [20] is used to detect the vulnerability. A set of translation rules to map a subset of C code into Promela are described.
- Fifth chapter presents some practical results based on proposed approaches.
- Finally, last chapter concludes the present work and establishes some potential future work.

Chapter 2

State of the Art

Contents

In order to perform their tasks software systems interact with other systems, the users or their environment to obtain the required information. If inputs are not properly processed and validated before being used inside the program then they might cause an unexpected behavior of the program, or even worse, of the system where the program is running. Such condition may be exploited by an attacker for its benefit and he may access critical data; impersonate a real user and/or damage the system.

This chapter presents the most frequent and known software vulnerabilities, also some formalisms to describe vulnerabilities together with the different existing methods and tools to deal with are provided. We are particularly interested on vulnerabilities related to the C programming language and the graphical formalisms used to describe them. Concerning detection techniques, we describe static and dynamic approaches which are the most used in the literature. This state of the art of software vulnerabilities has been published in [24].

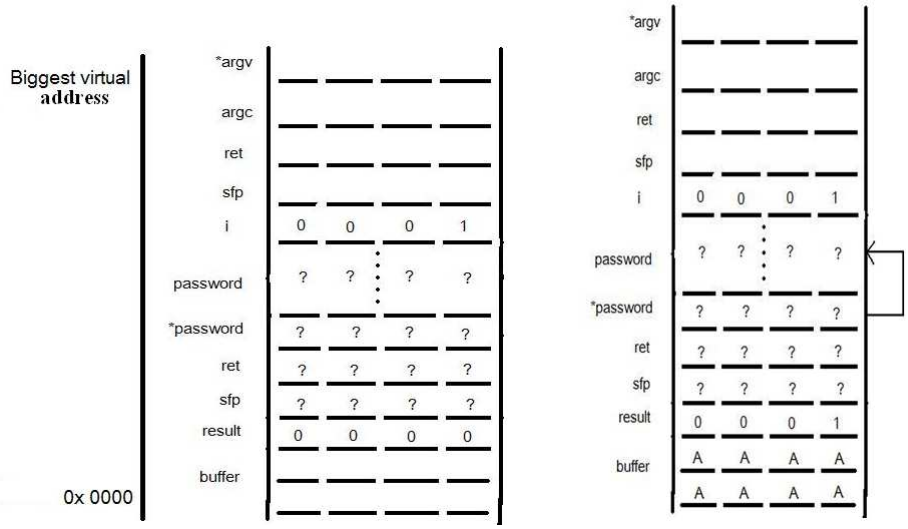
2.1 Known Vulnerabilities

2.1.1 Buffer Overflow

It occurs in fixed length buffers when data is written beyond the boundaries of the current defined capacity. This could lead to mal functioning of the system since the new data can corrupt the data of other buffers or processes. The buffer overflow can be used also to inject malicious code to alter the normal execution of the program and take control of the system. C programming language is particularly affected by this vulnerability due to its dynamic management of the memory, in fact, some critical applications like aeronautics forbids the use of pointers or dynamic memory allocations to avoid this kind of problems. The following C program is vulnerable:

```
int main(int argc, char **argv) {
char buffer[1024];
strcpy (buffer, argv[1]);
}
```

Because $argv[1]$ can contain more than 1024 characters (is bigger than variable buffer). To understand what effects a buffer overflow has and, in particular, how it can be exploited,



(a) Stack before input (b) Stack after input

Figure 2.1: Evolution of the stack

we first have to take a closer look at memory management (See Figure 2.1) with the following program:

```
#include <stdio.h>
#include <string.h>

int check_password (char *password) {
  int result = 0;
  char buffer[8];

  printf(" password: ");
  gets(buffer);
  if (strcmp(buffer, password) == 0){
      result = 1;
  }
  return result;
}

int main (int argc, char *argv[]) {
  int i;
 for (i=1; i<argc; i++) {
    printf(argv[i]);
    if (check_password("secret")) {
         printf("success\n");
    }
```

```
    else {
       printf("failed\n");
    }
  }
}
```

In this example, the program will ask user to enter a password. If password is correct program will answer "success". If user gives a wrong password, it will answer failed. Figure 2.1 is the graphically presentations of the virtual memory management where every memory cell has 4 bytes(32bits-cpu) in size, and the input is equal to *AAAAAAAA1*. Figure 2.1(a) (resp. 2.1(b)) shows us the stack memory before (resp. after) input. In figure 2.1(b), we can see the last character of the buffer overwrites the value of result with 1. Whatever the value of the buffer, the program will answer "success".

2.1.2 XSS or Cross Site Scripting

This vulnerability is associated to web applications. An attacker injects code in web pages that are accessed by other users. And then uses it to bypass access controls, perform phishing, identity theft or expose connections. Such vulnerability is very widespread and happens anywhere a web application uses input from a user without validating it. An attacker can exploit XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Consequently, the malicious script can access any cookies, session tokens, or other sensitive information saved by the browser and used with that site. Such scripts can even rewrite the content of the HTML page.

2.1.3 SQL Injection

Any application that uses SQL database must be protected against SQL injection. An attacker can get sensitive information from the database by injecting crafted inputs that contain hidden SQL commands. If they are not well filtered, they can be executed by the SQL interpreter and expose the content of the database. For example, if an application requests to enter the user name to login, and the attacker enters the following text: ' or '1'='1, then the application may execute the following SQL command:

```
SELECT * FROM users WHERE name = '' OR '1'='1';
```

So, the attacker will get a valid user name since the evaluation of the statement '1'='1' is always true. In a similar manner the attacker might get confidential information, alter the content or even delete the records of the database impacting the service and/or business.

2.1.4 Format String Bugs

Similar to buffer overflow, it happens when external data is given to an output function (*syslog, printf, fprintf, sprintf, and snprintf*) as format string argument. The format arguments tell the function the type and the sequence of arguments to pop and then the format for output. Such format string bugs most commonly appear when a programmer wishes to print a string containing user supplied data. The programmer may mistakenly write $printf(InputBuffer)$ instead of $printf(\text{"}\%s\text{"}, InputBuffer)$. The first version may interpret buffer as a format string, and parses any formatting instructions it may contain. The second version simply prints a string to the screen, as the programmer intended. The first version can lead to the Denial of Service. In this case, when an invalid memory address is requested, normally the program is terminated. For example, if the attacker uses %s%s%s%s as input, the system will output segmentation error and stop the program. There is another string vulnerability that allows an attacker rewrite the data in stack memory. For example, if we have the following code:

```
int i, j;
i=j=0;
printf("abc%ndef%n",&i,&j);
```

After executing this code, the value of i will be rewritten by 3 [1], the value of j will be rewritten by 6. Also, a well trained attacker can overwrite the function return address with a malicious shellcode address by using %n.

2.1.5 Integer Vulnerabilities

They can be of two different types, sign conversion and arithmetic operations bugs. The first occurs when a signed integer is converted to an unsigned integer. The second

1. In printf() function, %n will rewrite the value of variables with the number of arguments read before it.

occurs when the result of an arithmetic operation is an integer larger/smaller than the maximum/minimum possible integer values. Integer vulnerabilities are not only caused by wrong input validation, they can also be caused by not verifying the result of arithmetic operations, which means that two validated inputs, used together in the same operation can create a vulnerability.

2.1.5.1 Integer overflow vulnerabilities

An integer overflow occurs at run-time when the result of an integer expression exceeds the maximum value for its respective type. For example, the product of two unsigned 8-bit integers may require up to 16-bits to represent, e.g., $(2^8 - 1) * (2^8 - 1) = 65025$, which cannot be accurately represented by a signed 8-bit integer. Also, if variable a holds the biggest integer value ($a = 2147483647$) and we execute a+1, then the result will ($a = -2147483648$) be instead of the right value 2147483648.

2.1.5.2 Integer underflow vulnerabilities

An integer underflow occurs at run-time when the result of an integer expression is smaller than its minimum value, thus "wrapping" to the maximum integer for the type. For example, subtracting $0 - 1$ and storing the result in an unsigned 16-bit integer will result in a value of $2^{16} - 1$, not -1. Since underflows normally occur only with subtraction, they are rarer than overflows with only 10 occurrences according the survey given in [17].

2.1.5.3 Integer sign conversion vulnerabilities

A signedness error occurs when a signed integer is interpreted as unsigned, or vice-versa. If a negative signed integer is cast to unsigned, it will become a large value. And if a large positive unsigned integer is cast to a signed integer, it will become negative. Because the sign bit is interpreted as the most significant bit (MSB) or conversely, hence -1 and $2^{32} - 1$ are misinterpreted to each other on 32-bit machines.

2.1.5.4 Integer down-cast vulnerabilities

An integer cast may increase (up-cast) or decrease (down-cast) the precision of the representation. Increasing the precision is always safe, and usually accomplished by zero-extending the casted value. However, decreasing the number of bits is potentially unsafe.

Now, in the next part we study how modeling software vulnerabilities can be helpful to understand vulnerabilities causes, their consequence and possible mitigation or detection methods.

2.2 Vulnerability Modeling

Most of the vulnerabilities could be prevented if software is developed more carefully, however, reading the vulnerabilities reports we notice this is not the case. One possible solution to reduce the number of vulnerabilities is in the improvement of the knowledge and understanding of software developers; in fact developers should not only care about the code and coding speed but also about the vulnerabilities related to the used programming language or system, their causes, consequences, possible threats, types attacks and counter measures. Graphical models might be an adequate tool to implement such solution as we study next.

2.2.1 Vulnerability Cause Graph

Vulnerability Cause Graph (VCG) [2,6] "is a directed acyclic graph that contains one exit node representing the vulnerability being modeled, and any number of cause nodes, each of which represents a condition or event during software development that might contribute to the presence of the modeled vulnerability". The VCG [3] showed in Figure 2.2 represents the vulnerability CVE-2005-3192, which corresponds to a buffer overflow in xpdf.

In this graph we can observe the different causes, nodes one to six and possible scenarios that could lead to the introduction of this kind of vulnerability. A scenario is composed by a sequence of nodes, in our example a scenario might be {1, 2, 4, 5, 7}.

The VCG is helpful to understand what can cause the vulnerability. So, if causes are well understood then they could be avoided in the development process. Since VCGs have been improved by SGMs, we will cover more details later.

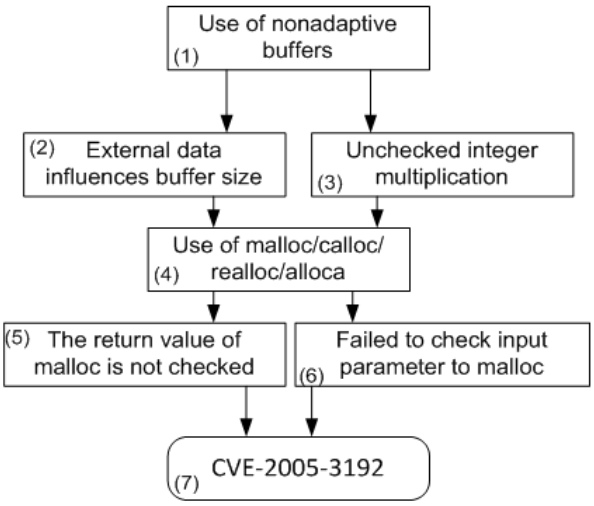


Figure 2.2: Vulnerability Cause Graph

2.2.2 Security Activity Graph

Security Activity Graphs (SAGs) [2,7] are a graphical representation that is associated with causes in a VCG. SAGs indicate how a particular cause can be prevented following a combination of security activities during the development process. Figure 2.3 represents a SAG [7] showing different alternatives to address the cause "Lacking design to implementation traceability".

Thus, in order to solve the design to implementation traceability problem during software development; we have several alternatives resulting from the combination of the different security activities and operators X (AND), + (OR):

- Generate a code from design OR
- Make design objects identifiable AND code comments linking core to design objects OR
- Make design objects identifiable and cross reference index between design and code.

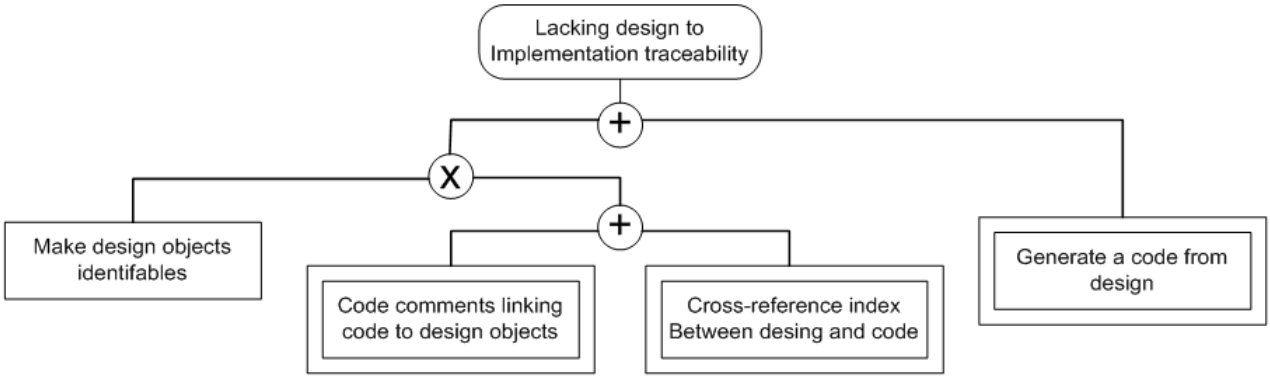


Figure 2.3: Security Activity Graph

2.2.3 Security Goal Indicator Tree

Security Goal Indicator Tree (SGIT) [32] focuses on positive features of the software which can be verified during the inspection process. A SGIT is then a graph where the root is a security goal and its subtrees are indicators or properties that can be checked for achieving that goal. However, since not all properties can be positively expressed it is possible to have also negative indicators (something that should not occur). These indicators have Boolean relations with the goal and have to be checked in order to validate the security goal. SGIT are created by security experts. A SGIT for the goal Audit Data Generation, taken from [32], is presented in figure 2.4, showing some dependency relations, and positive and negative indicators. Also the small box pointing to the indicator "An audit component exists" means that a specialization tree can be deployed for this indicator.

2.2.4 Security Goal Model

The Security Goal Model (SGM) [8] "can be used in place of security activity graphs (SAG), vulnerability cause graphs (VCG), and security goal indicator trees (SGIT)"; since SGMs can be more accurate and rich in expression than previous mentioned models.

In the case of software vulnerabilities, figure 2.5 shows a SGM representing a known buffer overflow in xine, a free multimedia player (CVE-2009-1274). We can observe that this graph is similar to VCG but it offers more details about the different causes and scenarios that could lead to the introduction of this kind of vulnerability. For instance, the

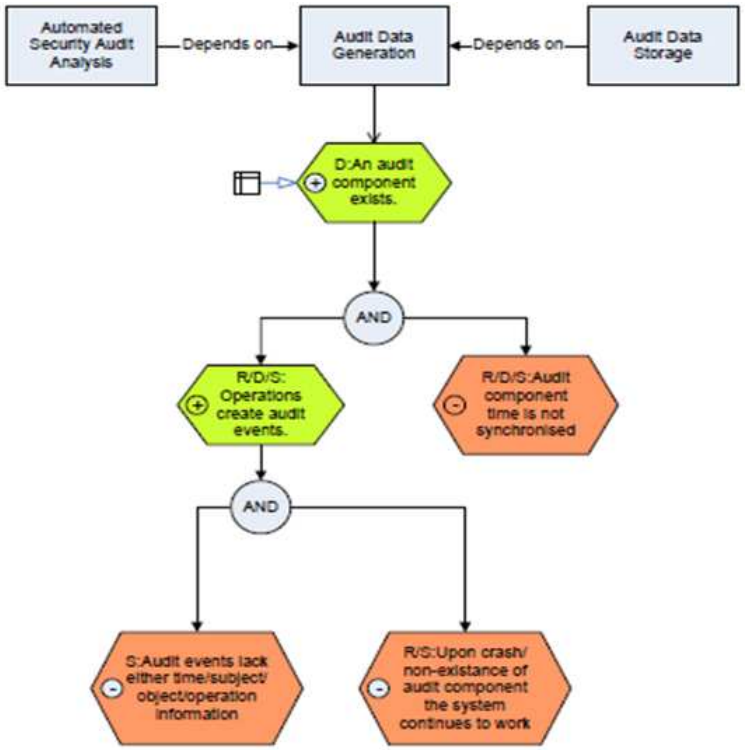


Figure 2.4: Security Goal Indicator Tree

black node represents a "positive" subgoal that helps to reduce the possibility of having the vulnerability, while the information ports and edges (dash arrows) provide more details about the relation among the different subgoals or causes, and the places which can be useful for detection purposes. In our case the subgoal "code controlled by range check" helps to reduce vulnerability presence in the case where a data entered by the user controls or is used inside a loop.

This ideas are helpful to understand the "enchainment" of events that may lead to a vulnerability, thus, they are a valuable input for a detecting tool as we explain later.

2.3 Vulnerability Detection

Models and inspections are useful to understand and prevent vulnerabilities; nevertheless it is also necessary to count on tools that can be used by programmers in order to detect vulnerabilities during the process of software construction.

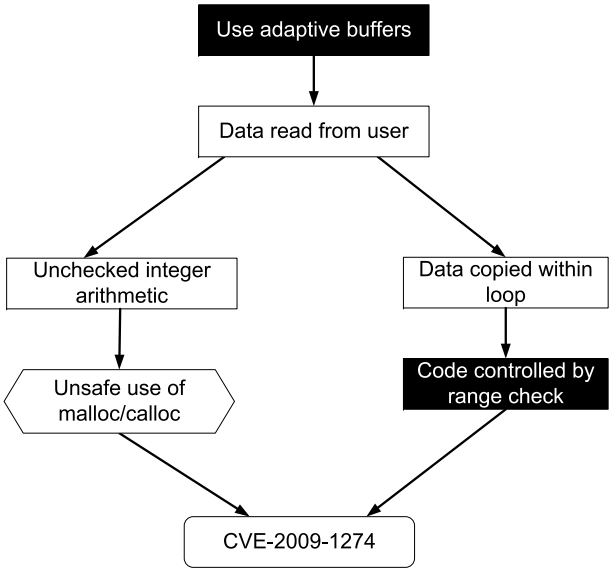


Figure 2.5: Security Goal Model

Some of these tools are based on static methods, thus it is not necessary to run the code to perform the detection. In the case of dynamic methods, the code is run inside a controlled environment to perform the detection or collect program traces that can be use for such purpose. In the next section we present some existing techniques to detect vulnerabilities.

2.3.1 Software Inspection

The software inspection process consists in reading or visually inspecting the program code or documents in order to find any defects and correct them early in the development process. When the defect is found soon the less expensive it becomes to fix. However, a good inspection depends then on the ability and expertise of the inspector, and the kind of defects he is looking for. Usually during the software inspection, it is necessary to look for any possible defects during the security inspections. Vulnerability Inspection Diagram (VID) is a manual inspection introduced in [14], the purpose is to benefit developers from the knowledge and experience of security experts in the detection of problems in the de-

velopment process. Thus a VID is a flowchart-like graph that guides developers to check the software to detect the presence of vulnerabilities based on the knowledge of experts. There is a specific VID for each vulnerability class.

2.3.2 Static Techniques

Static techniques are those applied directly to the source code without running the application, the objective is to evaluate or get specific information directly from the source code without executing it. There are different techniques to perform static analysis; here we mention some of them.

2.3.2.1 Pattern matching

Consists in searching a "pattern" string inside the source code and give as results the number of occurrences of it. For instance if we consider C language, the pattern could be any call to possible dangerous functions (vulnerable) like "getc". Pattern matching can be implemented using a simple tool like the Unix command "grep", however this method generates much false positives because there is no analysis of the results, additionally its effectively is limited since depends on the exact writing of the strings, thus additional white spaces will limit the results.

Flawfinder contains a built-in database of potentially dangerous functions, and uses pattern matching process to find possible vulnerabilities in the code. In order to reduce false positives the results are sorted by risk level [38]. The risk level is associated to the vulnerability of the function used and to the type of function parameters, for example the use of a constant variable is less risky.

2.3.2.2 Parsing

Parsing is more complex than lexical analysis, thus when the source code is parsed, a representation of the program is built using a parsing tree in order to analyze the syntax and the semantics of the program. For example the parsing technique is used to detect SQL command injection attacks [35].

2.3.2.3 Type qualifier

The addition of type qualifiers in a program can be useful to analyze the properties or content of variables in order to find vulnerabilities. For example; Cqual [16], is a type-based static analysis tool for finding bugs in C programs, which means that programmers can extends the existing C types to add annotations to the program; those annotations can be then checked by the tool and detect possible problems. In Cqual user's guide is given the following example:

```
$tainted char *getenv(const char *name);
int printf($untainted const char *fmt, ...);

int main(void)
{
char *s, *t;
s = getenv("LD LIBRARY PATH");
t = s;
printf(t);
}
```

When the code is analyzed by the tool, there will be an error indicating the use of a tainted data (t) where an untainted is expected (argument of *printf*).

2.3.2.4 Data flow analysis

The purpose is to evaluate the source code in order to determine the possible set of values that a variable or an expression may have during the execution of the program. This technique is specially suited for buffer overflow detection.

A control flow graph CFG is used to evaluate sections of the program where the assignation of a given value to a variable is done, and how it is propagated inside the program.

Kem et al in [25] use data flow analysis, they create rules describing vulnerability patterns to detect locations and paths of the pattern in the program. The detector is implemented in three parts: a pattern matcher which finds locations of vulnerabilities in source program, a flow graph constructor which extracts the control flow and data flow from the program, and a flow analyzer which finds program's vulnerable execution paths.

2.3.2.5 Taint analysis

It is a special case of data flow analysis where any data coming from un-trusted sources, e.g. introduced by a user, is a potential problem to the system, thus it is marked as tainted. Tainted data flow is monitored because it cannot reach critical functions unless it is processed and changed to untainted.

Livshits and Lam [29] propose a static analysis framework to find vulnerabilities in Java applications. They define a Tainted Object Propagation problem class to deal with improper user input validation. Java bytecode and vulnerability specifications are employed to perform a taint object propagation and find vulnerabilities using the Eclipse platform.

2.3.2.6 Model checking

Model Checking is a technique to automatically test if a property is verified on a system, so it can be also used to detect vulnerabilities. However, usually model checking is a complex technique because the elaboration of the model is difficult, but once obtained it is easier to test the properties of the system.

A security verification framework with multi-language support was developed based on GCC compiler [19]. Their approach uses a conventional push down system model checker for reach ability properties to verify software security properties; it is composed of three phases: security property specifications, program model extraction and property model checking, this last has as output the detected errors with execution traces.

Constraint analysis has been combined with model checking in order to detect buffer overflow vulnerabilities [37]. They trace the memory size of buffer-related variables and the code instrumented with constrains assertions before the potential vulnerable points. The vulnerability can be detected with the reach ability of the assertion using model checking. They decrease the cost of model checking by slicing the program.

Model checking has been used to detect vulnerabilities [5, 11] bugs or problems in C programs. For instance, Holzmann developed the Modex [30] tool to extracts models from ANSI-C code using a test-harness specified by the user in a file and then test distributed systems using the Spin model checker. Modex have also been employ by Kim et al [25] to test for concurrency bugs in the Linux kernel while Bao et al [2] test abstract com-

ponents, however previously to the model extraction they compile the C program into a C intermediate language (CIL) [31] to reduce the syntactic constructs and simplify the translation.

Another approach based on model checking is the one of Jiang et Jonsson [23] who test the correctness of concurrent algorithms. They automatically translate a subset of C into Promela specification; they describe the properties to test and run Spin to verify if the specification is correct. Wan et al [37] combine model checking and program analysis. The purpose is to detect buffer overflows using constraint based analysis and program slicing to instrument assertions before vulnerable points and verify the reach-ability with model checking.

2.3.3 Dynamic Techniques

In order to dynamically detect vulnerabilities it is necessary to execute the program code, and then analyze the behavior or the answers of the system and gives a verdict. In the next part we study some of the techniques to perform dynamic detection.

2.3.3.1 Fault injection

Fault injection is a testing technique that introduces faults in order to test the behavior of the system, some knowledge about the system is required to generate the possible faults. With fault injection, it is possible to find security flaws in a system [36] by injecting them into the system under test and observing its behavior. The failure to tolerate faults is an indicator of a potential security flaw in the system, a model is used to decide what faults to inject.

2.3.3.2 Fuzz testing

The idea of this test is to provide random data as input to the application in order to determine if the application can handle it correctly. Fuzz testing is easier to implement than fault injection because the test design is simpler and previous knowledge about the system to test is not always required, additionally it is limited to the entry points of the program. Web scanners are in this tool category. Fuzz testing can also be improved to

have a better coverage of the system. For instance recording real user inputs to fill out web forms and then utilize the collected data in the fuzz testing process to better explore web applications (reach ability) [27].

2.3.3.3 Dynamic taint

Similar to taint analysis, however in this case the tainted data is monitored during the execution of the program to determine its proper validation before entering sensitive functions. It enables the discovering of possible input validation problems which are reported as vulnerabilities [12].

2.3.3.4 Sanitization

One possibility to avoid vulnerabilities due to the use of user supply data is the implementation of new incorporated functions or custom routines whose main idea is to validate or sanitize any input from the users before using it inside a program. In [3] they present an approach using static and dynamic analysis to detect the correctness of sanitization process in web applications that could be bypass by an attacker. They use data flow techniques to identify the flows of input values from sources to sensitive sinks or the places where the value is used. Later they apply the dynamic analysis to determine the correct sanitization process.

In Table 2.1, we present a list of tools for dynamic code analysis.

2.4 Conclusion

Software security has become an important research area due to the massive use of software programs in multiple kinds of applications and environments. It is necessary to guarantee that those programs do not contain vulnerabilities that represent a potential source of problems. Vulnerabilities are not new, however the impact of their presence has increased because of the "interconnection" capabilities of programs, that facilitate their use and access but also attacks.

In order to help developers to better understand vulnerabilities and how avoid and detect them in the code we can count on models like VCG and SGM, which show how vulnerabilities are caused. Despite the graphical aspect of such models that facilitates the communication with the different stakeholders, they lack rigorous semantics. Thus, they cannot be used as basis for automatic vulnerabilities detection. This is why we propose in chapter 3 a formal language, called *Vulnerability Detection Conditions*(VDC), that permits to formally describe the occurrence of vulnerabilities to detect them automatically. We also define an intermediate format, called *template*, to represent vulnerabilities that is less informal than VCG and allows an automatic translation into VDC. Rules to generate such templates from VCG are also provided.

Despite of all precautions we can take during software development, we need to ensure the program does not contain any vulnerability. The selection of tool and detection technique for vulnerabilities is related to the type of application to evaluate, the programming language and the type of vulnerability to detect. A classic technique to detect vulnerabilities is the inspection of the source code, this method can be applied several times during the construction phase as advantage but requires specialists to perform the task as drawback and it is time consuming and is performed by an expert. The static techniques cover all possible execution paths but require the source code while dynamic techniques have the difficulty of requiring the preparation of test cases and the possibility that not all paths in the program are covered, but the advantage that the problems if any, are found in the running code. Dynamic techniques have also less false positives than static ones. Moreover, the tools/libraries supporting dynamic techniques detect runtime errors but they do not allow users to define vulnerabilities to be checked on the analyzed executable program.

The next two chapters present two complementary dynamic techniques to detect vulnerabilities. The first one is based on passive testing technique [4] and uses the concept of VDC, while the second uses the model checking together with active testing technique [9].

Tool	Name Developed by	Description
Valgrind	Valgrind developers	Valgrind runs programs on a virtual processor and can detect memory errors (e.g., misuse of malloc and free) and race conditions in multithread programs
Insure++	Parasoft Insure++	is a memory debugger computer program, used by software developers to detect various errors in programs written in C and C++.
Dmalloc	Gray Watson	Dmalloc is a memory debugger C library that helps programmers to find a variety of memory allocation programming errors for dynamic memory. It replaces parts of standard programming library provided by the operating system for malloc and other software with its own versions which help the programmer detect buffer overflows and other critical programming issues
DynInst	University of Wisconsin-Madison and University of Maryland	DynInst is a runtime code-patching library that is useful in developing dynamic program analysis probes and applying them to compiled executables. Dyninst does not require source code or recompilation in general, however, non-stripped executables and executables with debugging symbols are easier to instrument.
Daikon	MIT	Daikon (system) is an implementation of dynamic invariant detection. Daikon runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions, and thus likely true over all executions.
IBM Rational AppScan	IBM	IBM Rational AppScan is a suite of application security solutions targeted for different stages of the development lifecycle. The suite includes two main dynamic analysis products: IBM Rational AppScan Standard Edition, and IBM Rational AppScan Enterprise Edition. In addition, the suite includes IBM Rational AppScan Source Edition a static analysis tool.
Purify	IBM	Purify is a memory debugger program used by software developers to detect memory access errors in programs, especially those written in C or C++. It was originally written by Reed Hastings of Pure Software. Pure Software later merged with Atria Software to form Pure Atria Software, which in turn was later acquired by Rational Software, which in turn was acquired by IBM. It is functionally similar to other memory debuggers, such as Insure++ and Valgrind.
Intel Thread Checker	Intel	Intel Thread Checker is a runtime threading error analysis tool which can detect potential data races and deadlocks in multithreaded Windows or Linux applications.

Table 2.1: List of dynamic code analyzers

Chapter 3

Detection of Vulnerabilities using Passive Testing

Contents

Security has become an important issue for the industry of software; being one of its main objectives to reduce as much as possible the presence of vulnerabilities in the software that is produced; reducing as well the risk of attacks and their impact in terms of money, downtime, loss of customers, and performance. Nevertheless, if we consider the complexity and size of programs we need then to count on tools and methods to test software and be able to detect as much vulnerabilities as possible. In previous part, we presented a review of known software vulnerabilities, detection methods and modeling techniques like for example SGM and VCG. These models are helpful to understand known vulnerabilities because they graphically represent how a given vulnerability was caused in a specific program. In fact, they show the sequence of actions and conditions alongside the program code; that leaded to the modeled vulnerability. Vulnerability models are then helpful to improve the knowledge of programmers, but also they are of great value from a vulnerability point of view since they provide a guide about the conditions and actions that should be checked inside a program code to detect such vulnerabilities. However, in these models the descriptions of the causes are done in natural language, which is good for human understanding but not very valuable for automatic detection. In this chapter we present a formal language called Vulnerability Detection Condition (VDC); a formal interpretation of the occurrence of the vulnerability. These VDCs can be obtained from VCGs or SGMs through an intermediate format called Template. A template is a table with specific and fixed fields conceived to systematically extract information from the vulnerability models or the description of the vulnerability and automatically generate VDCs. Our approach, has been used to develop a passive testing tool which takes VDCs as input in order to monitor the execution of the program under test and detect the presence of the vulnerability given as input, as explained in this chapter. This chapter gives also a formal syntax and semantics of SGM in order to formalize the derivation of templates from them.

3.1 Vulnerability Detection Condition

Vulnerability Detection Conditions is the formalism we have defined to describe the presence of a vulnerability in a software. It allows to express that, for instance, a dynamically allocated memory must not be used (read or written) without first checking that the

allocation operation succeeded. The main idea behind the definition of the VDC formalism is to point out the use of a dangerous action under some particular conditions, for instance "it is dangerous to use unallocated memory".

Definition 1 *(Vulnerability Detection Condition). Let Act be a set of action names, Var be a set of variables, and P be a set of predicates on (Var ∪ Act). A vulnerability detection condition VDC is of the form (square brackets denote an optional element):*

$$VDC ::= a/P(Var, Act) \mid a[/P(Var, Act)]; P'(Var, Act)$$

where a denotes an action, called a master action, that produces the vulnerability, $P(Var, Act)$ and $P'(Var, Act)$ represent any predicates on variables Var and actions Act.

A vulnerability detection condition $a/P(Var, Act)$ means that the master action a produces a vulnerability when it occurs under specific conditions denoted by predicate $P(Var, Act)$.

A vulnerability may also occur due to the action that follows the master action. That case is represented by

$$a[/P(Var, Act)]; P'(Var, Act)$$

This means that the master action a used under the optional conditions $P(Var, Act)$ is followed by a statement whose execution satisfies $P'(Var; Act)$. Naturally, if action a is not followed by an action, the predicate $P'(Var, Act)$ is assumed to be true.

Intuitively, VDCs are composed of actions and conditions. An *action* denotes a particular point in a program where a task or an instruction that modifies the value of a given object is executed. Some examples of actions are variable assignments, copying memory or opening a file. A *condition* denotes a particular state of a program defined by the value and the status of each variable. For a buffer, for instance, we can find out if it has been allocated or not.

More complex vulnerability detection conditions can be built inductively using the different logical operators according to the following definition:

Definition 2 *(General Vulnerability Detection Conditions). If VDC_1 and VDC_2 are vulnerability detection conditions, then $(VDC_1 \vee VDC_2)$ and $(VDC_1 \wedge VDC_2)$ are also vulnerability detection conditions.*

3.1.1 VDC examples

In order to clarify the concept we present some examples of VDCs. First, consider that we want to define a vulnerability detection condition to detect if a certain value y is assigned to a memory variable x, but the memory space for x has not yet been allocated. We can define the VDC as follows:

$$VDC_1 = Assign(x, y)/IsNot\_Allocated(x)$$

In the case of programming languages like C/C++, there are some functions that might lead to a vulnerability if they are applied on out-of-bounds arguments. The use of a tainted variable as an argument to a memory allocation function (e.g. `malloc`) is a well-known example of such a vulnerability, expressed by the vulnerability detection condition VDC_2 below. A variable is tainted if its value is obtained from a non-secure source. This value may be produced by reading from a file, getting input from a user or the network, etc. Note that a tainted value can be untainted during the execution of the program if it is checked to determine if it has an acceptable value.

$$VDC_2 = memoryAllocation(S)/tainted(S)$$

A good programming practice is to verify the return value from any allocation function. The following vulnerability detection condition VDC_3 detects the absence of such verification:

$$VDC_3 = Assign(u, memoryAllocation(S)); notChecked(u, null)$$

3.1.2 Creating VDC

As we have mentioned previously, the aim of VDCs is to formally define the causes described by the vulnerability model. An informal description of a vulnerability states the conditions under which the execution of a dangerous action leads to a security breach. So, it should include the following elements:

1. *A master action*: an action denotes a particular point in a program where a task or an instruction that modifies the value of a given object is executed. Some examples

of actions are variable assignments, copying memory or opening a file. A master action *Act_Master* is a particular action that produces the related vulnerability.

2. A set of conditions: a condition denotes a particular state of a program defined by the value and the status of each variable. For a buffer, for instance, we can find out if it has been allocated or not. Once the master action is identified for a scenario, all the other facts are conditions $\{C_1, \ldots, C_n\}$ under which the master action is executed. Among these conditions, a particular condition C_k may exist, called *missing condition*, which must be satisfied by an action following *Act_Master*.

In our work we developed a method consisting in four steps:

1. Analyze the model that represents the vulnerability
2. Extract the testing information using templates
3. Automatically process the templates to obtain the VDCs and
4. Define the global VDC for the vulnerability.

Each step is described in detail in the next part. To this aim, we give first the syntax and the semantics of the SGM model to help the readers understand our approach.

3.2 The SGM Vulnerability Model

3.2.1 The SGM Syntax

A security goal model (SGM) is a directed acyclic graph. Vertices represent subgoals; solid edges represent dependencies between subgoals; and dashed edges can be thought of as modeling information flow. For VDC generation purpose, we only consider the solid edges, the others are skipped. Also, we adapt the definition given in [33] in order to improve the formalization of scenarios generation. To this aim, let $\mathfrak{N}$ be a set of all possible nodes.

Definition 3 *A security goal model T is a 7-tuple $(N, N_0, n_{exit}, succ, desc, struct, conj)$, where N is a finite set of nodes ($N \subseteq \mathfrak{N}$) such that N_0 ($N_0 \subseteq N$) denotes the set of the initial nodes for the scenarios that lead to the vulnerability represented by the root of the SGM n_{exit} ($n_{exit} \in N$), succ is a relation that gives for each node its successor*

nodes ($succ \in N \leftrightarrow N$), *desc is a function that returns the textual description of each node* ($desc \in N \rightarrow String$); *struct is a function that gives the SGM associated with each composite node* ($struct \in N \rightarrow SGM \cup \{\perp\}$) *where value* $\perp$ *is the image of a simple node by function struct, and conj is a function that gives the set of nodes that composes a conjunction node* ($conj \in N - \{n_{exit}\} \rightarrow P(\mathfrak{N} - N)$).

To be suitable for VDC translation, a SGM model should meet the following requirements:

1. each node of N_0 has no antecedent [1]:

$$\forall n.(n \in N_0 \Rightarrow n \notin ran(succ))$$

2. node n_{exit} has no successor [2]:

$$n_{exit} \notin dom(succ)$$

3. for each node n_1, such that ($n_1 \in N$), there should exist a path starting from a node of N_0 that includes n_1 and n_{exit}. That means that the following two properties are verified:

   (a) node n_1 is reachable from a node of N_0:

$$\forall n_1.(n_1 \in N \Rightarrow \exists n_0.(n_0 \in N_0 \wedge n_1 \in succ^*[\{n_0\}]))$$

   (b) node n_{exit} is reachable from node n_1 [3]:

$$\forall n_1.(n_1 \in N \Rightarrow n_{exit} \in succ^*[\{n_1\}])$$

1. If $R \in X \leftrightarrow Y, ran(R) = \{y \mid y \in Y \wedge \exists x.(x \in X \wedge x \mapsto y \in R)\}$
2. If $R \in X \leftrightarrow Y, dom(R) = \{x \mid x \in X \wedge \exists y.(y \in Y \wedge x \mapsto y \in R)\}$
3. $succ^*$ denotes the reflexive transitive closure of relation *succ*.

3.2.2 The SGM Semantics

For the VDC generation purpose, we define the semantics of a SGM model as a transformation function that translates the 7-tuple into a set of scenarios (or paths) (called *scenario suites*), each of which describes a valid path to obtain the modeled vulnerability. The scenarios can then be interpreted in an appropriate manner to create VDCs. These set of scenarios S, is used to build the test suite that is going to be used by the detection tool to verify if the program under evaluation is executing certain actions under some specific conditions, if it is the case the considered vulnerability is detected. Before defining this semantics, we introduce some useful notations:

- $succ^2 = \{(x_1,\ x_2,\ x_3) \mid (x_1,\ x_2) \in succ \wedge (x_2,\ x_3) \in succ\}$
- $succ^3 = \{(x_1,\ x_2,\ x_3,\ x_4) \mid (x_1,\ x_2,\ x_3) \in succ^2 \wedge (x_3,\ x_4) \in succ\}$
- for any rank i: $succ^i = \{(x_1,\ x_2, \ldots,\ x_{i+1}) \mid (x_1,\ x_2,\ x_3,\ x_i) \in succ^{(i-1)} \wedge (x_i,\ x_{i+1}) \in succ\}$
- a set of functions $map\_set_{i(i\geq 2)}$ that transform any $i-tuple$ to a set of its elements; each function map_set_i is defined by:

$$map\_set_i \in \underbrace{N \times \ldots \times N}_{i\ times} \rightarrow \mathbb{P}(N)$$

with:

$$map\_set_i((x_1, \ldots,\ x_i)) = \{x_1, \ldots,\ x_i\}$$

Definition 4 *Let T be an SGM $(N, N_0, n_{exit}, succ, desc, struct, conj)$. The semantic transformation of T, $S\_ \in \boldsymbol{SGM} \rightarrow \mathbb{P}(\bigcup_{i=1,..\infty} map\_set_{i+1}[succ^i])$ is such that :*

1. *each scenario sc contains an initial node and the exit node*

$$\forall sc.(sc \in S(T) \Rightarrow sc \cap N_0 \neq \emptyset \wedge n_{exit} \in sc)$$

2. *each node of T belongs to one scenario at least:*

$$\forall n.(n \in N \Rightarrow \exists sc.(sc \in S(T) \wedge n \in sc))$$

3. *each sub-set scs of* $\mathbb{P}(\bigcup_{i=1,..\infty} map\_set_{i+1}[succ^i])$ *that verifies conditions (1) and (2) is included in* $S(T)$*:*

$$scs \subseteq S(T)$$

3.3 Procedure to Create VDC

This section describes the procedure to create a VDC from a SGM according to the different syntactical and semantical definitions presented before.

3.3.1 Analyze the Model that represents the Vulnerability

Before deriving the scenarios from a SGM model T ($T = (N, N_0, n_{exit}, succ, desc, struct, conj)$), we have to ensure that it meets some specific requirements:

- if there is a node A shared by two conjunction nodes B and C, then make B and C disjoint by duplicating the node A.To do that, let node A' be a new node:
  - $T.N$, $T.N_0$, $T.n_{exit}$ and $T.succ$ remain the same
  - make the description of node A' equal to that of A: $(A'.desc = A.desc)$ where $A.desc$ denotes the description of node A.
  - make the structure of node A' equal to that of A: $(A'.struct = A.struct)$ where $A.struct$ denotes the structure of node A.
  - $T.conj = (\{B\} \lhd T.conj) \cup \{(B, B.conj - \{A\} \cup \{A'\})\}$ where $B.conj$ gives the components of node B. We chose to replace node A by A' in B.
- If there is a conjunction node A modeled by a a set of nodes B, replace it with sequential nodes. To this aims, we have to define a path $pa = (n_i, n_f, succ')$ with the nodes B as follows:

  1. n_i and n_f $(n_i \in B \wedge n_f \in B)$ denote respectively the initial and final nodes of pa,

  2. function $succ'$ is such that:

$$succ' \in (B - \{n_f\}) \rightarrowtail\!\!\!\!\!\rightarrow (B - \{n_i\})$$

this means that:

(a) Except the final node n_f, each node has exactly one successeur,

(b) Except the initial node n_i, each node has exactly one predecessor.

the path pa, being defined, the replacement of a conjunction node is performed as follows:

- $T.N = T.N - \{A\} \cup B$
- if A belongs to $T.N_0$, $T.N_0 = T.N_0 - \{A\} \cup \{n_i\}$
- $T.n_{exit}$ remains the same
- $T.succ = ((\{A\} \lhd T.succ) \rhd \{A\}) \cup (T.succ^{-1}[\{A\}] \times \{n_i\}) \cup (\{n_f\} \times T.succ[\{A\}]) \cup succ'$ [4]
- $T.desc = (\{A\} \lhd T.desc) \cup B.desc$ where $B.desc$ gives the description of each node of B.
- $T.struct = (\{A\} \lhd T.struct) \cup B.struct$ where $B.struct$ gives the structure of each node of B
- $T.conj = (\{A\} \lhd T.conj) \cup B.conj$ where $B.conj$ gives the possible components of each node of B

- If there is any node A modeled by a suitable SGM G ($G = (N, N_0, n_{exit}, succ, desc, struct, conj)$), replace it with its corresponding model. To do that, we have to relate all the antecedents of A to each initial node of G and the antecedent nodes of $G.n_{exit}$ to each successor of A. Formally, components of T become as follows:
  - $T.N = T.N - \{A\} \cup G.N - \{G.n_{exit}\}$
  - if A belongs to $T.N_0$, $T.N_0 = T.N_0 - \{A\} \cup G.N_0$
  - $T.n_{exit}$ remains the same
  - $T.succ = ((\{A\} \lhd T.succ) \rhd \{A\}) \cup (T.succ^{-1}[\{A\}] \times G.N_0) \cup (G.succ^{-1}[\{G.n_{exit}\}] \times T.succ[\{A\}])$
  - $T.desc = (\{A\} \lhd T.desc) \cup G.desc$
  - $T.struct = (\{A\} \lhd T.struct) \cup G.struct$
  - $T.conj = (\{A\} \lhd T.conj) \cup G.conj$

4. If $R \in X \leftrightarrow Y$, $A_1 \subseteq X$ and $B_1 \subseteq Y$, then $A_1 \lhd f = \{x \mapsto y \mid x \mapsto y \in R \wedge x \notin A_1\}$ and $f \rhd B_1 = \{x \mapsto y \mid x \mapsto y \in R \wedge y \notin B_1\}$

- Discard the qualitative subgoals of the SGM and keep only quantitative ones. Qualitative subgoals cannot be checked or evaluated without human intervention. *Documentation is unclear* is an example of such a cause. Since our interest is automatic testing, we are concerned only with quantitative subgoals. A quantitative subgoal is directly related to the software code, so it can be automatically checked. An example is the use of `malloc` as memory allocation function. To formalize this step, we add to the previous definition of SGM, a function *Qualitative* defined as $(Qualitative \in N \rightarrow BOOL)$ to indicate whether a node is qualitative of not. So if a node A is qualitative, the initial SGM T becomes:
  - $T.N = T.N - \{A\}$
  - If $A \in T.N_0$, $T.N_0 = T.N_0 - \{A\} \cup \{n \mid n \in T.N \wedge succ^{-1}[\{n\}] = \{A\}\}$
  - $T.n_{exit}$ remains the same because we assume that $(Qualitative(n_{exit}) = False)$, that is $A \neq n_{exit}$.
  - $T.succ = ((\{A\} \lhd T.succ) \rhd \{A\}) \cup (T.succ^{-1}[\{A\}] \times T.succ[\{A\}])$
  - $T.desc = (\{A\} \lhd T.desc)$
  - $T.struct = \{A\} \lhd T.struct$
  - $T.conj = \{A\} \lhd T.conj$
- Replace counteracting nodes with an equivalent contributing nodes. When testing, we want to check if the "bad" actions or conditions are execute in order to determine whether the vulnerability is present or not. To formalize this step, we add to the previous definition of SGM, a partial function *counteract* defined as $(contrib \in N \nrightarrow N')$ to provide the corresponding contributing node for counteracting node. So if a node A is counteracting, the initial SGM T becomes:
  - $T.N = (T.N - \{A\}) \cup \{contrib(A)\}$
  - $T.N_0 = T.N_0 - \{A\} \cup \{contrib(A)\}$ if $(A \in N_0)$, otherwise $T.N_0$ remains the same.
  - $T.n_{exit}$ remains the same because we assume that n_{exit} is not a counteracting node.
  - $T.succ = ((\{A\} \lhd T.succ) \rhd \{A\}) \cup (T.succ^{-1}[\{A\}] \times \{contrib(A)\}) \cup (\{contrib(A)\} \times T.succ[\{A\}]))$
  - $T.desc = (\{A\} \lhd desc) \cup \{contrib(A) \mapsto desc'\}$ where $desc'$ is the description of the contributing node.
  - $T.struct = (\{A\} \lhd T.struct) \cup \{contrib(A) \mapsto struct'\}$ where $struct'$ denotes the

potential SGM associated with the contributing node, $\perp$ otherwise.

- $T.conj = (\{A\} \lhd T.conj) \cup \{contrib(A) \mapsto conj'\}$ where $conj'$ denotes the potential components of the contributing node.

The resulting graph is now adequate to obtain the VDCs. Nevertheless, in order to facilitate the scenario processing we use numbers to identify subgoals. So, we add an injective function *number* defined by: $number \in N \rightarrowtail NAT$ because two distinct nodes should have different numbers.

3.3.2 Extract Testing Information using Templates

Once the scenarios are defined we have to collect all the possible details given by the subgoals. The idea is to identify the variables, parameters, actions and conditions that contribute to the vulnerability. For that we have created two templates, one corresponding to master actions and another to the conditions under which the master actions are executed. These templates, produced manually, are automatically processed to generate the VDCs.

In the SGM, every possible scenario must contain one master action Act_Master that produces the related vulnerability. All the other vertices of this path denote conditions $\{C_1, ..., C_n\}$.

Among these conditions, a particular condition C_k may exist, called missing condition, which must be satisfied by an action following Act_Master. Let $\{P_1, \ldots, P_k, \ldots, P_n\}$ be the predicates describing conditions $\{C_1, \ldots, C_k, \ldots, C_n\}$. The formal vulnerability detection condition expressing this dangerous scenario is defined by: $Act\_Master/(P_1 \wedge ... \wedge P_{k-1} \wedge P_{k+1} \ldots \wedge P_n); P_k$

After the identification of master actions and conditions we take the corresponding template to analyze each subgoal. The master action and condition templates are herewith explained.

3.3.2.1 Master action template

This template is designed to collect all the information related to the master action of the SGM and possible input/output parameters. The master action template with its

corresponding items and a brief explanation of them are shown in Table 3.1.

Table 3.1: Master Action Template

Item	**Description**	**Value**
1. Node number	Number used to identify each node of the SGM: $number(A)$ with A denoting the master action	Integer
2. Previous node	This field indicates the number of the previous node in the SGM; it is duplicated from the SGM to make the template more self-contained: $number[succ^{-1}[\{A\}]]$	Integer
3. Next node	This field indicates the number of the next node/nodes in the SGM; it is duplicated from the SGM to make the template more self-contained: $number[succ[\{A\}]]$	Integer(s)
4. Function name	Indicate the name of the master action function : derived from $desc(A)$	Text (predefined)
5. Input parameter name	Indicate the name of the input parameter of the master action function	Free text
6. Input parameter type	Indicate the type of the input parameter of the master action function	Variable types
7. Variable that receives function result	Indicate the name of the variable that receives the result of the execution of the function considered	Free text
8. Type of the variable that receives function result	Indicate the type of the output parameter of the master action function	Variable types

From the template, the master action expression is derived by combining some of the items according to the following general expressions:

- $function\_name(inputparameter)$: the master action is related to the execution of $function\_name$ which receives $inputparameter$ as input.
- $function\_name(outputparameter, inputparameter)$: if the $outputparameter$ is given; the master action is related to the use of $function\_name$ which receives $inputparameter$ as input to calculate the value of $outputparameter$.

3.3.2.2 Condition template

The condition template is intended to describe the conditions under which the execution of the master action becomes dangerous, i.e., produces the modeled vulnerability. The condition template is described in Table 3.2.

Table 3.2: Condition Template

Item	**Description**	**Value**
1. Node number	Number used to identify each node of the SGM: $number(A)$ with A denoting the node of SGM	Integer(s)
2. Previous node	This field indicates the number of the previous node of the SGM; it is duplicated from the SGM to make the template more self-contained: $number[succ[\{A\}]]$	Integer
3. Next node	This field indicates the number of the next node of the SGM; it is duplicated from the SGM to make the template more self-contained: $number[succ[\{A\}]]$	Integer
4. Search	Indicate the element considered in the node	Functions, variables, list
5. Name	Indicate the name of the element considered in the node	Free text or predefined (case of functions)
6. Type	Indicate the type of the element considered in the node	Predefined
7. Condition follows master action	Indicates if the current condition follows or not the execution of the master action	Yes or no
8. Condition	Condition expressed by the node	Reserved text
9. Condition element	Elements involved in the condition	Text

The expression derived from condition template is written according to the formula:

$$Condition(name, condition\_element)$$

This indicates that the condition is given by $condition\_element$ acting on element $name$.

3.3.3 Automatically Process of Templates to obtain the VDCs

In this step the information collected with the master action and condition templates are automatically processed to generate the expressions of the VDCs according to the corresponding testing scenario.

For that, all nodes of the graph are numbered in the template, indicating also the number of the predecessor/successor nodes. The purpose is to identify the nodes and find all the paths starting from the initial node to the exit node (the vulnerability). These paths correspond the testing scenarios. Once the templates are filled, a predicate is associated with each node, and the scenarios identified according to the previous definitions, the templates are processed to generate the VDCs using an algorithm.

This algorithm considers a set of nodes stored in a collection where each node is represented by a record data type (like a JAVA class) with the following attributes:

- *Number*: denotes the node number as defined in the template,
- *Nextnodes*: denotes a collection of the node numbers of the successors nodes as defined in the template,
- *FollowMasterAction*: specifies if the node follows the master action or not, as defined in the template,
- *Predicate*: denotes the predicate associated with each node, its type is string.

FOR *each scenario S* **DO**

search the exit node B /**B* is such that $B.Nextnodes = null$*/

let A be the master action

pre = ""; post="" /*variables *pre* and *post* are initialized to empty Strings*/

FOR EACH *node C of S* **DO**

IF $C.Number \neq A.Number \wedge C.Number \neq B.Number$ **THEN**

/* check if *C* does not follow *A**/

IF $C.FollowMasterAction$ **THEN**

IF $post = ""$ **THEN** $post = ";" \bullet C.Predicate$

ELSE

$post = post \bullet " \wedge " \bullet C.Predicate$

END

ELSE

IF $pre = ""$ **THEN** $pre = "/" \bullet C.Predicate$

/* where $\bullet$ denotes the concatenation operator on strings*/

ELSE $pre = pre \bullet " \wedge " \bullet C.Predicate$

END

END

END

END

print A.Predicate $\bullet$ *pre* $\bullet$ *post*

END

3.3.4 Define the Global VDC for the given Vulnerability

The semantic transformation explained in section 3 helps to find the scenario suite, a set of scenarios that show all the different paths that cause the modeled vulnerability. From a testing perspective we have to consider this scenario suite, it means we have to test all the scenarios in order to detect the considered vulnerability.

Therefore, we define the global VDC representing the modeled vulnerability as the disjunction of the all vulnerability detection conditions of each scenario (VDC_i denotes the VDC associated with each path i):

$$S = (VDC_1 \vee \ldots \vee VDC_n)$$

3.4 Examples of VDC Creation

In this section, we illustrate the process of the VDC generation through some vulnerability examples. For each vulnerability, we give its description using the VCG or SGM formalisms, them we show how to translate it to VDC.

3.4.1 VDC for CVE-2005-3192 vulnerability

Consider the VCG shown in Figure 3.1 for CVE-2005-3192 vulnerability, which is a buffer overflow in Xpdf 3.01.

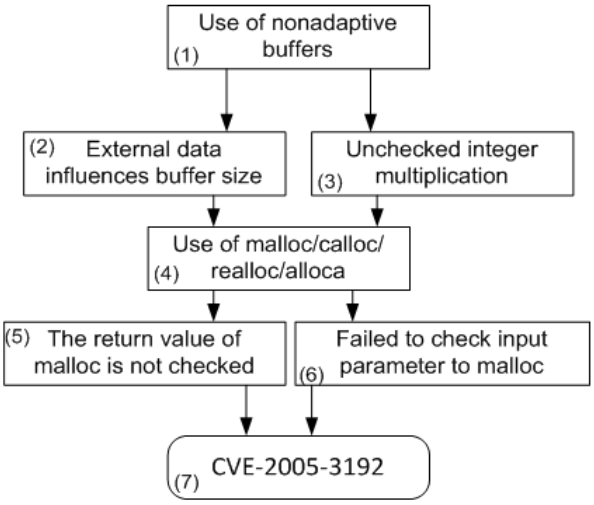


Figure 3.1: VCG for CVE-2005-3192

The first step of the process is to assign an identification number for each node of the graph as shown in Figure 3.1. Then, the different scenarios can be generated as illustrated below. First, we have to calculate the different relations $succ^i$:

- $succ = \{(1,2),(1,3),(2,4),(3,4),(4,5),(4,6),(5,7),(6,7)\}$
- $succ^2 = \{(1,2,4),(1,3,4),(2,4,5),(2,4,6),(3,4,6),(3,4,5),(4,5,7),(4,6,7)\}$
- $succ^3 = \{(1,2,4,5),(1,2,4,6),(1,3,4,5),(1,3,4,6),(2,4,5,7),(2,4,6,7),$
  $(3,4,6,7),(3,4,5,7)\}$
- $succ^4 = \{(1,2,4,5,7),(1,2,4,6,7),(1,3,4,5,7),(1,3,4,6,7)\}$

Now, we have to compute functions map_set_i:

- $map\_set_2[succ] = \{\{1,2\},\{1,3\},\{2,4\},\{3,4\},\{4,5\},\{4,6\},\{5,7\},\{6,7\}\}$
- $map\_set_3[succ^2] = \{\{1,2,4\},\{1,3,4\},\{2,4,5\},\{2,4,6\},\{3,4,6\},\{3,4,5\},$
  $\{4,5,7\},\{4,6,7\}\}$
- $map\_set_4[succ^3] = \{\{1,2,4,5\},\{1,2,4,6\},\{1,3,4,5\},\{1,3,4,6\},\{2,4,5,7\},$
  $\{2,4,6,7\},\{3,4,6,7\},\{3,4,5,7\}\}$
- $map\_set_5[succ^4] = \{\{1,2,4,5,7\},\{1,2,4,6,7\},\{1,3,4,5,7\},\{1,3,4,6,7\}\}$

Finally, we deduce the set of scenarios: {1,2,4,5,7},{1,2,4,6,7}, {1,3,4,5,7} and {1,3,4,6,7}. And for each one we have to define its vulnerability detection condition.

In our example, the master action that may lead to the vulnerability is the use of a memory allocation function (node 4), which is common to all the scenarios. To collect the information regarding the master action we fill the master action template.

Table 3.3: Master Action Templates for CVE-2005-3192

Item	Description	Node
1	Node number	4
2	Previous node	2, 3
3	Next node	5, 6
4	Function name	malloc
5	Input parameter name	buffer_size
6	Input parameter type	integer
7	Variable that receives function result	buffer
8	Type of the variable that receives function result	pointer

When the template is processed, the master action expression is:

$$malloc(buffer, buffer\_size)$$

Which means the master action is the allocation of memory for the variable buffer using the function *malloc* which has the variable $buffer\_size$ as input. The other nodes are analyzed with the condition template, considering the variables and functions indicated by the master action, the result is shown in the next table:

Table 3.4: Condition Templates for CVE-2005-3192

Item	Description	Node	Node	Node	Node	Node
1	Node number	1	2	3	5	6
2	Previous node	Null	1	1	4	4
3	Next node	2, 3	4	4	7	7
4	Search	Variable	Variable	Variable	Variable	Variable
5	Name	buffer	buffer_size	buffer_size	buffer	buffer_size
6	Type	Pointer	Integer	Integer	Pointer	Integer
7	Condition follows master action	No	No	No	Yes	No
8	Condition	Fixed	Result	Result	Unchecked	Unchecked
9	Condition element		user_input	multiplication	Null	less than Max integer

The predicates derived from the template are:

Table 3.5: Predicates for CVE-2005-3192

Node	Predicate
1	Fixed(buffer)
2	Result(buffer_size, user_input)
3	Result(buffer_size, multiplication)
5	Unchecked(buffer, NULL)
6	Unchecked(buffer_size, less than Max_integer)

Once the predicates are ready, it is necessary to get the VDCs for each of the scenarios.

For instance, using the algorithm for scenario $\{1, 2, 4, 5, 7\}$, we have:

- The master action is:
$$"malloc(buffer, buffer\_size)"$$
- Node 1 is evaluated, it does not follow master action: $pre = "/Fixed(buffer)"$
- Node 2 is evaluated, it does not follow master action: $pre = "/Fixed(buffer) \wedge Result(buffer\_size, user\_input)"$
- Next node is master action, it is skipped
- Next node is node 5, it follows master action:
$post = ";Unchecked(buffer, NULL)"$
- Node 7 is the exit node and iteration for this scenario finishes

The complete VDC expression for this scenario is printed:

$$malloc(buffer, buffer\_size)/Fixed(buffer) \wedge Result(buffer\_size, user\_input)$$
$$;Unchecked(buffer, NULL)$$

The VDC expressions for the rest of the scenarios are:

- Scenario $\{1, 2, 4, 6, 7\}$:

$$malloc(buffer, buffer\_size)/Fixed(buffer) \wedge Result(buffer\_size, user\_input) \wedge$$
$$Unchecked(buffer\_size, lessthanMax\_integer)$$

- Scenario $\{1, 3, 4, 5, 7\}$:

$$malloc(buffer, buffer\_size)/Fixed(buffer) \wedge Result(buffer\_size, multiplication)$$
$$;Unchecked(buffer, NULL)$$

- Scenario $\{1, 3, 4, 6, 7\}$:

$$malloc(buffer, buffer\_size)/Fixed(buffer) \wedge Result(buffer\_size, multiplication) \wedge$$

$$Unchecked(buffer\_size, lessthanMax\_integer)$$

3.4.2 VDC for CVE-2009-1274 vulnerability

Consider the SGM for CVE-2009-1274 in figure 3.2, which shows how a buffer overflow vulnerability is caused in the *xine* media player.

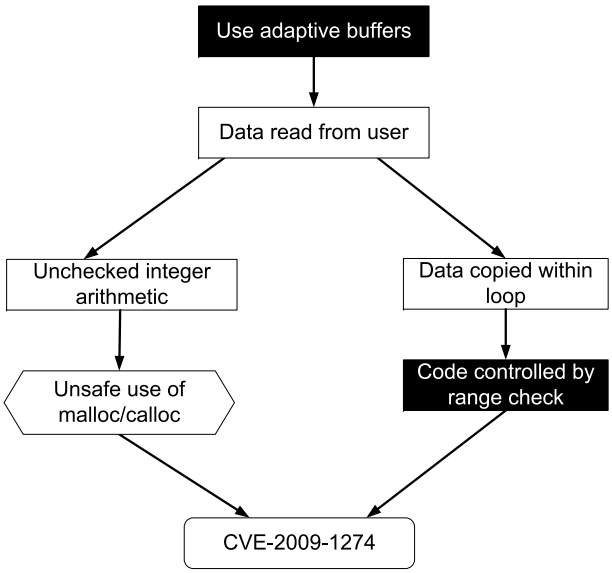


Figure 3.2: SGM for CVE-2009-1274

Analyzing this model we observe the following features: there are six different subgoals. Two of them are counteracting subgoals: use adaptive buffers and code controlled by range check; while subgoal unsafe use of malloc/calloc is associated with a SGM. Thus, we have to transform this graph for creating the VDCs: we replace the subgoal unsafe use of malloc/calloc with its associated SGM, we also replace the counteracting subgoals with contributing ones and the resulting graph is shown in Figure 3.4.

Applying the semantic transformation to the SGM of Figure 3.4, the resulting scenario suite contains three scenarios that cause the modeled vulnerability (CVE-2009-1274):

S = {{1,2,3,4,5,9}, {1,2,3,4,6,9}, {1,2,7,8,9} }

Now a vulnerability detection condition has to be defined for each of these scenarios.

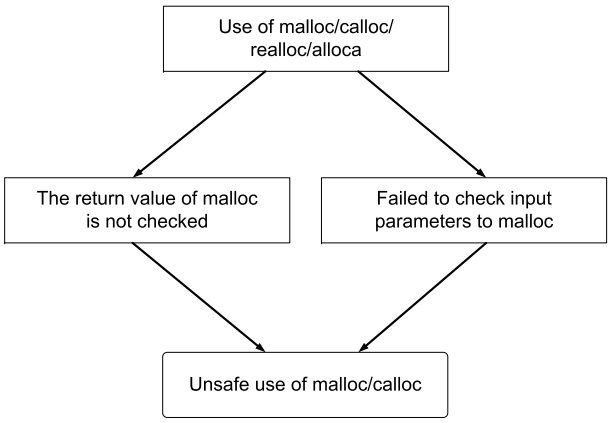


Figure 3.3: SGM for Subgoal Unsafe Use of malloc/calloc

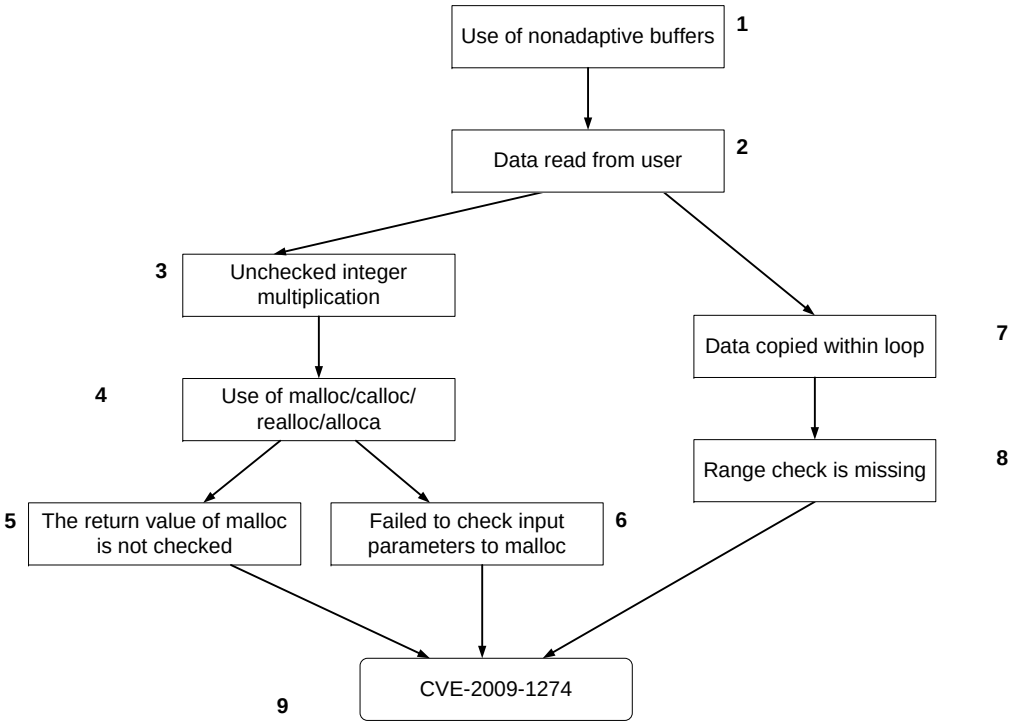


Figure 3.4: Transformed SGM for a Buffer Overflow in *xine*

The next part consists in identifying master actions. In our case we can identify two dif-

ferent master actions that lead to the vulnerability, given by nodes 4 and 7. The templates associated with these nodes are as follows.

Table 3.6: Master Action Templates for CVE-2009-1274

Item	**Node**	**Node**
1. Node number	4	7
2. Previous node	3	2
3. Next node	5,6	8
4. Function name	Alloc	CopyData
5. Input parameter name	buffer_size	user_input, loop_counter
6. Input parameter type	integer	string, integer
7. Variable that receives function result	buffer	buffer
8. Type of the variable that receives function result	pointer	pointer

Summarizing, we have that variable $buffer\_size$ has to be considered at least in the templates for nodes 4, 5 and 6. Node 7 is processed in a similar manner and the results of the analysis for both master actions are shown in Table 3.6.

The master action expressions are:

Alloc(buffer, buffer_size) and *CopyData(loop_counter, user_input).*

Table 3.7: Condition Templates for CVE-2009-1274

Item	**Node**	**Node**	**Node**	**Node**	**Node**	**Node**
1. Node number	1	2	3	5	6	8
2. Previous node	Null	1	2	4	4	7
3. Next node	2, 7	3	4	10	9	10
4. Search	Variable	Variable	Variable	Variable	Variable	Variable
5. Name	buffer	buffer_size	buffer_size	buffer	buffer_size	loop_counter
6. Type	pointer	integer	integer	pointer	integer	integer
7. Condition follows master action	No	No	No	Yes	No	No
8. Condition	Fixed	Result	Result	Unchecked	Unchecked	Unchecked
9. Condition element		user_input	*null*	*null*	buffer_bounds	counter_bounds

The other nodes are analyzed with the condition template, considering the variables and functions indicated by master actions. The process is shown with one node, number 2, which is about reading data from the user. Such a condition creates problems if the data is used in an integer arithmetic operation or if the data is copied inside a loop. We define a variable `user_input` in node 2, that holds the data provided by the user. The

same variable has to appear in nodes 3 and 7 to keep the relation. Table 3.7 contains the results of the analysis for nodes 1, 2, and 3, 5, 6 and 8.

The predicates derived from the templates are listed in table 3.8.

Table 3.8: Predicates for CVE-2009-1274

Node	**Predicate**
1	Fixed(buffer)
2	Result(buffer_size, user_input)
3	Result(buffer_size, arithmetic)
5	Unchecked(buffer, *null*)
6	Unchecked(buffer_size, buffer_bounds)
8	Unchecked(loop_counter, counter_bounds)

Finally, the vulnerability detection condition for scenario $\{1, 2, 3, 4, 5, 9\}$ is given by the expression:

$$alloc(buffer, buffer\_size)/Fixed(buffer) \wedge Result(buffer\_size, user\_input)$$

$$\wedge Result(buffer\_size, arithmetic)$$

$$; Unchecked(buffer, NULL)$$

This vulnerability detection condition expresses a potential vulnerability when the memory space for a non-adaptive buffer is allocated using the function `malloc` (or similar) whose size is calculated using data that is obtained from the user and the return value from memory allocation is not checked with respect to *null*.

In a similar way the VDCs for scenarios 2 and 3 are generated and the VDC for CVE-2009-1274 is given by the expression:

$$VDC = VDC_1 \vee VDC_2 \vee VDC_3$$

3.4.3 VDC for CVE-2006-5525 vulnerability

This vulnerability is more complex since the VCG includes several composed nodes, conjunction nodes and some qualitative causes as shown in Figure 3.5.

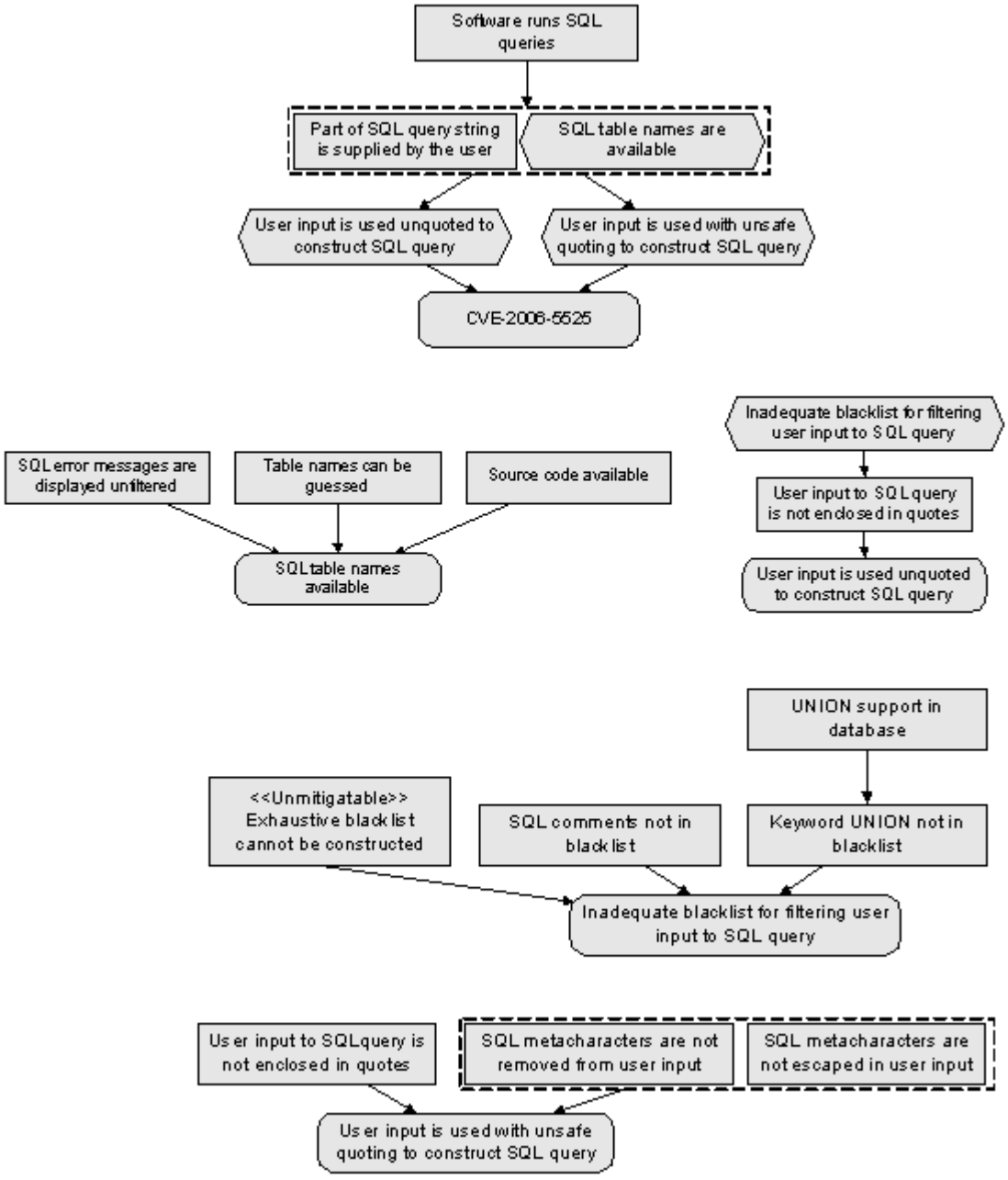


Figure 3.5: VCG for CVE-2006-5525

First, it is necessary to simplify the VCG in order to identify the scenarios. The simplification is done following these steps:

1. Discarding all qualitative nodes. In this case: "Tables names can be guessed", "Source code available" and "Exhaustive blacklist cannot be constructed".
2. The conjunctions nodes are converted into two sequential nodes.
3. The composed nodes are replaced by their components.

After this simplification the resulting VCG is shown in Figure 3.6:

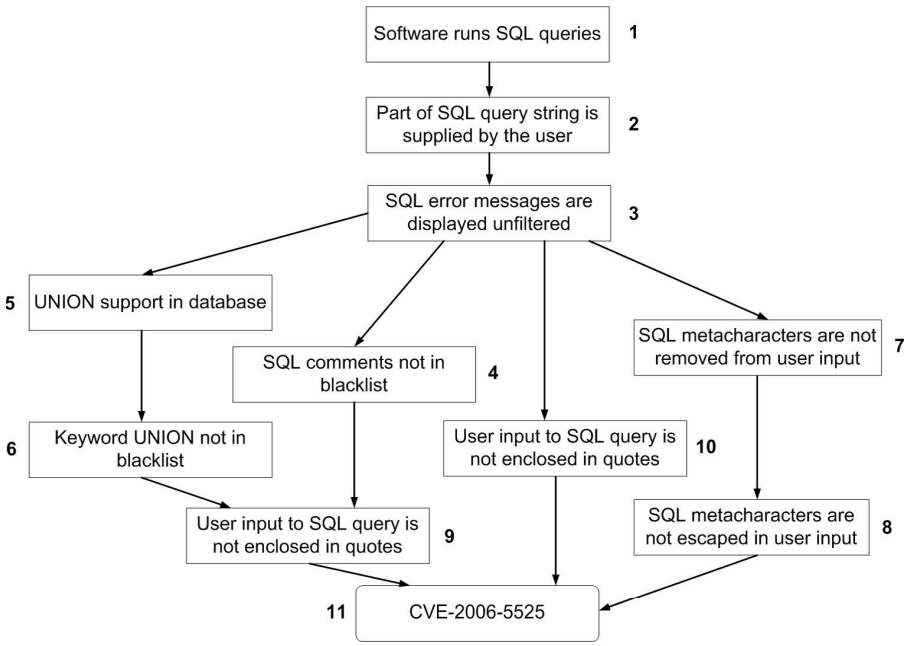


Figure 3.6: Simplified VCG for CVE-2006-5525

In this case, the vulnerable action is the use of SQL queries as indicated in node one, so filling the master action template we obtain:

Table 3.9: Master Action Template for CVE-2006-5525

Item	Description	Node
1	Node number	1
2	Previous node	Null
3	Next node	2
4	Function name	Sql_query
5	Input parameter name	s
6	Input parameter type	string
7	Variable that receives function result	
8	Type of the variable that receives function result	

The expression for the master action is $Sql\_query(s)$; or the execution of a SQL query using a variable s of string type.

The condition templates for the other nodes are:

Table 3.10: Condition Template for CVE-2006-5525, 1/3

Item	Description	Node	Node	Node
1	Node number	2	3	4
2	Previous node	1	2	3
3	Next node	3	4, 5, 7, 10	9
4	Search	Variable	Sql error message	Black list
5	Name	s	sql error message	black list
6	Type	string	sql error message	black list
7	Condition follows master action	No	Yes	No
9	Condition	Result	Unfiltered	Missing
10	Condition element	user_input		sql comments

Table 3.11: Condition Template for CVE-2006-5525, 2/3

Item	Description	Node	Node	Node
1	Node number	5	6	7
2	Previous node	3	5	3
3	Next node	6	9	8
4	Search	sql query	black list	variable
5	Name	sql query	black list	s
6	Type	sql query	black list	s
7	Condition follows master action	No	No	No
9	Condition	Support	missing	contains
10	Condition element	UNION	UNION	metacharacter

Table 3.12: Condition Template for CVE-2006-5525, 3/3

Item	Description	Node	Node	Node
1	Node number	8	9	10
2	Previous node	7	4,6	3
3	Next node	11	11	11
4	Search	variable	variable	variable
5	Name	s	s	s
6	Type	string	string	string
7	Condition follows master action	no	no	no
9	Condition	not escaped	contains	contains
10	Condition element	metacharacters	unquoted content	unquoted content

The predicates are then:

Table 3.13: Predicates for CVE-2006-5525

Node	Predicate
2	Result(s, user_input)
3	Unfiltered(ErrorMessage)
4	Missing(blacklist, sql comments)
5	Support(Sql_query, UNION)
6	Missing(blaclist,UNION)
7	Content(s, metacharacters)
8	Not_escaped(s, metacharacters)
9	Content(s, unquoted_content)
10	Content(s, unquoted_content)

The VCG in Figure 3.5 has 4 different scenarios that could lead to the vulnerability and their expressions for each of them are:

– Scenario $\{1, 2, 3, 4, 9, 11\}$:

$$Sql\_query(s)/Result(s, user\_input) \wedge Missing(blacklist, sqlcomments) \wedge$$

$$Content(s, unquoted\_content); Unfiltered(ErrorMessage)$$

– Scenario $\{1, 2, 3, 5, 6, 9, 11\}$:

$$Sql\_query(s)/Result(s, user\_input) \wedge Support(Sql\_query, UNION) \wedge$$

$$Missing(blaclist, UNION) \wedge Content(s, unquoted\_content); Unfiltered(ErrorMessage)$$

– Scenario $\{1, 2, 3, 10, 11\}$:

$$Sql\_query(s)/Result(s, user\_input) \wedge Content(s, unquoted\_content);$$

$$Unfiltered(ErrorMessage)$$

– Scenario $\{1, 2, 3, 7, 8, 11\}$:

$$Sql\_query(s)/Result(s, user\_input) \wedge Content(s, metacharacters)$$

$$\wedge Not\_escaped(s, metacharacters); Unfiltered(ErrorMessage)$$

3.5 Detection of Vulnerabilities using Passive Testing and VDC

In active testing special inputs are crafted to test the behavior of the system under test while in passive testing the behavior is evaluated through the monitoring of its execution trace without entering any special data. The trace is compared to formal models to determine the presence or not of faults on it.

A passive testing technique combined with our approach has been used in a prototype tool developed by Montimage. The tool called TestInv-Code [33] detects software vulnerabilities in C programs using VDCs as inputs and analyzing the traces of the program execution. The VDCs are expressed in XML, which are then translated to a set of predicates or patterns that may appear in the code under evaluation. The tool aims at detecting vulnerabilities in an application by analyzing the traces of the code while it is executing. By traces we mean here the disassembled instructions that are being executed. They are produced by executing the program under the control of the *TestInv-Code* tool, similar to what a debugger does.

In order to use the *TestInv-Code* tool the first step consists in defining the vulnerabilities causes that are of interest. Starting from informal descriptions of the vulnerabilities and VDCs models, a set of conditions that lead to a vulnerability are derived. These conditions are formally specified as regular expressions that constitute the first input for *TestInv-Code* tool.

Thus, passive testing using *TestInv-Code* proceeds along the following steps:

1. *Informal definition of vulnerable scenarios.* A security expert describes the different scenarios under which a vulnerability may appear.

2. *Definition of VDC.* A VDC expressing formally the occurrence of the related vulnerability is created for each possible situation that leads to the vulnerability.
3. *Instantiation of the VDC.* The predicates of the VDCs need to be instantiated with specific information related to the programming environment used. This will result in regular expressions that can be matched against the trace.
4. *Vulnerability checking.* Finally, *TestInv-Code* checks for evidence of the vulnerabilities during the execution of the program. Using the VDCs and the corresponding regular expressions, it will analyze the execution traces to produce messages identifying the vulnerabilities found, if any, and indicating where they are located in the code.

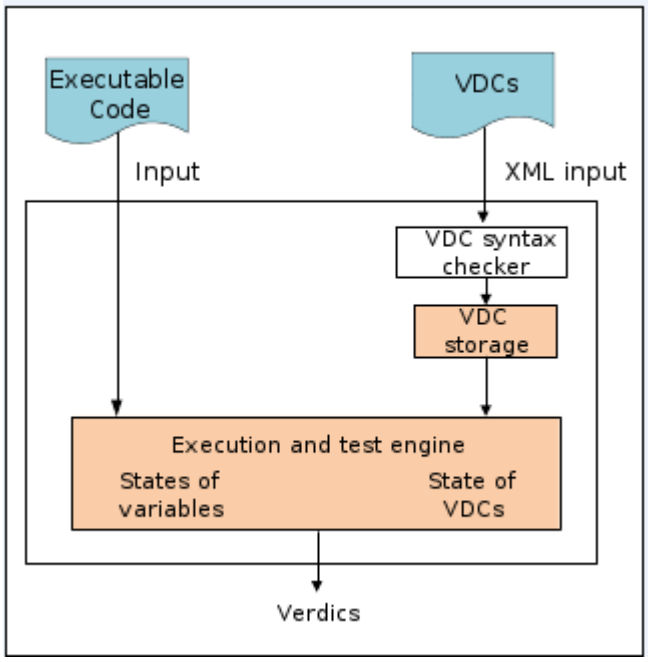


Figure 3.7: Passive Testing for Vulnerability Detection

Consider for instance the following $VDC1$:

$$Assign(x, y)/Is\_notAllocated(x)$$

where variables x and y are the generic representation of any variable/expression inside a C program. In practice to perform the vulnerability detection, those variables have to

be instantiated with real variables/expression from the C program. If we consider the following C program as input of the tool:

```
void main(){
1. int *a;
2. int b;
3. *a=b+2;}
```

then the tool will instantiate, at each point of the program, the predicates that appear in $VDC1$ to produce the following formula:

- At point 1: $Is\_notAllocated(a)$
- At point 2: $Is\_notAllocated(a)$
- At point 3: $Assign(a, b)$ and $Is\_notAllocated(a)$. Then in step 4, the tool detects the vulnerability in the program since VDC1 is verified at point 3.

Figure 3.7 depicts the passive testing architecture for vulnerability detection. As shown, the *TestInv-Code* tool takes as input:

1. *The vulnerability causes.* The file containing the vulnerabilities causes formally specified using VDCs and the corresponding regular expressions.
2. *The executable.* The Executable Linked Format (ELF) file for the application that is to be tested. This file contains the binary code of the application and it should include debug information if we want the tool to be able to determine the line of code where the vulnerability occurs and processes them to produce the final verdicts.

3.6 Conclusions

Vulnerability models as VCG and SGM are useful to understand how vulnerabilities are created in a program since they show the sequence of causes ("events" or "actions") that may lead to known vulnerability. These models could be valuable to train programmers about vulnerabilities but unfortunately they cannot be directly used in vulnerability detection. Their constraint is due to the use of natural language to express the caused.

In this chapter we have presented a method to formally describe the causes indicated in the vulnerability model. In a first step, the causes expressed in natural language are

analyzed using templates; these templates are predefined tables that help us to formalize them. Later the content of the templates are used to generate the VDCs expressions which are the formal expression of the sequence of causes that creates the vulnerability.

The formal expression of VDCs makes them suitable to be used in automated tools, and it has been proved in practice by Montimage and their passive testing tool TestInv-Code. This tool takes VDCs expressed in XML as input which are then translated to a set of predicates or patterns that are later verified in the execution trace of the program under evaluation and finally emits a verdict about the presence of the vulnerability.

Chapter 4

Vulnerabilities Detection using Model Checking

Contents

This chapter presents a model-checking based approach for dynamic detection of vulnerabilities in a C program. In our work, we are interested in C programs that read data from users because C language is a popular and performance programming language that can be vulnerable if it is not use with precautions. Considering that some C standard functions are vulnerable due to the lack of automatic bounds checking; then it is important to avoid vulnerabilities when creating programs in C. In our approach we define translation rules and security assertions in order to generate a formal specification in Promela from the considered C code. On the generated formal specification, the Spin model checker is used to detect the presence or not of vulnerabilities. If there is any assertion violation, due to a possible vulnerability, a counter example is given by Spin. It is then used as a test case of the real C program to confirm the vulnerability presence. The use of model checking permits a maximal coverage of the state space, that is, better vulnerability detection. The goal of our work is to show the feasibility of the approach that combines model checking and fault injection and needs improvements to take all the vulnerable concepts of the C language. This work has been published mainly in [28].

4.1 Overview of Spin Model Checker

Promela is a formal language originally devoted for the analysis and verification of communication protocols [20]; its syntax is very similar to that of C language, but it also includes control flow statements based on Dijkstra's guarded commands. A Promela formal specification is composed of set processes that can communicate among them through shared memory represented as global variables. Each process consists of a set of actions that are considered asynchronous and interleaved, which means that in every step only one enabled action is performed and without any additional assumptions of the relative speed of the process execution.

The spin model checker can do random or interactive simulations to perform the validation of the Promela model by scanning the state space, if there is a property violation then a counterexample is generated. The main advantage of Spin is the generation of optimized verifiers from a Promela model [22]. The verifier is setup to be fast and to use a minimal amount of memory. The exhaustive verifications performed by Spin are conclusive, estab-

lishing whether or not a system's behavior is error-free. In fact, very large verification that cannot be performed with automated techniques can be run in Spin with the "bit state space" technique. This method collapses the state space to a few bits per system state stored.

4.2 Overview of the Approach

Our proposal to deal with vulnerabilities using the Spin model checker includes the following phases (see Figure 4.1):

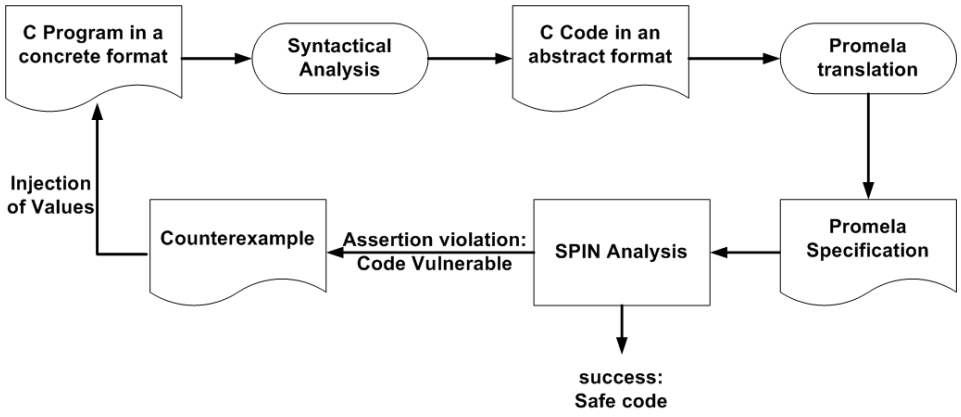


Figure 4.1: Detecting Vulnerabilities in C using Spin Model Checker

1. In a first step, the C code is transformed into an abstract form in order to be processed to generate a Promela specification. To this end, we have defined a sub-set of C language that includes, among others, the declarations of buffers and arithmetic data.
2. In the second step, a Promela formal specification is generated from the C program to analyze. The Promela specification contains the useful information about the vulnerable elements that appear in the C code. In case of buffer for instance, we generate its size and also the size of the data it holds at each moment. To detect vulnerabilities, we add some assertions to state that each vulnerable element must

always be in a safe state. For buffer for example, we add an assertion to specify that the size of the data it holds is always less than its size.

3. To detect the presence/absence of vulnerabilities, the Spin model checker is used on the generated Promela specification. The presence of a vulnerability is detected by an assertion violation. In that case, a counterexample is returned by Spin to give the values of the different variables that cause the violation of the related assertion.

4. To highlight the part of the C code that corresponds to the detected vulnerability, we apply a fault injection approach by replaying the counterexample found in the previous step on the C program.

The following section describes the generation of the Promela formal specification from a C program in order to detect vulnerabilities. It is important to note that the goal of the work presented here is to show the feasibility of the approach. This is why the approach does not deal with all the concepts the C language. Some ideas to extend approach are included as future work. Our approach considers a subset of the ANSI C language satisfying the following assumptions:

- the ANSI-C program has been already preprocessed, e.g., all the $\#define$ directives are expanded (inline expansion).
- the supported C types are primitive integer and array.
- function calls are expanded (inline expansion), the return statement is replaced by an assignment (if the function returns a value).
- all the arithmetic operations contain two operands, that is, they are of the form $(z := x\ op\ y)$. Multi-operand operations are transformed into binary operations by introducing temporary variables. For instance, operation $(a = a_1 + a_2 + a_3)$ is replaced by the following two binary operations: $a_{temp} = a_1 + a_2$, $a = a_{temp} + a_2$. This transformation is necessary in order to detect any arithmetic overflow. Indeed, multi-operand operations may mask some overflows like in statement $(x = maxint + 1 - 1)$ where the sub-assignment $(x = maxint + 1)$ produces an overflow while its equivalent $(x = maxint)$ does not.
- the declaration of variables, buffers,arrays pointers, and files,
- the predefined C functions like input/output functions ($printf$, $scanf$, etc.) and also

the functions on buffers (*strcpy*, *strncpy*) and files ($fscanf$, $fgets$),

– the assignment, the sequence and the choice (**IF**) statements.

Considering these elements, the present chapter presents the detection of the following vulnerabilities: arithmetic overflow/underflow, buffer overflow, incorrect array index, vulnerabilities related to pointers.

4.3 The Promela Model of a C Program

An important step in our approach is the creation of a Promela model based on the C program. The C program structure we consider include: functions, variables, statements and expressions and comments; while the Promela [9] model consists of processes, message channels and variables.

In this transformation process C functions are mapped to Promela processes. For instance, the main function in C is translated into the init process in Promela. C statements and expressions are mapped to their Promela equivalent while the comments are removed. Regarding the C variables, we only consider those who can have overflows or underflows as explained in the next section.

4.3.1 Mapping of C Functions

To facilitate the presentation of our approach, we consider that the C program contains a unique function, that is, the particular main function that denotes the entry point of the program. In other words, we make the hypothesis that all other functions are expanded in the main function.

The main function is translated into the *init* Promela process that is implicitly active (i.e., running) in the initial system state as shown in the rule mapping of Table 4.1.

4.3.2 Mapping of Buffers

To detect buffer overflows, we have to generate information about the maximum size of each declared buffer and also the amount of the data that it contains at any time. This information is stored in a constant $buffer\_max$ and a variable $buffer\_used$. The

C Code	Promela Code

«Global Variable Declarations»
main() {
«Local Variable Declarations»
S
}
	«Global Variable Declarations»
	init() {
	«Local Variable Declarations»
	S′
	}

Table 4.1: Translation C to Promela

Buffer declaration	$buffer\_max$
char buff[N]	N
char buff[N]="$hello\_world$"	N
char buff[];	0
char buff[] = {$'a'$,$'b'$,$'c'$}	3
*char *buff* ="$hello\_world$"	11

Table 4.2: Definition of the size of a buffer

value of constant $buffer\_max$ is defined according to the declaration of the buffer. Some examples are given in table 4.2.

Variable $buffer\_used$ is updated each time a new value is assigned to the buffer using different C instructions that modify the value of the buffer. Table 4.3 gives some examples.

For functions $gets$, $fscanf$ and $scanf$, we have to deal with the limit cases, that is, the buffer may contain data of maximum size: $buffer\_used = buffer\_max + N$ where N denotes any big number. For our experiments, we have chosen $buffer\_used$ equal to $buffer\_max + 1$ to simulate a potential attacker. To detect a potential vulnerability at each point of the program, we have to add the following assertion after each assignment on variable $buffer\_used$

$$\textbf{ASSERT}(buffer\_used \leq buffer\_max)$$

4.3.3 Mapping of C Language Variables

Since we are considering C programs that read data from users, it is necessary to simulate such process in the model in order to have an automatic verification. However, we first have to review the variable types in C and its correspondence in Promela types.

Buffer assignment	**PROMELA Translation**
$char\ buff[N]$="$hello\_world$"	$buffer\_used = 11;$
$buff = malloc(sizeof(char*)\ *\ N)$	$buffer\_max = N;$ $buffer\_used = 0;$
$gets(buff)$	$buffer\_used = buffer\_max + 1$
$fscanf(file\_name, "\%s", buff)$	
$scanf("\%s", buff)$	
$fgets(buff, size, fp)$	$buff\_used := size;$
$read(fd, buff, size)$	
$strcpy(dest\_buff, source\_buff)$	$dest\_buff\_used = source\_buff\_used;$
$strncpy(dest\_buff, src\_buff, size)$	$dest\_buff\_used = size;$

Table 4.3: PROMELA translation of C instructions on buffers

In Table 4.4 we present the default variable types in C language, while Table 4.5 presents the Promela types.

Name	Size	Range
short int	2 bytes	-2^{15} *to* $2^{15}-1$
unsigned short int	2 bytes	0 *to* $2^{16}-1$
unsigned int	4 bytes	0 *to* $2^{32}-1$
int	4 bytes	-2^{31} *to* $2^{31}-1$
unsigned long int	4 bytes	0 *to* $2^{32}-1$
long int	4 bytes	-2^{31} *to* $2^{31}-1$
unsigned char	1 byte	0 *to* $2^{8}-1$
char	1 byte	-2^{7} *to* $2^{7}-1$
bool	1 byte	true or false
float	4 bytes	
double	8 bytes	
long double	12 bytes	

Table 4.4: C language variables

As we can observe comparing both tables there is no complete correspondence between all the C variables and the Promela variables which means we have to map the variables. Nevertheless we are not interested in all C variables but those that are more susceptible to overflow or underflow like integer types or those who can be easily converted to integers like char types. These C variables are all transformed to integers in Promela, but establishing

Name	Size	Range
bit	1 bit	0 to 1
bool	1 bit	False, true
byte	8 bit	$0\ to\ 2^8 - 1$
short	16 bit	$-2^{15} - 1\ to\ 2^{15} - 1$
int	32 bit	$\text{-}2^{31} - 1\ to\ 2^{31} - 1$

Table 4.5: Promela variable types

bounds that will help in the detection of vulnerabilities, as shown in Table 4.6. For instance, a variable v in C of type short integer is equivalent to the Promela variable v of type *int* where $MIN \leq v \leq MAX$ with $MIN = -32767$ and $MAX = +32767$.

C type	Promela type	Constraints (min, max value)
short int	Int	$-2^{15} \leq int \leq 2^{15} - 1$
Unsigned short int	int	$0 \leq int \leq 2^{16} - 1$
unsigned int	int	int
Int	int	int
Unsigned long int	int	int
Long int	int	int
unsigned char	int	$0 \leq int \leq 2^8 - 1$
char	int	$-2^7 \leq int \leq 2^7 - 1$
bool	bit or bool	

Table 4.6: C variables transformed into Promela

Let us remark that since the *Unsigned long int* type has been mapped into int whose domain is smaller, it is possible to get a false positive during the verification of the Promela code.

4.3.4 The C Language Input Functions

To simulate any of the input functions of C (e.g., $scanf$) that read a variable v of type t, whose minimum and maximal values are t_min and t_max respectively, we use a Promela process called $input\_t$, which randomly generates a value to the corresponding variable. This Promela process is specified as follow (t' denotes the Promela translation of type t):

```
int t_min= MIN;
int t_max= MAX;
t' result;
proctype input_t(){
result = t_min;
do
  :: if
     ::(result  > t_max-step1) -> result= t_min+step2;
     ::(result <= t_max-step1) -> result= result+step1;
     fi
  :: break;
od
}
```

Let us give some explanations about process $input\_t$. This process is used to produce a value for a variable of type t whose translation into Promela is t'. The definition of process $input\_t$ is based on a global variable result, which is initialized to the minimum value of type t'. In order to consider all the possible values of result, the variable is incremented by an amount equal to $step1$ until the maximal value is reached. If t_max is reached before any assertion violation, we check other values by going back ($result = t\_min + step2$) with $step2 \neq step1$. We chose $step1$ different from $step2$ in order to obtain different values when going back. Indeed, if $step1$ and $step2$ are equal, the process will generate the same values as the first iteration. Also, we consider $step1$ and $step2$ less or equal to t_max in order to avoid arithmetic overflow/underflow since the beginning of the program. At any moment, we can break the loop in order to produce the final value of result.

Recall that our goal is to read a value for a variable v of type t. So, each C $scanf$ function to read a variable v is translated into Promela by the two following statements:

```
run input_t();
timeout -> v=result;
```

To read the Promela variable v, process $input\_t$ is launched, then we have to wait that it ends its execution, once finished predicate $timeout$ becomes true to state that no

statement is executable in any other active process. In that case, value of result is assigned to variable v.

4.3.5 Arithmetic Overflow/Underflow

To detect arithmetic overflow/underflow, we have to check the result of each arithmetic operation. Since an arithmetic operation can be composed by one or more variables or expressions then we have to consider only binary operations, that is, we transform each multi-operation into several binary operations by introducing temporary variables; the goal is to detect the specific operation or value that causes the problem. For instance, the following C arithmetic statement:

$$a = b + c + d + e$$

can be grouped as $a = (((b + c) + d) + e)$ and then we use the temporary variables to perform the operation:

$a1 = b + c;$
$a2 = a1 + d;$
$a = a2 + e;$

Also, to detect type conversion overflow/underflow, we transform any assignment ($v1 = v2$) into a binary expression ($v1 = v2 + 0$). We make such a transformation in order to use the same approach to detect all kinds of overflow/underflow vulnerabilities.

After transforming all arithmetic operations into binary ones, type overflow/underflow can be detected by defining the following Promela process that check the result z of an operation ($z = x + y$) with respect to the values of both operands x and y. For instance, when variables x and y are both positive, the value z must be greater than both ($z \geq x \wedge z \geq y$) otherwise a type overflow is detected. Process *check* is defined as follows:

```
proctype check(int x, y, z){
   if
    ::(x>=0) -> if
                ::(y>=0) -> assert(z>=x & z>=y);
```

```
                ::(y<0) -> if
                           ::(x> -y) -> assert(z>=0);
                           ::(x< -y) -> assert(z<0);
                          fi
            fi
    ::(x<0) -> if
                ::(y>=0) -> if
                              ::(y>= -x) -> assert(z>=0);
                              ::(y< -x) -> assert(z<0);
                           fi
                ::(y<0)-> assert(z<=x & z<=y);
               fi
   fi
}
```

Process *check* being defined, we translate each C arithmetic operation ($zz = xx + yy$) by the following Promela statements:

```
zz = xx+ yy;
run check(xx,yy,zz);
```

Let us remark that process *check* can also detect type conversion vulnerability.

4.3.6 Incorrect Array Index

For every C array a of $size\_a$ items whose index of variable type t is i, then any time an element of the array is going to be accessed we add the following two assertions previous to the element access :

```
assert(i<size_a);
assert(i>=0);
```

4.3.7 Vulnerabilities on Pointers

This section describes how we deal with two main vulnerabilities on pointers but without considering neither the arithmetic operations on pointers nor the aliasing concept. On pointers, two known vulnerabilities are: double free on pointers and deleting unallocated pointers. Double free vulnerability means that in a corresponding C code a pointer $*p$ is declared and the programmer tries to delete the allocated memory for p twice, while deleting unallocated pointers means that the programmer tries to delete a memory which is not allocated before. To detect such vulnerabilities, we define a variable $alloc\_p$ to memorize the number of times a memory is allocated. This variable is updated as follows: initialized to 0, $alloc\_p$ is incremented each time an allocation statement on p is used; it is decremented each time a free statement is applied on it. To detect vulnerabilities on pointers, we generate the following Promela statements from the C program:

- Allocation statement on p: such a statement is translated into:
  ```
  assert(alloc_p == 0);
  alloc_p = alloc_p+1;
  ```
  the first statement checks that the memory has not already allocated.
- Free statement on p: such a statement is translated into:
  ```
  assert(alloc_p == 1);
  alloc_p = alloc_p-1;
  ```
  the first statement checks that the memory is really allocated before any statement. In this way, a double free memory can be detected.
- Other statements on p: this case denotes the use of variable p. So, we have to check that the used memory is allocated by generating the following Promela assertion:
  ```
  assert(alloc_p == 1);
  ```

4.4 Injecting Data into a C Code for Detecting Vulnerabilities

The counterexample returned by SPIN should correspond to a real vulnerability in the initial C program, but it may not be always the case due to the inherent limitations

of Promela language and also because we do not prove the correctness of the translation rules. So, in order to guarantee that the counterexample is valid, an injection test is performed. Then, the C program is executed and fed with input data based on the resulting counterexample in order to verify that a vulnerability is present in the program. The process of injecting data is not trivial and different options are possible.

1. The produced counterexample directly provides input data for the initial C program.

2. The values of variables corresponded to input data values have been changed during the program execution and SPIN produced the last values of those variables.

We note that in the former case corresponding input data could be injected manually by a user but automatic injection is preferable, i.e., it is desirable to automate the injection process. In both cases, in order to confirm that the alarm was not false we modify the initial C code injecting special corresponding assignments in some parts of the initial code. For this reason, we study C code instructions, so-called input functions, which deal with input data. A C function is an input function if it reads the value (or values) of a variable (or several variables) from a keyboard or from a file. These input functions are $scanf$, $getc$, $read$, $fscanf$ e.t.c. Two cases are considered below.

- Let SPIN produce the value e for a variable v as a counterexample. In this case, we scan the C program in order to find an input function that reads variable v. The C program is then modified by direct injecting the instruction $(v = e)$ after the corresponding input function. We run the modified program and if no error message about incorrect data appears but the result of the program is incorrect then a program is vulnerable, and we output this information. If such an error message appears for the value e of the variable v then we conclude that the program is safe w.r.t. this vulnerability.
- Let SPIN produce the value e for a variable v as a counterexample; however, in the PROMELA code the v value has been recalculated several times and e is its current value. In this case, after running an input function in a corresponding PROMELA code we put instruction $printf(v)$ in order to get the counterexample that has to be injected into the initial C code. Afterward we proceed as in Case 1.

4.5 Tool Support

To make the presented approach workable, we are developing a tool called SecInject - Security Injection whose architecture is depicted in Figure 4.2.

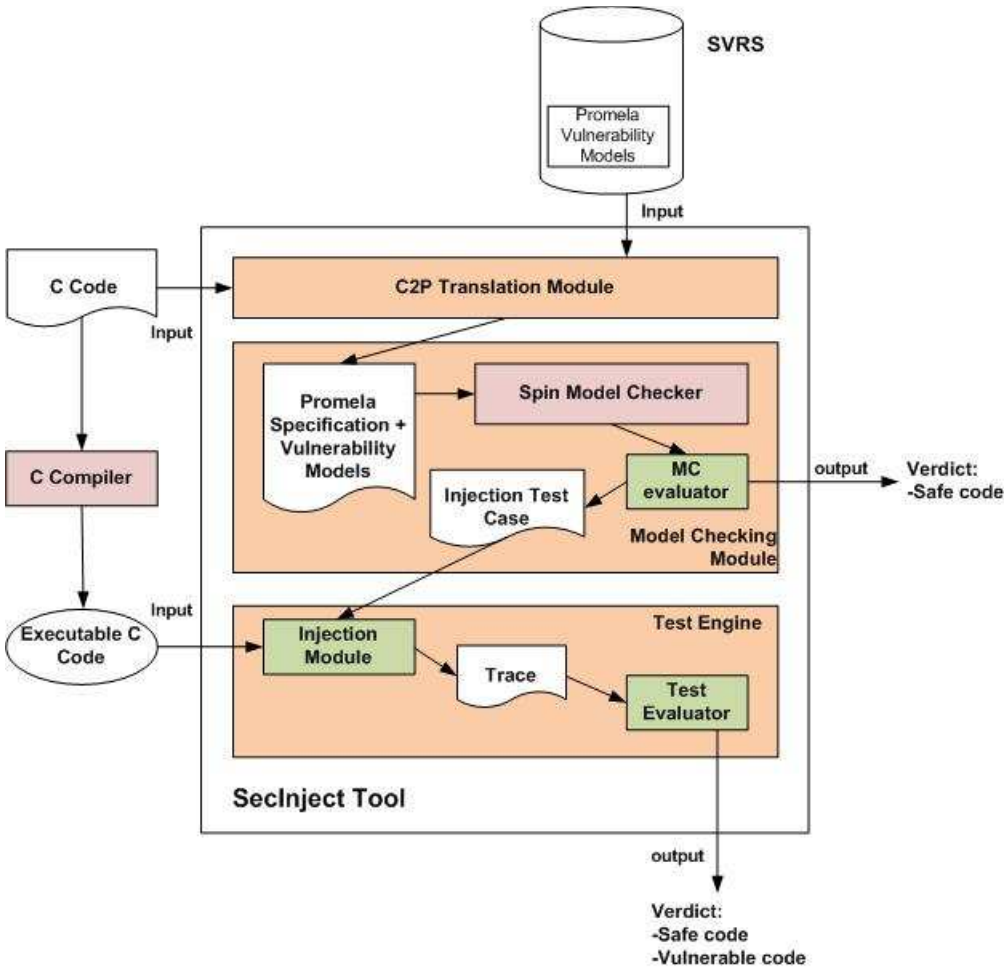


Figure 4.2: SecInject Tool Architecture

The following part briefly explains the architecture of the SecInject tool which is composed of three main modules:

1. C2P Translation module: Takes two inputs and combines then into a unique Promela specification. First input is the C code to be tested, which is translated into Promela. The second input is a Promela model of the vulnerability that is taken from the Shields SVRS repository.

2. The Model Checking module: Executes the Spin model checker with the unique Promela specification generated in the previous module to determine if the specification is correct, if yes, a verdict of safe code is emitted, otherwise an injection test case if prepared with the counterexample.
3. The Test Engine: Takes the test case and injects the specified values to the executable C code and evaluates the response to give a verdict confirming the detection of the vulnerability if any or safe code on the contrary.

4.6 Illustrative Example

To illustrate our approach, let us apply the different translation rules to the following C program:

```
#include <stdio.h>
int main(void){
   int n,n1,n2;
   printf("Enter first integer n1=  ");
   scanf("%d", &n1);
   printf("\nEnter second integer n2= ");
   scanf("%d", &n2);
   if (n1>=n2) n=n1+n2;
   else n=n2-n1;
}
```

and then we obtain the Promela specification below:

```
int n;
int n1;
int n2;
int v;
int t_min = -1073741824;
int t_max = 1073741824;
```

```
int step1 = 2;
int step2 = 3;

proctype input_t(){
v=1;
do
::
     if
       :: (v > t_max) -> v = t_min+step2 ; break;
       :: (true) ->  v = v +step1;
      fi
:: break;
od
}

proctype check(int x, y, z){
   if
    ::(x>=0) -> if
         ::(y>=0) -> assert(z >= x); assert(z >= y);
::(y<0) -> if
   ::(x > -y) -> assert(z >= 0);
   ::(x < -y) -> assert(z < 0);
   fi
fi
    ::(x<0) -> if
        ::(y>=0) -> if
        ::(y >= -x) -> assert(z >= 0);
   ::(y< -x) -> assert(z < 0);
   fi
        ::(y<0)-> assert(z <= x); assert(z <= y);
        fi
```

```
   fi
}
init {
   run input_t();
   timeout -> n1=v;
   run input_t();
   timeout -> n2=v;

  if
     :: (n1 >= n2) -> n = (n1 + n2);  run check(n1, n2, n);
     :: else -> n = (n2 - n1);  run check(-n1, n2, n);
  fi;
}
```

This Promela specification is verified by the Spin model checker. It detects an assertion violation for the addition and gives a counterexample with the values of n1 and n2 that cause it:

$$n1 = 1073741834 \; and \; n2 = 1073741834$$

Now, in order to determine if these values create a vulnerability in the C program, we inject both values to the executable C code and we probe the result is not correct, so the C program is vulnerable.

4.7 Conclusion

The advantage of the model checking based approach is that the C code does not need to be changed in order to test it for vulnerability detection; however the difficulty resides in obtaining the right model of the code under evaluation. Our approach considers C programs that read data from users, since this is one of the main sources of vulnerabilities because if the data provided is not correctly validated it can cause a vulnerability during run time with undetermined consequences.

Our method takes a C program that read data from users and transforms it to a Promela

model using some transformation rules. Then some assertions are added to state that each vulnerable element must always be in a safe state. The model and the assertions conforms a Promela specification that is verified using Spin model checker. An assertion violation is a sign of a vulnerability in the C program, so the counterexample given by Spin is used by a fault injector to demonstrate the presence of the vulnerability in the original C program. Some tests were done to show the validity of our method, which proved to be useful to find specific values for which a C program is vulnerable.

The approach presented in this chapter is close to that introduced in [37] that uses also model-checking technique. However, we think that our approach is more general since it does not deal with buffers only but it considers more C constructs and vulnerability kinds. In addition, our approach does not modify the initial C code since the assertions are generated automatically when producing the PROMELA code. Finally, our approach can be automated by adapting the Modex [30] tool dedicated to the verification of multi-threaded software that is written in the C programming language. We did not select this option because the Modex tool is complex and it is very hard to use it.

Chapter 5

Evaluation

Contents

This chapter presents some practical experiences of the approaches considered in this thesis. First, we present the work done in collaboration with Montimage enterprise. They have developed a tool to graphically describe the VDCs as well as a tool to detect vulnerabilities using VDCs. The tool is evaluated on a real size application XINE [1] which was written in C language. More details can be found in [13]. Finally, the model checking approach is applied to some well known algorithms.

5.1 Evaluation of the VDC-based Approach

This section describes the VDC tool editor developed by Montimage in order to design and store VDCs. It also mentions some of the features of TestInv-C vulnerability testing tool.

5.1.1 Montimage VDC Editor and TestInv-C Tool

The VDC editor was developed as a GOAT [2] plug-in. It offers security experts the possibility to create and store vulnerability detection conditions (VDCs). The VDC editor has the following functionalities:

- The creation of new VDCs corresponding to vulnerability causes from scratch and their storage in an XML format.
- The visualization of already conceived VDCs.
- The editing (modification) of existing VDCs in order to create new ones.

The VDCs are stored in an XML format file that constitutes one of the inputs for the Montimage vulnerability detection tool, called TestInv-Code. The VDCs are then instantiated in the tool in order to establish the conditions to be evaluated during program execution and detect a given vulnerability. A vulnerability is discovered if a VDC signature is detected on the execution trace.

In the editor a VDC is mainly composed of 3 parts:

1. *Master condition:* The triggering condition, also called master action (denoted a).

1. http://www.xine-project.org
2. http://www.ida.liu.se/divisions/adit/security/goat/

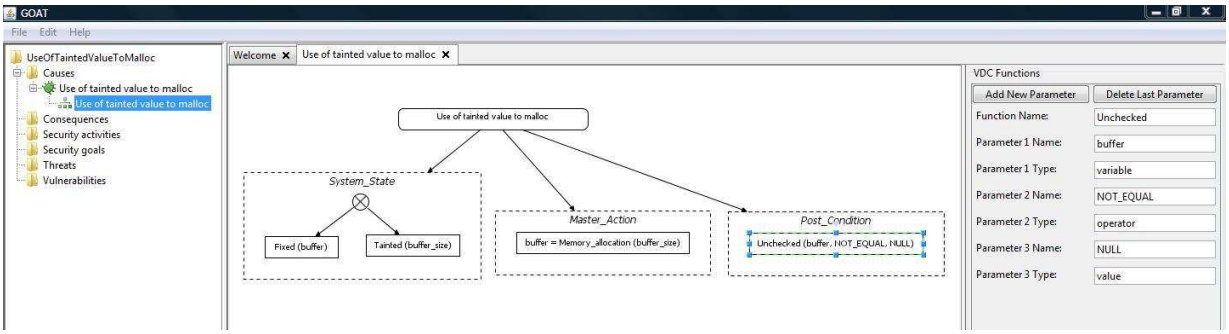


Figure 5.1: VDC for "Use of Tainted Value in *malloc*" in GOAT.

When analyzing the execution trace, if this condition is detected, we should verify if the state and post conditions of the VDC hold as well. If yes then a vulnerability has been detected. A master condition is mandatory in a VDC.

2. *State condition:* A set of conditions related to the system state (denoted $P(Var, Act)$). The state condition describes the states of the specified variables at the occurrence of the master action. We make the state condition also mandatory since predicate $P(Var, Act)$ is often present. If it is absent, the user can put value $True$ for this predicate.

3. *Post condition:* A set of conditions related to the system future state (denoted $P'(Var, Act)$). If a master action is detected in the state condition context, then we should verify if the post condition holds in the execution that follows. If this is the case, a vulnerability has been detected. Post condition is not mandatory in a VDC.

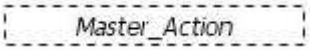
Figure 5.1 shows the top-level VDC for the buffer overflow vulnerability modeled in GOAT. This vulnerability detection condition concerns the use of a tainted value in a memory allocation. The textual representation of this VDC is:

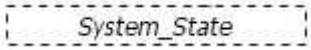
$$buffer = Memory\_allocation(buffer\_size)/(Fixed(buffer) \wedge Tainted(buffer\_size)); \\ Unchecked(buffer, Not\_equal, NULL)$$

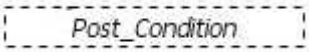
The graphical notation used to edit VDCs is defined as follows:

– *Root*: denoted by 

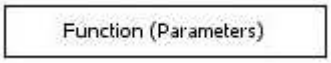
, the root node of VDC which represents the entire VDC.

- *Master action*: denoted by 

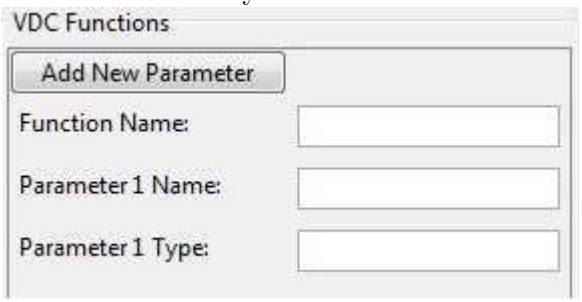
, it represents the triggering condition. If detected, the state and post conditions need to be checked.
- *State condition*: denoted by 

, it represents a set of conditions that should have occurred prior to the master action in order to detect the vulnerability.
- *Post condition*: denoted by 

, it represents a set of conditions that should occur following the master action in order to detect the vulnerability.

  SimpleCondition-Node: denoted by 

, it represents the label of a simple condition in VDCs.
- *VDC Function Parameters*: denoted by:



  a condition in a VDC is represented by a function with parameters (at least one). Each parameter has a name and a type (variable, operator or value).
- *And Gate*: denoted by 
, the "And" gate is used to build a conjunction of at least 2 conditions. The result is complex condition.
- *Or Gate*: denoted by 
, the "Or" gate is used to build a disjunction of at least 2 conditions. The result is complex con-dition.

5.1.2 XINE Application

XINE[3] an open source application and free multimedia player that plays back audio and video which was written in C, was selected as case study for the tool TestInv-C since it is a real world application, is open source code (code available free of copyright), and

3. http://www.xine-project.org

because it contains a number of known vulnerabilities which can be used to demonstrate the effectiveness of our detection approach.

The application contains a set of modules and libraries however we are interested in *xine-lib* [4] (xine core), module developed in C language. We selected an obsolete version of xine-lib that is known as vulnerable.

5.1.3 Case Study: XINE CVE-2009-1274 Vulnerability

The version v1.1.15 of XINE application has several vulnerabilities, in our case of study we dealt with with CVE-2009-1274 as explained below:

- Summary: Integer overflow in the qt_error parse_trak_atom function in de-muxers/demux_qt.c in xine-lib 1.1.16.2 and earlier allows remote attackers to execute arbitrary code via a Quicktime movie file with a large count value in an STTS atom, which triggers a heap-based buffer overflow.
- Published date: 04/08/2009
- CVSS Severity: 5.0 (MEDIUM)

The exploitation occurs when someone is trying to play with XINE a Quicktime encoded video that an attacker has modified to make one of its building blocks (the “time to sample" or STTS atom) have an incorrect value. The malformed STTS atom processing by XINE leads to an integer overflow that triggers a heap-based buffer overflow probably resulting in arbitrary code execution. The patch to this Vulnerability is in version v1.1.16.1, also included in v1.1.16.3.

CVE-2009-1274 can be considered as part of the family or of vulnerabilities class named “Integer Overflow", which has ID CWE 190 in the Common Weakness Enumeration database. The description of CWE 190 is summarized as follows “The software performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control". Figure 5.2 depicts the associated VCG [34].

4. Xine-lib source code can be downloaded from: http://sourceforge.net/projects/xine.

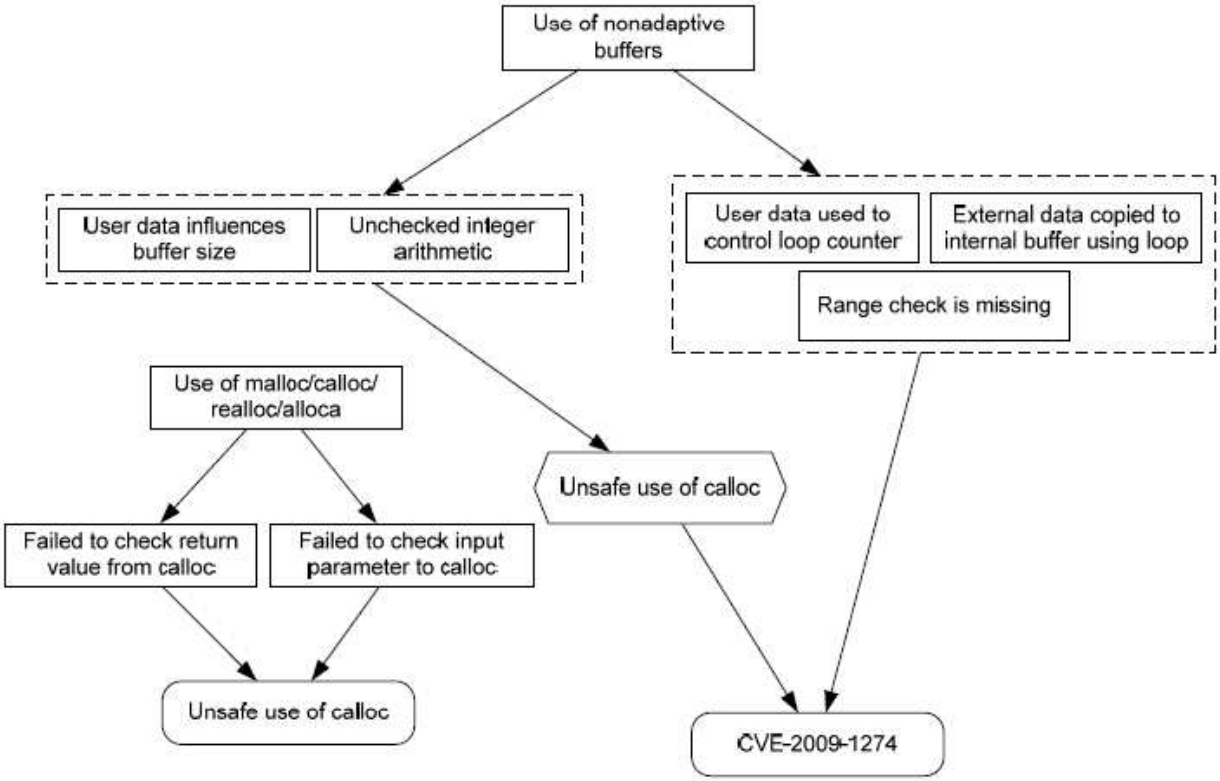


Figure 5.2: VCG for CVE-2009-1274

5.1.4 Vulnerability Modeling

Using the VCG that describes CVE-2009-1274 vulnerability, we have derived 3 VDCs and their corresponding regular expressions to be used as input to *TestInv-Code* tool.

1. Calloc(buffer, buffer_size) / Fixed(buffer) ∧ Result(buffer_size, user_input)∧ Result(buffer_size, addition); Unchecked(buffer, NULL)
2. Calloc(buffer, buffer_size) / Fixed(buffer) ∧ Result(buffer_size, user_input) ∧ Result(buffer_size, addition) ∧ Unchecked(buffer_size, buffer_bounds)
3. CopyVar(loop_counter, user_input) / Fixed(buffer) ∧ Unchecked(loop_counter, counter_bounds); CopyData(buffer, user_input, loop_counter)

Using VDC editor, the VDC models for each cause scenario can be built. For instance, figure 5.3 illustrates the VDC model for the first scenario. Created VDCs are inputs to TestInv-C tool.

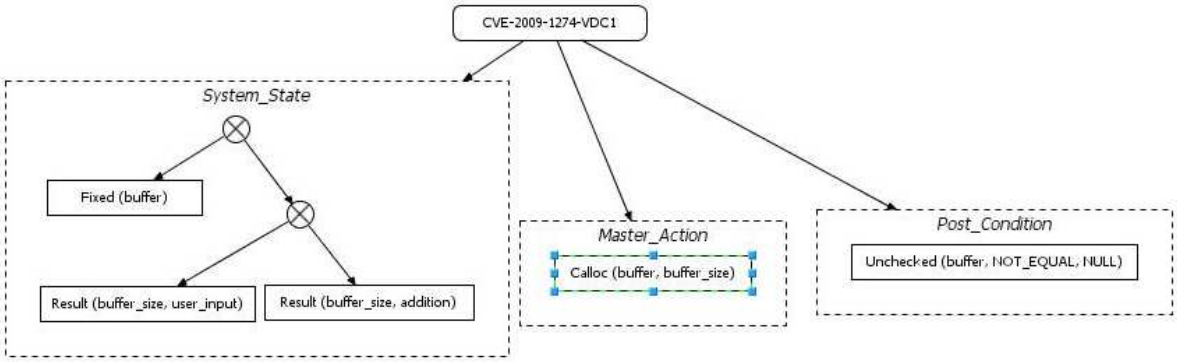


Figure 5.3: VDC Model for CVE-2009-1274 vulnerability

5.1.5 Application of TestInv-Code

In order to analyze the xine-lib it is necessary to run it. Actually, to be able to reach the plug-in that contains the error (the quicktime file demuxer), the muxine application was run with a quicktime file. The TestInv-C tool allows performing the analysis on all the application's functions (including those of the library and the plug-ins). The tool's user can also identify a function or set of functions to be analyzed. This feature is necessary to avoid performance issues, particularly in applications that perform intensive data manipulations (like video players). The complete list of available functions is obtained automatically. Another feature that helps improve the performance of the tool is the possibility of limiting the number of times a piece of code inside a loop is analyzed. Below the *XINE* code executed:

```
Code fragment from demux_qt.c
...
1907 trak->time_to_sample_table = calloc(
1908  trak->time_to_sample_count+1,
         sizeof(time_to_sample_table_t));
1909 if (!trak->time_to_sample_table) {
1910   last_error = QT_NO_MEMORY;
1911   goto free_trak;
1912 }
1913
```

```
1914 /* load the time to sample table */
1915 for(j=0;j<trak->time_to_sample_count;j++)
...
```

where `trak->time_to_sample_table` is tainted since it is set from information taken from the external QuickTime file.

The tool detects the vulnerability CVE-2009-1274 when it is launched on the mux-ine application using a quicktime video file. It is done activating the option to analyze all functions (of the application, the library and the plug-ins) or just the function parse_trak_atom in the quicktime plug-in. The result of the vulnerability testing provided by TestInv-Code is shown in figure 5.4.

5.1.6 Analysis

The same VDCs can be reused to analyze any code under the same programming environment to detect the same types of vulnerabilities. For instance, we applied the same VDCs on ppmunbox, a program developed by Linköpings university to remove borders from portable pixmap image files (ppm) and the same vulnerability was detected.

This vulnerability is located in the following section of ppmunbox.c file:

```
Code fragment from ppmunbox.c
...
76:/* Read the dimensions */
77:if(fscanf(fp_in,"%d%d%d",&cols,&rows &maxval)<3){
78: printf("unable to read dimensions from PPM file");
79: exit(1);
80 }
81:
82:/* Calculate some sizes */
83:pixBytes = (maxval > 255) ? 6 : 3;
84:rowBytes = pixBytes * cols;
85:rasterBytes=rows;rasterBytes=rowBytes*rows;
```

```
86:
87:/* Allocate the image */
88:img = malloc(sizeof(*img));
89:img->rows = rows;
90:img->cols = cols;
91:img->depth = (maxval > 255)?2:1;
92:p = (void*)malloc(rasterBytes);
93:img->raster = p;
94:
95:/* Read pixels into the buffer */
96:while (rows--) {
...
```

To illustrate the applicability and scalability of TestInv-Code, it has been tested with six different open source programs to determine if known vulnerabilities can be detected using a single model. The following paragraphs describe the vulnerabilities and give a short explanation of the results obtained, which are summarized in table 5.1.

Table 5.1: Summary of TestInv-Code results with different VDCs

Vulnerability	**Software**	**Detected ?**
CVE-2009-1274	XINE	Yes
Buffer overflow	ppmunbox	Yes
CVE-2004-0548	aspell	Yes (two)
CVE-2004-0557	SoX	Yes
CVE-2004-0559	libpng	Yes
CVE-2008-0411	Ghostscript	Yes

In the experiment, we checked the scalability of the tool by the application of a high number of VDCs (more than 100) to a software with intensive data use (as in the case of video decoders). The tool performance remained good in comparison to known dynamic code analysis tools in the market like Dmalloc, DynInst, and Valgrind. Indeed, the detection based on our tool does not insert a big overhead (the execution time is almost equal to the program execution time).

To optimize the analysis, the tool is being modified so that the user can select specific functions to check in the program. But in this case all the input parameters for this function are marked as tainted even if they are not. Another solution that is being studied is to only check the first iteration of loops in the program to avoid checking the same code more than once.

At present, we have checked applications written in C, which do not have a complex architecture. We are now starting to experiment more complex applications with architectures that integrate different modules, plugins, pointers to function, variable number of parameters or mixing different programming languages.

5.2 Evaluation of the SPIN Model-checking Based Approach

In chapter 4 we presented our model-checking approach to detect vulnerabilities in C programs. In this section we apply it to student's implementations of different purpose array algorithms such as minimal (maximal) item searching, sorting, average value calculating etc. Although those programs might be easily written they are widely used in a number of complex C codes.

As explained previously, in order to dynamically detect vulnerabilities in a C code for calculating an average value of array items we have to translate the corresponding C code into PROMELA code according to given rules presented. As test case, we consider a C code where array items are of the unsigned short type and *sred* is the variable where the average value is saved, the memory is allocated statically and the variable n which corresponds to the real array size (dimension) is an input to the code. When experimenting with the given implementation we compared the *sred* value with the maximal value for unsigned short type - 65535. Correspondingly an assertion ($sred < 65536$) was added into the corresponding PROMELA program. The SPIN model checker produced a counterexample for the assertion violation. The latter means that there exist such values for an array items that *sred* is larger than 65535. SPIN detected the type overflow vulnerability and produced the value 10005 as a counterexample for each array item value. For the case

where ($n = 10$), the assertion violated on the seventh cycle iteration for sred being equal 70035. The corresponding counterexample was injected into the initial C code that was not interrupted as well as no error message was produced. The result produced was 34514 while the right value should be 100050; integer 34514 equals 100050 w.r.t. module 65536. This result illustrates that a given C code has a vulnerability, i.e., the C code is unsafe according to the presence of type overflow vulnerability. In order to dynamically detect this vulnerability we inserted the following instruction into PROMELA code '$assert(n < 10)$'. The corresponding codes are presented below; the corresponding PROMELA code is developed by a student of Tomsk State University Anton Ermakov.

C code of calculating an average value of array items	PROMELA code

#include <iostream>
#include <conio.h>
using namespace std;
int main()
{
  unsigned short n=10, a[10];
  for(int i=0; i<n; i++)
  {
  scanf("%d",&a[i]);
  }
  unsigned short sred=0;
  for(int i=0; i<n; i++)
  {
  sred+=a[i];
  }
  sred/=n;
  printf("sred = %d",sred);
  getch();
}
	int result;
	proctype input_int ()
	{
	  ...
	}
	init
	{
	  int i;
	  for(i:1..10){
	  run input_int ();
	  a[i]=result;
	  printf("%d ",a[i]);
	  }
	  for(i:1..10){
	  sred=sred+a[i];
	  assert(sred<65536);
	  }
	  sred=sred/n;
	  assert(sred<65536);
	  printf("the value of sred=%d",sred);
	}

An array overflow vulnerability was also detected in other C codes for array algorithms when the memory was statically allocated and later a user used the real size of an array. Below there is a list of C programs where this vulnerability was detected.

1. C program for finding a minimal (maximal) item of a given array.
2. C program of the bubble sort;
3. C program of the insertion sort;
4. C program of finding primes in the interval [1, n] for a given n (Eratosthenes sieve).

When detecting vulnerabilities in the above C programs we never asked SPIN to generate another counterexample. The latter confirms that the rules for translating a C program into PROMELA instructions are correct. This correctness is also confirmed by the fact that when SPIN detects a vulnerability an initial C code indeed has such vulnerability and this fact is confirmed by the test injection

5.3 Conclusion

This chapter reports on the application of both VDC and model checking based approaches on real sized applications. The results obtained are very promising since the vulnerabilities are successfully detected with no false positive. Compared to the other existing tools/methods, our methods seem more workable and can be easily extended to consider other vulnerabilities. Indeed, the presented TestInv-C tool is extensible, which means that new vulnerabilities can be integrated easily by just adding their corresponding VDCs into the repository and defining some additional checking functions. This feature is interesting for the evolution and exploitability of the tool.

MyProjects-->*Xine Demo Testing Project* -->*TestLabs*-->*Xine Test Lab 1*

Testlab Execution Results Report

Project Name:	Xine Demo Testing Project
Project Description:	This is a TestInv-C Demo testing project for Xine application.
Testlab Name:	Xine Test Lab 1
Testlab Description:	This is a test lab that includes the VDCs for vulnerability CVE-2009-1274.
Execution start time:	2010-05-17 20:31:52

Testlab coverage

VDC Model
CVE-2009-1274-VDC1
CVE-2009-1274-VDC2
CVE-2009-1274-VDC3

VDCs coverage: 60%(3/5)

Test results summary

TestLab	Xine Test Lab 1	Passed VDCs	0/3
		Failed VDCs	3/3

Test results

VDC	Defect	Priority	Verdict
CVE-2009-1274-VDC1	Defect: CVE-2009-1274-VDC1	Essential	
CVE-2009-1274-VDC2	Defect: CVE-2009-1274-VDC2	Essential	
CVE-2009-1274-VDC3	Defect: CVE-2009-1274-VDC3	Essential	

Figure 5.4: Screenshot of TestInv-Code Result for *xine* Vulnerability

Chapter 6

Conclusion

Contents

This thesis was achieved in the context of the European Project: "SHIELDS, detecting known security vulnerabilities from within design and development tools". The main objective of this project was to contribute with innovative formal and tool-oriented approaches to detect software vulnerabilities. In the literature, there are several available tools/approaches that have been developed to detect vulnerabilities; nevertheless they are specific vulnerabilities oriented. Consequently, in the practice several tools should be used in order to cover a wider range of vulnerabilities that may be present in our application. Additionally, some tools are not documented enough to precise which vulnerabilities they deal with.

6.1 Contributions

In our work we addressed those limitations by defining a more general and extensible approach that allow to cover multiple vulnerabilities. Thus our contribution in this field can be summarized as:

1. First, defining a formal language called Vulnerability Detection Conditions (VDCs) to describe the presence of a software vulnerability without any ambiguity. The aim is to formally describe a vulnerability as the execution of a particular action in the program under very specific conditions. At the beginning this formal language was created to formalize the information given in natural language by a graphical representation of vulnerabilities called Security Goal Model (SGM). This model provides a visual representation of the different scenarios that lead to the occurrence of a known vulnerability. Although this model favors the communication between the different stakeholders of the development team, they lack a precise semantics that does not allow their use to automatically detect vulnerabilities. With the translation of these graphical notations into VDCs we assign a formal semantics and we establish the basis to automate the vulnerability detection process.

2. Second, formalization of SGMs. we have defined a syntax and semantics that helps in the transformation of a initial SGM into a SGM that is more suitable for VDC translation. In our study we have only considered SGMs that model vulnerabilities.

Additionally, we have proposed an intermediate format called templates, that formalize the description in natural language of the vulnerability causes provided by the vulnerability model.

The global approach to detect a software vulnerability begins with its SGM model, which is analyzed to fill the templates with the proper information. Later they are processed and translated into one or several VDCs. Some formal rules are defined to map SGMs into templates and to generate from them the VDCs. These VDCs can then be easily translated to conditions that can be checked during the execution of the program under evaluation. In fact, this approach was used By Montimage to develop a tool that uses VDCs combining dynamic code analysis and passive testing techniques to detect vulnerabilities in the execution trace of the program. It is important to note that this approach is general and might be applied to any programming language (like C, JAVA, etc.) since VDCs are defined in a generic manner. Finally, in practice we noticed that VDCs can also be directly derived from the vulnerability description (given in natural language) without the need of having the vulnerability model, giving more flexibility to this approach.

3. The third contribution consists of an alternative approach that uses model checking to detect vulnerabilities in C programs. It is based on the SPIN model checker and its associated formal input language Promela. We have defined a subset of the C language that might contain vulnerable statements, and we have proposed formal rules to translate them into Promela. The presence/absence of vulnerabilities are modeled as assertions. On the obtained Promela specification, the SPIN tool is launched to check the safety of the code or detect the presence of vulnerabilities by providing counterexamples. The approach combines a passive/active technique, since at the beginning the program is not executed but its model; once the counterexample is given it is used as input to the original program to demonstrate the existence of the vulnerability; avoiding having false positives.

6.2 Perspectives

The work presented here could be extended :

1. For the VDC-based approach, a repository may be built to store the templates of the most common vulnerabilities and their associated VDCs. Then they can be reused by programmers to secure theirs codes. It would be like providing a set of bricks that help to check a program against several kinds of known vulnerabilities. As noted before, the VDC formal language is generic, and can be extended to other programming languages. Also, more properties of the SGMs can be considered to enrich the transformation process, for instance, the information edges.
2. The model checking-based approach might be extended considering C recursive functions and more elaborated statements like loops and dynamic allocation of the memory; in order to apply it real-sized applications and be able to evaluate its scalability and performance. Also, it is necessary to consider the different compilers in this approach.

Bibliography

[1] http://www.shieldsproject.eu/.

[2] S. Ardi, D. Byers, and N. Shahmehri. Towards a Structured Unified Process for Software Security. In *Proceedings of the 2006 international workshop on Software engineering for secure systems*, SESS '06, pages 3–10. ACM, 2006.

[3] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *IEEE Symposium on Security and Privacy*, pages 387–401. IEEE Computer Society, 2008.

[4] E. Bayse, A. Cavalli, M. Núnez, and F. Zaïdi. A passive testing approach based on invariants: application to the wap. *Computer Networks and ISDN Systems*, 48(2):247–266, 2005.

[5] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST: Applications to Software Engineering. *INT. J. SOFTW. TOOLS TECHNOL. TRANSFER*, 2007.

[6] D. Byers, S. Ardi, N. Shahmehri, and C. Duma. Modeling Software Vulnerabilities with Vulnerability Cause Graphs. In *In Proceedings of the International Conference on Software Maintenance (ICSM06*, 2006.

[7] D. Byers and N. Shahmehri. Contagious errors: Understanding and avoiding issues with imaging drives containing faulty sectors. *Digital Investigation*, 5(1-2):29–33, 2008.

[8] D. Byers and N. Shahmehri. Unified modeling of attacks, vulnerabilities and security activities. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 36–42. ACM, 2010.

[9] A. Cavalli, E. Montes de Oca, W. Mallouli, and M. Lallali. Two Complementary Tools for the Formal Testing of Distributed Systems with Time Constraints. In D. Roberts, A. El-Saddik, and A. Ferscha, editors, *12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications, 27-29 October 2008, Vancouver, BC, Canada, Proceedings*, pages 315–318. IEEE Computer Society, 2008.

[10] CERT/CC. CERT/CC Statistics. (accessed october 2007).

[11] H. Chen and D. Wagner. MOPS: An Infrastructure for Examining Security Properties of Software. In V. Atluri, editor, *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, pages 235–244. ACM, 2002.

[12] B. Chess and J. West. Dynamic Taint Propagation: Finding Vulnerabilities without Attacking. *Inf. Secur. Tech. Rep.*, 13(1):33–39, January 2008.

[13] SHIELDS Project Consortium. Final report on inspection methods and prototype vulnerability recognition tools. SHIELDS Project Deliverable D4.3. http://www.shields-project.eu/.

[14] SHIELDS Project Consortium. Formalism definitions and representation schemata. SHIELDS Project Deliverable D2.1. http://www.shields-project.eu/.

[15] Coverity. Prevent. (accessed september 2008).

[16] Cqual. A tool for Adding Type Qualifiers to C. http://www.cs.umd.edu/ jfoster/cqual/.

[17] D. Brumley and T. Chiueh and R. Johnson and H. Lin and D. Song. RICH: Automatically Protecting against Integer- Based Vulnerabilities. In *In Symp. on Network and Distributed Systems Security*, 2007.

[18] Fortify. Fortify Software. Fortify SCA (accessed september 2008).

[19] R. Hadjidj, X. Yang, S. Tlili, and M. Debbabi. Model-Checking for Software Vulnerabilities Detection with Multi-Language Support. In L. Korba, S. Marsh, and R. Safavi-Naini, editors, *Sixth Annual Conference on Privacy, Security and Trust, PST 2008, October 1-3, 2008, Fredericton, New Brunswick, Canada*, pages 133–142. IEEE, 2008.

[20] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Inc., 1991.

[21] G.J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.

[22] G.J. Holzmann and D. Bosnacki. The Design of a Multicore Extension of the SPIN Model Checker. *IEEE Trans. Software Eng.*, 33(10):659–674, 2007.

[23] K. Jiang and B. Jonsson. Using SPIN to Model Check Concurrent Algorithms, Using a Translation from C to Promela. In *Proc. 2nd Swedish Workshop on Multi-Core Computing*, pages 67–69. Department of Information Technology, Uppsala University, 2009.

[24] W. Jimenez, A. Mammar, and A. Cavalli. Software Vulnerabilities, Prevention and Detection Methods: A Review. In A. Bagnato, editor, *SEC-MDA workshop, 24-June 2009, Enschede, The Netherlands*, 2009.

[25] H. Kim, T. Choi, S. Jung, H. Kim, O. Lee, and K. Doh. Applying Dataflow Analysis to Detecting Software Vulnerabilities. In *10th International Conference on Advanced Communication Technology*, pages 255–258, 2008.

[26] Klocwork. K7. (accessed september 2008).

[27] C. Kruegel, E. Kirda, and S. McAllister. Expanding Human Interactions for In-Depth Testing of Web Applications. In *11th Symposium on Recent Advances in Intrusion Detection (RAID), Boston, MA*, 2008.

[28] N. G. Kushik, A. Mammar, A. Cavalli, N. V. Yevtushenkoa, W. Jimenez, and E. Montes de Oca. A SPINbased Approach for Detecting Vulnerabilities in C Programs. *Automatic Control and Computer Sciences*, 46(7):131–143, 2012.

[29] V. Livshits and M. Lam. Finding Security Vulnerabilities in JAVA Applications with Static Analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18. USENIX Association, 2005.

[30] Modex. Modex web page http://cm.bell-labs.com/cm/cs/what/modex/index.html.

[31] G. Necula, S. Mcpeak, S. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *In International Conference on Compiler Construction*, pages 213–228, 2002.

[32] H. Peine, M. Jawurek, and S. Mandel. Security Goal Indicator Trees: A Model of Software Features that Supports Efficient Security Inspection. In *HASE*, pages 9–18. IEEE Computer Society, 2008.

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

[34] SHIELDS. Detecting Known Security Vulnerabilities from within Design and Development Tools. "D1.4 Final SHIELDS approach guide".

[35] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–382. ACM Press, 2006.

[36] A. Mathur W. Du. Vulnerability Testing of Software System Using Fault Injection. In *Proceeding of the International Conference on Dependable Systems and Networks (DSN 2000), Workshop On Dependability Versus Malicious Faults*, 2000.

[37] L. Wang, Q. Zhang, and P. Zhao. Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), 28-29 September 2008, Beijing, China*, pages 165–173. IEEE, 2008.

[38] D. Wheeler. Flawfinder. April 2007. http://www.dwheeler.com/flawfinder/.