Final thesis

Modeling and Pattern Matching Security Properties with Dependence Graphs

by

Pia Fåk

LITH-IDA-EX--05/067--SE

2005-08-22
Final thesis

Modeling and Pattern Matching Security Properties with Dependence Graphs

by Pia Fåk

LITH-IDA-EX--05/067--SE

Supervisor: John Wilander
Dept. of Computer and Information Science
at Linköpings universitet

Examiner: Professor Mariam Kamkar
Dept. of Computer and Information Science
at Linköpings universitet
Modeling and Pattern Matching Security Properties with Dependence Graphs

Pia Fåk

With an increasing number of computers connected to the Internet, the number of malicious attacks on computer systems also raises. The key to all successful attacks on information systems is finding a weak spot in the victim system. Some types of bugs in software can constitute such weak spots. This thesis presents and evaluates a technique for statically detecting such security related bugs. It models the analyzed program as well as different types of security bugs with dependence graphs. Errors are detected by searching the program graph model for subgraphs matching security bug models.

The technique has been implemented in a prototype tool called GraphMatch. Its accuracy and performance have been measured by analyzing open source application code for missing input validation vulnerabilities. The test results show that the accuracy obtained so far is low and the complexity of the algorithms currently used cause analysis times of several hours even for fairly small projects. Further research is needed to determine if the performance and accuracy can be improved.

information security, static analysis, dependence graphs, pattern matching
Abstract

With an increasing number of computers connected to the Internet, the number of malicious attacks on computer systems also raises. The key to all successful attacks on information systems is finding a weak spot in the victim system. Some types of bugs in software can constitute such weak spots. This thesis presents and evaluates a technique for statically detecting such security related bugs. It models the analyzed program as well as different types of security bugs with dependence graphs. Errors are detected by searching the program graph model for subgraphs matching security bug models.

The technique has been implemented in a prototype tool called Graph-Match. Its accuracy and performance have been measured by analyzing open source application code for missing input validation vulnerabilities. The test results show that the accuracy obtained so far is low and the complexity of the algorithms currently used cause analysis times of several hours even for fairly small projects. Further research is needed to determine if the performance and accuracy can be improved.

Keywords: information security, static analysis, dependence graphs, pattern matching
Acknowledgements

A number of people have been of great help to me during my work with this thesis. Some of them are listed below; I am much indebted to them all. If someone is forgotten—please forgive me.

John Wilander, my supervisor, has taken good care of me during my thesis project and provided me with everything I have needed. Whenever I have been stuck, he has helped me out and suggested solutions. His comments on the thesis have greatly added to its clarity and structure.

Mariam Kamkar, my examiner, has in spite of her very busy professional situation agreed to spend some time on the assessment of this thesis.

Patrik Wikström, my opponent, has with his very relevant comments and questions increased the quality of the thesis.

David Byers has on several occasions sacrificed some of his valuable time to let the GraphMatch project benefit from his experience in static analysis.

GrammaTech Inc., who let researchers use their tool CodeSurfer without fee has made the thesis project possible to carry out. They have also provided me with a non-released plug-in for extracting graphical representations of CodeSurfer graphs. In particular, I would like to thank Chi-Hua Chen, who has very promptly and correctly answered all my questions regarding CodeSurfer.

Everyone at the Programming Environments Lab have been very helpful and friendly during my stay with them. Many of them have come up with very helpful suggestions on graph pattern matching and testing of static security analysis tools.

Albin Sunnanbo, my boyfriend, has not only provided emotional support but also very useful advice on programming and thesis writing.
# Contents

1 Introduction 1
   1.1 Background ........................................... 2
   1.2 Objectives ............................................ 3
   1.3 Method Overview ...................................... 4
   1.4 Assumed Prior Knowledge ......................... 5
   1.5 Notes on Terminology .............................. 5
      1.5.1 Properties and Policies ..................... 6
      1.5.2 Example Terminology ......................... 7
      1.5.3 Policy Violations and Security Bugs ........ 7
   1.6 Thesis Overview .................................... 8

2 Dependence Graphs 9
   2.1 Terminology ........................................ 9
   2.2 Introduction to Dependence Graphs ............ 10
      2.2.1 The Program Dependence Graph ............ 11
      2.2.2 The System Dependence Graph .......... 12
   2.3 Modeling Security Properties using Dependence Graphs 14
      2.3.1 Example ..................................... 15
   2.4 Detecting Policy Violations using Dependence Graphs 16
      2.4.1 Notes on Terminology ..................... 17
      2.4.2 Embedded Negative Patterns ............ 17
      2.4.3 Modeling Individual Vertices ............ 18
      2.4.4 Non-Unifiable Negative Properties .... 19
3 Implementation 21

3.1 Overview ........................................... 21
  3.1.1 Motivation 1: Control of Resource Consumption ... 23
  3.1.2 Motivation 2: Independence ..................... 24
3.2 The Builder ....................................... 24
3.3 The Extractor and Second Builder ....................... 25
  3.3.1 Intermediary Data Format ......................... 25
  3.3.2 Extracted Information .......................... 25
3.4 The Matcher ........................................ 26
  3.4.1 Program Slicing ................................ 27
  3.4.2 Matching Example ............................... 30
  3.4.3 Matching Property Patterns ...................... 32
    Computing Dependency Closures ....................... 32
    Matching a Single Pattern ........................... 34
    Example ........................................... 37
  3.4.4 Matching Individual Vertices ..................... 46
    Basic Approach ................................... 46
    Improvement 1: Using Dependence Variables .......... 47
    Improvement 2: Inserting Artificial Definitions ...... 48
    Improvement 3: Allowing for External Validation .... 50

4 Results and Conclusions 53

4.1 Test Results ....................................... 53
  4.1.1 Synthesized Test Code ............................ 53
    Intraprocedural Test Cases .......................... 54
    Interprocedural Test Cases .......................... 57
  4.1.2 Open Source Application Code .................... 58
    Initial Problems .................................. 59
    Closure Depth Experiments .......................... 60
    Accuracy of Analysis ................................ 60
4.2 Future Work ....................................... 63
  4.2.1 The Complexity Issue ............................ 63
  4.2.2 The Accuracy Issue .............................. 64
  4.2.3 The Other Issues ................................ 64
CONTENTS

4.3 Conclusions ................................. 65

5 Terminology .................................. 67

Bibliography ................................... 70

Index ........................................... 73
Chapter 1

Introduction

*Behind every computer security problem and malicious attack lies a common enemy—bad software.*
— John Viega and Gary McGraw, *Building Secure Software*

With an increasing number of computers connected to the Internet, the number of malicious attacks on computer systems also raises. CERT Coordination Center at Carnegie Mellon University report an increase in the number of reported attack incidents on computer systems from 6 in 1988 to 137,529 in 2003. As from 2004, they will no longer provide any incident statistics, due to the fact that automated attack tools create such large numbers of incident reports that the figures become meaningless [1].

Clearly, every way of enhancing computer security is welcome. This thesis will present and evaluate a technique for detecting security related bugs in software. In this introductory chapter we will present some necessary background information as well as the objectives of the thesis work and the means of reaching them. A brief overview of the report and some notes on terminology used will also be given.
1.1 Background

The key to all successful attacks on information systems is finding a weak spot in the victim system, a vulnerability, and exploiting it. The weak spot may be a human, an insecure password, a back door deliberately built into the system, or a combination of several seemingly harmless factors. Many vulnerabilities are however undeliberately created by developers when building the software that runs on the system. Even the smallest programming mistake may cause unexpected and unwanted behavior in their programs, behavior that can be exploited by an attacker. We will refer to such exploitable unwanted behavior as a security bug, or sometimes just bug.

Although there are innumerable ways in which software can be vulnerable to a malicious attack, some classes of security bugs are very common and commonly exploited. Readers acquainted with computer security will recognize buffer overflows, format string attacks, double free() flaws, race conditions, insufficient input validation and SQL injection as examples of entry points for attackers. Many of these common security bugs can be detected by automated analysis of the source code, so called static analysis.

Lately, research efforts have resulted in a number of static security analysis tools, such as SPlint by Larochelle and Evans [2], BOON by Wagner et al [3], various type inference based tools using CQual [4][5], MOPS by Chen and Wagner [6], the Stanford Checker by Ashcraft and Engler [7] and many others. The majority of these only analyze C programs and require large efforts to adapt for new types of security bugs. Some of them also require their users to add annotations to their code in order to be useful. Since new programming languages and new common security bugs are likely to appear in the future, a more flexible approach would be appropriate in future security analyzers.

This thesis introduces a flexible technique for security bug detection. It models the analyzed program as well as different types of security bugs with dependence graphs. Errors are detected by searching the program dependence graph model for subgraphs matching security bug dependence graph models.

Dependence graphs are not associated with any specific programming language. Successful attempts to build correct dependence graphs for
C/C++ [8][9] and Java [10] have so far been made. The graph matching technique may be used with any language that can be modeled with dependence graphs.

Another advantage of the dependence graph matching method is that any security bug that may be modeled with dependence graphs may also be detected by it. A tool implementing the technique could reach a high rate of flexibility by letting the user choose security bugs to scan for from a database of models. Models may also be added to the database as needed.

1.2 Objectives

This thesis is a part of the ongoing software security research at the Programming Environments Lab at the Department of Computer and Information Science at Linköping University. Its purpose is to evaluate the dependence graph modeling and pattern matching technique for security bug detection. More precisely, we want to determine if it can be used in practical software development. We believe that software developers are, much like people in general, lazy. Therefore, we hold it highly probable that software developers want a security analysis tool to:

- emit few but relevant warnings on security bugs
- require practically no effort to use
- run as a part of every compilation or
- run as a night-time batch job

The accuracy that may be obtained by static analysis is limited by various factors. Some problems that should be solved to obtain an exact analysis are theoretically proven to be undecidable such as some alias problems [11]. Other problems are not undecidable, but still not practically possible to solve due to prohibitive computation times.

When building static security analysis tools, there is always a choice between producing false positives (raising false alarms) and false negatives (not reporting an actual error). There are different opinions on which of these two alternatives to prefer. Michael Howard and David LeBlanc write
in their book on secure programming, *Writing Secure Code*, that there is no use spending time on finding out if a reported potential security bug really can be exploited if it is easy to fix [12]. On the other hand, Musuvathi and Engler have experienced during their work with static analysis and model checking that reporting many bugs to software vendors is less efficient than reporting just a few. Thousands of reported bugs may result in very few of them being fixed. Their idea is that a user wants to be presented with the “5-10 bugs that really matter” [13].

Our opinion here goes along the lines of Musuvathi and Engler’s. Producing a lot of warnings that may well be false alarms may be contraproductive, so missing a few real security bugs is a better alternative. A still better alternative would be to rate bugs according to their chance of being false positives in combination with assessed severity. That would give the developer a chance to fix the most important bugs.

The above discussion in combination with the assumptions stated earlier allows for the formulation of the following requirements on an analysis tool. It should:

- produce few false alarms and rate detected security bugs according to severity.
- not require developers to add annotations to their code or otherwise drop hints to the tool what to do.
- either run in approximately the same time as an ordinary compilation, or at least be able to perform a complete analysis in a couple of hours time.

The goal of this thesis is to find out if these requirements can be fulfilled by a tool based on dependence graph pattern matching.

1.3 Method Overview

The dependence graph modeling and matching technique has been evaluated by way of implementing a prototype tool, which we call *GraphMatch*. Its accuracy and performance has been measured and analyzed. The prototype is currently capable of recognizing one type of potential security
bug in C programs: missing input validation. This limits the generality of results obtained, since detection of bugs may vary in difficulty. However, relevant conclusions on the technique can still be drawn due to the fact that the prototype uses algorithms that should work for any bug model.

Two categories of test cases have been used: Test code especially synthesized for the purpose and real code from an open source application. We used the synthesized test cases to verify that certain security bug cases were detected as expected. The application code was used to investigate GraphMatch’s behavior with respect to more realistic problems. The test results were analyzed with respect to analysis times and accuracy in terms of false positives and negatives.

1.4 Assumed Prior Knowledge

The readers of this thesis is assumed to be acquainted with the basics of computer science and computer programming. In particular, knowledge of basic C syntax and semantics is needed to understand the many examples that are based upon miniature C programs. Some algorithms will also be described using a general pseudocode notation.

We also assume that readers are acquainted with the basics of graph theory. Graph terminology such as vertex, edge, subgraph and path will be used but not explained.

1.5 Notes on Terminology

Technical terms will be introduced as they are needed in the remaining chapters of this thesis. The most important of them are listed in chapter 5 together with a brief explanation and reference to their original introduction context.

Here and now, we will discuss some terms that are essential for a correct interpretation of the rest of the thesis.
1.5.1 Properties and Policies

Two of the most central terms used in this thesis are security policy and security property. These are intuitively understandable concepts, although their exact meaning may vary from context to context. In this section we will provide their exact definitions as used in this thesis and discuss their relations to more well-known definitions. Readers who are contented with a less exact comprehension of the terms may skip this section.

We define a security policy to be a rule that defines acceptable and unacceptable programming practices with respect to a certain programming action. A programming action is something that a program does as a consequence of one or several source code statements, such as adding two integers, assigning a value to a variable or sending a chunk of data through a network socket. We call the action concerned by a security policy a policied action. It may be a file access, use of data affected by external input, memory buffer copying or any other action that requires extra care for security reasons.

We use security properties to describe classes of acceptable and unacceptable behavior with respect to a certain policy. We say that the program surrounding a policied action has a certain security property if it fulfills a predicate associated with the property. A negative security property is a property that is not accepted by a given security policy. Similarly, a positive security property is a property that is accepted by a given security policy.

A security policy can be described either by a negative security property that contains all programming practices that are not acceptable by the policy, or by a positive security property that contains all programming practices that are accepted by the policy. Security policy violations can be described by a set of negative security properties that represents all possible programming practices unacceptable by the policy.

The definitions above differ from earlier uses of the terms. Bowern Alpern and Fred B. Schneider define a property as a set of executions of a program in their paper on liveness properties from 1985 [14]. Membership in a property is determined by a predicate on a single execution. Their definition is similar to the security property definition used in this thesis. Here, though, we are not as interested in single executions as in the set of
all possible executions around a point of interest (such as a policied action). This is because the nature of dependence graphs is to model every possible execution instead of singled-out ones.

Similarly, Schneider defines a security policy as something that can be specified by giving a predicate on sets of executions in his article on enforceable security policies [15]. This is quite similar to the definition used here, since the security properties describing a certain security policy are predicates on the set of all executions around the policied action. Other uses of the term security policy involve any set of security rules for anything from whole organizations to use of individual network protocols.

1.5.2 Example Terminology

The input validation policy will be used as an example policy throughout this thesis. Informally, it can be put as “All external input should be validated before use”. The policied action here is the use of data originating from external sources. The policy can be described by the positive correct input validation property, informally defined as “The external data is correctly validated before this use”. The negative property describing correct input validation can be divided into several different cases, one of which is the missing input validation property, informally defined as “The external data is not validated at all before this use”.

This thesis will frequently use the input validation policy and its properties as examples. Whenever input validation is mentioned, we mean the policy in itself. When missing input validation or correct input validation is mentioned, we mean the properties just explained.

1.5.3 Policy Violations and Security Bugs

It is worth noting that security bugs are always security policy violations, but the contrary is not always true. A security policy violation may not always lead to an exploitable bug. For example, all violations of the input validation policy may not result in security bugs, since there may be occasions where validations is not necessary. We sometimes use the term potential security bug as an equivalent of security policy violation.
1.6 Thesis Overview

The listing below gives a brief presentation of the contents of the remaining chapters of this report.

Chapter 2 introduces dependence graphs and their possible use as a modeling tool for security properties. It also elaborates on the possibilities of detecting security policy violations in existing software by using property models as patterns.

Chapter 3 describes the implementation of GraphMatch, a tool for detection of security policy violations using the technique discussed in chapter 2.

Chapter 4 accounts for the experimental results of analysis performed with GraphMatch. A discussion of future work and final conclusions based on the test results forms the last part of the chapter.

Chapter 5 contains a comprehensive list of the most important terms used in the thesis, together with brief explanations and references to their original contexts.
Chapter 2

Dependence Graphs

The dependence graph is the central abstraction used in this thesis. This chapter will introduce the reader to its general design and the particular role it could serve in modeling of security properties and detection of security policy violations.

2.1 Terminology

Three terms related to program analysis in general must be explained before proceeding to the main dependence graph description.

**program point:** A program point is a statement, a control predicate or some other point of interest in a program, such as a variable declaration or the entry point of a function.

**definition:** A definition of a variable $v$ occurs at a program point where a value is assigned to $v$. This is sometimes also referred to as a *kill* of $v$.

**conditional definition:** A conditional definition, or *conditional kill* occurs at a program point where a value might be assigned to $v$, but $v$ is not definitely killed.
A use of a variable \( v \) occurs at a program point where the value of \( v \) (rather than its name) may be required.

The example shown in figure 2.1 and further explained below should clarify the definitions.

```
int x = 0;
int y = 0;
int* xp;
if(x < y) {
    xp = &x;
} else {
    xp = &y;
}
*xp = 2;
```

Figure 2.1: Terminology example source code.

All declarations and assignments in the code above are program points. So is the if branching point with its associated predicate \((x < y)\). The two initializations of \( x \) and \( y \) as well as the assignments to \( xp \) are definitions. The control predicate \((x < y)\) represents a use of both \( x \) and \( y \). The assignment of \(*xp\) in the last line is a typical example of a conditional kill. Depending on the predicate evaluation, \( xp \) might point to either \( x \) or \( y \). Therefore, both \( x \) and \( y \) are conditionally defined by the statement.

### 2.2 Introduction to Dependence Graphs

A dependence graph is a directed graph that provides an explicit representation of data and control dependencies between statements in a program. The general idea is to represent each program point as a vertex in a directed graph with edges representing dependencies between two points.
Dependence graphs have a variety of applications and therefore a number of different definitions. Ferrante et al. first introduced the Program Dependence Graph (PDG) in 1987 [16], although they were not the first to define a graph representation of program dependencies. Their graphs definition provides a modeling abstraction for monolithic, single-procedural programs. Dependencies between procedures in the same program, i.e. *interprocedural dependencies*, are not modeled.

Here, another definition, the System Dependence Graph (SDG), introduced by Horwitz et al. in 1990 [17], is used. The SDG models programs as collections of procedures connected by interprocedural dependencies. Each procedure is modeled with a PDG.

The subsections below will give a further, but not complete, description of the PDG and SDG. Readers who are interested in the exact definitions should refer to the original paper [17].

### 2.2.1 The Program Dependence Graph

Horwitz et al. represent three types of dependencies in their graphs: *control dependencies* and two types of *data dependencies*: data-flow dependence and def-order dependence. Def-order dependencies concern the order of definitions of a variable, which is useful for example when using dependence graph for code optimization. Data-flow dependencies concern the flow of data between program points, which is what we are interested in in this thesis. From now on, we will therefore use the term *data dependencies* when referring to data-flow dependencies. Dependencies are represented by directed edges between vertices. Each vertex represents a program point.

A program point $p_2$ is **control-dependent** on another point $p_1$ if $p_1$ is a control predicate and the execution of $p_2$ depends on the evaluation of the predicate at $p_1$. There is also a control dependency between the entry point of a function and each program point within the function that is not nested within a conditional or loop statement.

A program point $p_2$ is **data dependent** on another point $p_1$ if $p_1$ defines a variable $v$ that is used at $p_2$ and there are no intervening definitions of $v$ between $p_1$ and $p_2$.

A small example will hopefully clarify the program dependence graph concept. Consider the program and PDG presented in figure 2.2. Control
dependencies are shown as solid arrows, while dashed arrows represent data dependencies. The reader will note that every definition of a variable causes dependencies to one or more use points of that variable. It is also worth commenting that the $b = a$ vertex is data-dependent on the $a = 0$ vertex even though there is a definition of $a$ between them. That is because the in-between definition is enclosed in a conditional; there may still be a direct dependency between $a = 0$ and $b = a$.

```c
int main() {
    int a = 0;
    int b = 1;
    if(a < b) {
        a = a + 1;
    }
    b = a;
}
```

Figure 2.2: A single-procedure program and its corresponding program dependence graph. Solid arrows represent control dependencies, dashed arrows represent data dependencies.

### 2.2.2 The System Dependence Graph

The SDG is a collection of PDGs connected with interprocedural dependence edges. The interprocedural dependencies are modeled by a number of new kinds of vertices and edges:
• *call-site* vertices representing each function call and *call* edges representing the control dependency between the call site and the entry point of the called function.

• pairs of *actual-in* and *formal-in* vertices connected by *parameter-in* edges. They represent parameters passed to a function and global variables used by it.

• pairs of *formal-out* and *actual-out* vertices connected by *parameter-out* edges. They represent return values from the function and global variables defined in the called function.

In many contexts it is not necessary to differentiate between parameter-in and parameter-out edges. We will then refer to them as *interprocedural data edges*. We will also refer to call edges as *interprocedural control edges*.

```
int global = 0;

int f(int a) {
    global = global + a;
    return global;
}

int main() {
    int x = 0;
    x = f(x);
}
```

Figure 2.3: A two-procedure example program. Its corresponding SDG is shown in figure 2.4.

A small example will hopefully clarify the interprocedural dependency concept. Consider the program presented in figure 2.3. The SDG for the program is shown in 2.4. Call and {formal|actual}-{in|out} vertices are shown in boldface. Call edges are drawn as control dependencies and parameter-{in|out} edges as data dependencies. They are, however, marked
2.3 Modeling Security Properties using Dependence Graphs

John Wilander shows in his article *Modeling and Visualizing Security Properties of Code using Dependence Graphs* [18] that dependence graphs can be used to model security properties. The modeling concepts developed by Wilander and briefly presented below constitute the basis of this thesis work.

with *call*, *p-in* and *p-out*, respectively. Note how the global variable is treated as a hidden parameter to \( f() \).

**Figure 2.4**: System dependence graph for the program on page 13. Solid arrows represent control dependencies, dashed arrows represent data dependencies. Vertices and edges shown in boldface represent interprocedural dependencies.
A security property model consists, just like a PDG or an SDG, of vertices and edges. While the vertices of a PDG or SDG each represent a program point in a certain program, the vertices of a security property model each represent a class of program points in any program. Such a class may be described for example as “all program points that contain a definition of an integer variable”. Note that when we say program point, program vertex might have been used instead, since each program vertex represents a program point.

The edges of a security property model are generally transitive, so that they may represent a chain of dependencies rather than a single direct dependency between two program points. In this thesis, all model edges will be regarded as transitive unless specified as otherwise.

### 2.3.1 Example

We use two example security properties to illustrate the modeling concept: correct input validation and missing input validation. Figure 2.5 shows their dependence graph models.

![Dependence graph models](image)

**Figure 2.5: Dependence graph models of the correct input validation property (left) and the missing input validation property (right). Solid arrows represent control dependencies, dashed arrows represent data dependencies. The vertices represent in order of appearance: ext_input: an untrusted data source, def: definition of a variable, val: validation of a variable, use: sensitive use of a variable.**
Each of the vertices shown in figure 2.5 represents a class of program points as described below.

**ext_input:** Program points where input from a data source that cannot be trusted occurs. The data source may be network traffic, data read from a terminal or a user-defined file.

**def:** Program points containing a definition of a variable.

**val:** Program points containing correct validation a variable.

**use:** Program points containing sensitive use of a variable, such as non-bounds-checked copying into a limited buffer, use in pointer arithmetic or as buffer size specifier when copying data between buffers.

The edges connecting the vertices of the correct input validation model signify that

- the value used in the definition at a **def** program point depends on a variable defined at a **ext_input** program point.

- the variable validated at a **val** program point depends on the variable defined at the **def** program point.

- the variable used at a **use** program point also depends on the variable defined at the **def** program point.

- the result of the validation at the **val** program point controls the execution of the **use** program point.

The edge connecting the vertices of the missing input validation model signify that the variable used at a **use** point depends on a variable defined at a **ext_input** point. Note that this is also a description of the policied action of the input validation policy.

### 2.4 Detecting Policy Violations using Dependence Graphs

As we explained in 1.5.1, negative security properties represent violations of security policies. This fact allows us to detect policy violations by using
property models as patterns and match them in program SDGs. That is, whenever it is discovered that an SDG contains a subgraph which matches a negative security property pattern, we know that the program represented by the SDG contains a security policy violation.

How the matching is done is an implementation concern. Chapter 3 will give a detailed description of the methods used in this thesis project. The following subsections will introduce some general problems associated with the pattern matching. First, however, we will introduce and clarify some additional terminology that will be used in the remainder of the thesis.

2.4.1 Notes on Terminology

This section introduces the use of dependence graph security property models as graph patterns. From now on, we will use the term security property pattern or just property pattern to refer to such a model.

Dependence graphs will be mentioned frequently. Whenever an SDG or PDG is mentioned, we mean a graph as defined above. Sometimes, we will refer to dependence graphs. In these cases, we refer to some general definition of the term that may apply to both SDGs and PDGs. In the upcoming discussion we will often need to contrast dependence graphs and vertices representing parts of a specific program from a dependence graph pattern. In these cases, the terms program SDG, program edge and program vertex will be used to specify the program’s dependence structure. Pattern edge and pattern vertex will be used when we refer to the vertices and edges of a pattern.

Match is another word that will be used often and in different contexts. When referring to a program vertex that matches a pattern vertex, the term vertex match will be used. When referring to a subgraph of an SDG that matches an entire property pattern, the term subgraph match will be used.

2.4.2 Embedded Negative Patterns

Attentive readers may have observed that the missing input validation pattern presented above is present as a part of its positive counterpart, the
correct input validation pattern. We call such a pattern an *embedded negative pattern* and the positive pattern which it is a part of an *embedding pattern*. If we wish to search for violations of the input validation policy, it is not enough to look for matches for the missing input validation pattern. We also need to ensure that the instances found are not embedded in the positive correct input validation pattern.

Note that this does not contradict the statement that a security policy violation can always be described by a set of negative properties. The limiting factor here is not the security property, but the modeling strategy. If we had a way of modeling the actual absence of validation, the problem would not exist.

### 2.4.3 Modeling Individual Vertices

Descriptions of individual vertices have so far been inexact. The reason for this is that it is difficult to provide their exact definitions. The fairly simple correct input validation pattern will serve to exemplify the difficulties that may occur.

Our first problem is defining what external input is. This may differ from application to application. For a command-line tool, external input may be command-line arguments, environment variables and files read. For a network daemon, external input may come from the network. For an OS kernel, it can be reasonable to regard all data copied from user space as external input.

Assuming that we have chosen a suitable input model for the program type we are analyzing, the next step will be to determine how a correct validation should be performed and what uses of the variable must be protected. These two factors depend on each other, but also on the type of variable we are handling. Take for example string variables and integer variables:

The main danger with string variables is that they may cause buffer overruns if they are copied into limited buffers. The copying might thus be the sensitive use, and a correct validation of string variables might be checking the string variable’s length.

Integer variables can cause trouble when used as a size specifier when writing to a buffer, when used as an offset from a pointer location or when
used as an array index. Michael Howard points out in his article on integer vulnerabilities that seemingly validated integers may still cause trouble such as buffer overruns due to signedness errors and integer over/underflows [19]. A basic rule is that correct validation of a signed integer should check both its upper and lower bound. For unsigned integers it is in many cases enough to check an upper bound.

To accommodate for different types of variables we will thus need specialized input validation models with respect to vertex models. For the remainder of this report, we will discuss only integer validation models.

2.4.4 Non-Unifiable Negative Properties

It is worth noting that the missing input validation property is not a complete description of the input validation policy. There are a number of different cases of insufficient validation. For integer variable validation, some of the cases are shown in figure 2.6. From the picture, one may easily conclude that there is no way of modeling the property containing all of the negative properties of input validation with one dependence graph. We call this condition a non-unifiable negative property.

In the case of input validation, we are lucky: all of the models of negative properties contain the common element of missing input validation. Thus we may detect all cases of input validation violation by searching for missing input validation and then determine if the piece of code in question also matches correct input validation or one of the other negative patterns shown in figure 2.6.
Figure 2.6: Different incorrect validation property models. Solid arrows represent control dependencies, dashed arrows represent data dependencies. The vertices represent in order of appearance: ext_input: an untrusted data source, def: definition of a variable, insuff. val: insufficient validation of a variable, e.g. when a signed integer is checked only for its upper bound, val: validation of a variable, use: sensitive use of a variable.
Chapter 3

Implementation

So far, this report has discussed modeling and pattern matching of security properties with dependence graphs from a theoretical point of view. This chapter deals with the implementation of GraphMatch, a security policy violation detection tool based on the ideas presented in chapter 2. In contrast to chapter 2 which contains background information on earlier research in the area, the material presented here constitutes the result and contribution of this thesis work.

GraphMatch is currently at a prototype stage. It scans C source code and detects violations of the input validation policy introduced in chapter 1. We have chosen to limit the class of policy violations to include only integer variables affected by external input used in pointer arithmetic.

3.1 Overview

Figure 3.1 shows an overview of GraphMatch’s design. It is built from four parts: the Builder, the Extractor, the Second Builder and the Matcher.

The Builder builds system dependence graphs from source code. GraphMatch uses a commercial tool, CodeSurfer from GrammaTech Inc. [8], for this job. CodeSurfer is freely available for research use and is distributed in two major versions, a standard and a programmable version. Here, the pro-
3.1. Overview

Figure 3.1: Overview of the design of GraphMatch.
grammable version has been used, since it provides a possibility of browsing SDG information through a scripting interface. As CodeSurfer stores its dependence graphs using a proprietary file format, there is a need for the Extractor and the Second Builder. The Extractor transfers SDGs from CodeSurfer’s internal representation to to an intermediary text format. The extracted data can be parsed by the Second Builder and transformed into GraphMatch’s internal graph representation format. The Matcher traverses the graph created by the Second Builder and finds matches for property patterns. Matches for negative patterns are reported to the user as security violations.

All parts except for the Builder are implemented as a part of the thesis project. The Second Builder and Matcher are implemented as a standalone C++ application, while the Extractor uses the scripting API provided by CodeSurfer for browsing of SDGs and other structures. The Extractor script contains about 400 lines of code. The Second Builder/Matcher C++ program contains approximately 5000 lines of code. About 1000 of these are used for the SDG representation, 1500 consists of code for the Second Builder and the remaining 2500 lines are Matcher code. These figures give a brief overview of the size and complexity of the tool.

CodeSurfer’s scripting utility is sufficiently powerful to allow for a tool like GraphMatch to be implemented in it. The option of doing so was considered at the start of the project, but later rejected for the reasons described in the next two subsections.

3.1.1 Motivation 1: Control of Resource Consumption

Dependence graphs are information-intensive structures. There is a need for GraphMatch to handle large volumes of data, which means that resources consumed is a crucial issue. CodeSurfer keeps track of a lot more information than needed for the graph matching problem. Since it is a closed source program, there is no way to control how it will perform when resources get sparse. A standalone application can use less information than CodeSurfer and hence be better optimized for its purpose.
3.1.2 Motivation 2: Independence

There are several reasons for not letting the implementation depend too much on CodeSurfer:

- A future goal of the GraphMatch project is to detect policy violations in various languages using one tool consisting of a single Matcher and several Builders, one for each analyzed language. Implementing the Matcher using the CodeSurfers scripting utility would be a step in the opposite direction.

- GrammaTech may stop developing and supporting CodeSurfer.

- In a future commercialization of GraphMatch, a continued usage of CodeSurfer might become expensive.

- While using the API provided by GrammaTech, one is limited by the operations it can perform on the graph.

3.2 The Builder

The Builder transforms source code into dependence graphs. As we have already mentioned, the programmable version of the tool CodeSurfer does the building job. CodeSurfer is a powerful general-purpose static analysis tool for C/C++ code. It is capable of building not only dependence graphs, but also abstract syntax trees and control-flow graphs. Information held in these structures can be browsed either by a graphical user interface or by a Scheme scripting API. For a detailed description of CodeSurfer, refer to GrammaTech’s homepage [8].

To our knowledge, there is no other tool available that builds dependence graphs with equal precision. The only alternative to using CodeSurfer would thus be to design our own dependence graph builder, which would be far too time-consuming for a thesis project.
3.3 The Extractor and Second Builder

The Extractor transfers SDGs from CodeSurfers internal representation to an intermediary text format. The extracted data can be parsed by the Second Builder and transformed into GraphMatch’s internal graph representation format.

3.3.1 Intermediary Data Format

For ease of implementation, we chose to use eXtensible Markup Language (XML) [20] for the intermediary data format used by the Extractor and the Second Builder. XML is a tag-based markup language much like HTML, but in contrast to HTML, it has no pre-defined tags. The user defines his or her own XML data representation by specifying the tags and their allowed content.

One of the advantages of using XML for information transfer between programs is that there are several XML parser implementations available as libraries. GraphMatch uses libxml++ [21], which is a C++ wrapper for the Gnome project libxml2 parser [22].

But the choice of XML also has one major drawback: The XML representation has a very high space consumption since all data and metadata is represented as text strings. Necessary information for a single SDG vertex consumes on average 0.6 kB. Since an SDG representing a fairly large program of some hundred thousand lines of code may contain millions of vertices, this data size penalty is considerable. However, for a prototype implementation the cost was considered acceptable compared to the effort of developing a more efficient information transfer protocol.

3.3.2 Extracted Information

CodeSurfer keeps a lot of information about each SDG, PDG and vertex. GraphMatch extracts a subset of this information in order to recognize pattern vertices and perform the matching of property patterns. The listing below describes which information is extracted for each SDG, PDG, and vertex:

**SDG:** The name of the analyzed program.
PDG: The name of the modeled function and the filename where it occurs.

Vertex: The vertex type, the source text associated with the vertex (the \textit{vertex text}), the line number where the vertex text occurs, variables used and defined, and the edges leaving the vertex.

\section*{3.4 The Matcher}

The Matcher detects policy violations in the SDG produced by the Second Builder. The detection mechanism attempts to find subgraphs of the SDG that matches negative security property patterns. We have designed and implemented some straightforward algorithms and strategies that will correctly solve the matching problem and the problem of matching embedded negative patterns (see section 2.4.2). The prototype implementation has been governed by the goal of obtaining a working program within given time constraints rather than the principle of perfection. We have therefore chosen simple algorithms whose correctness could be verified. The task of designing better ways of solving the problem has been left to prospective future continuations of the project.

The matching strategy is based on a primitive operation which we call a dependency closure. The dependency closure operation is very similar to a more well-known program analysis concept: program slicing. The description of the Matcher will therefore start with a description of the slicing operation. We will proceed by describing a matching case that will be used as an example in the main part of the Matcher description. The said main part will follow, divided into two parts: one that describes the process of matching property patterns to subgraphs of SDGs (section 3.4.3) and one that describes how to recognize SDG vertices that match a certain pattern vertex description (section 3.4.4).

This order may seem somewhat backward, but some knowledge of the subgraph matching process is needed for the discussion of vertex matching. Until we get to section 3.4.4, we will assume that we have \textit{some} way of identifying SDG vertices matching each pattern vertex.
3.4.1 Program Slicing

The *slicing* concept was first introduced by Mark Weiser in 1984 [23]. A slice of a program is computed with respect to a program point $p$ and a variable $v$. It contains all program points that might affect the value of $v$ at $p$. This is sometimes also referred to as a *backward slice*. A *forward slice* of a program with respect to a program point $p$ and a variable $v$ contains all the program points that might be affected by the value of $v$ at $p$.

Ottenstein and Ottenstein showed that intraprocedural program slices can be computed by computing a transitive closure on both control and data dependence edges of PDGs in 1984 [24]. Horwitz et al continued the Ottensteins’ work in 1990 by describing how exact interprocedural slices can be built using SDGs [17].

The problem when computing interprocedural slices is that calling contexts may be mixed up if the Ottensteins’ transitive closure method is used. We will use the piece of code shown in figure 3.2 to illustrate the problem.

Suppose that we want to compute the forward slice with respect to $x$ and the initialization of $x$. The result of computing the slice using the transitive closure method is shown in figure 3.3.

We see that we will falsely reach the conclusion that the last definition of $y$ is affected by the initialization of $x$. The error occurs when the traversal leaves the PDG of $f()$. Both parameter-out edges will be followed, though only the one leading back to the original calling context should be used. We call the path found by following the wrong edge an *infeasible path*.

Horwitz et al solve this problem by introducing *summary edges* connecting each actual-in vertex with the corresponding actual-out vertex. On the summary edge augmented graph, the forward slicing operation can be performed in two passes.

The first pass adds all vertices reachable from the start vertex by following intraprocedural edges, summary edges and parameter-out edges to the slice. The second pass extends the slice by adding all vertices reachable from a vertex in the slice by following intraprocedural edges, summary edges, call edges and parameter-in edges. Figure 3.4 and 3.5 show how the first and second slicing pass are performed on a summary edge augmented version of the SDG shown in figure 3.3.
3.4. The Matcher

```c
int f(int a) {
    return a;
}

int main() {
    int x = 3;
    int y = 6;
    x = f(x);
    y = f(y);
}
```

Figure 3.2: An example of a program where calling contexts may cause an inexact slice. See also figure 3.3

Figure 3.3: SDG representing the program in figure 3.2. Shadowed vertices are members of the transitive closure taken from the \( x \) definition point. The square frame enclose the vertices that belong to the PDG representing \( f() \).
Figure 3.4: Result of the first slicing pass with respect to \( x \) and its initialization point. Members of the slice are shadowed in grey.

Figure 3.5: Result of the second slicing pass with respect to \( x \) and its initialization point. Members of the slice are shadowed in grey.
3.4.2 Matching Example

This section introduces a matching scenario that will be used in the following sections as an example. Suppose that we wish to detect all instances of the correct input validation property in the piece of code shown in figure 3.6.

```c
#include <stdio.h>

#define BUFSIZE 10

int ext_input() {
    int x = 0;
    scanf("%d", &x);
    return x;
}

void use_int(char* buf, int i) {
    buf[i] = 'B';
}

int main() {
    char arr[BUFSIZE] = "AAAAAAAAA";
    int x = ext_input();
    if(0 <= x && x < BUFSIZE - 1) {
        use_int(arr, x);
    }
}
```

Figure 3.6: An example program that will used in matching examples. Its corresponding SDG is presented in figure 3.7.

Figure 3.7 shows the SDG representing the example code. Note that each vertex is marked with a unique identifier, px, where p stands for program. The identifiers will be used when we refer to the vertices in the following sections. Also note that p12 contains a call to scanf() which is not expanded into a complete function call. To keep the example simple, we have chosen to view library function calls as atomic operations rather
Figure 3.7: SDG representing the code presented in figure 3.6. Solid arrows represent control dependencies, dashed arrows represent data dependencies. Each procedure except for main() is enclosed in its own frame.

Figure 3.8: Models of the correct and missing input validation properties. Solid arrows represent control dependencies, dashed arrows represent data dependencies. The vertices represent in order of appearance: ext_input: an untrusted data source, def: definition of a variable, val: validation of a variable, use: sensitive use of a variable.
The patterns that will be used to identify input validation policy violations are the correct input validation property pattern and the missing input validation property pattern presented in section 2.3. Figure 3.8 reminds the reader of how they look and introduce identifiers for their respective vertices (c for correct and m for missing).

We assume that the pattern vertices are modeled as follows:

- **c0/m0**: A program vertex containing a `scanf()` operation (remember that we view standard library calls as atomic operations)
- **c1**: A program vertex defining an integer variable.
- **c2**: A program vertex containing one of the operators `<`, `>`, `<=` or `>=`.
- **c3/m1**: A program vertex containing an array subscription or an addition or subtraction between pointer and integer.

### 3.4.3 Matching Property Patterns

We will here present the algorithm used to find program SDG subgraphs matching a certain property pattern. It is based on the computation of what we call a dependency closure, which is used to match the pattern’s transitive edges. We will start by describing the dependency closure concept. Then we will proceed to a description of the main matching algorithm. Finally, we will touch the subject of matching embedded negative patterns and analyze the worst case costs of algorithms used.

#### Computing Dependency Closures

Dependency closures are either data dependency closures or control dependency closures and contain all vertices that are reachable on feasible execution paths along the specified edge type. A dependency closure is very similar to a forward slice as presented in section 3.4.1. The difference between the two is that only one type of edge (data or control) is used in the computation of a dependency closure.

We compute a control dependency closure with respect to a certain program point and variable by taking the transitive closure over intra and
Implementation

ClosurePass($S, C, kinds$)
\begin{verbatim}
1 /* $S$ is an SDG */
2 /* $C$ is the closure, a set of vertices */
3 /* kinds is a set of edge kinds to be used */
4 /* WorkList is the set of vertices that */
5 /* will be used as a base for extension */
6 WorkList $\leftarrow C$
7 while WorkList $\neq \emptyset$
8 do select $v \in$ WorkList
9 WorkList $\leftarrow$ WorkList $\setminus \{v\}$
10 $V \leftarrow \{w \mid w \notin C, v \rightarrow_k w \in \text{Edges}(S), k \in \text{kinds}\}$
11 $C \leftarrow C \cup V$
12 WorkList $\leftarrow$ WorkList $\cup V$
\end{verbatim}

DependencyClosure($S, v, edgetype$)
\begin{verbatim}
1 /* $S$ is an SDG */
2 /* $v$ is the start vertex of the dep. clos. */
3 /* edgetype is either data or control */
4 /* $C$ is the closure, a set of vertices */
5 $C \leftarrow \emptyset$
6 if edgetype = control
7 then $C \leftarrow$ TransitiveClosure($v, control$)
8 if edgetype = data
9 then $C \leftarrow$ ClosurePass($v, intra - data, parameter - out$)
10 $C \leftarrow$ ClosurePass($C, intra - data, parameter - in$)
\end{verbatim}

Algorithm 3.4.1: Compute the dependency closure in the SDG $S$ starting from a vertex $v$ and following edges of $edgetype$. Note that $edgetype$ may only represent $data$ or $control$, whereas the $kinds$ argument to the ClosurePass procedure is a set that may contain more specific edge kind specifiers, i.e. parameter-in.
interprocedural edges. When dealing with control edges, there is no need to worry about calling contexts.

Data dependency closures must be computed using the two pass approach presented in section 3.4.1, due to the risk of following infeasible paths. In the first pass we add to the data dependency closure each vertex that is reachable through any sequence of intraprocedural data edges, summary edges and parameter-out edges. In the second pass we add each vertex that is reachable through any sequence of intraprocedural data edges, summary edges and parameter-in edges to the closure.

Algorithm 3.4.1 formally describes how a dependency closure is computed. It is practically identical to the forward slicing algorithm presented by Horwitz et al, but included here for the convenience of readers not acquainted with their work.

**Matching a Single Pattern**

The matching is performed in a series of successive steps, one for each transitive edge in the pattern. We will use the terms *current pattern source*, *current pattern target* and *current edge* to refer to the source vertex, target vertex and self of the pattern edge being matched. We start each step with a set of *base vertices*. The base vertices are program vertices that match the current pattern source. For each of the base vertices, we compute a dependency closure of the type indicated by the current edge. The next set of base vertices is computed by searching the closure for vertex matches of the current pattern target.

The new set of base vertices thus calculated are to be used when matching edges leaving the target pattern vertex of the current edge. For pattern vertices that have no incoming edges, *top vertices*, the base set is computed by a search in the control dependency closure of the top vertex of the program SDG. This closure includes all the vertices in the graph.

Algorithm 3.4.2 formally describes the process outlined above. The **MatchStep** procedure performs the matching of one transitive edge by by the nested calls of **Filter** and **DependencyClosure** on line 10. The **MatchStep** procedure calls itself recursively to be able to use the same set of base vertices more than once in cases where pattern vertices have more than one outgoing edge to match. To simplify the description, the
FILTER($p, V$)
1 /* $p$ is a pattern vertex */
2 /* $V$ is a set of vertices */
3 for each $v$ in $V$
4 do if $\neg$Matches($p, v$)
5 then $V \leftarrow V \setminus \{v\}$
6 return $V$

MATCHSTEP($P, p_{src}, Base, S$)
1 /* $P$ is a pattern */
2 /* $p_{src}$ is a pattern vertex */
3 /* Base is a set of program vertices, */
4 /* all matching $p_{src}$ */
5 /* $S$ is an SDG */
6 /* edge is either data or control */
7 for each $v$ in $Base$
8 do for each <$p_{tgt}, edge$> such that $p_{src} \rightarrow_{edge} p_{tgt} \in E(P)$
9 do Closure $\leftarrow$ DEPENDENCYCLOSURE($v, edge, S$)
10 MATCHSTEP($P, j, Filter(p_{tgt}, Closure)$)

MATCHEDGES($P, S$)
1 /* $P$ is a pattern */
2 /* $S$ is an SDG */
3 Closure $\leftarrow$ DEPENDENCYCLOSURE($Top(S), control, S$)
4 StartBase $\leftarrow$ Filter($Top(P), Closure$)
5 MATCHSTEP($P, Top(P), StartBase, S$)

Algorithm 3.4.2: Find all matches for a pattern $P$ in an SDG $S$. This simplified version assumes that the pattern contains only one top vertex and does not remember what it has matched. The Top and Matches are meant to be viewed as atomic operations.
Algorithm 3.4.3: Find all matches for a pattern \( P \) in an SDG \( S \). This simplified version assumes that the pattern contains only one top vertex. The \texttt{Top}, \texttt{Matches} and \texttt{container} procedures are meant to be viewed as atomic operations.
presented algorithm assumes that there is only one top vertex in the pattern. Interested readers may easily fill in the necessary alterations to handle more than one top vertex.

You may notice that algorithm 3.4.2 does not produce any result. The program vertices matching a pattern vertex are not saved anywhere and nothing is returned. The reason for presenting such a useless algorithm is to prepare the reader for the useful but more complicated algorithm will be described next.

As pattern vertices may have several different successors and predecessors, keeping track of the interrelationships of their matches in program SDGs in an efficient way is complicated. The thing that we left out from algorithm 3.4.2 is the interrelationship trackkeeping.

We have chosen to use match containers tailor-made for the pattern for the purpose of remembering which target vertex matches is connected to which source vertex matches. A match container contains a collection of vertex holders, one for each pattern vertex that should be matched and a collection of link holders, one for each pattern edge that should be matched. Figure 3.9 shows a schematic picture of a match container for the correct input validation pattern.

As we traverse the edges of a pattern, we successively fill the empty vertex and link holders of match containers with the matching program vertices we have found. Each time two different vertices compete for the same vertex holder in a match container, the container is copied to make room for both alternatives.

Algorithm 3.4.3 is a version of algorithm 3.4.2 augmented with the use of match containers. In contrast to the latter, this new algorithm actually produces a result. It is still somewhat simplified: it does not describe how the link holders of match containers are filled. That would add unnecessary complexity to the description without adding any significant value.

Example

We will illustrate the matching principle using our example program with the correct input validation pattern. Below is a step-by-step description of the process. Note that we describe the process in a recursive manner: each level of indentation represents a level of recursive nestedness. All procedure
name are taken from algorithm 3.4.3. We will use a special notation to denote match containers: \([h_1 : v_1; h_2 : v_2 \ldots h_n : v_n]\), where \(h\) denotes a vertex holder and \(v\) a vertex. Empty holders and links are not shown. The description starts at the MATCH\_EDGES procedure with \(P\) set to the correct input validation pattern and \(S\) set to the example program SDG.
0: Find the first set of base vertices by searching the control dependency closure of \( p0 \) for matches for \( c0 \). Figure 3.10 illustrates the closure its single vertex match, \( p12 \).

Call \texttt{MatchStep}(P, c0, \{p12\}, \{[]\}, S).

1: Call \texttt{ExtendContainers}([[]], c0, p12).

Result: \{[c0:p12]\}, shown in figure 3.11.

Find matches for \( c1 \) in the data dependency closure of \( p12 \). Figure 3.12 illustrates the closure and its vertex matches.

Call \texttt{MatchStep}(P, c1, \{p10, p5, p2, p7, p14\}, \{[c0: p12]\}, S).

2: Call \texttt{ExtendContainers}([c0: p12], c1, p10).

Result: \{[c0: p12; c1: p10]\}.

Find matches for \( c2 \) in the data dependency closure of \( p10 \).

One match is found: \( p4 \).

Call \texttt{MatchStep}(P, c2, \{p4\}, \{[c0: p12; c1: p10]\}, S).

3: Call \texttt{ExtendContainers}([c0: p12; c1: p10], c2, p4).

Result: \{[c0: p12; c1: p10; c2: p4]\}.

Find matches for \( c3 \) in the data dependency closure of \( p4 \).

One match is found: \( p16 \).

Call \texttt{MatchStep}(P, c3, \{p16\}, \{[c0: p12; c1: p10; c2: p4]\}, S).

4: Call \texttt{ExtendContainers}([c0: p12; c1: p10; c2: p4], c3, p16).

Result: \{[c0: p12; c1: p10; c2: p4; c3: p16]\}.

As \( c3 \) has no descendants, return the result directly.

3: As \( c2 \) has no more descendants, pass the match container set further up the call chain.

2: Find matches for \( c3 \) in the data dependency closure of \( p5 \).

One match is found: \( p16 \).

Call \texttt{MatchStep}(P, c3, \{p4\},
\{[c0: p12; c1: p10; c2: p4; c3: p16]\}, S).
3: Call `ExtendContainers`([`c0: p12; c1: p10; c2: p4; c3: p16`],
c3, p16).

Result: `{c0: p12; c1: p10; c2: p4; c3: p16}`.

As c3 has no descendants, return the result directly.

2: Unite $C_{final}$ with `{c0: p12; c1: p10; c2: p4; c3: p16}`. Continue to the next lap of the loop. As all the laps are very similar, we do not describe the rest of them.

Return $C_{final}$, which now contains all match containers from all branches, as shown in figure 3.14.

2: Pass the match container set further up the call chain.

0: Return the match container set returned by `MatchStep`.

**Worst Case Matching Cost**

The computation of a dependence closure may in the worst case follow all edges of its type in an SDG. Supposing that the operations of checking that a vertex is not already in a closure and finding out if there is an edge between to arbitrary vertices in an SDG can be performed in constant time, the cost of computing a dependency closure is $O(E)$, where $E$ is the number of edges in the SDG. Although it is highly improbable that all edges in the SDG should be followed, we can specify no stricter bound on the complexity.

As outlined in algorithm 3.4.3 the cost of filtering a dependency closure is bounded by $O(V)$, where $V$ is the number of vertices in the SDG. If we assume that the filtering can be done as a part of the closure computation, the complexity of the whole closure computation and filtering process can be reduced to $O(E)$. Each call to `MatchStep` will in the worst case cause $V$ dependency closure computations, since the number of base vertices is bounded by $V$.

A call to `MatchStep` will also generate a number of new calls to `MatchStep`. The number of calls is bounded by $O(V)$, since there are at most $V$ vertex matches for each pattern vertex descendant and the number of descendants can be assumed to be much smaller than $V$. If we for a moment pretend that the container set extensions take constant time, we
Figure 3.10: Result of the first base vertex search. Vertices that are members of the computed closure are shadowed in light grey. Matches for the pattern target vertex are filled with dark grey. Paths followed to compute the closure are shown in boldface.

<table>
<thead>
<tr>
<th>c0: p12</th>
<th>c0-c1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1:</td>
<td>c1-c2:</td>
</tr>
<tr>
<td>c2:</td>
<td>c1-c3:</td>
</tr>
<tr>
<td>c3:</td>
<td>c2-c3:</td>
</tr>
</tbody>
</table>

Figure 3.11: Match container created to store the result of the first base vertex search.
3.4. The Matcher

Figure 3.12: Matches for the c1 pattern vertex in the dependency closure of p12. Vertices that are members of the computed closure are shadowed in light grey. Matches for the pattern target vertex are filled with dark grey. Paths followed to compute the closure are shown in boldface.

<table>
<thead>
<tr>
<th>c0</th>
<th>c0-c1</th>
<th>c1</th>
<th>c1-c2</th>
<th>c2</th>
<th>c1-c3</th>
<th>c2-c3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c1</td>
<td>p10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c2</td>
<td>p11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c3</td>
<td>p12-c1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.13: Match container created to store the first result of the base vertex search from p12.
arrive at the conclusion that the matching process has a cost bounded by $O(E \cdot V^h)$, where $h$ is the depth of the pattern.

Although the matching is described in a depth-first manner, we will use breadth-first reasoning to arrive at a worst case cost of container extensions. The cost bound of container extension will coincide with the number of containers created. For each pattern vertex matched, at most $V_{new}$ match containers may be created for each match container that already exists. Since we start with one single match container, the number of match containers created when matching a pattern of $p$ vertices will be bounded by $O(V^p)$.

The total matching cost thus becomes $O(E \cdot V^h + V^p)$ and the maximum number of match containers to return $O(V^p)$. How close to the worst case cost we get in practice depends on how we define the pattern vertices. If every pattern vertex has a match in every program vertex, we have the worst case before us. If instead each pattern vertex has exactly one matching program vertex, the algorithm will complete in $O(E \cdot e)$ time, where $e$ is the number of edges in the pattern.
Matching Embedded Negative Patterns

Section 2.4.2 introduced the missing input validation pattern as an embedded negative property pattern. It also described that such patterns can only be detected by reporting instances of the negative pattern that do not contribute to form the whole embedding positive pattern. In this section we will present how GraphMatch handles detection of policy violations described by embedded negative properties.

GraphMatch’s embedded matching strategy is straightforward and naïve, but works well in practice. The idea is to first find matches for negative patterns. Then, using the negative matches as starting points, we try to find matches for embedding positive patterns. If such a positive match is found, we say that it invalidates negative match. If we cannot find such a positive match, the negative match is reported as a policy violation.

GraphMatch allows its users to define patterns as either positive or negative. Positive patterns are assumed to be embedding patterns for some negative property. The user may also specify relationships between positive and negative patterns in the form of relation points. A relation point is a pair of pattern vertices, one from the positive pattern and one from the negative. The relation point tells GraphMatch which vertices of a positive and a negative match should be the same for the negative match to be invalidated.

In our example, the missing input validation pattern would be specified as a negative pattern, the correct input validation pattern would be defined as a positive pattern. They would have two relation points: the vertex pairs c0-m0 and c3-m1.

As soon as a complete positive match is found, it is checked against each of the match containers in the negative match set. Supposing that the relationship of the positive and negative pattern is defined by a set of relation points \( \{p_1 - n_1, p_2 - n_2 \ldots p_n - n_n\} \), we check if the program vertex held in each \( p_i \) vertex holder of the positive match container is the same as the program vertex held in the corresponding \( n_i \) vertex holder in the current negative match container. If all \( p_i \)s and \( n_i \)s hold the same program vertex, the negative match container is considered invalidated and discarded. Else, it is saved for possible future invalidation by another positive match or to be reported as a policy violation.
To decrease the number of positive subgraph matches computed in vain, we employ a simple strategy: each extension of a positive subgraph match by a vertex match $v$ is checked against relation points and existing negative subgraph matches. If the current pattern vertex is associated with a negative pattern vertex $n$ by a relation point, we check whether $v$ is a member of the set of vertices that match $n$ in the negative matches. If it is a member, the extension is accepted. Else we know that a complete positive match containing $v$ may never invalidate any of the negative matches.

Supposing that we want to detect violations of the input validation policy in the example program, we must start by finding all instances of the missing input validation property. A quick manual examination of the program SDG shows that the graph contains only one single match for the property. It consists of the two vertices $p_{12}$ (matching $m_0$) and $p_{16}$ (matching $m_1$) and the data dependency path between them. The set of negative match containers for this example will thus contain only one container, which is shown in figure 3.15.

![Figure 3.15: Match container representing the single match for the missing input validation pattern present in the example program.](image)

Two pairs of vertices are present in the set of relation points of the missing and correct input validation patterns: $c_0-m_0$ and $c_3-m_1$. In our example, the $c_0$ and $c_3$ pattern vertices have only one match each, and those are the same as the matches for $m_0$ and $m_1$ present in the negative match container set. Thus, all vertex matches found will be accepted at first. As soon as the first of the positive matches shown in figure 3.14 is completed, the negative match will be invalidated. The other two match containers will never be completed, since the set of vertex matches for $m_1$ will be emptied by the earlier invalidation. Thus no $c_3$ vertex matches will be accepted when their turn come.
3.4.4 Matching Individual Vertices

To be able to identify matches for a certain pattern vertex, we need a concise way of describing the class of program vertices it represents. This is done by providing every pattern vertex with a set of attributes, similar to the program vertex information extracted from the CodeSurfer SDG. A number of matching parameters are used, such as the vertex type, types of variables used or defined at the vertex, and regular expressions representing the vertex text. The parameters specify what properties a program vertex must have to be accepted as matching the pattern vertex. Combinations of parameters can be specified using AND, OR, and NOT semantics.

It has proven to be difficult to design matching parameters so that the class they define include all program vertices they are supposed to but exclude all others. The inexactness is partly due to difficulties to define exactly what to look for, as discussed in chapter 2 and partly to the fact that GraphMatch does not use enough information from CodeSurfer to perform more exact analyzes on program vertices. Extracting more vertex information would be possible, but the limited time available for the thesis project did not allow for its implementation. Time constraints have also forced us to discard more precise solutions using the information currently extracted.

Basic Approach

Just as we outlined above, the basic vertex matching approach is to define and use matching parameters for each pattern vertex. In many cases, one pattern vertex may represent a small group of program vertices. We will clarify the matching parameter concept using the vertices of the correct input validation pattern as examples.

ext_input: Contains two vertices separated by a non-transitive control edge:

vertex 1 type: call-site.
vertex 1 text: name of a function that fetches user input.
vertex 2 type: actual-out.
or
vertex 1 type: entry-point.
vertex 1 text: main().
vertex 2 type: formal-in.
( represents a command-line argument)
def: defines: any integer variable.
val: vertex type: 'control point'.
vertex text: contains <, >, <=, or >=
use: uses: one integer variable, one pointer variable.
vertex text: contains + or –.
or
uses: one integer variable, one array variable.
vertex text: contains a pair of matching brackets ([ ]).

Three major improvements of the basic approach have been made to increase the accuracy of the vertex matches. They are described below.

Improvement 1: Using Dependence Variables

The matching criteria for the pointer use vertex above caused some trouble in the form of false positives. We will describe the problem by giving an example. Figure 3.16 shows a piece of code where no external input-affected data is used in pointer arithmetic. Yet it will yield a match if the matching criteria presented in the previous section are used. Note that the array assignment on the last line is a definition of an element in the array of integers and at the same time a case of pointer arithmetic. The array subscription with $i$ is equivalent to an addition of the base pointer for arr and the integer $i$.
The false match will be detected due to the fact that there is a data dependency between the `getchar()` return value and the array subscript point due to the use of a. The array subscription will match the pointer use criteria, which will cause a false match to be reported by GraphMatch. To differentiate between cases where there is a real pointer arithmetic use of the variable depending on external input and where there is not, we need
3.4. The Matcher

```c
int arr[10];
int i = 0;
int a = getchar(); /* a depends on external input */
arr[i] = a; /* no input-depending variable */
/* used to address the array */
```

Figure 3.16: Problematic pointer use example, that will generate a false match for the missing input validation pattern.

to include the variables involved in the followed dependence edges in the matching criterion.

By using the parent program vertices of the vertex we are interested in as matching factors, we can compute which variables may have caused a dependence edge. Note that the parent vertices of interest are those included in the same data dependency closure as the child we are matching. The computation of dependence variables is simple: we take the intersection of variables defined at the parent vertices and the variables used at the child vertex. If there is no match for these variables’ names in the significant places in the vertex text (such as inside brackets or next to an arithmetic operator), the vertex does not match the description.

In the example from figure 3.16, the parent vertex would be the vertex where `a` is defined and the child vertex the array subscription vertex. Since `a` but not `i` is defined in the former and used in the latter, we conclude that `a` caused the dependence edge. An array subscription using `a` as an index would be dangerous, whereas one that uses `i` is not. Thus the false match can be eliminated.

**Improvement 2: Inserting Artificial Definitions**

The matching criterion of the definition vertex above as well as the dependence variable attempt to increase precision of matches proved to cause some problems when `global-actual-out` vertices were involved in the call chain. A global-actual-out vertex is a special case of an actual-out vertex,
notably used in CodeSurfer’s dependence graph. Global-actual-out vertices appear whenever a function defines a non-local variable, either passed to it as a pointer argument or as a global variable. CodeSurfer does not model the global-actual-out vertex as defining any variable. This behavior is correct, since in a real program the global variable or pointer target is defined inside the called function but not on exiting the function. However, from the calling function’s point of view, there is a definition of some local or global variable at that point. Consider the example code shown in figure 3.17.

```
char arr[] = "AAAAAAAAAA";
int dangerous;
scanf("%d", &dangerous);
arr[dangerous] = 'B';
```

Figure 3.17: Example of global-actual-out problem

Here, the `scanf()` function will define the `dangerous` variable through the pointer passed to it. This definition and its result will be modeled by CodeSurfer as a global-actual-out vertex which the array subscription depends on. Thus, GraphMatch will consider the array subscription as harmless, since no dependence variable is found, and thus no dependence variable can be identified as the array subscription variable.

To resolve this problem, we have added some artificial definitions at each global-out-vertex. Whenever an actual-in-vertex is found during the graph build, its corresponding actual-out or global-actual-out vertex is identified (through the summary edge starting at the actual-in vertex). Any variable that may be pointed to by an actual-in pointer is then added to the definition variables used in the corresponding actual-out vertex.

This approach makes the identification of the array subscription example above work correctly, but cases where global variables are used cannot be resolved in this way. Global variables are not modeled as used by global-actual-in vertices, so any dependencies might have to be followed very far back in order to identify what global variable is really used and defined by
3.4. The Matcher

```c
int validate(int a) {
    return (0 <= a && a < MAX);
}

int main() {
    int dangerous = ext_input();
    if(validate(dangerous)) {
        sensitive_use(dangerous);
    }
}
```

Figure 3.18: Example of comparison point and use point in different functions.

Improvement 3: Allowing for External Validation

We discovered that our first definition of validation did not cover some important special cases, namely cases where the actual comparison part of the validation is not present in the same function as the sensitive use point. Figures 3.18 and 3.19 show two examples of this. In these examples, the use point will not be directly control-dependent on the validation point present in `validate()`.

In the example of figure 3.18, the result of the comparison in `validate()` is not used directly at a control point, but defines the return value of `validate()`, which is used in a control point, creating a data dependency between the two points. A pattern that matches this type of validation is shown in figure 3.20.

In the example of figure 3.19, the result of the comparison in `validate` is used in a control point, but one that controls what definition of the return value is performed. Again there is a data dependency between the return value definition points and the control point in `main()`. A pattern that matches this type of validation is shown in figure 3.21.
int validate(int a) {
    if(0 <= a && a < MAX) {
        return 1;
    }
    return 0;
}

int main() {
    int dangerous = ext_input();
    if(validate(dangerous)) {
        sensitive_use(dangerous);
    }
}

Figure 3.19: Example of comparison point and use point in different functions.

Even if we use the validation patterns presented here, correct validations may still not be detected correctly if they are nested in more than one calling context like the one shown in figure 3.19. The multi-vertex validation patterns also have the disadvantage of increasing the number of vertices and depth of the pattern. We recall from section 3.4.3 that the number of pattern vertices and the pattern depth are involved as exponents in the worst-case computation time. Increasing the number of pattern vertices will thus increase computation times in a dramatic manner.
Figure 3.20: Version of the correct input validation pattern that will match the code example in figure 3.18.

Figure 3.21: Version of the correct input validation pattern that will match the code example in figure 3.19.
Chapter 4

Results and Conclusions

This chapter accounts for the results of security scanning with GraphMatch. A discussion of future work and final conclusions based on the test results forms the last part of the chapter.

4.1 Test Results

The test cases used to evaluate GraphMatch and the results of testing are described in this section. We will present two types of test cases: synthesized test code and code from a real open source application. Each group of test cases will be presented together with a description of its results.

4.1.1 Synthesized Test Code

The synthesized cases used to test GraphMatch were pieces of code displaying behaviour acceptable and unacceptable by the input validation policy. They were subdivided into two categories: intraprocedural cases exemplifying different types of external input and pointer arithmetic and interprocedural cases were the external input, validation and pointer arithmetic use parts of the pattern all resided in different functions.
#include <stdio.h>
#define BUF_SIZE 10

int main(int argc, char* argv[]) {
    char buf[BUF_SIZE] = "AAAAAAAAA";
    char buf2[BUF_SIZE];
    char buf3[2];
    int dangerous;

    fgets(buf2, BUF_SIZE, stdin);
    buf3[1] = '\0';
    buf3[0] = getchar();

    /* 'dangerous' affected by external input */
    dangerous = atoi(buf3);

    /* 'dangerous' used in pointer arithmetic */
    memcpy(buf + dangerous, buf2, BUF_SIZE - dangerous);

    /* make sure nothing is consumed by optimization */
    printf("%s
", buf);
}

Figure 4.1: External input test case for \texttt{getchar()}. No validation is performed before the input from \texttt{getchar()} is used as an offset.

The test cases were chosen with a verification perspective in mind. They reflect what we think that GraphMatch should be able to handle, but we do not claim that they cover all possible cases of external input, pointer arithmetic use and validation.

**Intraprocedural Test Cases**

The intraprocedural cases were further divided into external input tests and pointer arithmetic tests. The first group was designed to test GraphMatch’s ability to detect external input from various sources, such as command
#include <stdio.h>

#define BUF_SIZE 10

int main() {
    char buf[BUF_SIZE] = "AAAAAAAAA";
    char buf2[BUF_SIZE];
    int dangerous;

    fgets(buf2, BUF_SIZE, stdin);

    /* 'dangerous' affected by external input */
    scanf("%d", &dangerous);

    if(0 <= dangerous && dangerous < BUF_SIZE) {
        /* 'dangerous' used in pointer arithmetic */
        buf[dangerous] = buf2[dangerous];
    }

    /* make sure nothing is consumed by optimization */
    printf("%s\n", buf);
}

Figure 4.2: Pointer arithmetic use test case for array subscription. 'dangerous' is validated before its use as an array index.

Intraprocedural test cases are included here.

Figure 4.1 shows a case from the external input group. The variable dangerous is affected by input from getchar() and reaches a pointer addition without validation. The other cases in the external input text group are identical to the case shown except for the manner in which the external input source reaches dangerous. A complete list of tested input sources are found in table 4.1 together with the test results.
Table 4.1: Results from the external input tests. **Return value:** Return value counted as user-controllable **Modified parameter:** Modified parameter counted as user-controllable **False negatives:** Number of actual security bugs not reported by GraphMatch. **False positives:** Number of false alarms raised by GraphMatch.

<table>
<thead>
<tr>
<th>Input source</th>
<th>Return value</th>
<th>Modified parameter</th>
<th>False negatives</th>
<th>False positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>argv</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fgetc()</td>
<td>yes</td>
<td>no</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fgets()</td>
<td>no</td>
<td>yes</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fread()</td>
<td>no</td>
<td>yes</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fscanf()</td>
<td>no</td>
<td>yes</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>getc()</td>
<td>yes</td>
<td>no</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>getchar()</td>
<td>yes</td>
<td>no</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>getenv()</td>
<td>yes</td>
<td>no</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>gets()</td>
<td>no</td>
<td>yes</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>getw()</td>
<td>yes</td>
<td>no</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>read()</td>
<td>no</td>
<td>yes</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>readdir()</td>
<td>yes</td>
<td>no</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>readlink()</td>
<td>no</td>
<td>yes</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>recv()</td>
<td>no</td>
<td>yes</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>recvfrom()</td>
<td>no</td>
<td>yes</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>recvmsg()</td>
<td>no</td>
<td>yes</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>scanf()</td>
<td>no</td>
<td>yes</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4.2 shows a case from the pointer arithmetic group. Input from `scanf()` reaches a validated array subscription. The other cases in the pointer use test group are identical to the case shown except for the type of pointer arithmetic used. A complete list of pointer arithmetic cases tested are found in table 4.2 together with the test results.

The **Return value** and **Modified parameter** fields of table 4.1 indicate
Table 4.2: Results from the pointer arithmetic use tests. **False negatives:** Number of actual security bugs not reported by GraphMatch. **False positives** Number of false alarms raised by GraphMatch.

<table>
<thead>
<tr>
<th>Type of pointer arithmetic</th>
<th>False negatives</th>
<th>False positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>array subscript</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>integer + pointer</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pointer + integer</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pointer - integer</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

what data affected by the external input source was counted as controllable by the user and not. For example, the `fread()` function writes data into a buffer given as a parameter, but also returns an integer indicating the number of bytes read. We have often chosen to ignore such indicator return values, since they are difficult to control for a user, but will generate false positives.

Table 4.1 shows that all the external input cases succeeded except for the `recvmsg()` socket primitive, which did not work out due to the fact that CodeSurfer does not model the actual buffer being modified by the `recvmsg()` call as modified by it. Table 4.2 shows that all pointer arithmetic cases were handled correctly by GraphMatch.

**Interprocedural Test Cases**

The interprocedural test cases were built from six base functions:

**two input functions**:

- `dangerous_input()`: Return value affected by external sources.
- `harmless_input()`: Return value not affected by external sources.

**two validation functions**:

- `proper_validation()`: Return value depends on proper comparison of arguments.
4.1. Test Results

- **no_validation()**: Return value does not depend on proper comparison of arguments.

  **two use functions**:

  - **sensitive_use()**: Integer argument used in pointer arithmetic.
  - **harmless_use()**: Integer argument not used in pointer arithmetic.

The test cases consisted of combinations of the six base functions, all very similar to the sample in figure 4.3. The scanning results from GraphMatch were just as expected for these cases: the only reported policy violation was the combination of `dangerous_input()`, `no_validation()` and `sensitive_use()`. All other cases were considered harmless by GraphMatch, since they either contained no external input source, had no pointer arithmetic use of integers or had sensitive use that was properly validated.

```c
#include "interproc.h"

int main() {
    char buf[] = "Du är en apa.";
    int x = dangerous_input(4);
    if(proper_validation(x, strlen(buf))) {
        sensitive_use(x, buf);
    }
}
```

Figure 4.3: Interprocedural combination test case that will match the correct input validation pattern.

### 4.1.2 Open Source Application Code

We tested GraphMatch on wu-ftp, version 2.6-2. It is a relatively small open source project containing approximately 20,000 lines of code. The SDG built by CodeSurfer for the ftpd part of the project contained 128,949 vertices.
Results and Conclusions

Initial Problems

The first thing we discovered was that the algorithm complexity we presented in the previous chapter (section 3.4.3) really showed its significance when working with a program of this size. Our first analysis attempts had to be interrupted since some quick running time extrapolations had it very likely that the complete analysis would take a week. We decided not to use the multi-vertex version of the validation pattern vertex presented in section 3.4.4. We judged that the risk of someone actually using a validation point like the ones exemplified there was small compared to analysis time penalty of added vertices.

Once we had a version of GraphMatch that would actually get through the analysis in little more than twenty-four hours time, we discovered the next problem: Most of the validation points identified by GraphMatch were entirely irrelevant for the use they were supposed to validate. A closer inspection of the problem showed that by allowing both edges emerging from the def node to be transitive, we allowed that several different variable definitions occurred on each branch.

Figure 4.4 shows an example that would match the correct input validation pattern even though the validation is irrelevant for the use of \( z \) as

```c
int x;
int y;
int arr[100];
x = getchar();
y = x * 3;
z = x - 35;
if(0 <= y && y < 100) {
    arr[z] = 34;
}
```

Figure 4.4: Example of irrelevant validation point. The comparisons performed are not relevant for the array indexing with \( z \).
array index. We eliminated this class of matches by specifying the `def`–`val` edge of the correct input validation pattern as non-transitive. With a direct dependency between the `def` and `val` vertices, the variable used in the validation is guaranteed to be the same as the one defined in the `def` vertex.

**Closure Depth Experiments**

We analyzed the code twice: once limiting the maximum depth of each dependency closure computed to 10 and once using an unlimited depth. Table 4.3 accounts for the results from the two test-runs. The analysis times presented in the table include approximately two minutes needed to read the graph from an XML file on disk and build the in-memory graph structure.

Table 4.3: Overview of test results using different maximum closure depths. **max depth**: Maximum closure depth used. **analysis time**: Time for analysis using a 2.66 MHz Pentium 4 processor. **neg. matches**: Matches for the missing input validation pattern found by GraphMatch. **warnings**: Warnings emitted by GraphMatch.

<table>
<thead>
<tr>
<th>max depth</th>
<th>analysis time</th>
<th>neg. matches</th>
<th>warnings</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11h 33min</td>
<td>1099</td>
<td>459</td>
</tr>
<tr>
<td>no limit</td>
<td>15h 4min</td>
<td>2041</td>
<td>3</td>
</tr>
</tbody>
</table>

We were hoping that limiting the search depth would reduce the analysis time without generating too many false positives. Reality, however proved us to be wrong. Although the analysis time was somewhat reduced, it was not worth the cost of 456 additional matches which are probably false positives.

**Accuracy of Analysis**

We manually examined the unlimited depth analysis results in some detail. On closer inspection, two of the three warnings emitted were false positives. Their `use` matches contained an addition of two integers, of which one was
Results and Conclusions

reached through a dereferenced pointer. The current GraphMatch implementation cannot differ between use of a pointer and use of the pointer target. Thus the addition matched the pointer use pattern vertex description. The third match was a case of array indexing, clearly without any checks on the array index used. However, although it was a correctly identified case of missing input validation, it was also clearly unexploitable.

A sample of 20 randomly selected instances of the missing input validation matches and their validation points as identified by GraphMatch were chosen for a manual inspection. Table 4.4 accounts for the inspection results. The correct vertices column accounts for the correctness of vertex matches, i.e. that every vertex in the subgraph match really represented the action it was supposed to.

The correct val found column contains the results of a quality check on the matches’ validation points. We found that although the quality of identified validation had increased significantly after replacing the transitive def-val edge with a non-transitive one, irrelevant points were still identified as validations.

In many cases, the identified validation was placed in the middle of some giant loop containing switch cases, gotos and breaks creating a spider web of control dependencies. The execution of a program point reachable through a control dependency edge starting at a control point inside such a loop might depend on several other control points as well. Depending on the path taken in the unstructured loop, different points may control the execution of the target. Thus, the identified “validation” could be bypassed by taking other paths. In these cases, we judged the validation as invalid.

The correct val exists column contains the results of a search for correct validation points in the surroundings of the pointer use matches. In cases where such a validation point does not exist and the validation identified by GraphMatch is incorrect, the risk is high that GraphMatch has overlooked an exploitable security policy violation. In many cases where GraphMatch had found an invalid val match, the confusing flow of control made it difficult to determine whether or not a correct validation point existed. Such cases are marked as unsure and are possibly actual vulnerabilities.

Table 4.4 shows that seven of the correct input validation pattern instances were correctly identified. Of the other 13, none had an actual clear validation point. That indicates that GraphMatch might have a high false
Table 4.4: Results from the wu-ftp tests. **sample**: Sample no. **vertices correct**: All vertex matches correct. **correct val found**: The identified validation point is correct. **correct val exists** A correct validation point exists in the code.

<table>
<thead>
<tr>
<th>sample</th>
<th>vertices correct</th>
<th>correct val found</th>
<th>correct val exists</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>7</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>8</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>9</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>12</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>13</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>14</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>15</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>16</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>17</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>18</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>19</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>20</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>total</td>
<td>100%</td>
<td>0%</td>
<td>40%</td>
</tr>
</tbody>
</table>
negative rate. Although the few matches which have been examined here are not enough to determine any definite rate of false positives or negatives, the results indicate that both numbers might be high if more data would be collected.

4.2 Future Work

The previous section showed that the results of running GraphMatch on a real code example were unsatisfactory, both in terms of accuracy of matches and in computation time. In this section we will discuss what might be done to improve these two qualities. We will also suggest some other issues that should be investigated.

4.2.1 The Complexity Issue

The extensive computation times presented in section 4.1.2 was not surprising considering the complexity of the algorithms used, as presented in chapter 3. The analysis time problem is a first rate issue that must be resolved before any other part of GraphMatch is touched. There are two main strategies available: a) finding a better exact algorithm and b) using some heuristic method with sufficient accuracy.

The task of finding SDG subgraphs matching a property pattern is very similar to the subgraph isomorphism problem, which has been proven to be NP-complete [25]. It is beyond the scope of a thesis project to determine the NP-completeness or non-NP-completeness of the dependence graph pattern matching problem. We may however note that the main difference between the two problems is that the property patterns allow edges to be transitive, i.e. to represent an arbitrarily long chain of edges and vertices. Intuitively, this fact implies that the dependence graph pattern matching problem is not less complex than the subgraph isomorphism problem.

Several exact and heuristic solution algorithms, and strategies to reduce the average computation cost for the subgraph isomorphism problem have been published. If the graph pattern matching problem can be transformed into the subgraph isomorphism problem, those algorithms may be employed.
4.2. Future Work

As the wu-ftp results proved earlier, the heuristic attempt to reduce analysis time used here did not turn out well. A limited search depth does not reduce the computation time enough to compensate for the accuracy lost. Perhaps a more sophisticated depth strategy which allows us to specify maximum depths for each individual edge would yield acceptable results.

4.2.2 The Accuracy Issue

The false positives and negatives found during analysis of wu-ftp were partly due to poor specification of vertex patterns. For example, false instances of pointer arithmetic were found because we could not differ between a dereferenced and non-dereferenced pointer. The cases of irrelevant validation points could also be diminished in numbers by providing stricter validation patterns. However, no strict vertex specifications could solve the problems with irrelevant validation points due to unstructured control flow.

More interesting cases of input validation policy violations could be detected by making use of information of integer signedness and requiring stricter validation of signed integers. Still more interesting results could be obtained by decorating the graph with type cast information and making use of that information when determining if a validation could be correct or not.

4.2.3 The Other Issues

Two major topics remain to be investigated if the complexity and accuracy issues are resolved with a satisfactory result.

The first topic is the set of policies that can be described using dependence graphs. At present we do not know the limitations of the modeling principle.

The second topic is the set of languages that are suitable for dependence graph analysis. At present we do not know how different language features affect modeling with dependence graphs and security bugs that may occur. A number of questions should be answered in a solution to this problem:

1. How to build dependence graphs for languages other than C.
2. What differences there are between dependence graphs for different languages and what consequences the differences might have for a graph pattern matcher.

3. Which security property models and parts of models could be reused for different languages and which are language specific.

4.3 Conclusions

We recall from chapter 1 that the thesis project’s goal was to determine if a tool like GraphMatch could be used as a part of practical software development. We assumed that the following requirements should hold for a practically usable security analysis tool:

- produce few false alarms and rate detected security bugs according to severity.
- not require developers to add annotations to their code or otherwise drop hints to the tool what to do.
- either run in approximately the same time as an ordinary compilation, or at least be able to perform a complete analysis in a couple of hours time.

At its current prototype state, GraphMatch does not fulfil the first requirement. The wu-ftp test indicated a high rate of both false positives and false negatives. However, some of the suggestions of future work in the previous section may be used to increase the accuracy.

Although GraphMatch at present does not rate its matches according to severity, its present implementation can easily be adapted to do so. The simplest way of implementing a rating would be to make use of the number of steps in the definition-use chain of each transitive edge match. It seems likely that the longer the chain is, the smaller is the risk of exploitation. Severity ranking could also be done using some ranking system for pattern vertices, e.g. rating input from network higher than input from the command-line.
The second requirement is fulfilled by GraphMatch. On the third requirement we are again far from perfection. When handling large projects of perhaps a couple of hundreds of thousands lines of code we will definitely not be able perform a satisfying analysis in a couple of hours time unless a better algorithm is found.

Our final conclusion is that we cannot yet determine if the dependence graph matching technique is useful or not. However, this thesis has covered some ground, resolved some problems and defined a new set of questions to investigate. Of these, the most important is to find out whether or not there is a polynomial-time algorithm that will perform graph pattern matching with acceptable accuracy.
Chapter 5

Terminology

This chapter contains a comprehensive list of all main terms used in the thesis. For each term there is a brief explanation and reference to the context where it was originally introduced.

**Backward slice:** See slice.

**Conditional definition:** A conditional definition, or *conditional kill* of a variable $v$ occurs at a program point where a value might be assigned to $v$, but $v$ is not definitely killed. See page 9.

**Definition:** A definition of a variable $v$ occurs at a program point where a value is assigned to $v$. See page 9.

**Dependency closure:** All program points that can be reached following either data or control edges on feasible paths in an SDG. See page 32.

**Embedded negative pattern:** A negative property pattern that constitutes a subgraph of a corresponding positive pattern. See page 17.

**False negative:** Failure to recognize an actual error. See page 3.

**False positive:** Reporting something harmless as an error. See page 3.
**Forward slice:** All program points that may be affected by the value of a certain variable at a certain point. See also *slice* and page 27.

**Negative security property:** A security property that is not accepted by a given security policy. See page 6.

**Non-unifiable negative property:** A condition when we can not provide one single dependence graph model for all the unacceptable practices defined by a policy. See page 19.

**PDG:** Program Dependence Graph, a graph representing intraprocedural, but not interprocedural dependencies in a program. See page 11.

**Policied action:** The programming action which a security policy concerns. The action might be file access, use of data affected by external input, memory buffer copying or any other action that requires extra care for security reasons. See page 6.

**Positive security property:** A security property that is accepted by a given security policy. See page 6.

**Potential security bug:** A security policy violation that is not guaranteed to be a security bug. See page 7.

**Program point:** A program point is a statement, a control predicate or some other point of interest, such as a variable declaration or the entry point of a function. See page 9.

**Property pattern:** See *security property pattern*.

**SDG:** System Dependence Graph, a graph representing both intraprocedural and interprocedural dependencies in a program. See page 12.

**Security bug:** A bug that could serve as an entry point for an attacker. See also *vulnerability* and page 2.

**Security policy:** A rule that defines acceptable and unacceptable programming practices from a security perspective. A security policy concerns some specific programming action. See page 6.
**Security property:** A class of programming practices that may be used in connection to policied actions. Membership in a property is determined by a predicate on an arbitrarily large part of the program surrounding a policied action. See page 6.

**Security property pattern:** A dependence graph model of a security property used as a pattern to search for in an SDG. See page 17.

**Slice:** All program points that may affect the value of a certain variable at a certain point. Also known as *backward slice*. See also *forward slice* and page 27.

**Static analysis:** Automated analysis of source code or binary code performed without running the program. See page 2.

**Subgraph match:** A subgraph of an SDG that matches a certain property pattern. See also *vertex match* and page 17.

**Use:** A use of a variable $v$ occurs at a program point where the *value* of $v$ (rather than its name) may be required. See page 10.

**Vertex match:** A program vertex that matches a certain pattern vertex. See also *subgraph match* and page 17.

**Vulnerability:** Any part of an information system that could serve as an entry point for an attacker. See also *security bug* and page 2.


actual-out vertex, 13

calling context, 27

CLOSUREPASS algorithm, 33

CodeSurfer, 21, 24
   information extracted from, 25
   scripting API, 21

central edge
   interprocedural, 13
   intraprocedural, 11

data edge
   interprocedural, 13
   intraprocedural, 11

definition
   conditional, 9
   of variable, 9

dependence
   control dependence, 11
   data dependence, 11
   interprocedural, 11

dependence graph, 10
   program dependence graph, 11
   system dependence graph, 11

dependency closure, 32

DEPENDENCYCLOSURE algorithm, 33

EXTENDCONTAINERS algorithm, 36

external input, 18

Extracted Information, 25

FILTER algorithm, 35

formal-out vertex, 13

GrammaTech, 21

infeasible path, 27

input validation, 7
   correct
      model, 15
      property, 7
   missing
      model, 15
      property, 7

kill, see definition

link holder, 37

match container, 37

MATCHEDGES algorithm, 36
   simplified algorithm, 35

matching
INDEX

cost, 40
embedded negative patterns, 44
example, 30, 37, 45
single pattern, 34
MatchStep algorithm, 36
simplified algorithm, 35

pattern vertex, 17
PDG, Program Dependence Graph,
   see dependence graph
policed action, 6
program point, 9
program slice, 27
   backward, 27
   forward, 27
   interprocedural, 27
   intraprocedural, 27
program vertex, 17
programming action, 6
property pattern, see security property, pattern

relation point, 44

SDG, System Dependence Graph, see
   dependence graph
security bug, 2
security policy, 6
security property, 6
   model, 15
   negative, 6
      non-unifiable, 19
   pattern
      embedded negative, 18
      embedding, 18
   positive, 6
securityproperty
   pattern, 17
sensitive use, 18
slicing, see program slice
subgraph match, 17
summary edge, 27
System Dependence Graph, 11
tgt, 36
use
   of variable, 10
validation, 18
vertex holder, 37
vertex match, 17
vulnerability, 2
Copyright

Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphåva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

English

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Pia Fåk
Linköping, 25th August 2005