FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Exploiting JavaScript Birthmarking Techniques for Code Theft Detection

João Pinto

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Cardoso

July 28, 2016

Exploiting JavaScript Birthmarking Techniques for Code Theft Detection

João Pinto

Mestrado Integrado em Engenharia Informática e Computação

July 28, 2016

Abstract

The purpose of this dissertation is to analyse *birthmarking* techniques in order to detect code theft. A birthmark of a software is, as the name suggests, the set of unique characteristics that allow to identify that software. In order to detect the theft, the birthmarks of two programs are extracted, the suspect and the original, and compared to each other to check if they are too similar or identical.

Nowadays the web applications are growing and JavaScript code is the most used in this field, therefore the theft in this area is a current problem. Because of that, the theft detection of programs developed in that language is the focus of this dissertation.

Some techniques of birthmarking are analysed in chronological order and accordingly to the relevance for the theme. Each technique is analysed individually and in the end a comparison between them is made. Given that most of the techniques were not created for JavaScript, their applicability to the language is analysed. With those analysis, some conclusions about the best candidates to the final solution are drawn. The main conclusion is that the best birthmarking techniques are the ones that use dynamic analysis. The techniques using static analysis are susceptible to obfuscation techniques and therefore easily defeated.

The result of the dissertation is a tool, that uses a dynamic birthmarking technique, to determine if two JavaScript programs were copied. The tool do not generate false positives and is capable to detect the copy after obfuscation in order to support code theft allegations.

Resumo

Este relatório visa a análise de técnicas de *birthmarking* para detetar o roubo de código. Um birthmark de um software, como o próprio nome indica, é um conjunto de características únicas que permitem identificar esse mesmo software. Para a deteção do roubo de código são extraídos birthmarks de dois programas, o original e o suspeito, e são comparados um com o outro, permitindo assim detetar o roubo caso sejam muito semelhantes ou iguais.

Como hoje em dia a internet é cada vez mais utilizada e o código JavaScript é usado em grande parte das aplicações web, o roubo de código nesta área é atualmente um grande problema. Tendo isto em conta, a solução final tem como objetivo esta mesma linguagem.

São analisadas, cronologicamente e tendo em conta a relevância para o tema, algumas das técnicas de birthmarking existentes. As técnicas são analisadas individualmente e no final é feito um resumo e comparação de todas as técnicas. Como a maior parte das técnicas existentes não foram pensadas para JavaScript, a sua aplicabilidade à linguagem é também analisada e são tiradas conclusões acerca de bons candidatos à solução final. A principal conclusão retirada foi a vantagem do uso de técnicas de análise dinâmica. As técnicas de análise estática são muito susceptíveis a ofuscação, portanto podem ser facilmente derrotadas.

O resultado da dissertação é uma ferramenta que, usando uma técnica dinâmica de birthmarking, determina se dois programas JavaScript foram copiados. A ferramenta não gera falsos positivos e é capaz de detetar a cópia mesmo após o programa ser ofuscado, de modo a suportar alegações de roubo de código.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

CVFV	Constant Values in Field Variables
SMC	Sequence of Method Calls
IS	Inheritance Structure
UC	Used Classes
FSM	Frequent Subgraph Mining
DAG	Directed Acyclic Graph
WPP	Whole Program Path
CFG	Control Flow Graph
GUI	Graphical User Interface
JVM	Java Virtual Machine

Chapter 1

Introduction

With the arrive of the "Digital Era", companies are likely to become more dependent on software in order to make their business grow. The number of software companies is increasing [16] and their value is escalating quickly. In this area there is an increase of Startups with high market value despite having few human resources. [24]. The core values in this particular type of companies are neither the infrastructures nor physical objects, but the software they produce, being the code of that software their most valuable asset.

In this context, software piracy and code theft is a huge problem that some companies face [20] and can lead to bankruptcy of some of them. Considering this, the detection of code theft is essential for the stability and continuity of those companies. In order to protect the code of companies we have two possible approaches, the prevention or the detection of the theft. On the prevention side, obfuscation of code [1] is one of the best examples of what can be done, it makes the code extremely difficult to understand and therefore hard to reuse/steal. After the theft occurred, there are some techniques that allow its detection like Watermarking [5] and Birthmarking [23].

Watermarking consists in adding a message to the code in order to allow the detection of its theft. This watermark is searched in the suspect code to prove it was stolen. The use of watermarking requires extra time to add it to the code, being this time spent without bringing any new functionality to the program, therefore implies an extra effort of the programmer [5]. One disadvantage of the watermark is, in the case of being too simple, it can be detected by the developer and removed. If it is too complex, the cost of adding it to the code is high given that it does not bring any functionality to the program[6]. In order to solve some of those disadvantages, the concept of software birthmark has been proposed [23]. This new approach allows to detect the theft even after the watermark has been removed by transformations [21].

[1] Obfuscation: Deliberate attempt to make a programming code hard to understand by other people. It can be done by multiple reasons: hide the code goal (on the case of malicious code), difficult the process of reverse engineering, prevent the violation of copyrights, prevent the modification of the code, etc. Obfuscation is widely used in JavaScript because it is a language that have a lot of client side implementations [17].

Birthmark of a code or program is the set of unique characteristics that makes it unique. The idea of detecting theft using this technique consists in extracting and comparing birthmarks of suspect programs with the original. After this comparison it is possible to say, with a certain confidence, if the program was stolen. Birthmarking techniques can usually be evaluated considering two properties: credibility and resilience [14].

Being *f* the function that extracts the birthmark of a program:

Credibility : Being *p* and *q* programs independently written that can be used to accomplish the same task. We say that *f* is credible if *f(p)* is different from *f(q)*. In summary, credibility is the ability that a birthmarking technique has, of not generating false positives. If two programs were in fact written independently, their birthmark has to be different.

Resilience : Being *q* a program obtained from *p* by applying semantic preserving transformations (T), we can say that *f* is resilient to T if $f(q) = f(p)$. Resilience is the ability to resist semantic preserving transformations like obfuscation, minification or optimization. If a birthmarking technique has a higher resilience, it can detect the theft even after the code being transformed.

Besides the two properties mentioned, birthmarking techniques are also divided in two main categories: static and dynamic. Static techniques [23, 14] rely only on the program itself, therefore no execution of the program is needed. On the other side, we have dynamic techniques [15, 1, 3, 18, 13, 22] where the birthmark is extracted given an input. Unlike the static, dynamic techniques imply the execution of the program.

1.1 Context

This dissertation was made in enterprise context at *JScrambler* [19], a software company whose focus is in the area of web security. The company has a product with the same name (*JScrambler*), used to protect Javascript and HTML code. The tool *JScrambler* has multiple functionalities like minification, compression, optimization, obfuscation and "*code traps*". The goal is to make the clients code safer and hide it, through obfuscation, preventing their misuse. This product was developed in JavaScript, using the framework *Node.js* [2].

1.2 Motivation and Goals

The goal of this dissertation is to develop a tool that allow code theft detection. The tool must be able to indicate, with a percentage of confidence, if a program is a copy of another. The tool receives as inputs the original and the suspect program and outputs a percentage of similarity between them. In order to develop the tool, the extraction and comparison method of the birthmark need to be defined. The comparison method is directly related with the type of birthmark extracted. To make the choice of which technique to use, an analysis of the current approaches in multiple languages is made, pointing out the characteristics of each one.

[2] https://nodejs.org/en/

JavaScript language has particularities that can exclude some techniques because they are not applicable. With that in mind, the goal is to adopt or evolve a technique that suits the purpose of dealing with programs written in JavaScript language.

To evaluate the success of the dissertation, the tests consist in compare multiple programs using the developed tool to detect their similarity. The first tests are used to test credibility and the goal is to obtain no false positives when comparing programs developed independently. The success of these tests indicate the level of credibility of the tool. The second set of tests are used to check the resilience against transformations. These tests include minification and obfuscation in multiple levels using the tool of the company (JScrambler [19]). The goal is to obtain a high similarity between the original programs and the obfuscated versions of them. Another set of tests are performed in order to evaluate the capability of detecting partial theft like a library for instance.

The adopted birthmarking technique is expected to have both, high credibility and resilience, allowing the results to have high accuracy. The main goal is to have a tool that do not detect false positives but is also capable of detecting the theft even after obfuscation. An overview of the input and output of the tool is presented in Figure 1.1.

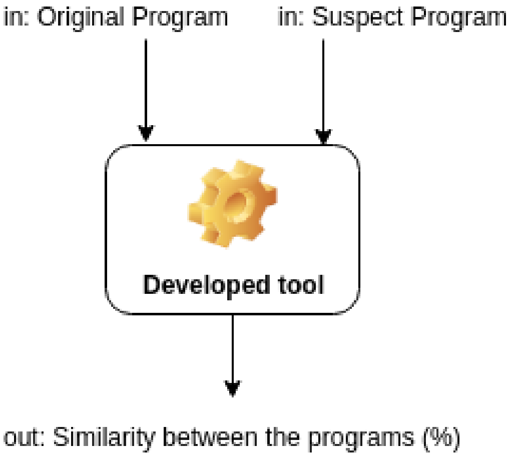


Figure 1.1: *Overview of the tool*

1.3 Document Structure

This document is divided in 6 chapters. Chapter 2 analyses the current birthmarking techniques and explains them. They are presented in a chronological order and accordingly to the relevance to the project. After describing all the techniques, an analysis comparing them is done. With that analysis some conclusions, about which techniques to use, are drawn. Chapter 3 describes the overall design and arquitecture of the solution developed. The modules of the tool and also its flow are explained. The path taken by the original and the suspect program is also presented in this chapter. Chapter 4 contains the details about the implementation with some pseudo-code and some examples of inputs and outputs. The experiments and results are described in Chapter 5.

An explanation about the experiences and the analysis and discussion of results are part of this chapter. The document ends with the conclusions and future work in Chapter 6.

Chapter 2

Current Birthmarking Techniques

2.1 Introduction

In this chapter multiple techniques of birthmarking are presented in a chronological order and accordingly to the relevance for the theme.

To the best of our knowledge the first approach to this problem used *Java Class Files* to extract birthmarks and is explained in Section 2.2. The second technique analysed (Section 2.3) is the *k-gram* [14], that uses *opcodes* from the executable file of the program. This technique proved to have higher resilience and credibility than the first one presented. Both methods are based on static analysis, therefore the possibility of being defeated by obfuscation is high. The first dynamic approach presented (Section 2.4), *Whole Program Path Birthmarks*, was proposed by G.Myles and C. Collberg [13] and introduced the first technique that used the execution of the program to extract the birthmark. Another dynamic technique, using API calls during the execution of a windows program, is analysed in Section 2.5. *Dynamic k-gram*[1], analysed in Section 2.6, is an evolution of the static *k-gram* that aims to solve the obfuscation problem.

Those first techniques were not developed for the JavaScript language. JavaScript focused techniques only appeared more recently and consist of dynamic analysis through the memory *Heap*. To finish this chapter, two techniques based on the *Heap* are described (Section 2.7 and 2.8).

2.2 Original Approach to Birthmark

Tamada, Haruaki, et al. [23] proposed in 2003 a method to extract a software birthmark. To the best of our knowledge this was the first approach on birthmarking. In this technique the birthmark is extracted from a *Java class file*. Using that file, the authors proposed four kinds of birthmarks: *Constant Values in Field Variables (CVFV)*, *Sequence of Method Calls (SMC)*, *Inheritance Structure (IS)* and *Used Classes (UC)*. In order to facilitate the understanding, a class example (Listing 2.1) is used.

```
import java.io.*;
import org.apache.tools.ant.Project;
import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.DirectoryScanner;
import org.apache.tools.ant.taskdefs.MatchingTask;
public class ViewSerialVersionTask extends MatchingTask{
  private static final String DEFAULT_BASE_DIR = ".";
  private File baseDir;
  public ViewSerialVersionTask(){
  }
  public void setBasedir(File baseDir){
    this.baseDir = baseDir;
  }
  public void execute() throws BuildException{
    if(baseDir == null) baseDir = new File(DEFAULT_BASE_DIR);
    DirectoryScanner scanner = getDirectoryScanner(baseDir);
  String[] list = scanner.getIncludedFiles();
    for(int i = 0; i < list.length; i++){
      log(list[i], Project.MSG_DEBUG);
      printSerialVersionUID(list[i]);
  }
  }

  private void printSerialVersionUID(String target){
    File inFile = new File(baseDir, target);
    if(!target.endsWith(".class")) return;
    try{
      String className = target.substring(0, target.length() - 6);
      className = className.replace(’/’, ’.’).replace(’\\’, ’.’);
      Class c = Class.forName(className);
      if(checkSerializable(c)){
        ObjectStreamClass osc = ObjectStreamClass.lookup(c);
        long serialVersionUID = osc.getSerialVersionUID();
        System.out.println(c.getName() + ": " + serialVersionUID);
      }
      } catch(Exception e){
        throw new BuildException(e.getMessage());
      }
  }
  private boolean checkSerializable(Class c){
    Class[] interfaces = c.getInterfaces();
    for(int i = 0; i < interfaces.length; i++){
      if("java.io.Serializable".equals(interfaces[i].getName()))
      return true;
    }
  return false;
  }
}
```

Listing 2.1: Java Class File Example (source: [23])

```
(java.lang.String, ".")
(java.io.File, null)
```

Listing 2.2: CVFV birthmark (source: [23])

2.2.1 *CVFV - Constant Values in Field Variables*

A Java class usually contains variables to save static/dynamic values of instantiated objects. If those variables are declared with constant values, they are essential to understand how the objects are instantiated. Based on this idea, Tamada, Haruaki, et al. [23] decided to create the CVFV birthmark.

Definition of CVFV birthmark: [23] Let p be a class file and $v_1, v_2, ..., v_n$ be field variables declared in p. Also, let $t_i (1 \leq i \leq n)$ be the type of v_i and $a_i (1 \leq i \leq n)$ be the initial value assigned to v_i in the declaration. (If a_i is not presented, we regard a_i as *"null"*). Then, the sequence $((t_1, a_1), (t_2, a_2), ..., (t_n, a_n))$ is called CVFV birthmark of p, denoted by $CVFV(p)$.

Accordingly to the definition and considering the class example (Listing 2.1), the CVFV birthmark would be the one described in Listing 2.2.

2.2.2 *SMC - Sequence of Method Calls*

The SMC technique takes advantage of the existence of many functions that are already implemented as methods of "*well-known*" classes (of JDK for example), and observes the use of them to create a birthmark. The sequence of those functions is hard to change automatically due to dependencies between methods, and require a lot of time to do manually. Those reasons motivated the creation of the SMC birthmark [23].

Definition of SMC birthmark: [23] Let p be a class file and C be a given set of *well-known* classes. Let $m_1, m_2, ..., m_n$ be a sequence of methods m_i 's invoked in p in this order, where m_i belongs to a class in C. Then, the sequence $m_1, m_2, ..., m_n$ is called SMC birthmark of p, denoted by $SMC(p)$.

Accordingly to the definition and considering the class example (Listing 2.1), the SMC birthmark would be the one described in Listing 2.3.

2.2.3 *IS - Inheritance Structure*

The IS birthmark is based on the inheritance structure of the classes. Because classes developed by users are easily modified, the technique only uses the *"well-known"* Java classes.

Definition of IS birthmark: [23] Let p be a class file and C be a given set of *well-known* classes. Let $c_1, c_2, ..., c_n$ be a sequence of classes such that $c_1 = p$, $c_i (2 \leq i \leq n)$ is a superclass of $c_i - 1$, and c_n is a root of class hierarchy (*java.lang.Object*). If c_i does not belong to a class in C, we replace c_i with *"null"*. Then, the resultant sequence $c_1, c_2, ..., c_n)$ is called IS birthmark of p, denoted by $IS(p)$.

```
org.apache.tools.ant.taskdefs.MatchingTask(),
java.io.File(String),
String[] org.apache.tools.ant.DirectoryScanner#getIncludedFiles(),
java.io.File(java.io.File, String),
void String#endsWith(String),
int String#length(),
String String#substring(int, int),
String String#replace(char, char),
String String#replace(char, char),
Class Class#forName(String),
java.io.ObjectStreamClass java.io.ObjectStreamClass#lookup(Class),
long java.io.ObjectStreamClass#getSerialVersionUID(),
StringBuffer(),
String Class#getName(),
StringBuffer StringBuffer#append(String),
StringBuffer StringBuffer#append(String),
StringBuffer StringBuffer#append(long),
String StringBuffer#toString(),
void java.io.PrintStream#println(String),
String Exception#getMessage(),
org.apache.tools.ant.BuildException(String),
Class[] Class#getInterfaces(),
String Class#getName(),
boolean String#equals(Object).
```

Listing 2.3: SMC birthmark (source: [23])

Accordingly to the definition and considering the class example (Listing 2.1), the IS birthmark would be the one described in Listing 2.4.

2.2.4 *UC - Used Classes*

This approach looks at the used classes, being once again only considered the "*well-known*" classes. Changing the "*well-known*" classes of a program is difficult without changing the functionality of it, therefore the idea of UC birthmark was created.

Definition of UC birthmark: [23] Le p be a class file and C be a given set of *well-known* classes. Let U be a set of classes u's such that u is used in p and $u \in C$. Let $u_1, u_2, ..., u_n$ $(u_i \in U)$ be a sequence obtained by arranging all elements of U in an alphabetic order. Then, the sequence $(u_1, u_2, ..., u_n)$ is called UC birthmark of p, denoted by $UC(p)$.

```
org.apache.tools.ant.taskdefs.MatchingTask,
org.apache.tools.ant.Task,
org.apache.tools.ant.ProjectComponent,
java.lang.Object.
```

Listing 2.4: IS birthmark (source: [23])

```
java.io.File,
java.io.ObjectStreamClass,
java.io.PrintStream,
java.lang.Class,
java.lang.Exception,
java.lang.String,
java.lang.StringBuffer,
java.lang.System,
org.apache.tools.ant.BuildException,
org.apache.tools.ant.taskdefs.MatchingTask,
org.apache.tools.ant.DirectoryScanner.
```

Listing 2.5: UC birthmark (source: [23])

Accordingly to the definition and considering the class example (Listing 2.1), the UC birthmark would be the one described in Listing 2.5.

2.2.5 Similarity analysis

In order to compare programs, the concept of similarity was created:

Definition of similarity: [23] Let $f(p) = (p_1, ..., p_n)$ and $f(p) = (q_1, ...q_n)$ be birthmarks with length n, extracted from class files p and q. Let s be the number of pairs (p_i, q_i)'s such that $p_i = q_i (1 \leq i \leq n)$. Then, similarity between $f(p)$ and $f(q)$ is defined by: $s/n * 100$. The similarity is a percentage of elements matched along $f(p)$ and $f(q)$ in the total elements in the birthmark (sequence).

2.2.6 Tests

The techniques presented in Sections 2.2.1, 2.2.2, 2.2.3 and 2.2.4 were tested with the following elements: bce1-5-1.jar (Jakarta BCEL v5.1), ant.jar (Apache Ant v1.5.2) e junit.jar (JUnit v3.8.1).

With the purpose of testing **credibility**, for each one of the *.jar* files, the comparison was made using pairs of *class files* contained in them. For each pair tested, each one of the techniques (CVFV,SMC,IS,UC) was tested alone and then combined with others. The goal of those tests was to obtain different birthmarks between classes.

The results shown that when the techniques are combined, they obtain the best results. In some cases the techniques could not differentiate the classes. A further analysis concluded that happen due to the classes being too small or almost identical.

For the **resilience** test, *jarg* (Java Archive Grinder) was used to optimize the package junit.jar. After the optimization, the original package was compared with the optimized one using the same method of grouping classes in pairs. The results shown that all the pairs of classes presented similarity above 80%. The minimum, maximum and average of similarity were 83%, 100% and 97,3%.

2.2.7 Conclusion

To the best of our knowledge, this approach was the first in the area of birthmarking and was able to obtain some interesting results. However, this technique was not tested with strong transformations (like obfuscation for instance) and therefore the results do not contemplate those transformations. Another characteristic of this technique is that it was developed specifically for the Java Language.

2.3 *K-gram Birthmark*

The *k-gram* technique was proposed in 2005 by Ginger Myles and Christian Collberg [14]. It is inspired on the division of files in k-grams or "chunks" in order to find similarity between documents or programs.

One *k-gram* is a continuous sub-string (of size *k*) that can be formed by words, letters, or in this case *Opcodes*. Listing 2.6 illustrates some examples of JVM Opcodes.

```
Constants:
00 (0x00) nop
01 (0x01) aconst_null
02 (0x02) iconst_null
...
Loads:
21 (0x15) iload
22 (0x16) lload
23 (0x17) fload
...
```

Listing 2.6: Examples of Java Opcodes (source: [12])

This technique uses the static analysis of the executable of the program in order to obtain the sequence of instructions (Opcodes). For each method, the set of unique *k-grams* is obtained by sliding a window of size *k* over that sequence. The birthmark of a method is the set of unique *k-grams*. The birthmark of the program is the union of the birthmarks of all methods. Because only the unique *k-grams* are used, the order of the Opcodes and their frequency are irrelevant. That characteristic is important because it makes the birthmark less susceptible to semantic preserved transformations.

The size of the window has influence in the results of this birthmark, therefore the value of k should be chosen wisely. For a $k = 7$ we will have sets of *k-grams* with 7 Opcodes each. The larger the window size is, the programs will have less *k-grams* in common.

2.3.1 Similarity analysis

A birthmark is the set of unique sequences of *Opcodes* of k size. Being $f(p) = \{p_1, ..., p_n\}$ and $f(q) = \{q_1, ..., q_n\}$ the birthmarks of two programs p and q, such that $p_1, ..., p_n$ and $q_1, ..., q_n$ are

Opcodes sequences. We consider two programs to be the same if $f(p) = f(q)$. Because we can apply transformations like obfuscation and optimization, this similarity is not exact, instead it is presented in the form of a percentage. Considering p the original program and q the suspect one, the definition of similarity is the following:

Definition of similarity: [14] Let $f(p) = \{p_1, ..., p_n\}$ and $f(q) = \{q_1, ..., q_m\}$ be *k-gram* birthmarks extracted from the sets of modules p and q. The similarity between $f(p)$ and $f(q)$ is defined by:

$$s(p,q) = \frac{|f(p) \bigcap f(q)|}{|f(p)|} \times 100$$

2.3.2 Tests

This technique was tested using 222 *Java jar-files* obtained from the Internet. Those files had different sizes, containing between 2 and 11,329 methods and 1 to 586 classes. The first test, to analyse the **credibility**, the programs were aggregated in groups of two (111 pairs). The percentage of similarity was obtained for values of k between 1 and 8. The results shown that as the value of k increase, the percentage of similarity between programs decreased. However, even with $k = 8$, five pairs of programs had similarity above 60% and one pair had 100%. After analysing those cases, was concluded that the programs with 100% similarity were identical. In all the five pairs above 60%, one element was a newer version of the other, justifying the high similarity. The results showed that for a high value of k the possibility of false positives was low.

The second test, to analyse **resilience**, consisted in applying some transformations and compare the original program with the transformed one. The target program for the transformations was *Conzilla*[1]. The program was submitted to transformations of three different optimization/obfuscation tools: *Codeshield* (deprecated), *SandMark*[2] and *Smokescreen*[3]. The experiments were made for a k between 2 and 8. Using *CodeShield* the percentage of similarity was 100% for all the values of k. With the transformation applied by *Smokescreen*, the similarity changed between 93% and 62% as the value of k increased. *SandMark* had 33 types of obfuscation and for 25 of them the similarity between the transformed program and the original was above 80%.

2.3.3 Conclusion

One conclusion of this technique is that the value of k must be chosen wisely, because it influences both the credibility and the resilience. As k increases the credibility also increases, however the resilience decreases. With the experiments done, the optimal value of k was considered to be 4 or 5, because were the values that given the best combination between resilience and credibility. With the tests made, namely the *SandMark* obfuscation, is only mentioned the values of 25 of the 33 obfuscations types, witch lead to believe that this technique is susceptible to some types of obfuscation.

[1] `http://www.conzilla.org/wiki/Overview/Main`

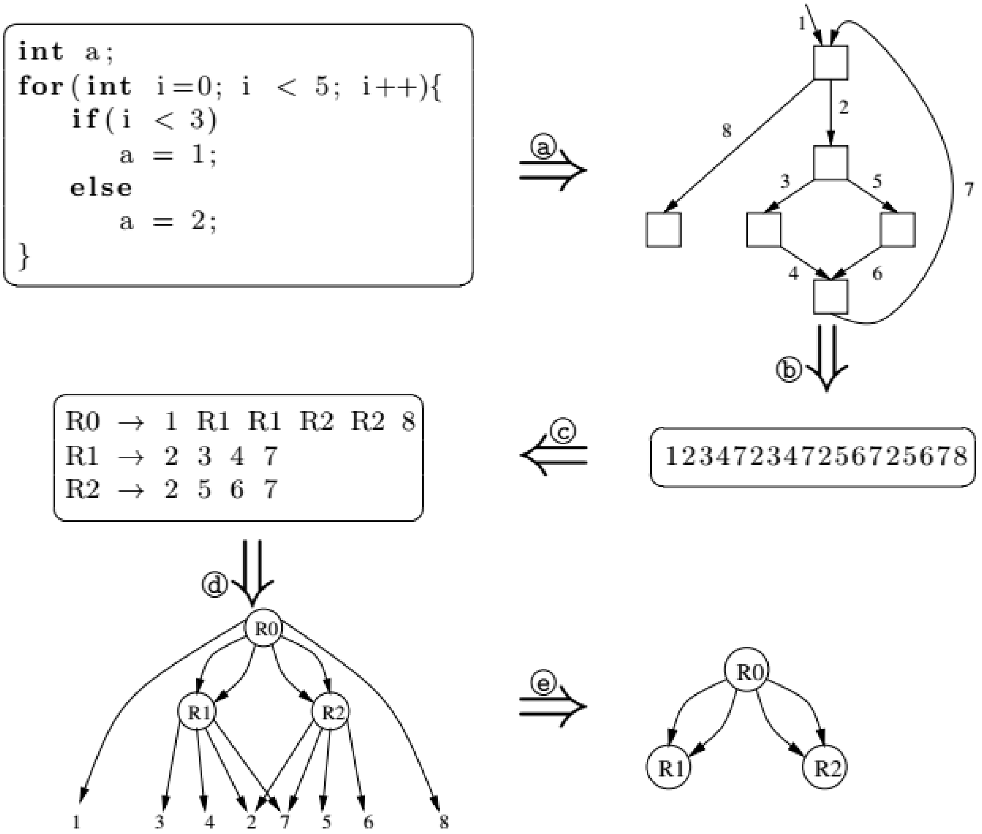
[2] `http://sandmark.cs.arizona.edu/`

[3] `http://www.softpedia.com/get/Programming/Other-Programming-Files/Smokescreen-Obfuscator.shtml`

2.4 *Whole program path birthmark (WPP)*

After realising that the birthmarking techniques at the time were ineffective against obfuscation, Ginger Myles and Christian Collberg [13] decided to present the first known dynamic birthmark to try to solve that problem. WPPB (*Whole Program Path Birthmarking*) was the name given to the technique that is based on the control flow of the program.

The birthmark is a directed acyclic graph (DAG) that represents the execution of the program. The process for obtaining this birthmark is illustrated in Figure 2.1. The first step consists in constructing the control flow graph (CFG) of the program and uniquely labelling each edge. Next, the program is executed with a given input in order to allow retrieving the path taken by the execution. As the program is executed, the edges are recorded to produce a trace (represented as a set of edges). That trace is processed by the SEQUITUR [10] to create a context free grammar. This grammar is then used to make a DAG in which the terminal nodes and edges will be removed. This removal is made because we only want to consider the internal nodes for the birthmark, due to being the most difficult to alter in the program.



```
int a;
for(int i=0; i < 5; i++){
    if(i < 3)
        a = 1;
    else
        a = 2;
}
```

```
R0 → 1 R1 R1 R2 R2 8
R1 → 2 3 4 7
R2 → 2 5 6 7
```

```
123472347256725678
```

Figure 2.1: *WPP birthmark construction process* (source: [13])

2.4.1 Similarity analysis

Because this birthmark is represented by a graph, the graph similarity is used to compare birthmarks. The process consists in finding the maximum common sub-graph between the two programs and compare its size (number of nodes) with the original. Considering $G1$ and $G2$ the graphs of the original and suspect programs respectively, and that mcs represents the maximum common sub-graph, the formula to calculate the similarity is the following:

$$d(G1,G2) = \frac{|mcs(G1,G2)|}{|G1|}$$

By multiplying the result by 100 we have the percentage of similarity between the two birthmarks.

2.4.2 Tests

To test the **credibility**, programs for solving two problems were developed using the recursive and the iterative approach. The problems were: calculate the factorial of a number and generate Fibonacci numbers. The goal of this test was to obtain different birthmarks (low similarity percentage) for the two approaches because they were developed differently. In the factorial problem the similarity obtained was 50% and in the Fibonacci the value of similarity was only 7%. This programs were also tested using the first birthmarking techniques presented in section 2.2 (CVFV,SMC,UC and IS) and was proven that they did not work for this small programs, however WPP birthmark had success in this test.

In order to test the **resilience** a program was obfuscated and optimized using multiple tools, namely *Zelix Klassmaster* (ZKM) [4], *Smokescreen* [5], *CodeShield* (deprecated) and *SandMark* [6]. The target of the obfuscation was a Java program *wc.jar* that works as the *wc* program of UNIX [7]. For all obfuscations used, the similarity between the original program and the transformed one were always 100%, therefore the test was a success. The first birthmarking techniques presented were also tested and the WPP birthmark proved to be superior to all of them.

Besides this test, a watermark was introduced in the program in order to evaluate its presence after obfuscation. After the program being obfuscated the watermark was destroyed, meanwhile the birthmark still could be used. This test was made in order to prove that a birthmark can prove code theft even after a watermark being destroyed due to transformations.

2.4.3 Conclusion

The goal of this technique was to prove that it was superior to the original proposed by Tamada, Haruaki, et al.[23] (Section 2.2). The birthmark proposed was in fact superior and could overcome

[4] http://www.zelix.com/klassmaster/

[5] http://www.softpedia.com/get/Programming/Other-Programming-Files/Smokescreen-Obfuscator.shtml

[6] http://sandmark.cs.arizona.edu/

[7] http://linux.about.com/library/cmd/blcmdl1_wc.htm

the flaw of the previous technique not working in small programs. The resilience of this technique was far superior than the old ones, mostly because they could not resist to obfuscations. A flaw of this birthmark is being susceptible to some cycle transformations, witch makes it useless against some *loop transformations*. Despite having good results, the technique was only tested in simple cases and it would be necessary to analyse it with a bigger set of tests in order to demonstrate its true value.

In general this was a big step in the area because it introduced the dynamic birthmarks category, witch can handle better transformations than the static ones.

2.5 *API calls*

In 2004 a new method of birthmarking , based on the API calls, was proposed [22]. This technique was created with the purpose of detecting theft of Windows applications. Because the API calls are not easily modified without modifying the behaviour of the program, the history of its calls is, in theory, a good birthmark. The birthmark is extracted using the *Microsoft Windows* API calls during the execution of the program, therefore it is dynamic.

Two versions of this birthmark were proposed: one that considers the sequence of API calls and the other considers the frequency of them. The sequence, of the API calls, is hard to change and therefore it is used to create a birthmark denominated "EXESEQ". The definition of this birthmark is the following:

Definition of EXESEQ birthmark: [22] Let p be a given program, I be a given input to p and W be a given set of API function names. Let $w_1, w_2, ..., w_n$ be a sequence of function calls called by executing p (execution order). If $w_i (1 \leq i \leq n)$ does not belong to W, we eliminate it from sequence. Then the resultant sequence $(w_1, w_2, ..., w_n)$ is called EXESEQ birthmark of p, denoted by $EXESEQ(p, I)$.

Considering that an attacker could change the order of the API calls, a second birthmark (EXEFREQ) was created and takes in account the frequency of those calls. The definition of that birthmark is the following:

Definition of EXEFREQ birthmark: [22] Let p be a given program, I be a given input to p and $(w_1, w_2, .., w_n)$ be a sequence of API function call birthmark of p $(EXESEQ(p, I))$. Let $(w'_1, w'_2, .., w'_m)$ be a sequence of function names witch obtained by eliminating duplicated function names from $EXESEQ(p, I)$. Also, let $k_i (1 \leq i \leq m)$ be a function name of w'_i and a_i be the number of appearances of k_i in $EXESEQ(p, I)$. Then, the sequence $((k_1, a_1), (k_2, a_2), ..., (k_m, a_m))$ is called EXEFREQ birthmark of p, denoted by $EXEFREQ(p, I)$.

2.5.1 Similarity analysis

To analyse the similarity between birthmarks *EXESEQ*, a *string matching* tool was used to detect similar sequences. The process of detection is illustrated at Figure. 2.2.

In order to compare *EXEFREQ*, a vector to represent the birthmark is created. Each element in that vector represents the number of calls of an API method, therefore the similarity is obtained

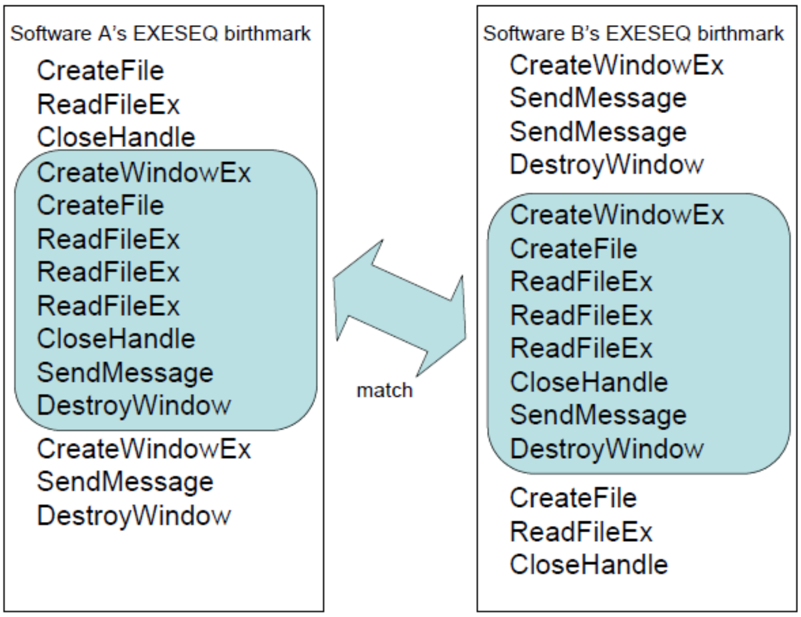


Figure 2.2: *Example of EXESEQ birthmark* (source: [22])

by computing the angle between vectors. An example of EXEFREQ birthmark is illustrated at Figure 2.3.

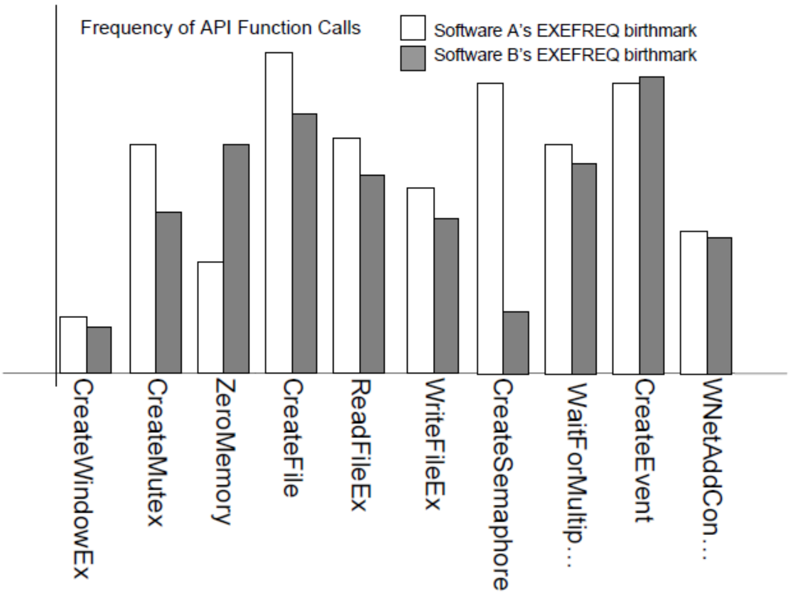


Figure 2.3: *Example of EXEFREQ birthmark* (source: [22])

2.5.2 Tests

The literature that presented the technique did not mentioned any tests or programs tested.

2.5.3 Conclusion

This birthmark was only theoretically evaluated. An obfuscator can change the calls of some internal functions, therefore modify the control flow of a program. The authors of this technique say that, unlike the internal functions, the modification of external libraries (namely API calls) is quite hard to do and therefore this technique can resist some types of obfuscation. If an attacker can change the API order without affecting the program specification, the EXESEQ birthmark is changed and can not detect the theft. However, in that situation the EXEFREQ birthmark remains the same, therefore can detect the theft occurrence. Overall the technique has decent resilience and credibility.

2.6 *Dynamic K-gram*

Because of the lack of tolerance against obfuscation of the static *k-gram* birthmark (section 2.3), the dynamic *k-gram* was created [1]. The previous technique used the static sequence of instructions, witch can be defeated by adding "dummy" methods. To solve this problem a dynamic version, witch implies the execution of the program, of the technique was created. The core ideas are the same of the static *k-gram* but it uses a trace of execution of the program instead of using the executable for extracting the sequence of instructions. These instructions are then filtered in order to prevent the sequence of being too big. The operands and redundant instructions are eliminated and then, like the static *k-gram*, a k sized window is slid over the instructions to get the birthmark.

2.6.1 Similarity Analysis

After getting the *k-grams*, the similarity analysis is done the same way as the static technique. (Section 2.3.1)

2.6.2 Tests

Because the goal of this technique was to obtain a higher resilience than the static *k-gram* (which has high credibility), tests to the credibility were not performed.

The program used for the experiments was *Conzilla*[8], a program with 32 classes and 79 methods. The thresholds (γ e γmin) of similarity were defined as 82% and 63%. If the value of similarity was higher than 82% (γ) it was considered a copy, if the value was lower than 62% (γmin) it was considered not a copy. For all the values between those the results were evaluated as inconclusive.

[8] http://www.conzilla.org/wiki/Overview/Main

To test the *resilience* , the program was transformed using three tools: *CodeShield* (deprecated), *SandMark*[9] and *Smokescreen*[10]. The transformed programs were compared, with the original, using both the static and the dynamic *k-gram* techniques. The results are represented in Figures 2.4 and 2.5. The static *k-gram* technique concluded that the optimal value of *k* is 4 or 5. If *k* is too small, a lot of false positives are generated. If we compare the techniques (static vs dynamic) with the optimal values of *k*, a good improvement is perceptible. We can verify that the static technique, for a value of *k* between 4 and 5, has bad resilience because the values of similarity are always bellow 70% and the lowest is 55%. On the other side, the dynamic technique has similarity always above 80% and reaches 85% on the best case.

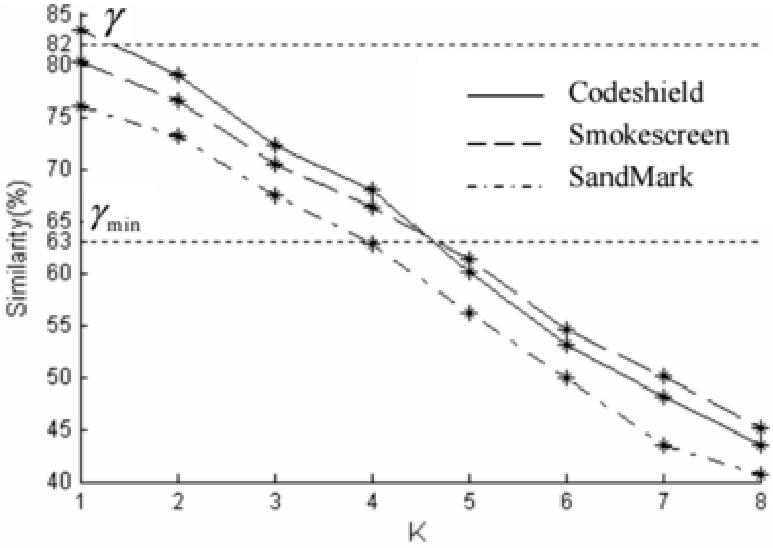


Figure 2.4: *Similarity values using static k-gram (source: [1])*

2.6.3 Conclusion

The goal of this technique (improve the resilience of the static *k-gram*) was accomplished with success by using the execution of the program. The results shown that the static *k-gram* could not detect the similarity in none of the experiments made, even using a small value of *k*. On the other hand the dynamic version presented was able to detect the similarity between the obfuscated programs using a k between 1 and 5. Given the results this technique was in fact a good improvement over the previous one.

2.7 *Run-time heap*

In 2011, P. P. F. Chan, L. C. K. Hui, and S. M. Yiu [3] proposed a birthmarking technique for the JavaScript language, to the best of our knowledge this was the first developed for the language. As

[9] http://sandmark.cs.arizona.edu/

[10] http://www.softpedia.com/get/Programming/Other-Programming-Files/Smokescreen-Obfuscator.shtml

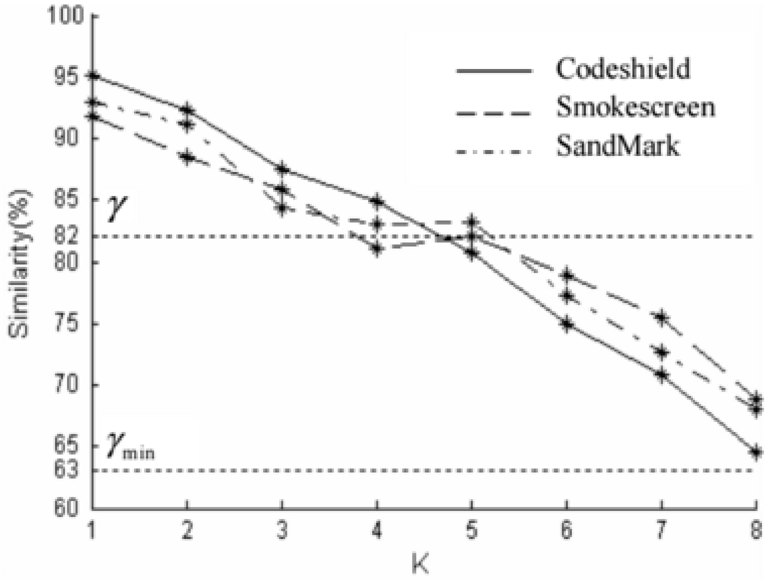


Figure 2.5: *Similarity values using dynamic k-gram (source: [1])*

the name suggests, this type of birthmark is based on the runtime heap analysis, which represents the dynamic behaviour of a program. The heap is a piece of the memory that keeps the objects created during the execution of the program. The idea is based on analysing the structure of the objects and their relations in order to understand if a program is a copy of the other.

The first step of this technique is extracting snapshots (through the browser) of the heap during the execution of the JavaScript programs. After obtaining those snapshots, they are filtered and interpreted in order to create a tree in which the nodes represent objects and their connections are represented by edges. The strategy was to take 20 snapshots during the execution of the program and join them together, like illustrated in Figure 2.6.

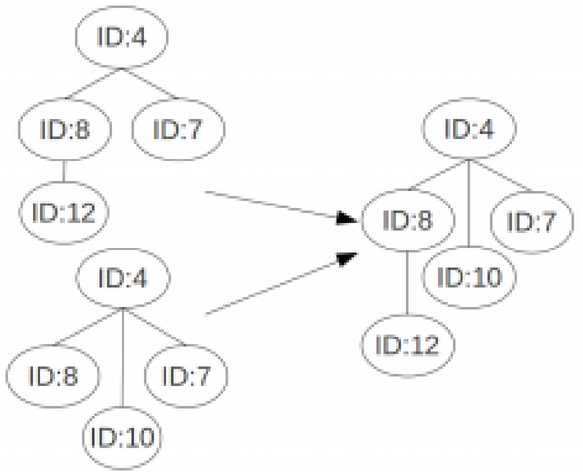


Figure 2.6: *Merging of 2 trees* (source: [3])

The trees are merged because the intention is to get the information about all the objects created by the program. The JavaScript engine (V8 [4]) can run the garbage collector and eliminate some objects that do not have any references, therefore the merging is essential to get all the objects. The next step is to filter the objects by eliminating some of the "standard" ones (HTMLDocument, JSON, DOMWindow, etc) and some auxiliary objects that do not reflect the behaviour of the program.

After those 2 phases, each program (the original and the suspect) has a representative tree of its behaviour. With those trees, the similarity analysis can start.

2.7.1 Similarity Analysis

A way of measuring the similarity between nodes was proposed and it considers the structure of the nodes, namely its edges, to get the similarity between two of them. Let Edg_A and Edg_B be the set of edges of node A (node from original program) and node B (node from suspect program) identified by their types, the similarity between them is given by:

$$N_sim(A,B) = \frac{|Edg_A \bigcap Edg_B|}{|Edg_A|}$$

Because the number of common edges between A and B is divided by the number of edges of the original node, the injection of references on the suspect node will not affect the similarity value. Since objects are represented by nodes in the heap, their similarity is calculated the same way. Each object of the original program is matched with the node of the suspect one that has the highest value of similarity. In order to facilitate the search, only the nodes with same types and with similar number of edges (number of edges differers only in m percent) are considered. After that match is done, each node (of the original program) belongs to a pair with the most similar node of the suspect program. Let $O_{sim}()$ be the similarity between objects, p be the original program, q be the suspect program, Obj_{i1} be an object from p and Obj_{i2} be its pair (belonging to q). The similarity between programs is given by:

$$Sim(p,q) = \frac{\sum O_{sim}(Obj_{i1}, Obj_{i2}) \times \text{Size of } Obj_{i1}}{\text{Total Heap size of } p}$$

Because each object can be also a tree (may have children objects), the algorithm $O_{sim}(X,Y)$ runs recursively through the tree of that node. The process of determining the similarity between objects can be costly if the tree is too big, the depth of the search is limited to a value d. For the experiments presented, the value of d was 3 and the value of m (difference between number of edges of the original and suspect node) was 50%.

2.7.2 Tests

To test this birthmark, multiple JavaScript programs from 4 different categories were used: 4 text editors, 4 graphic libraries, 5 games/game engines and 7 programs using jQuery [11]. The details of

[11] https://jquery.com/

the programs were not given. The programs were installed in a server and each one was opened in a modified browser that took a snapshot of the heap each 5 seconds until reaching 20 snapshots. While the snapshots were being taken, the program was being randomly used in order to activate the creation of more objects in memory. At the end of this phase and after merging the snapshots, each program had a file with the needed informations for the birthmark.

To test **credibility** the programs were compared with each other. All the comparisons obtained a value of 0% similarity between different programs. When the comparison was made with the same programs the results were always 100% with the exception of two cases where the values were 95,5% and 96,7%. The results shown that the technique had good credibility and therefore do not generate false positives. During the experiments some of the pairs could not be tested because they did not possess any "custom" objects therefore their birthmark could not be extracted. Those programs shown that the technique does not work for programs too small and simple, however if a program is that simple there is no point on detecting its possible theft.

The **resilience** test was done by obfuscating 8 of the programs using the tool *Jasob3* [12]. The obfuscated programs were then compared with their original versions in order to check its resemblance. The results shown that all the programs had a similarity value above 95,5% and 5 of them had 100%. This test revealed that the technique has high resilience against obfuscation.

To test a partial theft detection, a birthmark of the jQuery library was taken and then compared to all the programs that were using it. All the programs demonstrated similarity with jQuery, proving that the technique is also capable of detecting partial copy as expected.

2.7.3 Conclusion

Accordingly to the literature this was the first technique developed for the JavaScript language. Despite being the first, it had a good credibility and resilience due to its dynamic nature. Despite all the good results of this technique, it can require high computational power to be applied due to the big size of the heap graph. Another disadvantage is that the theft of small programs (without custom objects) may not be detected and therefore the technique does not work for all the programs.

2.8 *HEAP based - Agglomerative clustering and improved frequent sub-graph mining*

In 2014, S. Patel and T. Pattewar [18] proposed a technique based on the heap analysis to extract and compare the birthmarks of two programs. The core ideas of this technique are very similar to the one presented in Section 2.7, changing in the way the snapshots are used and compared. The process first step is to take multiple snapshots of the heap (graph representation) of the same program and merge them together. One difference from the previous technique is in the process of merging the graphs, to accomplish this task an agglomerative clustering algorithm was used.

[12] http://www.jasob.com/

Another difference is the way of selecting the birthmark of the programs, in this case the birthmark is selected by obtaining the biggest most frequent sub-graph of the complete one. To accomplish that, algorithms of FSM (frequent sub-graph mining) and Improved FSM were used. The birthmark of the original program is compared with the full graph (extracted from the snapshot) of the suspect one. The goal is to find the sub-graph of the original program (birthmark) inside the graph of the suspect, therefore proving it was copied.

The tests done with this technique showed a resilience and credibility similar to the one presented on Section 2.7.

2.9 Overview

Table 2.1: *Comparative table of the current birthmarking techniques (1)*

Technique	Year	Type	Use	JavaScript adaptability	Used Tests
Original - first approach to Birthmarking[23]	2003	Static	Use Java class files	Can be applicable	Junit.jar apache.jar Jakarta BCEL
K-gram[14]	2005	Static	Op-code level, Uses the executable of the program to extract the sequence of instructions.	Op-code level, Uses the executable of the program to extract the sequence of instructions.	222 java jar-files with multiple sizes
Whole program path[13]	2004	Dynamic	Uses the trace of an execution of the program	Depends on the possibility of generating a trace of execution of the program using V8 engine.	Same problem solved recursively and iteratively to test credibility. Obfuscation of WC.jar to test resilience
API calls[22]	2004	Dynamic	Extracts the API calls of an execution of the program	Not aplicable	No tests mentioned. Possible attacks were discussed
Dynamic k-gram[1]	2008	Dynamic	Extracts the opcodes from a trace of execution of the program	Depends on the possibility of generating a trace of execution of the program using V8 engine.	Program Conzilla obfuscated by CodeShield, SandMark and Smokescreen
Heap - JS-birth[3]	2011	Dynamic	Heap (snapshot of the heap taken in the browser)	Developed for JS	JavaScript apps:,4 text editors, 4 graphical libraries, 5 games/game engines and 7 jQuery programs. Obfuscated with Jasob 3
Heap - using Agglomerative Clustering and Improved Frequent Subgraph Mining[18]	2014	Dynamic	Heap (snapshot of the heap taken in the browser)	Developed for JS	300 combinations of sites from 10 diferent sectors

Table 2.2: *Comparative table of the current birthmarking techniques (2)*

Technique	Credibility	Resilience
Original - first approach to Birthmarking[23]	Medium	Low
K-gram[14]	High	Medium/Low
Whole program path[13]	Medium/High	Medium/High
API calls[22]	Medium	Medium
Dynamic k-gram[1]	High(few tests made, assuming it is as good as the static version)	Medium/High
Heap - JS-birth[3]	High	High
Heap - using Agglomerative Clustering and Improved Frequent Subgraph Mining[18]	High	High

The credibility and resilience values of Table 2.2 were inferred by analysis of papers and literature, they are qualitative and not 100% accurate because each technique was tested in different ways (with different programs, for different technologies, etc).

Table 2.1 shows a summary of the techniques analysed. Birthmarking techniques have been evolving and each new technique always aimed to solve some flaws of the previous ones. In the first approach presented (Section 2.2), the identification of the birthmark was made using Java Class files. This technique proposed four different ways of identifying the birthmark: CVFV, SMC, IS and UC. The experiments made had interesting results and despite being simple, they were able to detect the copies in multiple occasions. Due to being too simple, soon more robust techniques started to appear with higher credibility and resilience.

The *k-gram* technique [14] uses information from the execution of the program, namely the sequence of instructions, to detect the copy. It was proposed in 2005 and had success in improving the credibility upon the available techniques, however its resilience was not very good. Other static techniques were proposed but they all had the same problems with resilience due to their static nature. Those techniques could be defeated by some sort of obfuscation, which lead to the work on dynamic techniques.

The first dynamic technique was proposed in 2004 [13] and used the control flow of the program as base. The execution of the program produced a list of nodes that it executed. With that list, a context free grammar is created and used to generate a graph that, after getting filtered, represents the birthmark of the program. This technique was the first to have a decent resilience and therefore was a great step forward by introducing the dynamic birthmarks. Also in 2004 was created another dynamic technique which was based on the analysis of the API calls of *Windows* applications. This technique was resilient against some obfuscations however it is not directly adaptable for JavaScript. In 2008 the dynamic version of *k-gram* [1] technique was proposed in order to solve some weaknesses of the static version. The extraction and comparison of the birthmarks are similar, differing in the origin of the sequence of instructions. The dynamic version used a trace of execution instead of the executable of the program. This technique demonstrated a high resilience, convincingly better than the static version.

At this point was unanimous that the dynamic techniques were more resilient than the static ones. In 2011 the first dynamic technique for JavaScript was proposed [3] and its core idea was analysing the memory heap in order to get the birthmark. That analysis results in a graph that represents the birthmark of the program. This technique demonstrated having very good resilience and credibility results. Later in 2014, a new technique proposed by [18] presented a different way of selecting the birthmark of the program. This technique uses FSM (frequent subgraph mining) in order to find the largest frequent subgraph of the heap and use it as birthmark of the program. The core ideas are the same as the previous heap technique (Section 2.7), but it uses a standard algorithm to merge the graphs together and FSM to chose the birthmark. The tests performed showed that the credibility and resilience are as good as the previous technique.

2.10 Conclusion

Through the analysis of the most relevant work, the conclusion is that the static techniques have a lot of limitations in resilience. With that fact, the dynamic techniques, despite being harder to implement, are superiors and allow the code theft detection more accurately. The most promising technique so far is the one based on the heap analysis and therefore will be the one focused on this work.

Chapter 3

Design and Architecture

The solution implemented was based on the ideas of Section 2.7 and 2.8 and used the heap to detect the similarity between two programs. Because one of the goals is to have good resilience against obfuscation, the solution uses a dynamic birthmarking technique. This technique is the most promising technique developed for JavaScript.

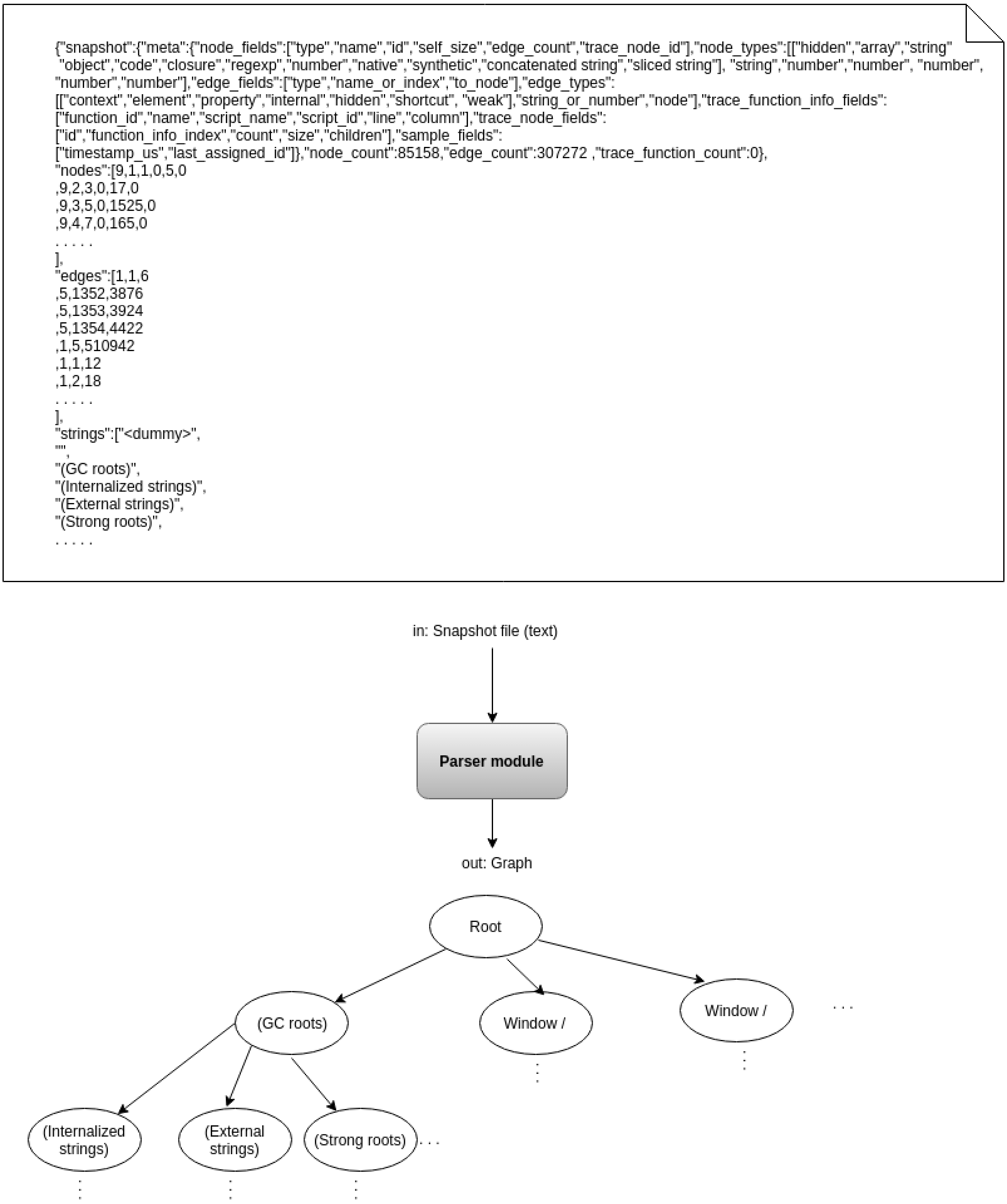
This chapter describes the overall design and architecture of the tool developed in this work. It is also explained how the tool works and the flow of the system. An overview about the modules of the system is described as well as their purposes. The sequence of processes that the snapshots, from the original and suspect sites, need to go through is also mentioned in this chapter.

3.1 Architecture of the system

The system is composed by five modules: Parser, Filter, Merger, Birthmark Extractor and Detector. The **Parser** module has the job of parsing the snapshot file and loading it into memory. It receives as input the snapshot of the heap and outputs a structure containing the information about it (a Graph). Figure 3.1 represents the behaviour of this module. More details about the structure of the snapshot are given in Section 4.2.

The loaded graph from the Parser module is usually big and therefore it requires a lot of memory for storing it. This is where the **Filter** module helps by eliminating some nodes from the graph. This module receives a graph as input and outputs another graph with less nodes than the original (Figure 3.2). The nodes that are not relevant to the representation of the program are eliminated. This module is very important and the choices of which nodes to filter are essential in order to get the smallest graph possible. The smaller the graph is, less effort is needed to compute it during the execution of the tool. The details of how we identify irrelevant nodes are explained in Section 4.4.

In order to use the tool, it is needed to take snapshots while the program is executed. To obtain a good representation of it, multiple snapshots are taken during the execution of the program. It is needed to take several snapshots because the JavaScript engine (V8) has a garbage collector that can run to free some memory. The garbage collector identifies dead regions in memory [7] that



```
{"snapshot":{"meta":{"node_fields":["type","name","id","self_size","edge_count","trace_node_id"],"node_types":[["hidden","array","string"
 "object","code","closure","regexp","number","native","synthetic","concatenated string","sliced string"], "string","number","number", "number",
"number","number"],"edge_fields":["type","name_or_index","to_node"],"edge_types":
[["context","element","property","internal","hidden","shortcut", "weak"],"string_or_number","node"],"trace_function_info_fields":
["function_id","name","script_name","script_id","line","column"],"trace_node_fields":
["id","function_info_index","count","size","children"],"sample_fields":
["timestamp_us","last_assigned_id"]},"node_count":85158,"edge_count":307272 ,"trace_function_count":0},
"nodes":[9,1,1,0,5,0
,9,2,3,0,17,0
,9,3,5,0,1525,0
,9,4,7,0,165,0
. . . . .
],
"edges":[1,1,6
,5,1352,3876
,5,1353,3924
,5,1354,4422
,1,5,510942
,1,1,12
,1,2,18
. . . . .
],
"strings":["<dummy>",
"",
"(GC roots)",
"(Internalized strings)",
"(External strings)",
"(Strong roots)",
. . . . .
```

Figure 3.1: *Parser Module*

can be released or re-used. It searches for "dead" objects in the heap and removes them. An object is considered "dead" either when there is no objects pointing to it, or the pointing objects are also "dead". An example representing the impact of the garbage collector on the heap graph is shown bellow:

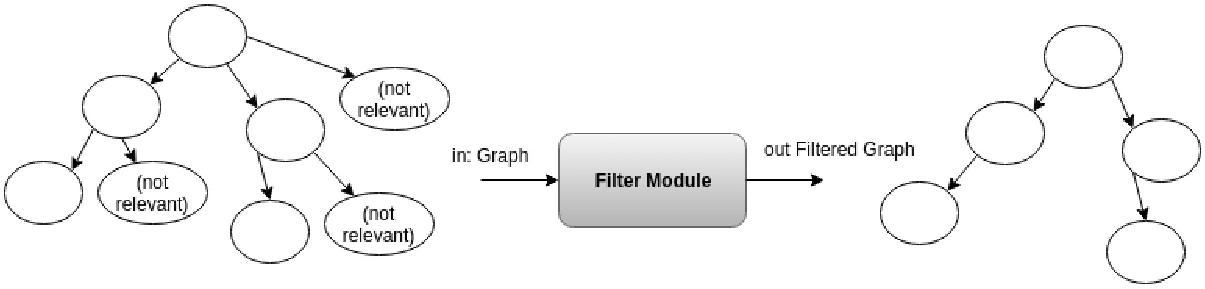


Figure 3.2: *Filter Module*

- **Snapshot 1**: Taken 15 seconds after the page loaded.
  - Number of Nodes: 177,505
  - Number of Edges: 336,125
- **Snapshot 2**: Taken 30 seconds after the page loaded.
  - Number of Nodes: 206,999
  - Number of Edges: 410,780
  - Eliminated Nodes: 309
  - Created nodes: 29,803
- **Snapshot 3**: Taken 45 seconds after the page loaded.
  - Number of Nodes: 217,308
  - Number of Edges: 439,823
  - Eliminated Nodes: 15,980
  - Created nodes: 26,289
- **Snapshot 4**: Taken 60 seconds after the page loaded.
  - Number of Nodes: 228,654
  - Number of Edges: 470,462
  - Eliminated Nodes: 1,467
  - Created nodes: 12,813

During the first minute of execution of the program, a lot of nodes were deleted (by the garbage collector) and added. From the first snapshot to the second (15-30 seconds), 309 nodes were deleted and 29,803 were created in the heap. From the second to the third (30-45 seconds), 15,980 nodes were deleted and 26289 were created. In this period, the garbage collector released a high amount of objects from memory that were "dead". In the last interval (45-60 seconds), only 1,467 nodes were eliminated and 12,813 were created. The low number of deleted nodes happened because on the previous period (30-45 seconds) a lot of nodes were eliminated by the garbage collector and therefore most of the "dead" objects were already deleted.

With the experiments done, it was concluded that the tool should use 4 snapshots taken during the first minute of execution of the program with an interval of 15 seconds between them. The tendency was a stabilization after the third snapshot taken and therefore all snapshots after the fourth have roughly the same content.

Because we have multiple snapshots of the same program, there is a need for a module that merges them. The **Graph Merger** module does exactly that, the combination of two graphs into one "super-graph". Its inputs are two graphs of the same program, and it outputs a graph containing all the nodes of both of them. An example of that procedure is illustrated in Figure 3.3. From the Graph 1 (left) to Graph 2 (right) some changes occurred: two nodes added (node with id "8" and "9"), two nodes removed (nodes with id "4" and "6") and 1 edge added (edge from node with id "5" to node with id "7"). The output is a graph containing all the nodes and edges from both graphs.

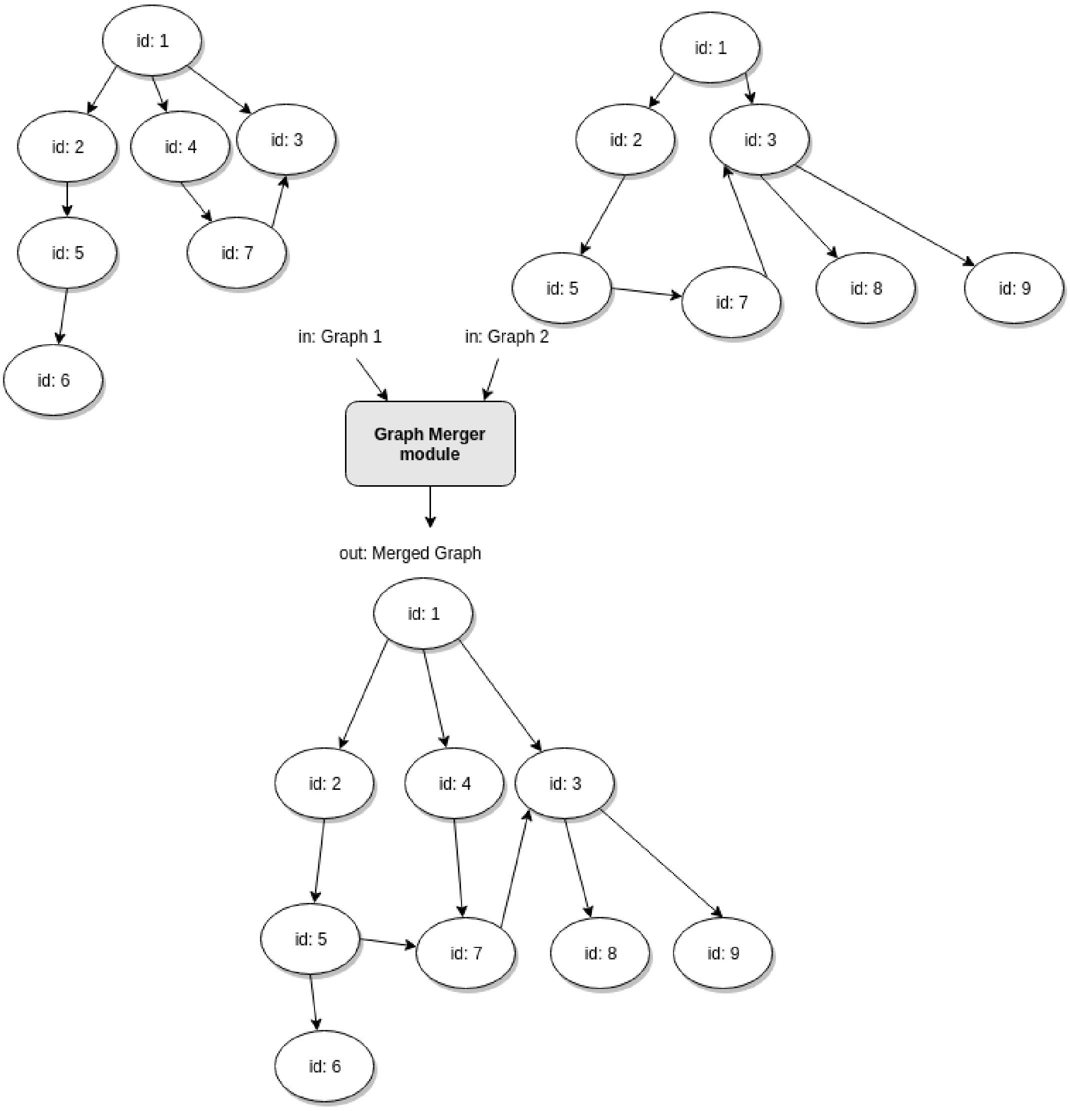


Figure 3.3: *Graph Merger module*

The module **Birthmark Extractor** is the one that extracts the birthmark from the original program, therefore the suspect program will not interact with it. This module has two versions, one that selects the largest subgraph and the other selects the largest frequent subgraph. The snapshots have a root node that connects multiple "windows" and each of those windows have a graph underneath. Figure 3.4 shows an example of that structure.

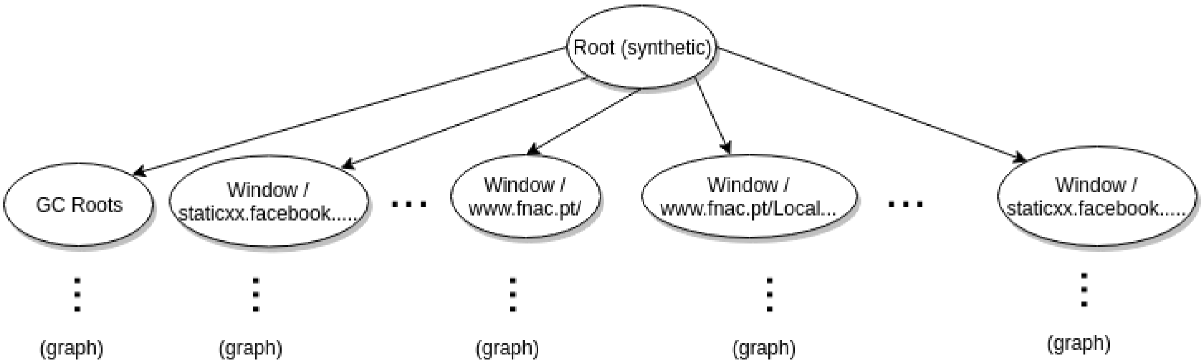

Root (synthetic)
GC Roots
Window / staticxx.facebook.....
Window / www.fnac.pt/
Window / www.fnac.pt/Local...
Window / staticxx.facebook.....
(graph)
(graph)
(graph)
(graph)
(graph)

Figure 3.4: *Example: snapshot graph of Fnac website*

The first version of the birthmark extractor selects the largest graph underneath the windows, because it is considered to better represent the program. The second version of the module looks at the entire graph and extracts the largest subgraph that is repeated. Basically it searches for a subgraph that appears more than once, supported by the idea that such subgraph will represent the behaviour of the program. Figure 3.5 illustrates an output example of this algorithm. The subgraphs highlighted inside the module represent the largest subgraph that occurs more than once, therefore they are chosen as the birthmark of the program.

In both versions, the result is a graph that will be considered the birthmark of the original program.

The last module is the **Detector** , and it is where the comparison between the two programs is done. This module receives the birthmark of the original program and the graph of the suspect. It then uses graph isomorphism to detect if they are copies and outputs a percentage of similarity between the programs. The goal of this module is to check if the original program birthmark (represented by a subgraph) exists inside the suspect program graph. The similarity percentage *Sim* is given by the number of nodes of the original birthmark *B* that were matched in the suspect graph *SG*, divided by the number of nodes of the birthmark:

$$Sim(B,SG) = \frac{|match(B,SG)|}{|B|}$$

Two examples of the output of this module are illustrated in Figure 3.6. The nodes matched by the module are highlighted in the suspect graph.

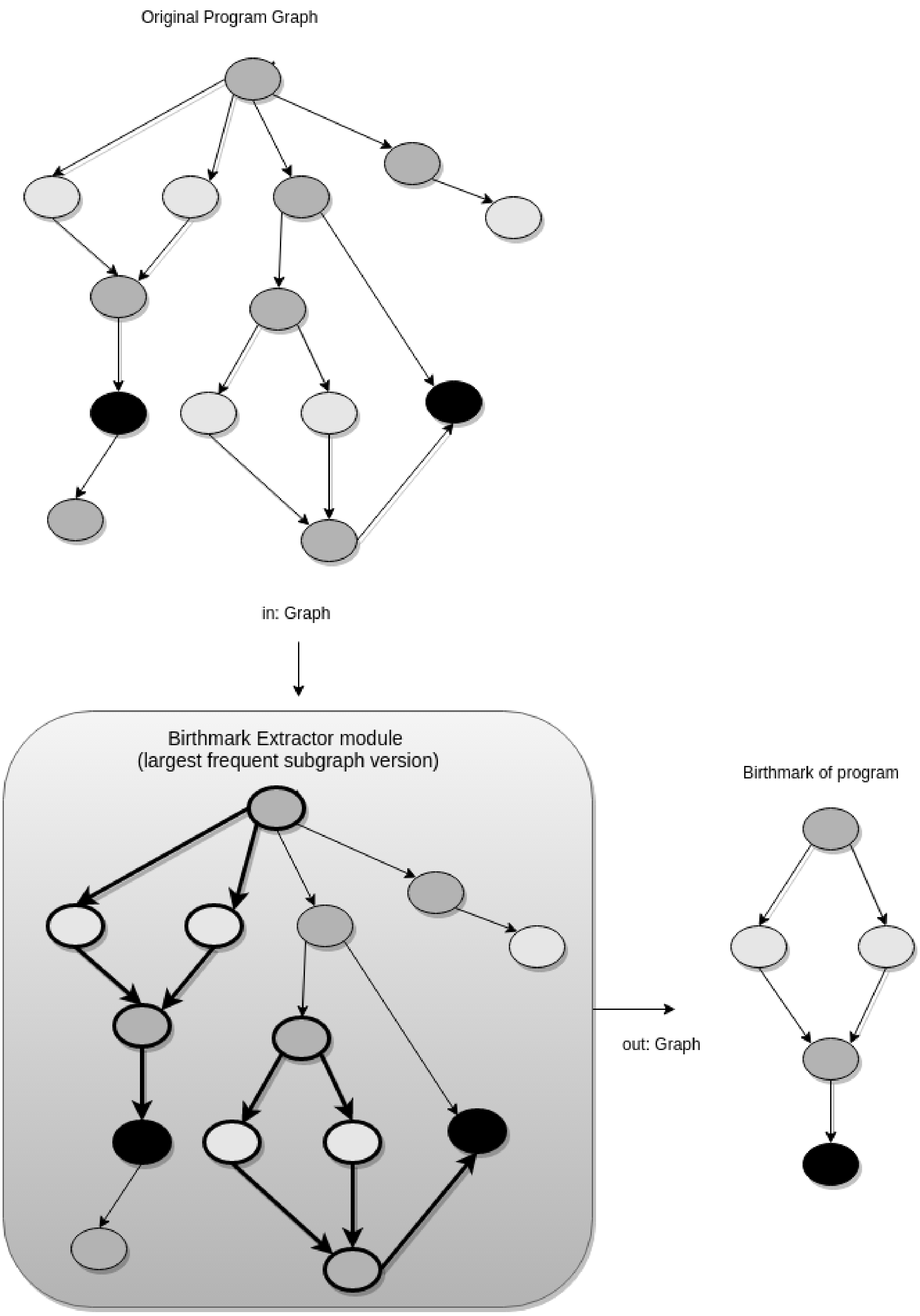

Original Program Graph
in: Graph
Birthmark Extractor module
(largest frequent subgraph version)
out: Graph
Birthmark of program

Figure 3.5: *Birthmark Extractor module - largest frequent subgraph version (different gray colours represent different node types)*

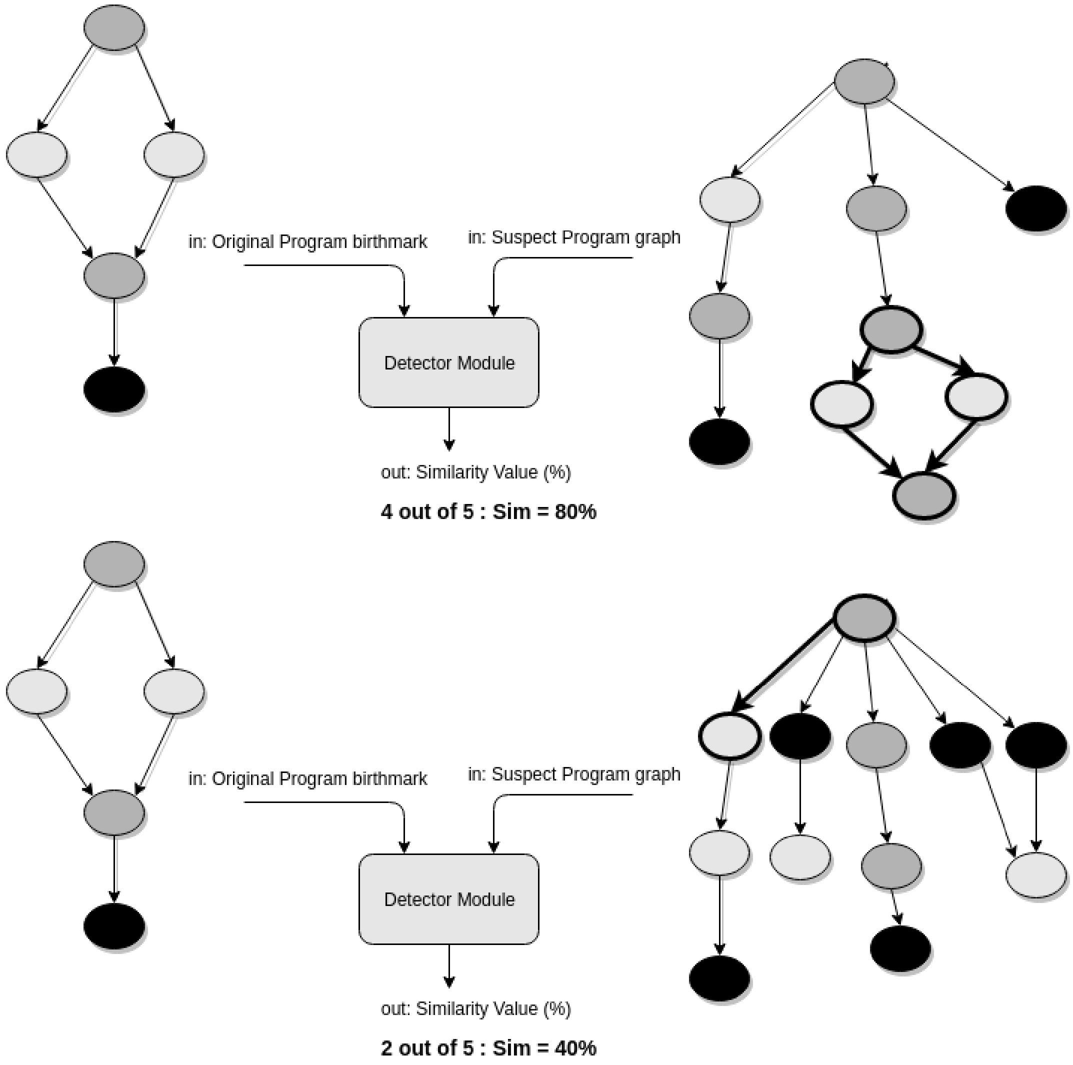

in: Original Program birthmark
in: Suspect Program graph
Detector Module
out: Similarity Value (%)
4 out of 5 : Sim = 80%
in: Original Program birthmark
in: Suspect Program graph
Detector Module
out: Similarity Value (%)
2 out of 5 : Sim = 40%

Figure 3.6: *Detector module*
(different gray colours represent different node types)

3.2 System Flow

The snapshots go through the modules following the sequence indicated in Figure 3.7. The figure illustrates the steps of the original program snapshots (left) and the suspect program snapshots (right). The first module is the **Parser** , in order to construct the graphs. The **Filter** is the next module in the sequence and it reduces the size of the graphs. Because we have 4 snapshots representing each program, the next phase consists in the union of the 4 snapshots into one and it is done by the **Graph Merger** module. At this point of the system, the suspect graph is ready to go to the **Detector** module, meanwhile the original graph still needs to go through another step. The **Birthmark Extractor** module is the destination of the original graph and it extracts the birthmark of it. After obtaining the birthmark of the original program, the **Detector** module has all the inputs it needs. This is the last step of the system and the result is a percentage of similarity between two programs indicating if there was a possible theft.

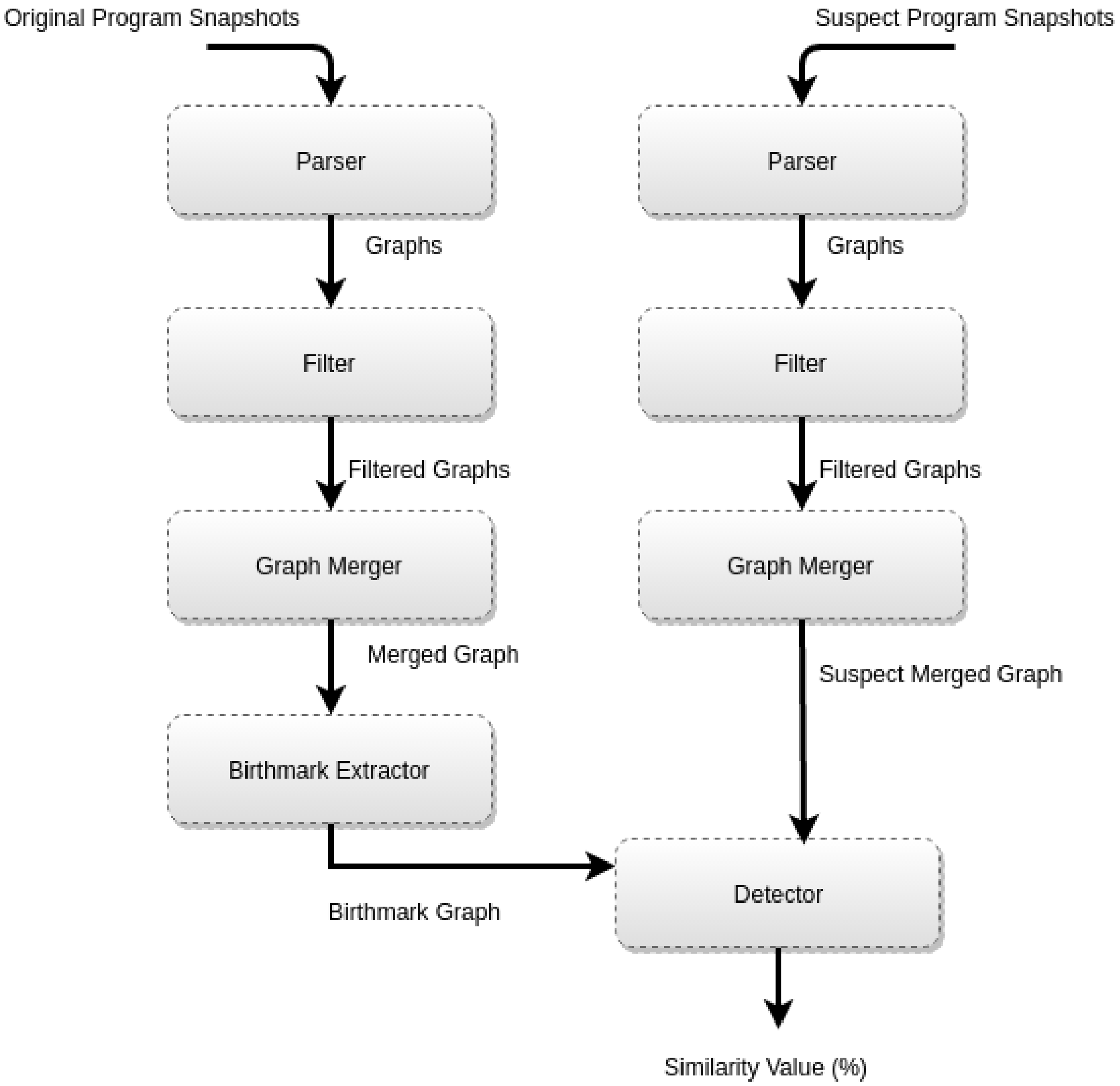


Figure 3.7: *Work flow of the tool developed*

A particularity about the system is that loading the snapshots into the tool, in order to merge them, consumes a lot of memory and therefore they are not loaded all beforehand. Instead, they are loaded as needed in order to economize memory usage. The process used to merge the 4 graphs of the same program is divided in the following steps: **a)** load snapshot 1 (into memory) **->** **b)** load snapshot 2 **->** **c)** merge snapshot 1 and 2 (result saved in 2) **->** **d)** load snapshot 3 **->** **e)** merge snapshot 2 and 3 **->** **f)** load snapshot 4 **->** **g)** merge snapshot 3 and 4. In the end of this sequence the snapshot 4 has the merged graph. This process is represented by the diagram in Figure 3.8 and it displays that there are no more than 2 snapshots (graphs) in memory at a given point.

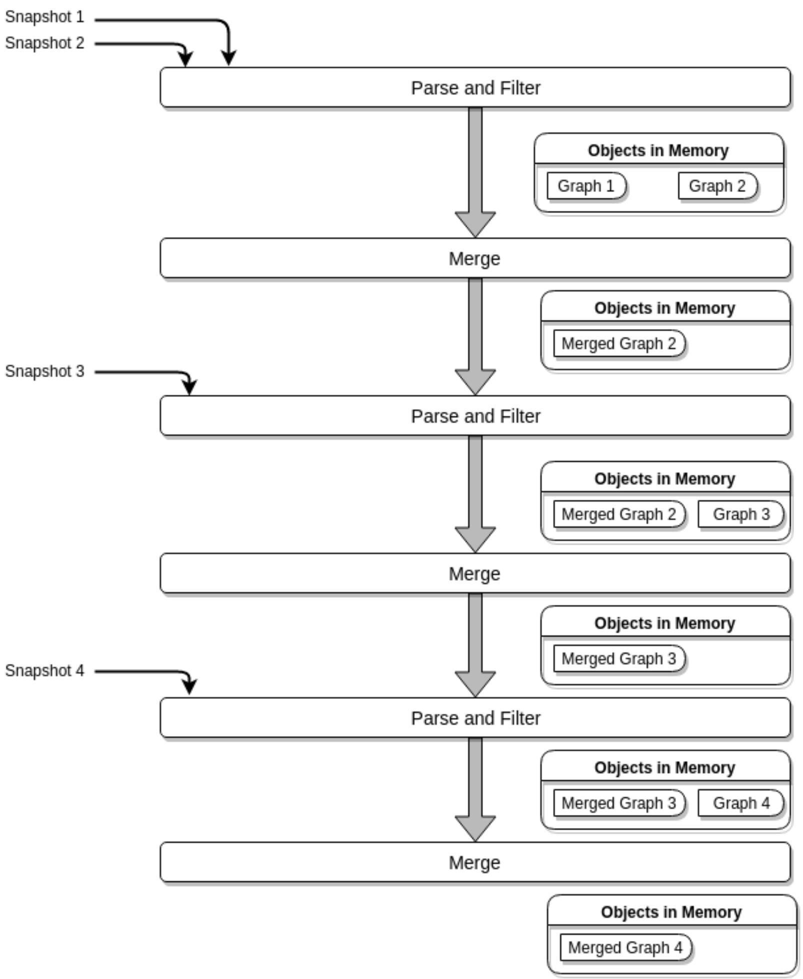


Figure 3.8: *Flow of merging 4 snapshots*

3.3 Summary

In this chapter are presented the modules of the tool as well as its behaviours. The snapshots of the original and suspect program follows almost the same path across the modules, with the exception of the Birthmark Extractor module. This module is only used for the original program in order to extract its birthmark. The Parser and Filter modules have a lower level of difficulty than the others (Merger, Birthmark Extractor and Detector). The higher difficulty is related with the graph operations that are made in those modules, which makes them harder and more complex.

Overall, the tools modules are presented in Section 3.1 and its flow in Section 3.2. More specific details about the modules are given in Chapter 4 that allow a better understanding of the system.

Chapter 4

Implementation

The entire system was developed in JavaScript. This chapter describes with more detail the modules developed and gives some particular insights of the implementation. The structure of the snapshot file is explained in Section 4.2. The module that parses the file is detailed in Section 4.3. The merger module (Section 4.5), filter module (Section 4.4), birthmark extractor module (Section 4.6) and detector module (Section 4.7) are also mentioned in this chapter.

4.1 Technologies Used

The system was developed using the JavaScript language and the platform Node JS. Node JS is built over the Google Chrome JavaScript engine (V8) [4]. All reads and writes to files use the File System API of Node JS[1]. In order to manage the dependencies of external libraries, npm[2] was used. All the developed code was managed and stored using a Git repository. Memoize[3], heapsnapshot-parser[4] and Babel[5] are some examples of external libs used. Memoize basically is used when we want to save the result of a function for a certain input. It is useful when an expensive function is called multiple times during the execution of the program. The lib "heapsnapshot-parser" was used to parse the heap snapshot generated by V8.

The JavaScript language has a standard specification that is called ECMAScript [11]. Some features of ES6 (2015) were used on the development, however some environments do not support it yet, therefore Babel was used to transpile the code from ES6 to ES5.

[1] https://nodejs.org/en/

[2] https://www.npmjs.com/

[3] https://lodash.com/docs#memoize

[4] https://github.com/jwalton/node-heapsnapshot-parser

[5] https://babeljs.io/

4.2 Heap Structure

The heap of a JavaScript program can be extracted by taking a snapshot of it. The snapshot is generated by the V8 engine [4] and the heap is represented by a graph whose nodes represent Objects and edges represent the connections between them. A text file with the information of the graph, where nodes are represented by 6 numbers and the edges by 3, can be saved. The initial part of the snapshot explains how the snapshot structure is formed and the meaning of each field. It can be seen as an header of the snapshot that contains relevant information for understanding it.

```
{"snapshot": {
    "meta": { "node_fields": [ "type", "name", "id", "self_size", "edge_count",
                   "trace_node_id"],
        "node_types": [
             [ "hidden", "array", "string", "object", "code", "closure", "regexp",
              "number", "native", "synthetic", "concatenated string", "sliced
                   string" ],
             "string", "number", "number", "number", "number", "number"
            ],
```

Listing 4.1: Partial snapshot example

In the beginning of the header (Listing 4.1) the fields of nodes are declared. A node is represented by a sequence of 6 numbers: type (id of the type), name (id of the string that represents the name), id (node id), self_size (object size in bytes), edge_count (number of edges in and out) and trace node id. A node can also be of 12 different types:

- **Hidden**: Hidden node, should be filtered.
- **Array**: An array of elements.
- **String**: A string.
- **Object**: A JavaScript object (excluding strings and arrays).
- **Code**: Compiled Code.
- **Closure**: Function closure.
- **Regexp**: Regular Expressions.
- **Number**: Number stored in the heap.
- **Native**: Native object.
- **Synthetic**: Objects generated to group snapshot items together.
- **Concatenated String**: Pair of pointers to strings.

- **Sliced String**: A fragment of another string.

After the information about the nodes, we have the same about edges (Listing 4.2).

```
"edge_fields": [ "type", "name_or_index", "to_node"],
"edge_types": [
     [ "context", "element", "property", "internal", "hidden", "shortcut", "weak"
         ],
      "string_or_number", "node"
],
```

Listing 4.2: Partial snapshot example

Edges are connections between objects and are represented by 3 fields: type (id of the type), name_or_index (id to the string that represents the name) and to_node (id of the node it points to). An edge can be of 7 different types:

- **Context**: Variable from a function context.
- **Element**: An element of an array.
- **Property**: A named object property.
- **Internal**: A link that cannot be accessed from JavaScript, should be filtered.
- **Hidden**: A link used to calculate sizes, should be filtered.
- **Shortcut**: A link that must not be followed during size calculation.
- **Weak**: A weak reference.

The types are classified with an id respectively (context - 0, ... , Weak - 6) After the information about the fields of the edges, some details about trace functions are given.

```
"trace_function_info_fields": [ "function_id","name","script_name","script_id","
    line","column"],
"trace_node_fields": ["id","function_info_index","count","size","children"],
   "sample_fields": ["timestamp_us","last_assigned_id"] },
"node_count": 184533,
"edge_count": 818340,
"trace_function_count": 0
},
```

Listing 4.3: Partial snapshot example

None of the snapshots taken during the experiments contained trace functions, therefore this fields are considered to be irrelevant for the problem context. Also in the header are presented counters with the number of nodes, edges and trace functions. In Listing 4.3, the heap had 184,533 nodes and 818,340 edges. The header ends with those counters and then the actual body of the snapshot starts. The body contains the information of the actual content of the heap and its structure is presented in Listing 4.4.

```
 "nodes": [
     9,1,1,0,15,0
  ,9,2,3,0,16,0
  ,9,3,5,0,1528,0
  ,9,4,7,0,326,0
  ,9,5,9,0,317,0
    ....
  ],
 "edges": [
   1,1,6
  ,5,1483,936
  ,5,1484,2838
  ,5,1485,4668
    ....
  ],
 "trace_function_infos": [],
 "trace_tree": [],
 "samples": [],
 "strings": [
  "<dummy>",
    "",
    "(GC roots)",
    "get valueType",
    "set valueType",
    ....
 ]
}
```

Listing 4.4: Partial snapshot example

This part of the snapshot contains the nodes, edges and strings. Nodes are represented by 6 numbers and edges by 3. Strings are referenced (on edge name and node name) by its position on array. For example:

The third node (9,3,5,0,1528,0) is a node of type "synthetic" (type 9) with id 5, name "(GC roots)" (position 1 on the array of strings), size 0 and 1,528 edges.

4.3 Parser

In order to parse the snapshot file and create a graph, a lib called "heapsnapshot-parser" was used. The output from that parse was a JavaScript Object containing:

- **nodes** - an array of Node objects, one for every object found in the heap snapshot.
- **nodesById** - a hash of Node objects, indexed by their ID.
- **edges** - an array of Edge objects, one for every edge found in the heap snapshot.

Node objects structure:

- **type** - (string) The type of the object.
- **name** - (string) The name of the object.
- **id** - (integer) A unique numeric ID for the object.
- **self_size** - (integer) Size of the object in bytes, not including any referenced objects.
- **trace_node_id**
- **references** - An array of Edge objects for nodes which this node references.
- **referrers** - An array of Edge objects for nodes which reference this object.

Edge objects structure:

- **type** - (string) The type of the edge.
- **name_or_index** - (string) The name (or index, for an array element) for this edge.
- **fromNode** - Node object from where this edge points
- **toNode** - Node object to where the edge points

4.4 Filtering the Heap

Usually, and depending on the program, a heap graph size is really big. A complex application can have hundreds of thousands of Nodes and a few millions edges. An example of a big graph can be the graph of a heap snapshot taken to the Facebook timeline, it had 816,625 nodes and 3,995,461 edges. Because the computation of this kind of graphs can be hard, they are filtered in order to reduce its size. Some nodes are removed because they are not relevant in this context. The tool visits all the nodes and the ones belonging to the "black list" are removed, as well as its links to other nodes. The blacklist of the nodes and edges contains:

- Nodes of type *Hidden*.

- Nodes of type *Array*, because the actual arrays are represented by a node of type *Object* with name "Array" [18].

- Nodes of type *String* and *Code*, because they have no references coming out of them.

- Edges of type *Context*, because its a connection inside a function context allowing the access of variables by its closure [18].

- Edges of type *Internal*, because they are created by the virtual machine and are not accessible by JavaScript code.

- Nodes related to the DOM of the page, because they are not relevant in this context. For example, a thief can copy all the code and only change its interface therefore the DOM nodes are not useful on detecting the theft.

4.5 Graph merger

This module was created to merge two graphs and it is needed to merge the snapshots together. The nodes id (assigned by V8 engine) remains the same across the snapshots and that's how we check how many were added and deleted. The algorithm used to merge 2 graphs is represented in Listing 4.5, and it is called to merge two consecutive snapshots.

```
 1 Inputs: snapshot1 (older snapshot) ; snapshot2 (newer snapshot)
 2
 3 delNodes <- get deleted nodes from 1 to 2
 4
 5 for all delNodes {
 6  copyNode <-make a copy of delNode
 7   add copyNode to snapshot2
 8
 9   for all parents of delNode {
10    if (parentNode exists in snapshot2) -> save it as parentNodeSnapshot2 {
11      add reference from parentNodeSnapshot2 to copyNode
12       add referrer from copyNode to parentNodeSnapshot2
13       }
14   }
15
16   for all children of delNode {
17    if (childrenNode exists in snapshot2) -> save it as childrenNodeSnapshot2 {
18      add referrer from childrenNodeSnapshot2 to copyNode
19       add reference from copyNode to childrenNodeSnapshot2
20     }
21   }
22 }
```

Listing 4.5: Algorithm to merge two graphs

The first step is to get the nodes that were present in graph 1 but are not in graph 2 (line 3). Those nodes, deleted by the garbage collector, are added to the graph 2 (line 7). After adding the node, the connections between its parents and children must be done. It is checked if the parents in old graph (snapshot 1) exists in the new graph (line 10), because they could also been deleted. If the parent exists, the connections between node and parent are made (lines 11-12). The same process is repeated to the children (lines 16-21). When the algorithm ends, the snapshot 2 contains a graph that is the combination of the two graphs.

4.6 Birthmark extractor

This module receives as input the filtered and merged graph, so it can extract the birthmark of the program. As said in previous chapter this module can have two versions, one selects the largest subgraph and the other searches for a frequent subgraph. The first version goes through all the "windows" (objects underneath the root node) objects and obtains the size of the graph underneath them in order to select the biggest (Listing 4.6). The method to get the size (line 6), is a recursive algorithm that performs a depth first search. In this algorithm we can limit the level of search in the graph in order to accelerate it. The selection of the biggest graph can be made in two different ways: the largest number of nodes or the largest size (in bytes) of the graph. By changing this parameters we obtain different outputs as the birthmark of the program.

```
 1 inputs: graph , limitLevel (0 if has no limit) , sizeInBytes (false - uses number of nodes ; true - uses byte size)
 2 
 3 birthmarkGraph - stores the birthmark graph
 4 get window object nodes
 5 for all window object Nodes {
 6   getSizeOfGraph(windowNode,limitLevel,sizeInBytes)
 7     update birthmarkGraph if size is higher than current
 8 }
 9 
10 return birthmarkGraph;
```

Listing 4.6: Birthmark extractor v1

The second version selects the largest subgraph that appears more than once in the graph. There are some algorithms to search for a frequent subgraph ([25, 8]), and the implementation follow a similar approach of that used in GRAMI algorithm ([9]). One difference is that GRAMI uses edges to extend subgraphs and this algorithm uses nodes. The idea consists in extending nodes forming a subgraph, check if it is frequent and keep extending it. In the end we have the largest subgraph that is frequent, as described in Listing 4.7.

```
input: graph

get frequent Nodes
get frequent Edges
birthmarkGraph - stores the largest graph that is frequent
for all frequent nodes as fNode {
  graph = make a new graph with only fNode
    graphExtended = graphExtension(graph,frequentNodes,frequentEdges, [fNode]);
    if graphExtended is bigger than birthmarkGraph {
      birthmarkGraph = graphExtended;
        }
   remove fNode from frequentNodes
13 }
14
15 return birthmarkGraph
```

Listing 4.7: Birthmark extractor v2

The first step is to get the frequent nodes and frequent edges (lines 3-4). A node is frequent if there is another node in the graph with the same type and with the same children types. An edge is frequent if the graph has another edge with the same name, type and the nodes it connects have the same type. This is done because we will only try to extend with frequent nodes and frequent edges. If a node is not frequent, a subgraph with that same node can not be frequent. The next step is, for each frequent node get the maximum extension of it (lines 6-13). The algorithm that does the extension of a node (line 8) is presented in Listing 4.8. When all frequent nodes have been extended, it is chosen the biggest extension as a birthmark.

```
 1 inputs: graph, fEdges (frequent edges), fNodes (frequent Nodes), leafNodes
 2
 3 graphExtended = copy of graph
 4 newLeafNodes = [];
 5 for all leaf nodes as lNode{
 6   for all children of lNode {
 7       edge - edge that connects lNode to its child
 8         if child belongs to fNodes and edge belongs to fEdges {
 9           extend graphExtended with child
10             add child to newLeafNodes
11         }
12      }
13 }
14
15 if(length of newLeafNodes > 0){
16     if(graphExtended is frequent) {
17           graphExtension(graphExtended,fEdges,fNodes,newLeafNodes)
18         }
19 }
20
21 return graph;
```

Listing 4.8: graphExtension

The "graphExtension" algorithm receives a graph and tries to extend it until the graph is not frequent. It returns the biggest extension that is frequent, in order to do that, each leaf node of the graph will be extended with its children (lines 5-13). Only the children that can be reached using frequent nodes and frequent edges will be extended (lines 8-11). If no extension could be done to the graph, the algorithm returns the current graph. If the graph was extended, it will be checked if it appears more than once in the full graph (line 16). The algorithm used to check if a subgraph is frequent, is similar to the one used in the Detector module (Section 4.7). Supposing that the extension is frequent, the algorithm will call it self recursively to further extend the graph. This implementation extends a node with all its children, therefore it is an extension by level. This is a modification of the original version implemented, which extended a node with only one child at a time.

The original version of the implementation generated all possible combinations of subgraphs and therefore was very heavy computationally. An example of a graph with 61,829 nodes and 84,410 edges resulted in an execution of 3 hours(after 3 hours the algorithm was still running). Therefore this algorithm was adapted to extend the node with all its children instead of only one at a time. An example of the differences between the original algorithm and the modified one is presented in Figure 4.1.

The modified algorithm does not guarantee the optimum solution for the problem of subgraph mining. However, in this context the goal of subgraph mining is to find a big object (represented by a subgraph) that is repeated in the heap, therefore it is likely that it has all the same children in both instantiations. In this case, the modified algorithm can also find the biggest frequent subgraph.

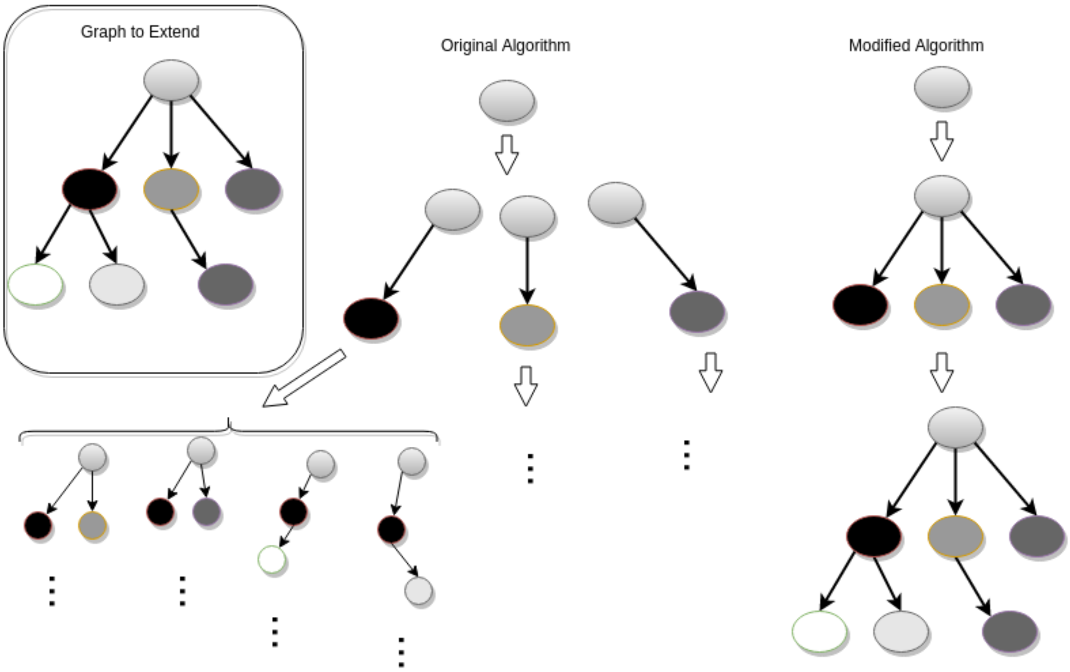


Figure 4.1: *Algorithm "graphExtension" original vs modified version (colours represent different node types)*

With the modified version, the same example took 64.501 seconds to execute, making it viable to use compared with the original version.

4.7 Detector

This is the module that checks the similarity between the programs. This module tries to find the birthmark of the original program (a graph) inside the graph of the suspect program. The problem it solves is similar to subgraph isomorphism [8, 25], which is basically trying to find a subgraph inside a large graph. The algorithm used is described in Listing 4.9, and it outputs a similarity (%) between the programs.

```
1  inputs: Birthmark of original program ; Graph of suspect program
2
3  candidates = find similar nodes to rootOfBirthmark inside suspect graph
4  bestMatch = 0; // represents the number of similar nodes between birthmark graph
       and the best candidate graph
5  for all candidates {
6    numberOfMatchNodes = getMatchNodes(rootOfBirthmark, candidate);
7      update bestMatch if numberOfMatchNodes is higher
8  }
9
10 return bestMatch/number of nodes of birthmark graph
```

Listing 4.9: Algorithm of Detector module

The algorithm limits the search by considering only the nodes equals to root of birthmark (line 3). These candidates are considered to be equal to the root node if they have the same type, size, and also the same children. In the case of the children, there is a minimum percentage *m* that represents the amount of children that are similar. If *m* is set to 70% and the original node has 10 children, the suspect node must have at least 7 children that are similar to the originals. This value can be changed in order to vary the number of candidates. If *m* increases, the number of candidates decreases. On the other hand if *m* decreases, the number of candidates increases. After having all candidates, it will be chosen the best one (lines 5-8). The function called in line 6, is presented in Listing 4.10. Basically, it receives two root nodes (the original and the suspect) and through a depth first search tries to match the children of the original with the ones from the suspect. It returns the number of nodes it was able to match.

```
inputs: nodeOriginal ; nodeSuspect
matchedNodes = 0;
if nodes are similar{
  matchedNodes++
  get matching children
    for each children matched{
    matchedNodes += getMatchNodes(childrenOriginal,childrenSuspect)
    }
}

return matchedNodes;
```

Listing 4.10: Algorithm for matching two graphs

This algorithm is called on Listing 4.9 with two root nodes. The first step is to check if the nodes are similar and therefore increment the number of "matchedNodes" (lines 3 and 4). After that, the children of the original node are matched with the children of the suspect node (line 5). For all the matched children, the algorithm calls itself recursively (line 7) to perform a depth first search and increment the number of "matchedNodes". When it ends, the value of "matchedNodes" is the number of nodes that were found in the suspect graph (example in Figure 4.2).

This algorithm has a vulnerability that consists in object injection. Figure 4.3 presents a theft that the tool would be unable to detect. The suspect graph is similar to the birthmark of the original program but with an extra object. The similarity is only 28% but is visible that the suspect program contains 100% of the birthmark from the original. Adding an extra dummy object is easy to accomplish, however inserting an object between two that were connected is very hard without changing the behaviour of the program. Therefore, despite existing such vulnerability it is hard for a thief to exploit it.

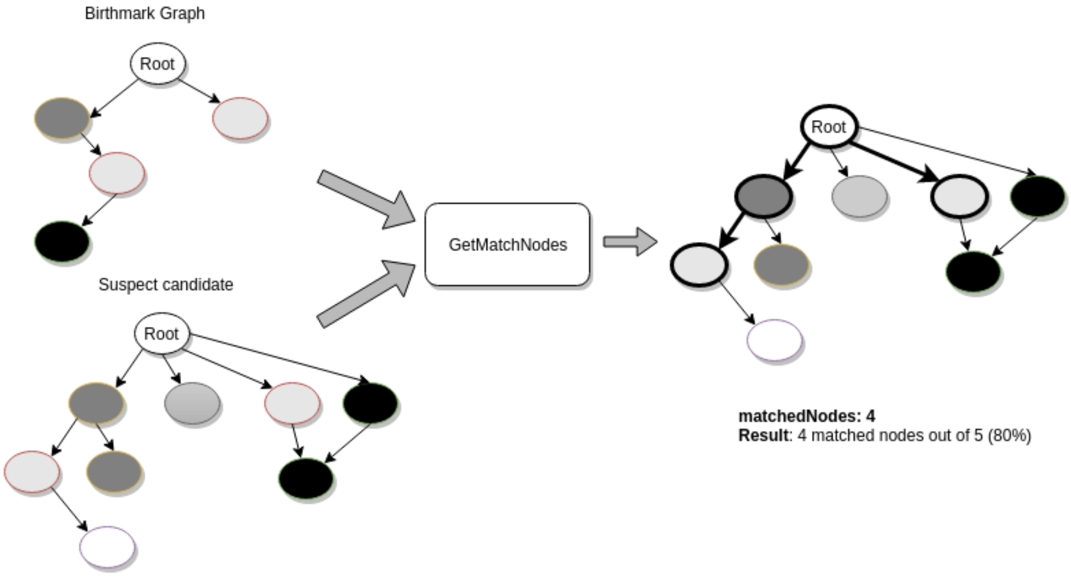


Figure 4.2: *Algorithm "getMatchNodes" output example (colours represent different node types)*

4.8 Summary

The implementation of all the modules and the necessary auxiliary functions resulted in close 1,800 lines of code divided across 10 files. These numbers do not include the libraries used. Overall, the implementation value is not on the size of the code, but on the difficulties and complexity of it. The operations with big graphs are costly and during the development of the tool, a high number of small optimizations were required to make it work better. A good example is the use of memoization to save the value of costly function that will be reused later during the program.

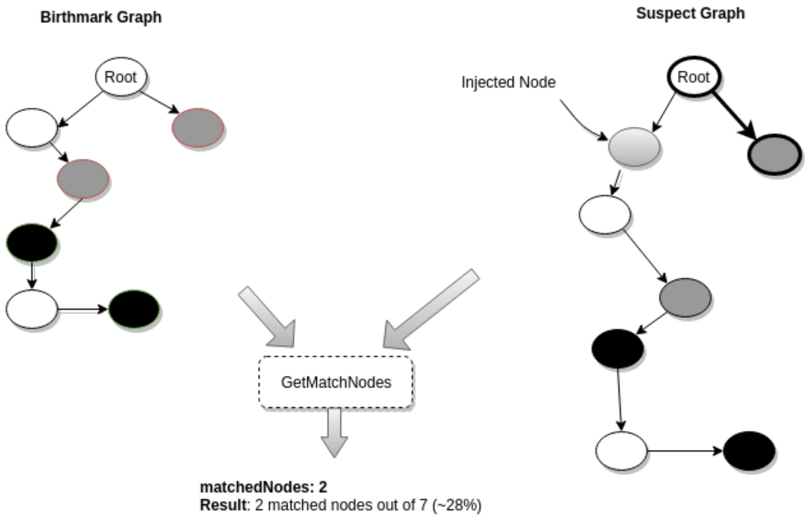


Figure 4.3: *Injection attack example*
(colours represent different node types)

Some functions during the graph mining can be called more than once for the same input and therefore this improved significantly the performance in some cases.

Chapter 5

Experiments and results

This chapter presents some preliminary experiments done to test the quality of the tool developed. Those experiments test different situations and its analysis allows some conclusions about the birthmarking technique implemented. This chapter also presents some discussion about the results of the tests done.

Because the tool need to be fed with heap snapshots, they were obtained using Google Chrome dev tools[1]. The programs were opened in Google Chrome and during the first minute of execution, 4 snapshots were taken (Figure 5.1).

The snapshots were taken with a browser in its "raw" state (without any extensions), if it has extensions active the results might be influenced by them. Because most extensions are basically JavaScript programs, they will be present in the snapshots taken and could lead to false positives. An improvement of the tool would be introducing a blacklist of extensions in order to exclude them from the snapshot. That addition can be made in the filter module, making it delete the nodes related to those extensions.

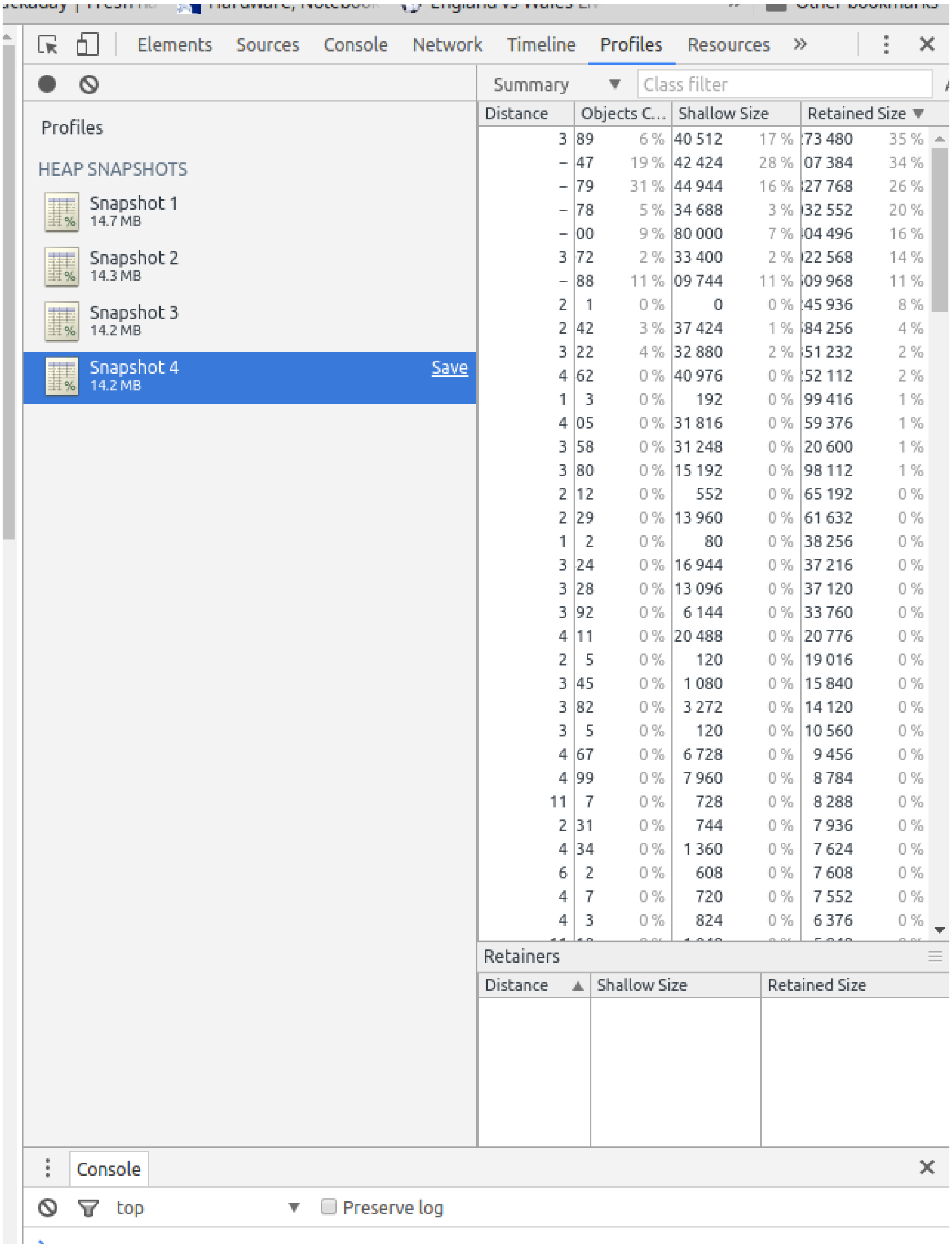
The tests were conducted using the 2 versions of the birthmark extractor module (version 1- largest subgraph version | version 2- frequent subgraph version). The results with version 2 did not corresponded to the expectations and therefore all the tables presented are related to the version1.

5.1 Programs and Websites used

Table 5.1 presents the multiple programs used to test the tool, each of them had 4 snapshots taken in order to feed the tool. Besides the name and description, the number of nodes and edges of each program are also displayed. Those numbers are related to the snapshots after they have been filtered and merged together.

The app "*backboneAlone*" was developed (by us) with the goal of capturing the birthmark of Backbone itself. It is a simple application that was written with some instantiations of views, routes and models.

[1] https://developer.chrome.com/devtools



Profiles

HEAP SNAPSHOTS

Snapshot 1 14.7 MB

Snapshot 2 14.3 MB

Snapshot 3 14.2 MB

Snapshot 4 14.2 MB Save

Distance	Objects C…		Shallow Size		Retained Size ▼	
3	89	6 %	40 512	17 %	73 480	35 %
–	47	19 %	42 424	28 %	07 384	34 %
–	79	31 %	44 944	16 %	27 768	26 %
–	78	5 %	34 688	3 %	32 552	20 %
–	00	9 %	80 000	7 %	04 496	16 %
3	72	2 %	33 400	2 %	22 568	14 %
–	88	11 %	09 744	11 %	09 968	11 %
2	1	0 %	0	0 %	45 936	8 %
2	42	3 %	37 424	1 %	84 256	4 %
3	22	4 %	32 880	2 %	51 232	2 %
4	62	0 %	40 976	0 %	52 112	2 %
1	3	0 %	192	0 %	99 416	1 %
4	05	0 %	31 816	0 %	59 376	1 %
3	58	0 %	31 248	0 %	20 600	1 %
3	80	0 %	15 192	0 %	98 112	1 %
2	12	0 %	552	0 %	65 192	0 %
2	29	0 %	13 960	0 %	61 632	0 %
1	2	0 %	80	0 %	38 256	0 %
3	24	0 %	16 944	0 %	37 216	0 %
3	28	0 %	13 096	0 %	37 120	0 %
3	92	0 %	6 144	0 %	33 760	0 %
4	11	0 %	20 488	0 %	20 776	0 %
2	5	0 %	120	0 %	19 016	0 %
3	45	0 %	1 080	0 %	15 840	0 %
3	82	0 %	3 272	0 %	14 120	0 %
3	5	0 %	120	0 %	10 560	0 %
4	67	0 %	6 728	0 %	9 456	0 %
4	99	0 %	7 960	0 %	8 784	0 %
11	7	0 %	728	0 %	8 288	0 %
2	31	0 %	744	0 %	7 936	0 %
4	34	0 %	1 360	0 %	7 624	0 %
6	2	0 %	608	0 %	7 608	0 %
4	7	0 %	720	0 %	7 552	0 %
4	3	0 %	824	0 %	6 376	0 %

Retainers

Distance ▲	Shallow Size	Retained Size

Figure 5.1: *Chrome Dev tools while taking snapshots*

All "*TODO*" programs are simple apps that simulate a TODO list and were downloaded from[2]. They allow to add tasks that can be removed, edited or marked as completed. Those apps have exactly the same appearance and behaviour, however they were all developed using a different JavaScript framework. A local server was created and they were run from it.

[2] http://todomvc.com/

All the other programs were accessed using the links shown in their description.

5.2 Testing apps with the exact same appearance and behaviour

The definition of credibility says that if two programs are independently written, their birthmark should be different. This experiment tests the credibility by using identical apps developed with different frameworks. The **goal** with this experiments is to check if two apps with the same appearance and behaviour are not reported as a false positive by the tool. The TODO apps were used with the exact same inputs in order to maximize the similarity between them. Table 5.2 shows the results of the experiments comparing all apps with each others. Each row of the table has the "original" app (*Snapshot1*), the "suspect" app (*Snapshot2*), birthmark size and the similarities(%) obtained. The percentage values are in the same order as the apps are in the table. The first percentage is related with the comparison with "*TODO-amperstand*", the second with "*TODO-angularjs*" and so on.

The tests with the version 2 generated false positives in all the cases. The result of the subgraph mining was a birthmark graph with around 50 nodes for all the apps and the similarity between them was always near 100%.

5.2.1 Analysis of results

The ideal results in the experiment presented in Table 5.2 would be: having one percentage equal to 100% and all the others close to 0%. The percentage equal to 100% representing the comparison of the program with itself and the others (close to 0%) representing the comparison with the other apps. All the tests were a success with the exception of "*vanillajs*". This case is a good example of what happens when there is not enough custom objects in the program that we use as birthmark. VanillaJS is basically pure JavaScript, therefore a lot of false positives (all of them) were obtained because the application was not complex enough to have a representative birthmark. The birthmark had only 101 nodes and did not represent the uniqueness of the program. Another interesting case to look is "*closure*", the percentages were quite high because the birthmark had only 239 nodes. If a birthmark is too small it could not be enough to represent the program, therefore will generate some false positives. This case did not generate false positives because the percentages were always bellow 46%, however they were higher than the other cases because of the small size of the birthmark.

5.3 Partial theft detection

The experiments with the goal of detecting partial theft are presented in the Table 5.3. The application "backboneAlone" had no content to show (no interface) in order to try to capture the

Table 5.1: Table with the snapshots used

Name	**Description**	**Merged and Filtered Graph**
amazon-es	Amazon from Spain. src: amazon.es	Nodes: 42587 Edges: 64312
amazon-fr	Amazon from France. src: amazon.fr	Nodes: 39809 Edges: 56648
backboneAlone	Empty app with backbone instances of views, models and routers.	Nodes:8525 Edges:11083
backboneApp1	App that uses backbone. src: https://earth.nullschool.net/	Nodes: 315205 Edges: 422933
backboneApp2	App that uses backbone. src: https://pt.foursquare.com/	Nodes: 115029 Edges: 67975
backboneApp3	App that uses backbone. src: https://www.documentcloud.org/public/search/	Nodes: 53564 Edges: 70855
backboneApp4	App that uses backbone. src: https://dashboard.stripe.com	Nodes: 87381 Edges: 133358
backboneApp5	App that uses backbone. src: http://planetica.pt/	Nodes: 28825 Edges: 39395
backboneApp6	App that uses backbone. src: http://mydietweb.com/	Nodes: 48417 Edges: 66726
TODO-angularjs	TODO app developed using angularjs	Nodes: 18217 Edges: 20457
TODO-amperstand	TODO app developed using amperstand	Nodes: 13231 Edges: 13701
TODO-amperstand-obfuscated	Obfuscated version	Nodes: 15365 Edges: 17337
TODO-backbone	TODO app developed using backbone	Nodes: 10351 Edges: 12073
TODO-backbone-obfuscated	Obfuscated version	Nodes: 11851 Edges: 14979
TODO-canjs	TODO app developed using canjs	Nodes: 16067 Edges: 16620
TODO-canjs-obfuscated	Obfuscated version	Nodes: 18139 Edges: 17768
TODO-closure	TODO app developed using closures	Nodes: 8499 Edges: 10455
TODO-closure-obfuscated	Obfuscated version	Nodes: 10715 Edges: 12245
TODO-jquery	TODO app developed using jquery	Nodes: 10023 Edges: 12231
TODO-jquery-obfuscated	Obfuscated version	Nodes: 11476 Edges: 14948
TODO-react	TODO app developed using react	Nodes: 22923 Edges: 32812
TODO-react-obfuscated	Obfuscated version	Nodes: 24236 Edges: 37288
TODO-vanillajs	TODO app developed using vanillajs	Nodes: 6687 Edges: 8084
TODO-vanillajs-obfuscated	Obfuscated version	Nodes: 6831 Edges: 8268
worten-product1	Webpage of a product in worten. src: https://www.worten.pt/inicio/landings/ o-mundo-esta-cada-vez-mais-worten/maquina-de-lavar-roupa-hotpoint-ariston-rsg-825-ja.htm	Nodes: 58409 Edges: 78384
worten-product2	Webpage of a product in worten. src: https://www.worten.pt/inicio/landings/ o-mundo-esta-cada-vez-mais-worten/portatil-mobilidade-14-asus-e403sa-wx0008t.html	Nodes: 61829 Edges: 84410

Table 5.2: Similar apps developed independently experiments

Snapshot1	**Snapshot2**	**Birthmark Size (n^{o} nodes)**	**Similarity (%)**
TODO-amperstand	TODO-*	1217	100% ; 13.7% ; 9.2% ; 8.3% ; 9.2% ; 5.9% ; 9.0% ; 4.4%
TODO-angularjs	TODO-*	2653	3.3% ; 100% ; 3.2% ; 3.4% ; 3.9% ; 3.2 % ; 3.6 % , 3.5 %
TODO-backbone	TODO-*	943	13.9% ; 12.6% ; 100% ; 11.8% ; 15.1% ; 10.1% ; 11.1% ; 9.0%
TODO-canjs	TODO-*	1216	17.1% ; 17.1% ; 17.1% ; 100% ; 17.1% ; 43% ; 17.1% ; 17.1%
TODO-closure	TODO-*	239	34.7% ; 35.9% ; 45,1% ; 36.8% ; 100% ; 33.1% ; 41.0% ; 25.5%
TODO-jquery	TODO-*	1003	11.2% ; 11.2% ; 11.2% ; 11.2% ; 11.2% ; 100% ; 11.2% ; 11.6%
TODO-react	TODO-*	617	17.8% ; 22.8% ; 48.9% ; 17.8% ; 17.8% ; 17.8% ; 100% ; 17.8%
TODO-vanillajs	TODO-*	101	99% ; 97% ; 99% ; 100% ; 99% ; 99% ; 97% ; 100%

Table 5.3: Partial theft experiments

Snapshot1	**Snapshot2**	**Birthmark Size (n^{o} nodes)**	**Similarity (%)**
backboneAlone	amazon-es	333	33.0%
backboneAlone	amazon-fr	333	30.9%
backboneAlone	backboneApp1	333	91.1%
backboneAlone	backboneApp2	333	75.3%
backboneAlone	backboneApp3	333	100%
backboneAlone	backboneApp4	333	78.0%
backboneAlone	backboneApp5	333	86.5%
backboneAlone	backboneApp6	333	86.5%
backboneAlone	TODO-amperstand	333	83.5%
backboneAlone	TODO-angularjs	333	26.4%
backboneAlone	TODO-backbone	333	100%
backboneAlone	TODO-backbone-obfuscated	333	100%
backboneAlone	TODO-canjs	333	30.9%
backboneAlone	TODO-closure	333	23.4%
backboneAlone	TODO-emberjs	333	30.9%
backboneAlone	TODO-jquery	333	30.9%
backboneAlone	TODO-react	333	23.4%
backboneAlone	TODO-vanillajs	333	23.4%
backboneAlone	worten-product1	333	75.7%
backboneAlone	worten-product2	333	75.7%

backbone framework on its purest form. The apps "*backboneApp1*" to "*backboneApp6*" were chosen because they are known to use backbone [3] [4]. The main **goal** with this set of experiments is to check if the framework is used in an app. This kind of tests can be used when someone wants to check if a specific library is being used unduly.

A real world scenario is: an attacker can steel a code and then develop on top of it, thinking that it will not be found because the stolen code is only a part of his program.

The tests with version 2 generated false positives in all the cases. The birthmark graph extracted had 48 nodes and the similarity was always 100%.

5.3.1 Analysis of results

The ideal results in the experiment presented in Table 5.3 would be: all cases with known use of backbone framework having percentages close to 100% and the others having low percentages. The comparison with the "*amazon-es*" and "*amazon-fr*" indicate a low percentage (below 33%) of similarity which indicates that Amazon do not use Backbone. The comparison with the "worten-product1" and "worten-product2" indicate a similarity of 75.7% in both cases, which strongly suggests that they use Backbone on their website. The tests with the "*backboneApps*" were a success because all of them had high similarity percentages. Two of them had values of 78% and 75.3% and all the others had similarities above 86%, including one value of 100%. The tests with "*TODO*" apps were also successful because only the apps with known use of Backbone had high similarity, all the others had values under 31%. The TODO app developed using Backbone had a similarity of 100% as well as the obfuscated version of it. The unusual case was the "*TODO-amperstand*" with a value of 83.5%. After a small research was concluded that the high similarity is due to the fact that Amperstand is very similar to Backbone in many aspects [5], therefore such results were expected.

5.4 Testing resilience against obfuscation

The definition of resilience says that if a program is obtained through another, only applying semantic preserving transformations, the birthmark should remain the same. Table 5.4 shows the experiments with the **goal** of testing the resilience of the birthmarking technique used. All the TODO-apps were obfuscated using JScrambler [6] with the strongest obfuscation options enabled. After being obfuscated, the original apps were compared with obfuscated version of themselves. JScrambler has a vast amount of transformations [7] and it is one of the most complete obfuscation tools on the market. The detection of the similarity with this kind of obfuscations is a good

[3] http://backbonejs.org/#examples

[4] https://index.woorank.com/en/reviews?country=pt&technology=backbonejs

[5] https://ampersandjs.com/learn/transitioning-from-backbone/#transitioning-from-backbone

[6] https://jscrambler.com/pt/

[7] https://jscrambler.com/pt/how-it-works

Table 5.4: Resistance against obfuscation experiments

Snapshot1	**Snapshot2**	**Birthmark Size (n^o nodes)**	**Similarity (%)**
TODO-amperstand	TODO-amperstand-obfuscated	1217	96.7%
TODO-backbone	TODO-backbone-obfuscated	943	98.6%
TODO-canjs	TODO-canjs-obfuscated	1216	90.1%
TODO-closure	TODO-closure-obfuscated	239	69.8%
TODO-jquery	TODO-jquery-obfuscated	1003	100%
TODO-react	TODO-react-obfuscated	617	96.7%
TODO-vanillajs	TODO-vanillajs-obfuscated	101	100%

indicator of the birthmark resilience. A real world scenario for this kind of tests is: an attacker steels a program and then uses obfuscation to disguise the code.

Tests with version 2 were not done because on Section 5.2 and 5.3 was concluded that they would produce false positives and therefore the results would have been 100% in every cases.

5.4.1 Analysis of results

The ideal results in the experiment presented in Table 5.4 would be: all of the cases having percentages close to 100%. These tests were a success and the tool was able to detect all the cases, with the exception of "*closure*". The "*TODO-closure*" only presented a 69.8% similarity with it obfuscated version, however all the others presented values above 90%. This shows that the birthmark has good resilience against transformations.

5.5 Scalability

In order to check scalability a few preliminary tests were done comparing the time of execution of the two versions of birthmarks. The machine used to run the tool has a core i7-4720HQ 2.4GHz, 12 gb of ram and a SSD storage of 256 gb. The goal of this tests is to understand how the tool behaves for different program sizes to analyse. Table 5.5 presents the preliminary tests done and Figure 5.2 presents a chart representation of those tests. The test cases were selected because of its size differences: "TODO-vanillajs" to represent a very small graph, "TODO-react" to represent a medium graph and "worten-product" to represent a big graph.

Table 5.5: Scalability tests

Snapshot1	**Snapshot2**	**Size of Graph (n^o nodes)**	**Largest node version (ms)**	**Frequent subgraph version (ms)**
worten-product1	worten-product2	120238 (58409+61829)	23486	87427
TODO-react	TODO-react-obfuscated	47159 (22923+24236)	10001	22237
TODO-vanillajs	TODO-vanillajs-obfuscated	13518 (6687+6831)	3125	3683

By analysing the experiments, the version 1 (largest node version) appears to have a linear evolution and the version 2 (frequent subgraph version) appears to have exponential evolution.

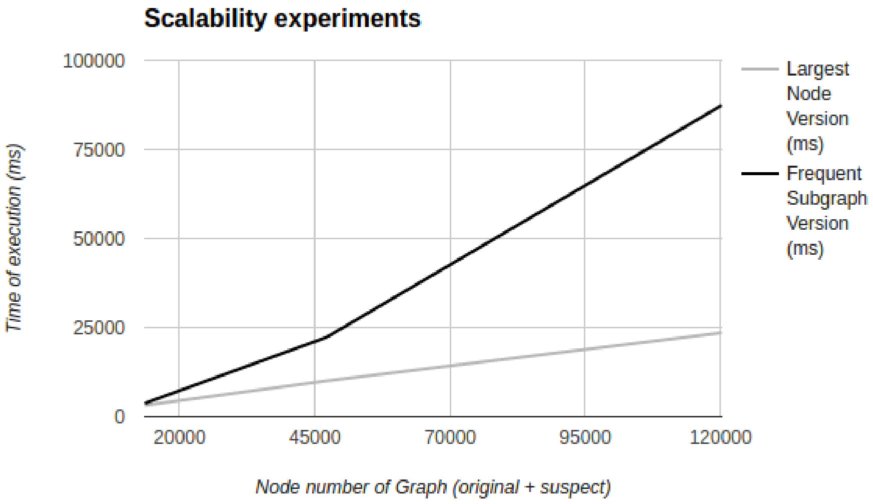


Figure 5.2: *Scalability test chart*

This experiment uses few test cases (only 3) and therefore a larger set of tests would help making a better scalability analysis.

5.6 Summary

In this chapter were presented some experiments that cover a large range of possible situations. The programs and the experiments were chosen to simulate a real usage and the results obtained by the tool were compared with the ideal ones expected. In all three scenarios tested (Sections 5.2, 5.3 and 5.4) the results were good and the exceptions were analysed and explained.

Chapter 6

Conclusions

This thesis consisted in the research of the current birthmarking techniques and the analysis of them. Most of the techniques were developed targeting other languages and therefore they are not adaptable to JavaScript. The analysis revealed that the most suitable technique for the problem was the one that used the heap to extract the birthmark. The tool was developed using the most recent proposed techniques.

One of the main difficulties was to understand the heap during the execution of a program. The JavaScript engine V8 allows to export snapshots of the heap, however the documentation of the heap structure is very scarce. A good understanding of the heap is necessary in order to filter the nodes that are not representative of the program itself. As we increase the nodes filtered, the graph to analyse becomes smaller and therefore the tool runs faster. After reading the available documentation and looking at the source code of V8, the knowledge of the heap structure was better but still not 100% clear. A better documentation about it would have been helpful during the development of the tool.

The implementation of this tool also revealed some difficulties because it was written in JavaScrpit. For this type of operations with graphs, a natively object oriented language is preferred. Finding bugs in the software and figuring out them was difficult because of the complex graphs that the tool is dealing with. A graphical visualization tool for the graph, while debugging, would have been very helpful in the process. Some applications were tested, namely Gephi[1] [2], to try to visualize the graph (by using the standard notation ".gml" to write the graph to a file) but because the graphs were too big, it was hard to get a good perception of them.

With this implementation of the tool, the results with subgraph mining claimed by S. Patel and T. Pattewar [18] were not achieved. Their proposal used Agglomerative Clustering and FSM with the goal of reducing the size of the birthmark. The smallest possible birthmark is useful because it facilitate the detection phase due to the graph, we are trying to find, being smaller. However, with the developed tool was concluded that the process of mining a graph to extract a smaller birthmark is more costly than having a bigger birthmark. As webapps are getting more complex, the mining

[1] https://gephi.org/

time will increase even further. The option of selecting the largest subgraph as birthmark will execute faster overall despite having a detection phase that takes longer.

Despite the tool being successful using the largest node approach, it would have been interesting comparing that method with the one that uses subgraph mining. The implementation of subgraph mining was not successful on the tool and the results did not correspond to the expectations. Due to time restrictions, only a small research was done about methods of subgraph mining. Because there are a lot of informations that can be represented by huge graphs (social media connections, city roads, traffic etc), this is a growing field with a lot of literature to analyse. If more time was available, a deeper research on subgraph mining algorithms and methods would have been done in order to improve the quality of the tool.

The amount of tests and experiments could also be higher in order to stress the tool further, however, the performed tests covered several situations and are a good indicator about the tool accuracy.

The overall thesis was successful because the built tool accomplishes the goal of detecting theft of JavaScript programs. The goal of using a birthmarking technique with high credibility and resilience was also accomplished.

6.1 Future Work

Despite having good results, the tool performance could be improved by the use of a database to store and process graphs. An example would be the use of *Neo4j* [2], a graph database that allows the tool to be scalable and may increase its performance.

Another improvement that can be done relies on the detection phase. The current implementation is subjective to an attack consisting in object injection (explained in Section 4.7). An algorithm capable of resisting to this attack would be a good improvement on the current solution. This modification would increase the resilience of the birthmarking technique against transformations.

Despite providing good results, the current selection of birthmark is quite simple (selecting the largest subgraph or the largest frequent subgraph). The heap is a very big graph and a deeper investigation of its behaviour could reveal a new way of selecting the birthmark of the program. An investigation of the behaviour of the JavaScript engine V8 and the way it manages memory could be done as future work. Another interesting possibility is the detection of partial theft of the original program. Current systems can detect if the full original program is used in another system, even if it is only a part of the suspect program (Section 5.3). However if only a portion of the original program is used, the theft is not detected. A technique that allows this detection would also be interesting as future work.

[2] https://neo4j.com/

Bibliography

[1] Yameng Bai et al. “Dynamic K-gram based software birthmark”. In: *Proceedings of the Australian Software Engineering Conference, ASWEC* (2008), pp. 644–649. ISSN: 1530-0803. DOI: 10.1109/ASWEC.2008.4483257. URL: http://ieeexplore.ieee.org/xpls/abs%7B%5C_%7Dall.jsp?arnumber=4483257.

[2] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. “Gephi: An Open Source Software for Exploring and Manipulating Networks”. In: *Third International AAAI Conference on Weblogs and Social Media* (2009), pp. 361–362. ISSN: 14753898. DOI: 10.1136/qshc.2004.010033. URL: http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154%7B%5C%7Dnpapers2://publication/uuid/CCEBC82E-0D18-4FFC-91EC-6E4A7F1A1972.

[3] P P F Chan, L C K Hui, and S M Yiu. “JSBiRTH: Dynamic JavaScript Birthmark Based on the Run-Time Heap”. In: *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual* (2011), pp. 407–412. ISSN: 0730-3157. DOI: 10.1109/COMPSAC.2011.60. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=%7B%5C&%7Darnumber=6032372.

[4] *Chrome V8 Google Developers*. URL: https://developers.google.com/v8/.

[5] Christian Collberg and C. Thomborson. “Software watermarking: Models and dynamic embeddings”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1999, pp. 311–324. ISBN: 1581130953. DOI: 10.1145/292540.292569. URL: http://dl.acm.org/citation.cfm?id=292569.

[6] C. Collberg et al. “Dynamic path-based software watermarking”. In: *PLDI ’04 Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. Vol. 39. 6. 2004, p. 107. ISBN: 1581138075. DOI: 10.1145/996893.996856. URL: http://dl.acm.org/citation.cfm?id=996856.

[7] Jay Conrod. *A tour of V8: Garbage Collection*. 2013. URL: http://www.jayconrod.com/posts/55/a-tour-of-v8-garbage-collection.

[8] L.P. Cordella et al. “A (sub)graph isomorphism algorithm for matching large graphs”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.10 (Oct. 2004), pp. 1367–1372. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2004.75. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1323804.

[9] Mohammed Elseidy, Ehab Abdelhamid, and Spiros Skiadopoulos. "GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph". In: *Proceedings of the VLDB Endowment* 7.7 (2014), pp. 517–528. ISSN: 21508097. DOI: 10.14778/2732286.2732289.

[10] *Identifying Hierarchical Structure in Sequences*. URL: http://www.sequitur.info/jair.

[11] Ecma International. *ECMA-262 ECMAScript 6th Edition Language Specification*. Tech. rep. 2015, pp. 1–566. URL: http://www.ecma-international.org/ecma-262/6.0/.

[12] Tim Lindholm et al. "The Java® Virtual Machine Specification". In: *Managing* (2014), pp. 1–626. URL: http://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf.

[13] Ginger Myles and Christian Collberg. "Detecting Software Theft via Whole Program Path Birthmarks". In: *Proc. Information Security 7th International Conference, ISC 2004* 3225 (2004), pp. 404–415. ISSN: 03029743. DOI: 10.1007/978-3-540-30144-8{_}34. URL: http://link.springer.com/chapter/10.1007/978-3-540-30144-8%7B%5C_%7D34.

[14] Ginger Myles and Christian Collberg. "K-gram based software birthmarks". In: *Proceedings of the 2005 ACM symposium on Applied computing - SAC '05* (2005), p. 314. DOI: 10.1145/1066677.1066753. URL: http://portal.acm.org/citation.cfm?doid=1066677.1066753.

[15] Shah Nazir et al. "A Novel Rules Based Approach for Estimating Software Birthmark". In: 2015 (2015).

[16] *New Report Confirms Startup Activity Increasing – After Years Of Decline - Forbes*. URL: http://www.forbes.com/sites/mariannehudson/2015/06/22/new-report-confirms-startup-activity-increasing-after-years-of-decline/%7B%5C#%7D374afda149c5.

[17] *Ofuscação de Código @revistabw*. URL: http://www.revistabw.com.br/revistabw/ofuscacao-de-codigo/.

[18] Swati J. Patel and Tareek M. Pattewar. "A Survey on Software Birthmark based Theft Detection of JavaScript Programs using Agglomerative Clustering and Frequent Subgraph Mining". In: *International Journal of Computer Applications* (2014), pp. 1–4. DOI: 10.1109/ICAECC.2014.7002457.

[19] *Protect your JavaScript | JScrambler*. URL: https://jscrambler.com/pt/.

[20] *Stolen Software: Piracy Hits More than Movies and Music | PCMag.com*. URL: http://www.pcmag.com/article2/0,2817,2399315,00.asp.

[21] Haruaki Tamada and Masahide Nakamura. “Design and evaluation of birthmarks for detecting theft of java programs”. In: *Proceedings of the IASTED International Conference on Software Engineering*. 2004, pp. 569–574. URL: http://pdf.aminer.org/000/260/561/design%7B%5C_%7Dand%7B%5C_%7Devaluation%7B%5C_%7Dof%7B%5C_%7Dbirthmarks%7B%5C_%7Dfor%7B%5C_%7Ddetecting%7B%5C_%7Dtheft%7B%5C_%7Dof%7B%5C_%7Djava.pdf.

[22] Haruaki Tamada, K Okamoto, and M Nakamura. “Dynamic software birthmarks to detect the theft of windows applications”. In: *International Symposium on. Future Software Technology (ISFST)* (2004). URL: http://www.sea.jp/Events/isfst/ISFST2004/CDROM04/Presented04/2P1-T1/isfst2004%7B%5C_%7DJ355.pdf.

[23] Haruaki Tamada et al. “Detecting the Theft of Programs Using Birthmarks”. In: *Information Science Technical Report NAIST-IS-TR2003014 ISSN 0919- 9527,* 2003, p. 13. URL: http://se.aist-nara.ac.jp/jbirth/.

[24] *Top 50 Inspiring and Most Read Startup Stories of 2013*. URL: http://yourstory.com/2013/12/50-most-read-stories-on-yourstory/.

[25] J. R. Ullmann. “An Algorithm for Subgraph Isomorphism”. In: *Journal of the ACM* 23.1 (Jan. 1976), pp. 31–42. ISSN: 00045411. DOI: 10.1145/321921.321925. URL: http://portal.acm.org/citation.cfm?doid=321921.321925.

BIBLIOGRAPHY

Index