IJARSCT

ISSN (Online) 2581-9429

International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)

International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal

Impact Factor: 7.53

Volume 4, Issue 2, August 2024

Software Plagiarism Finder

Azeena S[1], Riyad A[2], Harikrishnan S R[3]

Student, MCA, CHMM College for Advanced Studies, Trivandrum, India [1]

Associate Professor, MCA, CHMM College for Advanced Studies, Trivandrum, India[2,3]

Abstract: *Software plagiarism, the illegal copying of code, negatively impacts both open-source communities and legitimate companies. Notable incidents include Verizon being sued by the Free Software Foundation for distributing Busy box in its routers, and Skype's licensing issues with Joltid. Plagiarism is easy to execute but hard to detect, with a 2012 study indicating that 5%-13% of apps in third-party markets are copied from the official Android market. Challenges in detection arise from the lack of source code and advanced code obfuscation techniques. Researchers have developed methods like software birthmarking, which extracts unique characteristics from programs to identify them. Birthmarks can be static or dynamic; static birthmarks analyse syntactic features but struggle against obfuscations and packing techniques. Dynamic birthmarks, derived from runtime behaviours, are more accurate and robust. However, the rise of multithreaded programs poses a challenge to existing detection methods, which are optimized for sequential programs*

Keywords: Software Plagiarism, Code Obfuscation, Birthmarking Techniques, Detection Challenges, Multithreaded Programs

I. INTRODUCTION

Software plagiarism, involving the illegal copying of code, is a significant problem affecting both open source communities and commercial entities. High-profile cases such as Verizon being sued by the Free Software Foundation for the unauthorized distribution of Busybox in its routers and Skype's licensing disputes with Joltid highlight the serious repercussions of software plagiarism. Despite its ease of execution, detecting software plagiarism remains a complex challenge. A 2012 study revealed that between 5% and 13% of apps in third-party markets were copied from the official Android market, underscoring the prevalence of this issue. One of the main challenges in detecting plagiarism is the frequent lack of access to source code and the use of sophisticated code obfuscation techniques that make it difficult to identify copied software. In response to these challenges, researchers have developed various methods to detect software plagiarism. One notable approach is software birthmarking, which involves extracting unique characteristics from a program to create a "birthmark" that can be used for identification. Birthmarks are typically divided into two categories: static and dynamic. Static birthmarks analyze syntactic features of code, such as its structure and syntax, but often struggle with obfuscations and packing techniques that alter the program's appearance without changing its functionality. On the other hand, dynamic birthmarks are derived from the runtime behaviors of a program. These are generally more accurate and robust, as they capture the actual execution patterns of the program, which are harder to obfuscate. However, the rise of multithreaded programming introduces new challenges for existing plagiarism detection methods. Many traditional detection techniques are optimized for sequential programs and struggle to handle the complexities introduced by multithreading, such as non-deterministic thread scheduling and concurrent execution. As multithreaded programs become increasingly common, there is a pressing need to develop and refine detection methods that can effectively address these challenges and ensure reliable identification of plagiarized software.

II. LITERATURE SURVEY

Deep Sentence Embedding Using the Long Short Term Memory Network: Analysis and Application to Information Retrieval: This paper introduces a model for sentence embedding that utilizes recurrent neural networks (RNN) with Long Short Term Memory (LSTM) cells. The LSTM-RNN model processes each word in a sentence sequentially, extracting and embedding information into a semantic vector. Thanks to its capability to retain long-term memory, the

LSTM-RNN progressively builds a more detailed representation of the sentence, culminating in a semantic vector from the final hidden layer that encapsulates the entire sentence. This model is trained in a weakly supervised manner using user click-through data from a commercial web search engine. The paper includes visualizations and analyses to elucidate the embedding process, demonstrating how the model prioritizes important words and identifies key terms. These key terms activate specific LSTM cells, with words on similar topics stimulating the same cell. The resulting sentence embeddings have broad applications, such as web document retrieval, where similarity is assessed based on the distance between embedding vectors.

On cross-lingual text similarity using neural translation models. Journal of Information Processing: Accurately measuring similarity between texts in different languages is crucial for applications like cross-lingual information retrieval and text mining. This paper explores this issue using neural networks, with a particular focus on neural machine translation (NMT) models. Instead of concentrating on the translation output, we examine the intermediate states within these models. We hypothesize that these intermediate states effectively capture the syntactic and semantic essence of the texts, providing a robust representation that mitigates translation errors. To test this hypothesis, we evaluate the effectiveness of these intermediate states in assessing cross-lingual text similarity, comparing them to other neural network-based text representations, such as word and paragraph embeddings. Our findings show that using intermediate states not only surpasses these methods but also performs better than a traditional machine translation-based approach. Additionally, we find that intermediate states and translated texts complement each other, even though they originate from the same NMT models.

Cross-lingual text similarity exploiting neural machine translation models:One traditional method involves translating texts to make the problem monolingual, while a newer approach leverages intermediate states from NMT models to minimize translation errors. Our goal is to enhance both methods separately and then integrate them—using both translations and intermediate states—within a learning-to-rank framework to assess cross-lingual text similarity. We evaluate our approach through empirical experiments on English–Japanese and English–Hindi translation corpora for sentence retrieval tasks. The results show that our method, combining translations with intermediate states, surpasses other neural network-based approaches and performs comparably to a leading machine translation system. Additionally, we consider an alternative approach that maps texts from different languages into a shared semantic space for direct comparison. Our study aligns with this direction, focusing on semantic document representation through NMT models. We build on Seki's work by refining the use of intermediate states with new techniques, including orthogonal matrix transformations and a learning-to-rank method to better integrate intermediate states with translations.

Similarity metric method for binary basic blocks of cross-instruction set architecture:Basic block similarity analysis is a crucial method used in many machine learning-based binary program analysis techniques. This analysis involves converting the semantic information of basic blocks into fixed-dimension vectors, known as basic block embeddings. However, current methods for basic block embedding face two main challenges: 1) they capture limited semantic information, and 2) they are designed for only one instruction set architecture (ISA). To address these issues, we propose a cross-ISA approach to basic block embedding that leverages a Neural Machine Translation (NMT) model to link different ISAs. Our embedding model effectively translates the rich semantics of basic blocks from any ISA into fixed-dimension vectors. We have implemented several enhancements to improve the model, including using a pretrained model to generate hard negative samples and introducing a novel assembly instruction normalization method during data preprocessing, which has proven to be more effective than previous techniques. Additionally, we developed a similarity metric and created a dataset with over a million samples to train and evaluate our method—an unprecedented scale in this field. Our prototype system, MIRROR, demonstrates significant improvements over existing baselines by producing more distinguishable basic block embeddings, which leads to more accurate evaluations.

III. WORKING OF PROPOSED SYSTEM

We present TOB (Thread-Oblivious dynamic Birthmark), designed to handle multithreaded programs by identifying unique program characteristics from binary executables, rather than source code. TOB addresses the gap between current software development practices and existing plagiarism detection technologies, which are optimized for sequential programs. We apply two plagiarism detection algorithms, System Call Short Sequence Birthmark (SCSSB)

ISSN 2581-9429 IJARSCT

IJARSCT

ISSN (Online) 2581-9429

International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)

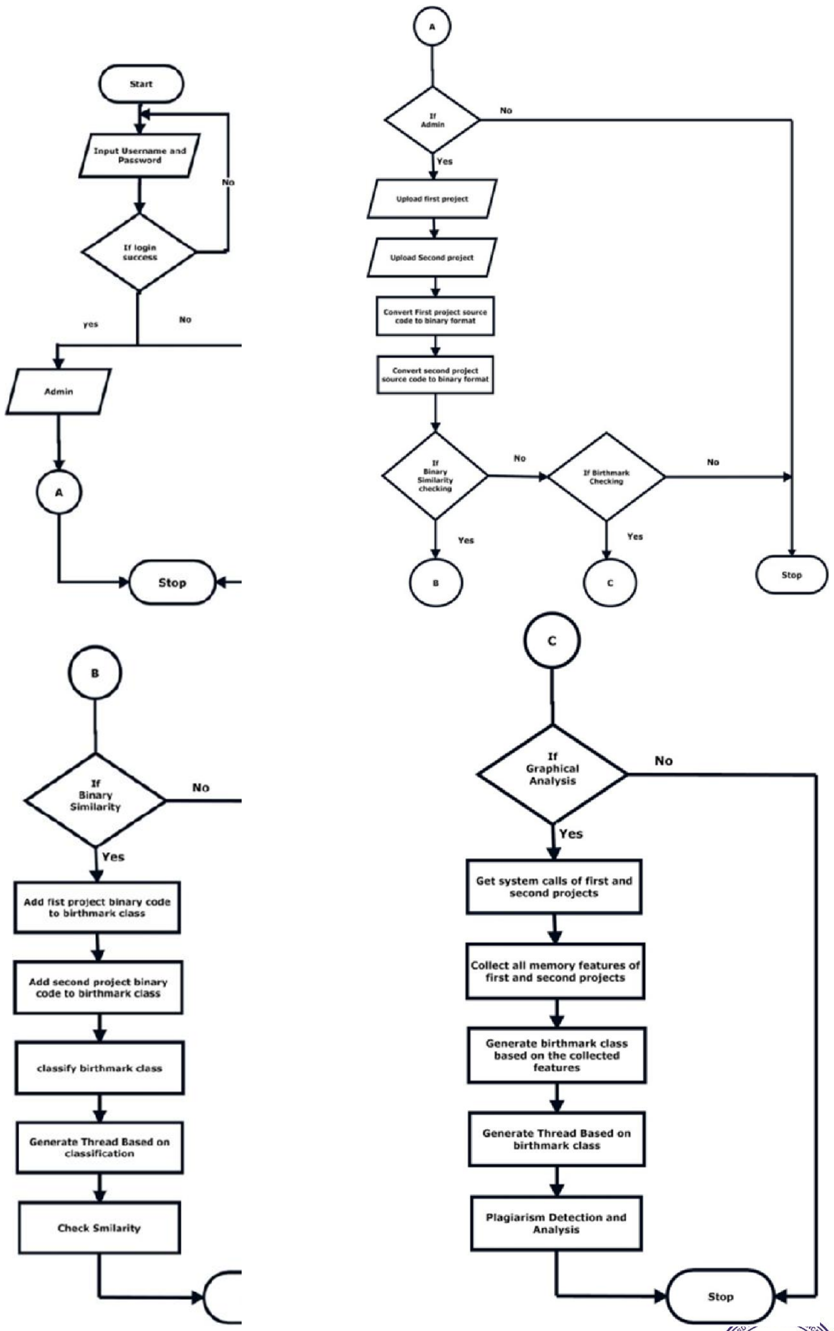
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal

Impact Factor: 7.53

Volume 4, Issue 2, August 2024

and DYKIS, using metrics like Cosine distance, Jaccard index, Dice coefficient, and Containment to analyze birthmarks from multiple executions. By integrating dynamic data flow analysis, TOB produces high-quality birthmarks resilient to obfuscation techniques. The advantages of TOB include its novel approach to addressing the impact of thread scheduling on plagiarism detection, its use of the var-gram algorithm, the development of TOB-PD (a comprehensive plagiarism detection tool), and its effectiveness in distinguishing between independently written programs that perform the same task without false positives.

ARCHITECTURE


Start
Input Username and Password
No
If login success
yes
No
Admin
A
Stop
A
If Admin
No
Yes
Upload first project
Upload Second project
Convert First project source code to binary format
Convert second project source code to binary format
If Binary Similarity checking
No
If Birthmark Checking
No
Yes
Yes
B
C
Stop
B
If Binary Similarity
No
Yes
Add fist project binary code to birthmark class
Add second project binary code to birthmark class
classify birthmark class
Generate Thread Based on classification
Check Smilarity
C
If Graphical Analysis
No
Yes
Get system calls of first and second projects
Collect all memory features of first and second projects
Generate birthmark class based on the collected features
Generate Thread Based on birthmark class
Plagiarism Detection and Analysis
Stop

IV. TECHNOLOGY USED

Open CV

OpenCV (Open Source Computer Vision Library) is a prominent open-source library dedicated to computer vision and machine learning. It was developed to establish a unified infrastructure for computer vision applications and to expedite the integration of machine perception into commercial products. Licensed under the BSD license, OpenCV allows businesses the flexibility to use and adapt the code freely. The library supports a range of programming languages including C++, Python, Java, and MATLAB, and is compatible with major operating systems such as Windows, Linux, Android, and macOS. Its primary focus is on real-time vision applications, leveraging MMX and SSE instructions to enhance performance. Currently, OpenCV is actively expanding its capabilities with the development of full-featured CUDA and OpenCL interfaces, which are crucial for high-performance computing tasks. The library includes over 500 algorithms and a vast number of supporting functions, all written in C++ with a templated interface that integrates seamlessly with STL containers.AForge.NET is a C# framework specifically designed for researchers in the fields of computer vision and artificial intelligence. The framework consolidates prior research and is continuously updated with new features and ideas. It is divided into three core components: image processing, neural networks, and evolution algorithms. Beyond merely aggregating various libraries and source codes, AForge.NET offers numerous sample applications that demonstrate its capabilities and provides comprehensive documentation in HTML Help format. This project not only aims to extend its functionality but also focuses on improving support through enhanced documentation and an expanded set of sample applications. The goal is to facilitate research and development in computer vision and artificial intelligence by providing a robust, well-documented framework that supports ongoing innovation and application development.

Visual Studio 2019

Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft, designed for building a broad spectrum of applications. These include desktop programs, websites, web apps, web services, and mobile applications. Visual Studio is compatible with several Microsoft development platforms, such as Windows API, Windows Forms, Windows Presentation Foundation, Windows Store, and Microsoft Silverlight. It supports the creation of both native and managed code. Visual Studio is a comprehensive, extensible, and free IDE for developing modern applications across Android, iOS, Windows, as well as web applications and cloud services. It does not natively support any specific programming language, solution, or tool; instead, it allows for the integration of functionalities through VSPackages. These packages are added as services within the IDE. Visual Studio offers three core services: SVsSolution, which manages project and solution enumeration; SVsUIShell, which handles windowing and UI elements like tabs, toolbars, and tool windows; and SVsShell, which manages the registration of VSPackages. The IDE also facilitates communication and coordination between these services.All editors, designers, project types, and other tools within Visual Studio are implemented as VSPackages, which are accessed via COM (Component Object Model). The Visual Studio SDK includes the Managed Package Framework (MPF), which provides managed wrappers around the COM interfaces, allowing packages to be developed in any CLI-compliant language. However, the MPF does not encompass all functionalities available through the Visual Studio COM interfaces. These services and frameworks enable the creation of additional packages that extend the capabilities of the Visual Studio IDE.

C#.Net

C# programs operate on the .NET framework, which includes the Common Language Runtime (CLR) and a set of class libraries. The CLR, Microsoft's implementation of the Common Language Infrastructure (CLI), serves as the execution environment for .NET applications, ensuring that different programming languages and libraries work together seamlessly. When a C# program runs, its assembly is loaded into the CLR, which performs Just-In-Time (JIT) compilation to translate Intermediate Language (IL) code into native machine instructions. The CLR also manages other essential services, such as automatic garbage collection, exception handling, and resource management. Code executed by the CLR is known as "managed code," contrasting with "unmanaged code" which is compiled directly into native machine language for a specific platform.Key features of C# include the absence of global variables or functions; all methods and members must be declared within classes, with static members of public classes providing a substitute for global variables and functions. Unlike C and C++, C# prevents local variables from shadowing variables in the enclosing block, reducing potential confusion. C# enforces a strict Boolean data type (`bool`) for conditional statements, avoiding the implicit conversions seen in C++ where integers can be treated as Boolean values. This approach helps prevent common programming errors, such as accidental assignments instead of comparisons.Memory management in C# is handled automatically through garbage collection, which prevents memory leaks by freeing unused memory without explicit programmer intervention. Pointers in C# are restricted to "unsafe" blocks, requiring special permissions to use, while safe object references point only to live objects or well-defined null values. Unsafe code can manipulate pointers to various types, but most object access is safely managed. Additionally, C# provides `try...catch` and `try...finally` constructs for exception handling, ensuring code execution in the `finally` block. Multiple inheritance is not supported in C#, but a class can implement multiple interfaces, simplifying design and reducing complexity. C# is designed to be more type-safe than C++, with only safe implicit conversions and explicit user-defined conversions, ensuring more predictable behavior. Enumeration members have their own scope, properties simplify the getter-setter pattern, and full type reflection is supported. C# has 77 reserved words as of version 4.0, and it does not use checked exceptions, aligning with scalability and versionability considerations.

Birthmarking Techniques

Birthmarking techniques are used to identify software by extracting unique characteristics, known as birthmarks, from programs. These techniques can be classified into two main types: static and dynamic.Static Birthmarkinginvolves analyzing the fixed attributes of a program, such as its code structure, syntax, or metadata, without executing it. This method typically examines features like function names, variable declarations, and control flow structures. Static birthmarks are valuable because they do not require the program to be executed, making them useful in scenarios where source code is accessible. However, they face limitations when dealing with code obfuscation techniques that alter the program's syntax or structure to avoid detection. Techniques such as code packing or renaming can obscure static birthmarks, making them less reliable in identifying software that has been intentionally modified to evade detection.Dynamic Birthmarking, on the other hand, relies on the runtime behavior of a program. This approach involves monitoring the program as it executes to capture patterns in its execution, such as system calls, function calls, and memory usage. Dynamic birthmarks are generated based on these execution traces and can reveal unique patterns that are more resistant to obfuscation techniques. Since dynamic birthmarking examines how the software behaves rather than just its static properties, it can be more robust against attempts to disguise or modify the code. However, dynamic techniques require the program to be run in an environment where its execution can be monitored, which might not always be feasible.Both static and dynamic birthmarking techniques have their strengths and weaknesses. Static birthmarking is advantageous for its simplicity and applicability to situations where code access is available. It is also generally faster since it does not require execution of the program. Dynamic birthmarking, while potentially more accurate in identifying software that has undergone modifications or obfuscations, can be more complex to implement and may require additional resources to monitor and analyze runtime behavior.

Plagiarism detection

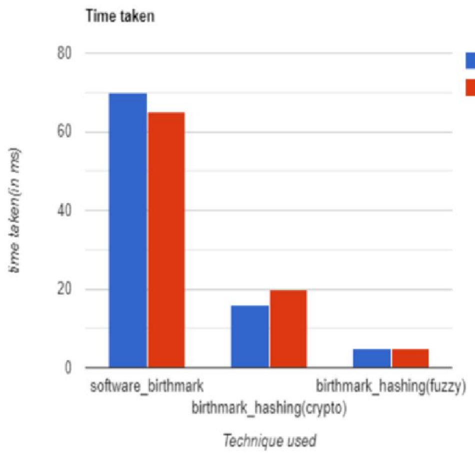
Previous methods for analyzing birthmarks involve calculating the similarity between two extracted birthmarks, typically ranging from 0 to 1. A similarity score of 0 indicates that the two programs are completely different, while a score of 1 suggests that one program is highly likely a copy of the other. The similarity between birthmarks B(p, I) and B(q, I) is represented as sim(B(p, I), B(q, I)). To determine whether one program is a copy of another, a similarity threshold ε is used. We categorize similarity scores into three groups to provide clarity on the results. Different inputs usually generate distinct birthmarks, but standardizing inputs across various projects is challenging due to differing formats. Consequently, this paper does not standardize inputs but instead uses unit test codes as raw inputs. Additionally, since a project may have multiple test codes, our method extracts several birthmarks for each project. For sequential birthmarks, similarity is calculated using pattern matching techniques like finding the longest common subsequences. Set-based birthmarks are created by dividing sequences into shorter subsequences, making comparisons more efficient. Methods commonly used for comparing sets include the Dice coefficient, Jaccard index, and Cosine distance. Graph-based birthmarks require more complex similarity calculations, which can be performed using graph monomorphism or isomorphism algorithms, translating graphs into vectors with techniques like random walk with restart, or mapping graphs to values using methods such as centroid and structure measurement. Unfortunately, these existing methods are not directly applicable to comparing thread-oblivious birthmarks.

V. PERFORMANCE ANALYSIS

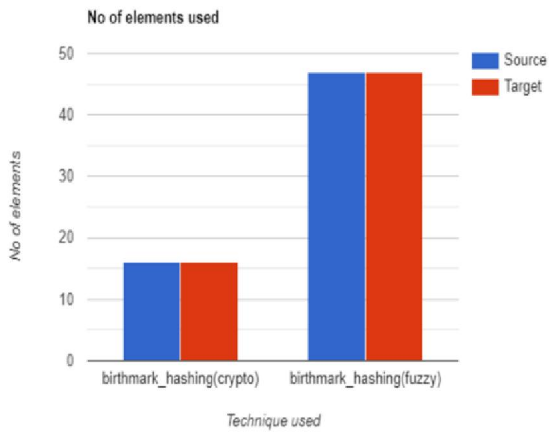
The two programs taken for plagiarism detection is compared using Jaccard index and Cosine distance. The programs that are found 100% similar are declared as same copy or plagiarised. Then they are further classified as chances of high plagiarism , chances of plagiarism and low chances of plagiarism.

• 100% - Plagiarism found
• 90%-99% - Very high chances of plagiarism
• 80%-90% - Chances of plagiarism
• 70%-80% = Low chances of plagiarism

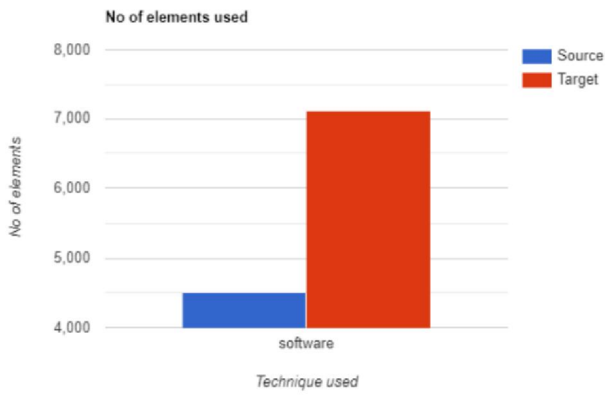
Also the time taken for the three cases : birthmark comparison, cryptographic - MD5 hash comparison and Non cryptographic comparison is plotted against each other. MD5 takes the minimum time among the three, but it cant be used for similarity checking. It rather gives yes/no answer that is whether it is copied or not. Even a single bit difference will result in different answer



Time taken by the three methods



Time taken by the other two methods



Time taken by software birthmark methods

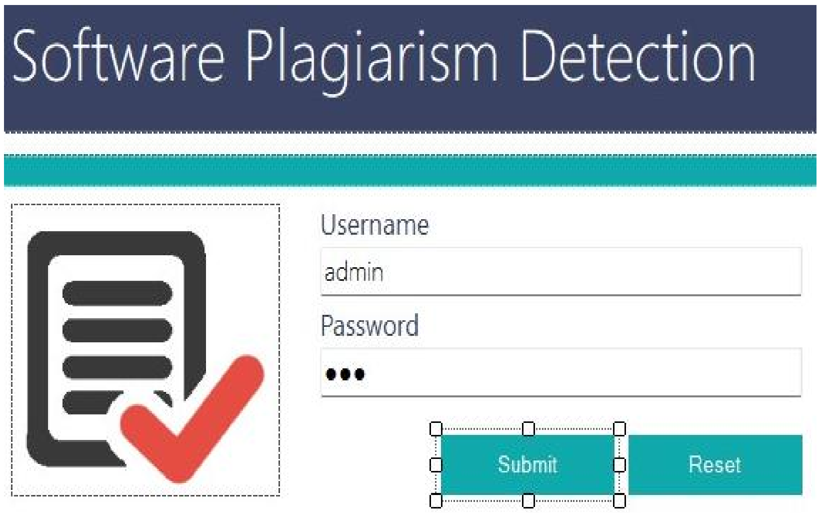
IJARSCT
ISSN (Online) 2581-9429
International Journal of Advanced Research in Science, Communication and Technology (IJARSCT)
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
Impact Factor: 7.53
Volume 4, Issue 2, August 2024

VI. FUTURE WORK

The proposed method also suffer the same limitation of dynamic birthmarks in exhaustively covering all behaviors of a program. Despite large number of executions, the inputs still constitute only a small proportion of the whole input space. One way to alleviate the concerns is to combine with testing techniques. One problem plagiarism detection researches face is the lack of real world plagiarism cases. In recent years, whole program plagiarism on mobile markets starts to rise, and many of the stolen apps have been processed with obfuscation techniques to evade plagiarism detection. Identifying potential pairs of apps that plagiarism may exist is extremely labor-intensive. Thus, this can be taken as one of the future work. Besides whole program plagiarism, there exists many cases that only part or a library of a program is copied. The main problem of using dynamic birthmarks to detect partial plagiarism is that they are mainly based on the similarity of program executions. Therefore, if there is only a small portion of code being copied, these approaches give low similarity scores. A straightforward solution is to instrument only the suspicious part. But this requires manual efforts and domain knowledge

VII. RESULT

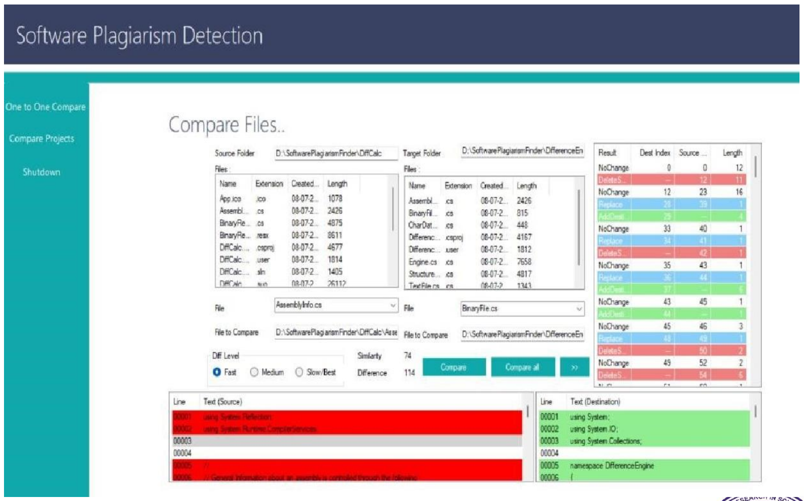
login page



Username: admin

Password: •••

Compare project



Source Folder: D:\SoftwarePlagiarismFinder\DiffCalc

Files :

Name	Extension	Created...	Length
App.ico	.ico	08-07-2...	1078
Assembl...	.cs	08-07-2...	2426
BinaryRe...	.cs	08-07-2...	4875
BinaryRe...	.resx	08-07-2...	8611
DiffCalc....	.csproj	08-07-2...	4677
DiffCalc....	.user	08-07-2...	1814
DiffCalc....	.sln	08-07-2...	1405
DiffCalc...	.suo	08-07-2	26112

Target Folder: D:\SoftwarePlagiarismFinder\DifferenceEn

Files :

Name	Extension	Created...	Length
Assembl...	.cs	08-07-2...	2426
BinaryFil...	.cs	08-07-2...	815
CharDat...	.cs	08-07-2...	448
Differenc...	.csproj	08-07-2...	4167
Differenc...	.user	08-07-2...	1812
Engine.cs	.cs	08-07-2...	7658
Structure...	.cs	08-07-2...	4817
TextFile.cs	.cs	08-07-2	1343

File: AssemblyInfo.cs
File to Compare: D:\SoftwarePlagiarismFinder\DiffCalc\Asse

File: BinaryFile.cs
File to Compare: D:\SoftwarePlagiarismFinder\DifferenceEn

Diff Level: [✓ Fast] Medium Slow/Best

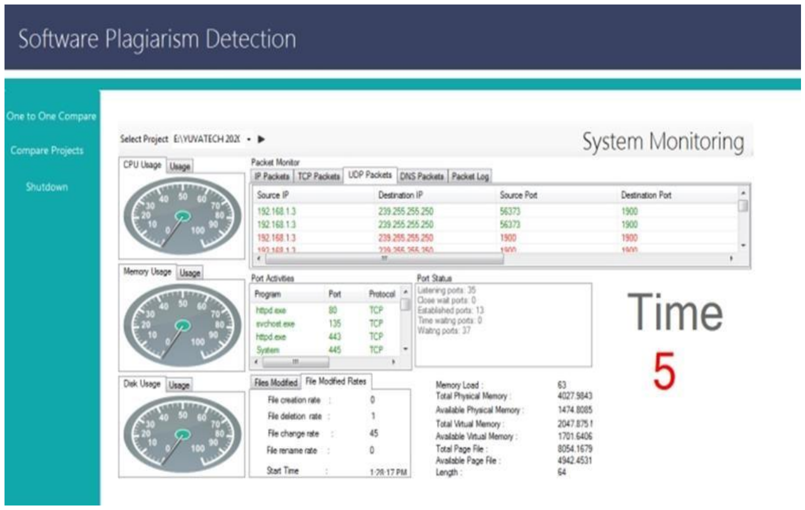
Similarity 74
Difference 114

Result	Dest Index	Source ...	Length
NoChange	0	0	12
DeleteS...	--	12	11
NoChange	12	23	16
Replace	28	39	1
AddDest	29	--	4
NoChange	33	40	1
Replace	34	41	1
DeleteS...	--	42	1
NoChange	35	43	1
Replace	36	44	1
AddDest...	37	--	6
NoChange	43	45	1
AddDest...	44	--	1
NoChange	45	46	3
Replace	48	49	1
DeleteS...	--	50	2
NoChange	49	52	2
DeleteS...	--	54	6

Line	Text (Source)
00001	using System.Reflection;
00002	using System.Runtime.CompilerServices;
00003	
00004	
00005	//
00006	// General Information about an assembly is controlled through the following

Line	Text (Destination)
00001	using System;
00002	using System.IO;
00003	using System.Collections;
00004	
00005	namespace DifferenceEngine
00006	{

Birthmark generation class first stage source project



Source IP	Destination IP	Source Port	Destination Port
192.168.1.3	239.255.255.250	56373	1900
192.168.1.3	239.255.255.250	56373	1900
192.168.1.3	239.255.255.250	1900	1900
192.168.1.3	239.255.255.250	1900	1900

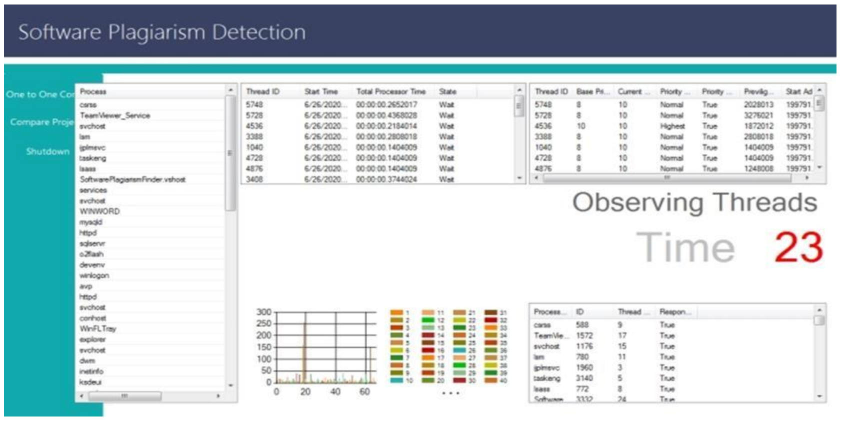
Program	Port	Protocol
httpd.exe	80	TCP
svchost.exe	135	TCP
httpd.exe	443	TCP
System	445	TCP

Listening ports: 35
Close wait ports: 0
Established ports: 13
Time waiting ports: 0
Waiting ports: 37

File creation rate	:	0
File deletion rate	:	1
File change rate	:	45
File rename rate	:	0
Start Time	:	1:28:17 PM

Memory Load :	63
Total Physical Memory :	4027.9843
Available Physical Memory :	1474.8085
Total Virtual Memory :	2047.875
Available Virtual Memory :	1701.6406
Total Page File :	8054.1679
Available Page File :	4942.4531
Length :	64

Thread birthmark generation class first stage first project

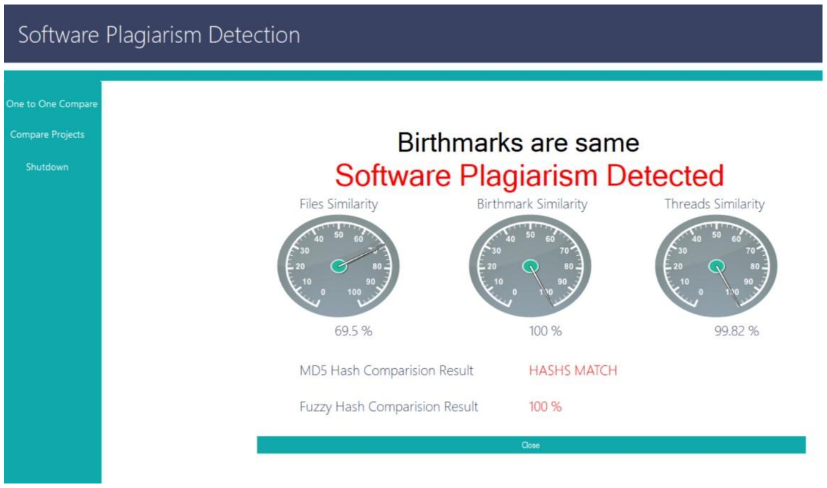


Process
csrss
TeamViewer_Service
svchost
lsm
jpImsvc
taskeng
lsass
SoftwarePlagiarismFinder.vshost
services
svchost
WINWORD
mysqld
httpd
sqlservr
o2flash
devenv
winlogon
avp
httpd
svchost
conhost
WinFLTray
explorer
svchost
dwm
inetinfo
ksdeui

Thread ID	Start Time	Total Processor Time	State
5748	6/26/2020...	00:00:00.2652017	Wait
5728	6/26/2020...	00:00:00.4368028	Wait
4536	6/26/2020...	00:00:00.2184014	Wait
3388	6/26/2020...	00:00:00.2808018	Wait
1040	6/26/2020...	00:00:00.1404009	Wait
4728	6/26/2020...	00:00:00.1404009	Wait
4876	6/26/2020...	00:00:00.1404009	Wait
3408	6/26/2020...	00:00:00.3744024	Wait

Thread ID	Base Pri...	Current ...	Priority ...	Priority ...	Previlig...	Start Ad
5748	8	10	Normal	True	2028013	199791
5728	8	10	Normal	True	3276021	199791
4536	10	10	Highest	True	1872012	199791
3388	8	10	Normal	True	2808018	199791
1040	8	10	Normal	True	1404009	199791
4728	8	10	Normal	True	1404009	199791
4876	8	10	Normal	True	1248008	199791

Process...	ID	Thread ...	Respon...
csrss	588	9	True
TeamVie...	1572	17	True
svchost	1176	15	True
lsm	780	11	True
jpImsvc	1960	3	True
taskeng	3140	5	True
lsass	772	8	True
Software	3332	24	True



Birthmarks are same
Software Plagiarism Detected

Files Similarity	Birthmark Similarity	Threads Similarity
69.5 %	100 %	99.82 %

MD5 Hash Comparision Result HASHS MATCH

Fuzzy Hash Comparision Result 100 %

VIII. SUMMARY

Software plagiarism involves the illegal copying of code from others. Various techniques have been developed to address this issue, including the use of software birthmarks based on system calls. System calls are considered fundamental indicators of program behavior, making them suitable for behavior-based birthmarks. However, the rise of multithreaded programming creates a gap between current software development practices and plagiarism detection technologies, as existing dynamic approaches are primarily optimized for sequential programs. This project also addresses this gap.The software birthmark generated using system calls tends to be large and requires significant time for comparison. To address this challenge, we introduce fuzzy hashing. This technique computes a hash of the birthmark and compares these hashes to determine similarity, resulting in a substantial reduction in comparison time. However, the proposed method shares the limitation of dynamic birthmarks in that it cannot exhaustively cover all program behaviors, as the inputs only represent a small fraction of the possible input space. Combining this approach with testing techniques could help mitigate this issue.A significant challenge in plagiarism detection research is the scarcity of real-world plagiarism cases. Recently, there has been an increase in whole program plagiarism in mobile markets, with many stolen apps using obfuscation techniques to evade detection. Identifying potential instances of plagiarism among these apps is extremely labor-intensive and represents a potential area for future work.Additionally, plagiarism may involve only parts or libraries of a program rather than the entire program. Dynamic birthmarks are often less effective in detecting partial plagiarism because they rely on the similarity of program executions. When only a small portion of code is copied, these methods may produce low similarity scores. A potential solution is to focus on instrumenting the suspicious parts of the code, but this requires manual effort and domain expertise.

REFERENCES

[1]. Z. Tian, T. Liu, Q. Zheng, E. Zhuang, M. Fan and Zijiang Yang (2017), ”Reviving Sequential Program Birthmarking for Multithreaded Software Plagiarism Detection,” IEEE Transactions on Software Engineering, Volume: 44, Issue: 5, pp.: 491 - 511.

[2]. C. Collberg, G.R. Myles, and A. Huntwork (2003), ”Sandmark-A tool for software protection research” IEEE Security Privacy, Volume: 1, Issue: 4, pp: 40-49.

[3]. Z. Tian, Ting Liu, Q. Zheng, M. Fan, E. Zhuang and Z. Yang (2016), ”Exploiting thread- related system calls for plagiarism detection of multithreaded programs” Journal of Systems and Software Vol. 119, pp: 136-148.

[4]. Z. Tian, Q. Zheng, T. Liu, M. Fan X. Zhang and Z. Yang (2014) ”Plagiarism detection for multithreaded software based on thread-aware software birthmarks” ICPC Conference

[5]. Z. Tian, T. Liu, Q. Zheng, F.i Tong, M. Fan and Z. Yang (2016) ”A New ThreadAware Birthmark for Plagiarism Detection of Multithreaded Programs”, IEEE/ACM 38th International Conference on Software Engineering Companion Austin, TX, pp. 734-736.

[6]. J. Kornblum, (2006) “Identifying almost identical files using context triggered piecewise hashing,” Journal Digital Investigation: The International Journal of Digital Forensics Incident Response, vol. 3, pp. 91–97.

[7]. T. Tsuzaki, T. Yamamoto, H. Tamada and A. Monden (2016), “A Fuzzy Hashing Technique for Large Scale Software Birthmarks”, IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS), pp. 867– 872.

[8]. Y.-C. Jhi, X. Jia, X. Wang, S. Zhu, P. Liu, and D. Wu (2015) “Program characterization using runtime values and its application to software plagiarism detection,” IEEE Trans. Softw. Eng., vol. 41, no. 9, pp. 925–943.