Source Code Anti-Plagiarism: a C# Implementation using the Routing Approach

Fabrizio d'Amore (0000-0002-6518-2445)
and Lorenzo Zarfati (0000-0003-3055-7873)

Sapienza University of Rome, Italy
damore@diag.uniroma1.it
lorenzozarfati@gmail.com

Abstract. Despite the approaches proposed so far, software plagiarism is still a problem which hasn't been solved entirely yet. The approach introduced throughout this paper is about a source code anti-plagiarism technique which aims at rendering the source code incomprehensible to a possible plagiarist and at the same time preventing source code modifications. The proposal is based on the concept of Router and makes use of both symmetric encryption and cryptographic hashing functions to provide such guarantees.

Keywords: Plagiarism · Software Plagiarism · Source code · Routing approach

1 Introduction

Plagiarism can be defined as the act of taking someone else's work or idea, and pass it as your own without giving the proper credits to the original authors. It can affect anything that has a value including academic research papers, intellectual property and software products, the last of which is causing huge money losses to companies worldwide. Naturally it is not limited to copy-paste activities as they are the technique which is the most trivial to be detected. There exist many plagiarism techniques that might be used depending on the target, for instance mosaic and self-plagiarism are two of the most common in research papers.

However when it comes to software plagiarism, both malicious and countermeasure activities change, and the challenges become even harder when compared with other domains where it must be considered. Plagiarism was not the main concern when the first software products started to be released, it is a problem which was considered only afterwards, while many products inevitably were affected already.

Our approach relies on the introduction of a software module, which we call "Router", that through some suitable cryptographic operations, can make it too difficult to copy or modify relevant parts of the source code. In our model we consider it meaningful the effort spent in the plagiarism: we do not need a provably secure approach but a heuristic that makes plagiarism costly as the process of rewriting the source code.

1.1 Previous work

During the past decades several techniques have been proposed to fight software plagiarism even though nowadays the problem is not entirely solved yet. The majority of approaches proposed so far mainly aim at detecting plagiarism once it has occurred, thus when it's too late, rather than preventing its occurrence.

Due to size limits we omit most of the discussion. For details refer to [3].

The proposed technique partially relies on obfuscation but together with symmetric encryption and cryptographic hash functions aims at preventing plagiarism attempts rather than just detecting them once they were carried out successfully.

2 A new approach

The idea on which the paper is based is an approach that has been investigated for several years, without however publishing the preliminary results. Terrinoni was the first master student to write a thesis [5] on it; later the idea was refined. It is inspired by the behavior of a well-known network device: the router, which is responsible of correctly routing packets depending on their destination address. The behavior is quite similar since the code implementing the approach consists in forwarding function references containing encrypted parameters toward the intended destination function and then return the output of such function (if any) to the original caller. From now on we will refer to the code responsible of performing all the features described in this paper with the term *Router* to make a clear distinction between the common networking device and the code implementing the approach.

The approach consists in rewriting function references (calls) to be protected with correspondent Router references, according to well-defined format which will be detailed in Section 3. Router calls input parameters will be encrypted by the Router itself during its initialization step and once this step terminates, and the software is re-compiled considering modified source files, encrypted parameters are decrypted only at runtime by the Router code which will then invoke the original function if and only if some particular conditions are met. It's very clear that this approach modify the software source code to be protected and it does so in only two ways: function calls replacement, which can be performed during the development itself or subsequently; the *Router.Init()* invocation must be inserted as the first statement of the software to be protected, so that a proper Router initialization step can be performed. The approach mainly relies on static features but it also depends on a specific runtime program state, therefore it may be classified as a blended technique. Before describing the approach further the definition of file dependency and closed hash values have to be clear.

Definition 1 (file dependency) *A source file f_i depends on another source file f_j ($f_i \rightarrow f_j$) if and only if (functions in) f_i reference (call) functions defined within source file f_j and both files are part of the same software.*

The definition does not consider references to files belonging to different software nor self-dependencies, the latter are not considered because they are also some of the most common. These considerations are made as a trade-off between security and performance: replacing all the function references might lead to severe performance degradation even though there might be scenarios where this becomes viable.

Definition 2 (closed hash) *Given a DAG $G = (V, A)$ where V is the set of source files that contain functions called by different source files, and each arc $(f_i, f_j) \in A$ is a file dependency: some function of f_i calls some function of f_j. The closed hash value h_i of the i-th file having m dependencies $f_{i_1}, \ldots, f_{i_m}$, is defined as:*

$$h_i = h'_i \oplus h_{i_1} \oplus h_{i_2} \oplus \cdots \oplus h_{i_m}. \tag{1}$$

where:

- *h'_i is the traditional hash value computed just on file f_i ($h'_i = h(f_i)$, being h the chosen cryptographic hashing function;*
- *$h_{i_1}, h_{i_2}, \ldots, h_{i_m}$ are the closed hashes of files $f_{i_1}, \ldots, f_{i_m}$ which file f_i depends upon.*

If the file f_k has no dependency, then the closed hash is its standard (traditional) hash ($h_k = h'_k = h(f_k)$).

Closed hash value computation for a given file might lead to a deadlock situation if there exist some circular dependency among source files, or equivalently, if one or more cycles exist in the directed graph obtained as described above (in Def. 2). Therefore when computing closed hash values the Router has to consider source files, thus graph vertices, in a well-defined order to prevent any deadlock scenario. It can be obtained by a topological sort and it is well-known that a topological sort exists if and only if the graph is acyclic. For this reason, first the Router runs a deterministic algorithm to break every cycle (if any) arbitrarily. Then, the topological sort is done (deterministically, again) and then closed hash values can be computed according its order (backward). These mentioned values are computed by the Router during its initialization step, together with random IVs (Initialization Vectors), one for each file that has information to be encrypted. Such keys are obtained xor-ing the xor of all the closed hashes of dependent files with a nonce (the IV) for preventing any known-ciphertext attack.

Of course the Router is a function written in another file. Let us briefly summarize the process:

0. The original source code has been produced but not yet secured by our approach;
1. all calls to functions in different files are replaced by a call to the Router, with parameters the name of the originally called function, and a list of comma separated original parameters;

2. the Router is called, for pre-processing the software and doing some verifications; in particular it symmetrically encrypts by using a proper key (see details later) names and parameters that describe the original calls (and changes itself!);
3. the code of the Router is obfuscated, in order to hide the logic of the original program;
4. the software is now protected and can be released and deployed.

The symmetric encryption key is computed only during the initialization step and a second time at runtime, when the software needs it to decrypt data previously encrypted. Since the key is never stored within the software it prevents easy key extractions from the software source code leaving a potential plagiarist with no option but performing a dynamic analysis with the aim to infer the Router behavior and remove all the modifications performed before the software was released. The process is summarized below:

$$k_j = h_{j_1} \oplus h_{j_2} \oplus ... \oplus h_{j_n} \oplus \mathrm{IV}_j. \tag{2}$$

where:

- k_j is the key used to encrypt function call information of functions belonging to f_j coming from other files (but now from the Router);
- $h_{j_1}, h_{j_2}, \ldots, h_{j_n}$ represent all the closed hashes of files which depend on f_j (destinations of the outgoing arcs);
- IV_j is a nonce generated during Router initialization, associated with the j-th file ((part of) whose functions should be called by external files, sources of the incoming arcs).

Thus the key computation process depends on closed hash values and since such values are always the same as soon as the same input DAG is provided, then it follows the key uniqueness theorem [5]. However, note that both the encryption key and closed hashes must be computed over decompiled source files rather than the original ones that, for instance, contain user comments and blank lines. This is strictly required because once the software is compiled and released, the Router code will have no way to access the original source files and therefore any hash value on different files will never match the one computed during the Router initialization step.

Once closed hash values and IVs are computed they are printed to standard output, together with file dependency info as Base-64 encoded strings which have to be copied within the Router very first lines of code, replacing homonymous declared yet not defined variables. This step can be easily automated and even though it is planned for future releases the implemented technique currently leaves this task to developers.

Every reference's first parameter to Router code is encrypted and contains all the data required to invoke a specific function, such as its name and parameters as well as the name of the enclosing class. Since such parameter was previously encrypted with a key depending on files (according to Equation (2)), it follows

that as soon as the decryption succeeds we know such files were not modified (we currently use `aes-256-cbc`, but we plan to switch to the more secure authenticated encryption[1]). On the other hand before invoking the original function, the Router ensures that runtime computed closed hash value of the calling file corresponds to the same value computed during the initialization step. If these values are equal and a secure cryptographic hash function is used, such as one of the SHA-2 or SHA-3 family, then we also know that any potentially involved file was not altered in any way. Thus if either the decryption of the string parameter fails or if any runtime computed closed hash does not match the previously computed one, then the Router terminates the running software and displays a message to the user indicating a plagiarism attempt was detected.

3 C# Implementation

The Router functionalities have been implemented through multiple C# classes which can be easily imported since they are part of a common C# VS (Visual Studio) project. We chose to use C# as it is usually the preferred choice when implementing executables and DLL files for Microsoft Windows. The prototype is available as open-source software at the URL https://github.com/msc-antiplag.

3.1 Software Dependencies

The Router code uses both reflection and cryptographic API to retrieve/invoke a specific function and to perform encryption and closed hashes related tasks, respectively. Apart these two APIs the software also has two additional dependencies:

- `Roslyn`. C# Code Analysis API.
- `ilspycmd`.[2] A command-line decompiler using the `ILSpy`[3] decompilation engine.

Due to size limits we omit most of the discussion. For details refer to [3].

3.2 Anti-Plagiarism Steps

Once the Router project is referenced by the VS project(s) of the application source code, we follow the steps mentioned below.

1. Replace original function calls (only those calling to other files) with correspondent Router `forwardCall`.
2. Run the Router initialization step.

[1] A method that guarantees confidentiality and authentication. Beginners can see, e.g., [4, 6].
[2] https://www.nuget.org/packages/ilspycmd/
[3] https://github.com/icsharpcode/ILSpy/releases/

3. Copy output values within the Router code and compile the software with the modified source files.
4. Obfuscate the Router code.

The calls replacement of the first step must be performed according to a predefined format which essentially consists in invoking a Router `forwardCall` instead, with all the parameters provided as a hyphen-separated values string. The mentioned format is shown below whereas an example with dummy objects is shown right after it.

```
[dstClssFullName]-[methodName]-[parametersTypes]-[parametersValues]

Class1.f1();                  (defined in Class1)
obj.f2();                     (obj is an instance of Class2)
obj.f3(7, "aString", varX);   (obj is an instance of Class3)
```

The expected function call replacement is shown below:

```
Router.ForwardCall("Class1-f1-null-null");
Router.ForwardCall("Class2-f2-null-null", obj);
Router.ForwardCall("Class3-f3-System.Int32,System.String-7,aString",
                   obj, X);
```

Steps 2. and 3. require little or no effort and are fast to be performed whereas the last one might require additional time. The Router code must be obfuscated because otherwise, anyone having access to the executable would decompile it, understands the internal logics and finally renders the protection scheme worthless by decrypting the ciphertext. It is recommended to apply obfuscation to both static and dynamic variables to obtain stronger guarantees (e.g., using SmartAssembly[4]).

4 Tests Outcomes

Due to size limits we omit most of the discussion. For details refer to [3].

5 Security Considerations

5.1 Comparison with Java implementation

From an attacker perspective, once the source code is obtained all that remains to do is to decrypt data and retrieve original function references as if the Router were never been used. When compared to previous Java proposal [1], the C# implementation provides stronger guarantees in terms of security as it does not leak any useful information to a potential plagiarist. On the other hand, previous

[4] https://www.red-gate.com/products/dotnet-development/smartassembly/

Java proposals leak the name of the class containing a called function, thus making the task of a plagiarist far easier. Every reference to the Router `forwardCall` function will appear as depicted below, with a longer (non-truncated) Base-64 encoded string as first parameter.

```
Router.forwardCall("wAp48JGP1bYqVzCYiwuNwSIKVA==",
                   "KeePass.Util.BinaryDataUtil");
```

First encoded parameter corresponds to the encrypted binary string containing all the information required to invoke the original function. Once decoded it provides no useful information to anyone having no access to the decryption key. The second plaintext parameter leaks no information since it's just used to shorten the overall delay of `forwardCall` and it might be obtained anyway by just decompiling the running software.

5.2 No trial-error guessing

In order to guess the original function references an adversary might either attempt to break the encryption scheme or guess all the original function references at once. However, decrypting something previously encrypted with `aes-256` is known to be unfeasible. Thus, it remains to guess all the functions original references and despite it has been made harder than before it is not impossible to be accomplished.

Due to size limits we omit most of the discussion. For details refer to [3].

5.3 Strongly Typed Programming Languages

When a plagiarist will attempt to guess original references he will be helped by the nature of the used programming language. Both C# and Java are strongly-typed. Furthermore the function type is known too. Since the usage of a strongly typed programming language somewhat weakens the C# implementation it is highly recommended to replace (i.e. protect) only those references to functions for which exist at least one other function definition returning the same output type.

5.4 Dynamic Analysis

A plagiarist cannot understand much about the protected function references and he might just have some intuition about the original ones, together with all its input parameters, by looking at the context in which such reference is found. For instance, it would be trivial to guess the original reference to function returning a string value if the surrounding code performs string manipulation and there exist only two function candidates: one performing string manipulation as well and the other performing network-related activities. Thus it will be a responsibility of the developer to choose what references should be replaced and what should not be. If this is done properly, then there would not be much

information obtainable through a static analysis. However, dynamic analysis can provide additional information. Depending on the obfuscation algorithms used, it might be possible to understand which files are accessed and then infer the technique used to compute both the encryption key and the runtime hash checking mechanism, even though this requires far more skills than a simpler static code analysis and is not guaranteed to succeed.

6 Conclusions

We presented a new approach to prevent the source code plagiarism. Our prototypes showed that the approach is viable, at least while programming in Java and in C# (see [3, 1, 2]). No other similar approach is known to the best of our knowledge.

We understand that our solution is not unassailable (some advanced schemes of attack are shown in [1]), however the main purpose is to render plagiarism so long, boring and complicated to make it competitive to re-write the software. The Router needs to be obfuscated, in order to prevent its analysis and understanding.

Our method could lead to a new scheme/framework in software production, making the addition of the anti-plagiarism technique integrated in the life cycle of the software product to be protected.

Future work will focus on the Javascript language, since it is very used in the construction of web sites/applications. Thus, our approach could prevent the copying of code from web site and its reuse in other sites, a habit that is very common in the modern web.

The authors acknowledge the contributions to this research by former master students Terrinoni [5], Cardelli [1] and Cavallaro [2], who got their master degree between 2017 and 2019, allowed to develop a more mature approach and tested the method for the Java language.

References

1. R. Cardelli. Anti-plagiarism detectors: Providing source code integrity for watermarking and protecting software. Master's thesis, Sapienza University of Rome, 2019.
2. S. Cavallaro. Anti-plagiarism in software development, a Java implementation. Master's thesis, Sapienza University of Rome, 2019.
3. F. d'Amore and L. Zarfati. Source code anti-plagiarism: a C# implementation using the routing approach, 2022. https://arxiv.org/pdf/2201.02241.pdf.
4. E.B. Kavun, H. Mihajloska, and T. Yalçin. A survey on authenticated encryption–ASIC designer's perspective. *ACM Comput. Surv.*, 50(6), December 2017.
5. J. Terrinoni. Anti-plagiarism detectors (APD): an innovation in source code protection. Master's thesis, Sapienza University of Rome, 2017.
6. F. Zhang, Z. Liang, B. Yang, X. Zhao, S. Guo, and K. Ren. Survey of design and security evaluation of authenticated encryption algorithms in the CAESAR competition. *Frontiers Inf. Technol. Electron. Eng.*, 19(12):1475–1499, 2018.