

METHOD AND SYSTEM FOR DETERMINING AFFILIATION OF SOFTWARE TO SOFTWARE FAMILIES

TECHNICAL FIELD

[0001] The present technology broadly relates to the field of computer technology; and, in particular, to methods and systems for determining affiliation of software to predetermined software families and/or authorships.

BACKGROUND

[0002] As it may be known, professional cybercriminals thoroughly elaborate an attack strategy and change it rarely, using the same malware for a long time with insignificant modifications.

[0003] At the same time, the developers of malicious SW (MSW) creating tools for cybercriminals could use the same software solution, for example, a function implementing cryptographic algorithm, for a long time in different samples of MSW created for different cybercriminal groups and associated with different MSW families.

[0004] Therefore, in the field of cyber security, it may be important to know with which MSW family a given sample of MSW is affiliated and/or who is an author (or a group thereof) of the given sample of MSW.

[0005] Signature analysis is a well-known method of MSW detection. This method is based on the search of a unique sequence of bytes in files including source code of MSW, i.e. signature which is indicative of a specific MSW. A respective signature associated with the given sample of MSW may be determined based on analyzing a source code associated therewith. Further, the respective signature can be stored into a virus signature database, to which an antivirus program may be provided access, thereby allowing for detection of the given sample of MSW.

[0006] However, this method is also well-known to cybercriminals. Therefore, nearly all types of modern MSW are constantly modified to change basic functionality thereof. As a result of such modifications, the files of a next version of the given MSW may acquire new properties, which may render the given MSW unrecognizable for antivirus signature analyzers as malicious, which may thus allow cybercriminals to conduct attacks without any obstacles.

[0007] Besides the modifications, various approaches of obfuscation are widely used. Broadly speaking, in the context of the present specification, “obfuscation” refers to a technique of modifying a source code of the given MSW such that functionality thereof is preserved, however, analyzing it to

determine operation algorithms becomes more complicated. The above-mentioned modifications to the given MSW could be performed either by a human or automatically, e.g. by so called polymorphic generator, which may be part of a malware.

[0008] At the same time, as a result of the obfuscation, operating functions of the given MSW are not significantly altered. For example, after the modification the given MSW will "look" differently only for signature analyzers, its code could be obfuscated and hence cannot be analyzed by a human; however, a set of operating functions of the given MSW performed before the obfuscation is likely to remain unchanged thereafter.

[0009] Certain prior art approaches are directed to determining authorship of different types of texts, such as literary, publicistic, or scientific based on stylometric analysis thereof.

[0010] An article written by Dauber *et al.*, published by Drexel University, Philadelphia, USA, and entitled ***“Stylometric Authorship Attribution in Collaborative Documents”***, discloses applying stylometry to a novel dataset of multi-authored documents collected from Wikia using both relaxed classification with a support vector machine (SVM) and multi-label classification techniques. Five possible scenarios are defined that show that one, the case where labeled and unlabeled collaborative documents by the same authors are available, yields high accuracy on the dataset while the other, more restrictive cases yield lower accuracies. Based on the results of these experiments and knowledge of the multi-label classifiers used, there is proposed a hypothesis to explain this overall poor performance. Additionally, there is performed authorship attribution of pre-segmented text from the Wikia dataset showing that while this performs better than multi-label learning it requires large amounts of data to be successful.

[0011] A PhD thesis written by S.Afroz at Drexel University, Philadelphia, USA, and entitled ***“Deception In Authorship Attribution”*** discloses authorship attribution methods in adversarial settings where authors take measures to hide their identity by changing their writing style and by creating multiple identities; using a large feature set to distinguish regular documents from deceptive documents with high accuracy and present an analysis of linguistic features that can be modified to hide writing style; adapting regular authorship attribution to difficult datasets such as leaked underground forum; and presenting a method for detecting multiple identities of authors. Further, demonstrating the utility of the approach with a case study that includes applying the technique to an underground forum and manual analysis to validate the results, enabling the discovery of previously undetected multiple accounts.

[0012] An article written by Alexander Granin, published by the web resource habr.com, and entitled ***“Text Analyzer”*** appears to disclose an automatic approach to determining authorship of texts based on Hamming Neural Network.

SUMMARY

[0013] Developers of the present technology have realized that the stylometric approaches, i.e. those based on analyzing text stylistics, for determining program code authorship, may not be an optimal solution. Regardless of the programming language in which the code is written, defining the author style in it would be extremely difficult by reason of the specific nature of the art. In cases, when the program source code is obfuscated, use of the stylometric approaches for samples of MSW analysis is not likely to bring expected results.

[0014] Therefore, non-limiting embodiments of the present technology are directed to methods and systems for determining affiliation of given software to a predetermined family of software and/or authorship based on specific features associated therewith that are derived from a source code thereof. It should be expressly understood that the method and systems described herein are not limited to MSW and may be used for any software.

[0015] More specifically, according to a first broad aspect of the present technology, there is provided a method for determining an affiliation of a given software with target software. The method is executable by a processor. The method comprises: receiving, by the processor, a software source code of the given software; determining, by the processor, reference data associated with software source code; parsing, by the processor, based on the reference data associated with software source code, the software source code to identify therein at least one function, the identifying comprising identifying, in the software source code, library functions and deleting the library function by applying, by the processor, a signature analysis to the software source code; parsing, by the processor, the at least one function to identify therein at least one function command, the at least one function command comprising a respective action and at least one argument associated with the at least one action; generating, by the processor, for the at least one function, based on the respective action and the at least one argument thereof, a respective one of a plurality of function identifiers associated with the given software, the generating comprising applying, by the processor, a hash function to the respective action and to each value of the at least one argument associated therewith to generate respective hash function values; and aggregating the respective hash function values; aggregating, by the processor, the plurality of function identifiers associated with the given software into an aggregated array of function identifiers given an order of occurrence of associated functions within the software source code; applying, by the processor, at least one classifier to the aggregated array of function identifiers associated with the given software to determine a likelihood parameter indicative of the given software being affiliated to a respective target software, the at least one classifier having been trained to determine the affiliation with the respective target software; in response to the likelihood parameter being equal to or greater than a predetermined likelihood parameter threshold: identifying the given software as being affiliated to the respective target

software; storing data indicative of the given software in a database of affiliated software; and using the data indicative of the given software for further determining affiliation to the given target software.

[0016] In some implementations of the method, the identifying the library functions further comprises identifying, in the software source code, source code portions inherently non-indicative of the affiliation to the target software, and deleting the source code portions inherently non-indicative of the affiliation to the target software from the software source code.

[0017] In some implementations of the method, the applying the hash function further comprises applying a first hash function to the respective action and applying a second hash function to the at least one argument, the second hash function being different from the first hash function.

[0018] In some implementations of the method, the generating the respective one of the plurality of identifiers further comprises applying a serialization process to the at least one action and the at least one argument associated therewith.

[0019] According to a second broad aspect of the present technology, there is provided a method for training a classifier to determine an affiliation with a given target software. The method is executable by a processor. The method comprises: receiving, by the processor, a plurality of target software source codes associated with the given target software; determining, by the processor, for each one of the plurality of target software source codes, respective reference data; identifying, by the processor, in each one of the plurality of target software source codes, based on the respective reference data, at least one target function; parsing, by the processor, the at least one target function to identify therein at least one target function command; generating, by the processor, based on each of the at least one target function command, a respective target function identifier associated with the at least one target function, the respective target function identifier comprising an associated number sequence; aggregating, by the processor, number sequences associated with respective target functions over the plurality of target software source codes, thereby generating a number array associated with the respective target software; identifying, by the processor, in the number array associated with the given target software, at least one pattern, wherein: the at least one pattern comprises a predetermined repetitive number sequence within the number array, and the predetermined repetitive number sequence is indicative of a frequency of occurrence of at least one associated target function command within the respective target software; storing the at least one pattern, thereby generating a training set of data, the training set of data comprising a plurality of patterns, a given pattern of the plurality of patterns including a respective label being indicative of an association between the given pattern and the given target software; training the classifier, based on the training set of data, to determine the affiliation of a given software to the given target software.

[0020] In some implementations of the method, the identifying the at least one target function further comprises identifying, in each of the plurality of software source codes, library functions and deleting the library functions.

[0021] In some implementations of the method, wherein the identifying the library functions further comprises identifying, in each of the plurality of software source codes, source code portions inherently non-indicative of the affiliation to the target software, and deleting the source code portions inherently non-indicative of the affiliation to the given target software from each of the plurality of software source codes.

[0022] In some implementations of the method, wherein the at least one target function command comprises at least one action and at least one argument associated with the at least one action; and the generating the respective target function identifier further comprises: applying a hash function to the at least one action and to each value of the at least one argument to generate respective hash function values, each one of the respective hash function values being a respective number sequence; and concatenating the respective hash function values.

[0023] In some implementations of the method, the target software is associated with at least one of: a predetermined software family, and a predetermined authorship.

[0024] In some implementations of the method, the method further comprises determining a frequency occurrence value associated with the at least one pattern, the determining being according to the following formula:

$$\lambda = \frac{L}{K},$$

where L is a frequency of occurrence of the at least one pattern within the number array associated with the respective target software, and K is a number of software source codes in the plurality of software source codes associated with the respective target software used for generating the training set of data.

[0025] In some implementations of the method, the method further comprises assigning a weight value to the at least one pattern.

[0026] In some implementations of the method, the weight value is further determined based on the frequency of occurrence value associated with the at least one pattern.

[0027] According to a third broad aspect of the present technology, there is provided a method for determining an affiliation of a given software with target software. The method is executable by a processor, The method comprises: receiving, by the processor, a software source code of the given software; executing, by the processor, the software source code in an isolated program environment to

identify at least one of a plurality of outgoing requests of the given software, the at least one of the plurality of outgoing requests comprising a respective action and at least one argument associated with the respective action; and generating, for the at least one of the plurality of outgoing requests, based on the respective action and the at least one argument, a respective one of a plurality of identifiers associated with the given software; aggregating, by the processor, the plurality of identifiers associated with the given software into an aggregated array of identifiers given a time order of identifying associated outgoing requests; applying, by the processor, at least one classifier to the aggregated array of identifiers to determine a likelihood parameter indicative of the given software being affiliated to a respective target software; in response to the likelihood parameter being equal to or greater than a predetermined likelihood parameter threshold: identifying the given software as being affiliated to the respective target software; storing data indicative of the given software in a database of affiliated software; and using the data indicative of the given software for further determining affiliation to the respective target software.

[0028] In some implementations of the method, the receiving the software source code further comprises identifying therein source code portions inherently non-indicative of the affiliation to the target software, and deleting the source code portions inherently non-indicative of the affiliation to the target software from the software source code.

[0029] In some implementations of the method, the generating the respective one of the plurality of identifiers further comprises: applying a hash function to the respective action and to each value of the at least one argument associated with the at least one of the plurality of outgoing requests to generate respective hash function values, each one of the respective hash function values being a respective number sequence; concatenating the respective hash function values.

[0030] In some implementations of the method, the applying the hash function further comprises applying a first hash function to the respective action and applying a second hash function to the at least one argument, the second hash function being different from the first hash function.

[0031] In some implementations of the method, the generating the respective one of the plurality of identifiers further comprises applying a serialization process to the at least one action and the at least one argument associated therewith.

[0032] In some implementations of the method, the respective target software is associated with at least one of: a predetermined software family, and a predetermined authorship.

[0033] In some implementations of the method, the at least one classifier has been trained to determine the affiliation to the respective target software based on a training set of data, and the method further comprising generating the training set of data, the generating comprising: receiving, by the processor, a

plurality of target software source codes associated with the respective target software; executing, by the processor, a given one of the plurality of target software source codes in the isolated program environment to identify at least one of a respective plurality of target outgoing requests of the respective target software, the at least one of the respective plurality of outgoing requests comprising a respective target action and at least one target argument associated with the respective target action; generating, for the at least one of the respective plurality of target outgoing requests, based on the respective target action and the at least one target argument, a respective one of a respective plurality of target identifiers associated with the respective target software; aggregating, by the processor, target identifiers associated with the respective target software over respective pluralities of target identifiers into an aggregated array of target identifiers given a time order of identifying associated target outgoing requests; identifying, by the processor, in the aggregated array of target identifiers associated with the respective target software, at least one pattern, wherein: the at least one pattern comprises a predetermined repetitive sequence of a given target identifier within the aggregated array of target identifiers, and the predetermined repetitive sequence of the given target identifier is indicative of a frequency of the respective target software executing the respective target action in the isolated program environment; storing the at least one pattern with a label indicative of an association between the at least one pattern and the respective target software for inclusion thereof into the training set of data.

[0034] In some implementations of the method, a length of the predetermined repetitive sequence is determined as a constant number.

[0035] In some implementations of the method, the length of the predetermined repetitive sequence is further determined iteratively, based on a current number thereof within the aggregated array of target identifiers.

[0036] In some implementations of the method, the method further comprises determining a frequency occurrence value associated with the at least one pattern, the determining being according to the following formula:

$$\lambda = \frac{L}{K},$$

where L is a frequency of occurrence of the at least one pattern within the aggregated array of target identifiers associated with the respective target software, and K is a number of software source codes in the plurality of software source codes associated with the respective target software used for generating the training set of data.

[0037] In some implementations of the method, the method further comprises assigning a weight value to the at least one pattern.

[0038] In some implementations of the method, the weight value is indicative of a type of the respective action associated with the at least one of the plurality of outgoing requests of the respective target software.

[0039] In some implementations of the method, the weight value is further determined based on the frequency of occurrence value associated with the at least one pattern.

[0040] According to a fourth broad aspect of the present technology, there is provided a system for determining an affiliation of a given software with target software. The system comprises a computer device comprising: a processor, a non-volatile memory storing instructions; the processor, upon execution of the instructions, being configured to execute one of the methods in accordance with the above aspects of the present technology.

[0041] Within the context of this description, unless expressly stated otherwise, the below-listed terms mean as follows:

[0042] **Platform or computer platform** is an environment where a given piece of software is executed. The platform includes both hardware (e.g. random-access memory, hard disk) and software (BIOS, operating system, etc.). Non-limiting example of a platform may be a Win32 API platform.

[0043] **Obfuscation** or code obfuscation is deliberate modifying an initial source code of a software program such that it is difficult for humans to understand, while preserving its functionality.

[0044] **Logging** is automatic recording actions performed by a software program in chronological order into a specific file, which may be referred to as a log or a report.

[0045] **Regular source code portions** are code snippets, which could be found in a variety of programs associated with a specific type of software. Such source code portions are used not only in software of specified purpose or by a certain author but practically ubiquitously. Example of regular source code portions includes, without being limited to, function prologues.

[0046] **Framework** is a program platform defining a structure of a program system – for example, a Django framework.

[0047] Further, in the context of the present specification, unless expressly provided otherwise, a computer system may refer, but is not limited to, an “electronic device”, an “operation system”, a “system”, a “computer-based system”, a “controller unit”, a “control device” and/or any combination thereof appropriate to the relevant task at hand.

[0048] In the context of the present specification, unless expressly provided otherwise, the expression “computer-readable medium” and “memory” are intended to include media of any nature and kind

whatsoever, non-limiting examples of which include RAM, ROM, disks (CD-ROMs, DVDs, floppy disks, hard disk drives, etc.), USB keys, flash memory cards, solid state-drives, and tape drives.

[0049] In the context of the present specification, a “database” is any structured collection of data, irrespective of its particular structure, the database management software, or the computer hardware on which the data is stored, implemented or otherwise rendered available for use. A database may reside on the same hardware as the process that stores or makes use of the information stored in the database or it may reside on separate hardware, such as a dedicated server or plurality of servers.

[0050] In the context of the present specification, unless expressly provided otherwise, the words “first”, “second”, “third”, etc. have been used as adjectives only for the purpose of allowing for distinction between the nouns that they modify from one another, and not for the purpose of describing any particular relationship between those nouns.

BRIEF DESCRIPTION OF THE DRAWINGS

[0051] Non-limiting embodiments of the present technology are described herein with reference to the accompanying drawings; these drawings are only presented to explain the essence of the technology and are not limiting the scope thereof in any way, where:

[0052] Figures 1A and 1B respectively depict flowchart diagrams of methods for determining an affiliation of a given software to a target software, in accordance with certain non-limiting embodiments of the present technology;

[0053] Figures 2A to 2F schematically depict stages of processing a software source code of the given software for generating specific features for determining the affiliation of the given software to the target software, in accordance with certain non-limiting embodiments of the present technology;

[0054] Figure 3 depicts a flowchart diagram of a method for identifying and deleting regular source code portions in the software source code, in accordance with certain non-limiting embodiments of the present technology;

[0055] Figure 4 depicts a flowchart diagram of a method for training a classifier used in one of the methods of Figures 1A and 1B for determining the affiliation of the given software with the target software, in accordance with certain non-limiting embodiments of the present technology;

[0056] Figure 5 depicts a schematic diagram of an example computing environment configurable for execution of one of the methods of Figures 1A, 1B, 3, and 4, in accordance with certain non-limiting embodiments of the present technology.

DETAILED DESCRIPTION

[0057] The following detailed description is provided to enable anyone skilled in the art to implement and use the non-limiting embodiments of the present technology. Specific details are provided merely for descriptive purposes and to give insights into the present technology, and in no way as a limitation. However, it would be apparent to a person skilled in the art that some of these specific details may not be necessary to implement certain non-limiting embodiments of the present technology. The descriptions of specific implementations are only provided as representative examples. Various modifications of these embodiments may become apparent to the person skilled in the art; the general principles defined in this document may be applied to other non-limiting embodiments and implementations without departing from the scope of the present technology.

[0058] Certain non-limiting embodiments of the present technology are directed to computer-implemented methods and systems for determining affiliation of given software to target software based on analyzing an associated software source code. In some non-limiting embodiments of the present technology the target software may include software of a predetermined software family and/or of a predetermined authorship.

[0059] According to certain non-limiting embodiments of the present technology, each one of the methods described herein below can be executed by a hardware processor – for example, a processor 501 of a computing device 500, which will be described below with reference to Figure 5.

Determining Affiliation to Target Software

[0060] With reference to Figure 1, there is depicted a flowchart diagram of a first method 100 for determining the affiliation of the given software with the target software, in accordance with certain non-limiting embodiments of the present technology.

STEP 110: RECEIVING, BY THE PROCESSOR, A SOFTWARE SOURCE CODE OF THE GIVEN SOFTWARE

[0061] The first method 100 commences at the step 110 where the processor 501 can be configured to receive a file containing a software source code of the given software for further analysis. According to certain non-limiting embodiments of the present technology, the file containing the software source code can be of various formats including, for example, without limitation, a *.cpp file format, a *.bas file format, as a project developed in one of a modern visual development environments, such as Visual Studio, and the like.

[0062] In some non-limiting embodiments of the present technology, if the given software is associated with a plurality of interrelated files (for example, it is a project in one of modern visual development environments), the processor 501 can be configured to select therefrom only files containing the software

source code. For example, if the project has been created in C/C++ programming language, the files containing the software source code can be *.cpp files; if the project has been created in assembler language, the files containing the software source code can be *.asm files, etc. The files containing the software source code of the given software could be selected, by the processor 501 using any method, and in some non-limiting embodiments of the present technology, by a script, which copies all project files with the above-specified extension into an input folder of the computing environment 500 implementing the first method 100.

[0063] The first method 100 thus advances to step 120.

STEP 120: DETERMINING, BY THE PROCESSOR, REFERENCE DATA ASSOCIATED WITH SOFTWARE SOURCE CODE

[0064] According to certain non-limiting embodiments of the present technology, at step 120, the processor 501 can be configured to analyze the software source code of the given software to determine certain reference data associated therewith. In some non-limiting embodiments of the present technology, the reference data may include a program platform and a programming language used to develop the software source code of the given software. In these embodiments, the platform could be determined, for example, by any known method. by brute-force searching; and the programming language could be determined, for example, by one of: a Linguist program (github.com/github/linguist) and by an Ohcount program (github.com/blackducksoftware/ohcount).

[0065] With reference to Figure 2A, there depicted a snippet 200 of the software source code of the given software, in accordance with certain non-limiting embodiments of the present technology. Analyzing the snippet 200 by the Linguist program, the processor 501 can be configured to determine that the software source code has been developed for the Win32 platform, in the JavaScript programming language.

[0066] According to certain non-limiting embodiments of the present technology, the processor 501 can be configured to conduct further analysis of the software source code using a function parser specifically preconfigured for the programming language, in which the software source code has been written. Thus, in the example of Figure 2A, the function parser, whose algorithm is based on JavaScript syntax, can be used for further analysis of the software source code.

[0067] The first method 100 hence advances to step 130.

STEP 130: PARSING, BY THE PROCESSOR, BASED ON THE REFERENCE DATA ASSOCIATED WITH SOFTWARE SOURCE CODE, THE SOFTWARE SOURCE CODE TO IDENTIFY THEREIN AT LEAST ONE FUNCTION

[0068] At step 130, using the function parser, the processor 501 can be configured to parse the software source code to identify therein at least one function and save it in a list of functions associated with the given software.

[0069] In the context of the present specification, the term “function” is to denote a portion of the software source code, which could be accessed within the software source code by reference thereto. In most cases, an identifier can be linked to a given function; however, many languages allow for anonymous functions. The address of the first (operator) instruction, being part of the given function, to which the control is passed when referring to the function, is inseparably associated with a name of the function. Having executed the given function, the control is returned back to a return address, i.e., to that portion of the software source code, from where the given function was called.

[0070] Thus, using the function parser, the processor 501 can be configured to determine boundaries, within portions of the software source code, which are associated with the functions within the software source code. For this purpose there could be used, for example, the commonly-known fact that, in the majority of high level programming languages, a function body of the given function is enclosed within a block operator (such as brackets, parentheses, and the like). Accordingly, in some non-limiting embodiments of the present technology, one of possible function parser operation algorithms of the function parser could include (1) detecting, in the software source code, pairs of words and/or symbol corresponding to a given block operator of the determined programming language, (2) verifying that the content within the block operator is the function body of the given function, and (3) storing the function body into a dedicated file of parsing results. In some non-limiting embodiments of the present technology, the verifying that the content within the block operator is the function body could be executed, for example, by searching, on the string preceding opening of the block operator, a symbolic expression corresponding to a function name in the syntax of the determined programming language.

[0071] In alternative non-limiting embodiments of the present technology, if the software source code is one of C-like programming languages, that is, for example, JavaScript, each function would begin with a function header having a predetermined format. In the example of Figure 2A, in the snippet 200 written in the JavaScript programming language, the function header could be identified by a key word "function" as labelled by numerals 210, 220, and 230. Each function header in the snippet 200 is followed by a respective function body, also enclosed in the block operator, which is, in this example, denoted by a pair of curly braces: {and}. For example, in Figure 2A the block operator is formed by braces 212 and 214, 222 and 224, 232 and 234. Accordingly, in this example, the processor 501 can be configured to identify the function headers, and subsequently, save portions of the software source code within the respective block operators following thereafter, into the list of functions associated with the given software. By so doing, the processor 501 may be configured to generate a list 250 depicted in Figure 2B,

according to certain non-limiting embodiments of the present technology. In the example of Figure 2B, the list 250 includes portions of the software source code of a first function 251, a second function 252, and a third function 252.

[0072] Further, in some non-limiting embodiments of the present technology, operations on remaining portions of the software source code of the given software beyond the block operators (i.e. the function boundaries) can be performed depending on whether the syntax of the programming language allows executing these portions. For example, in specific non-limiting embodiments of the present technology, if the software source code has been written in C or C#, the remaining portions thereof are ignored. In the example of Figures 2A to 2B, where the software source code has been developed in JavaScript, the remaining portions can be considered to be associated with another function, different from those having been previously determined by parsing. Such a function can be identified as a "main function" of the software source code and further processed analogously to all functions in the list 250. Thus, at step 130, the processor 501 can be configured to identify and obtain source code of the functions associated with the given software under analysis and save it the dedicated file.

[0073] Further, in some non-limiting embodiments of the present technology, the processor 501 may be further configured to identify, within the list 250 of functions associated with the given software, library functions and delete them therefrom. To that end, the processor 501 may be configured to apply a signature analysis. The library functions are the standard tools. They are widely used by a variety of programs, therefore, their presence in the software source code or in working processes is not specific to a certain software family and/or authorship. Excluding the library functions from the further analysis thus enables to significantly simplify the analysis and, at the same time, obtain better training results due to the fact that the decision rules are trained based on commands uniquely associated with the given software under analysis.

[0074] According to certain non-limiting embodiments of the present technology, the signature analysis and the deleting the library functions can be performed by the processor 501 executing an auxiliary script. An algorithm of the auxiliary script could represent, for example, a sequential comparison of each function within the list 250 with a prearranged set of signatures (regular expressions). Each of these signatures corresponds to a specific library function preliminarily described as a signature; when a function corresponding to any signature is detected, a whole portion of the software source code composing a function body and header of the function are deleted. Thus, in the example of Figure 2B, from the list 250 of function associated with the software source code, the second function 252 can be identified as a library function and thus will be deleted. Upon completion of processing by the auxiliary script, the processor 501 can be configured to update the list 250 of functions,

which after the deletion of the second function 252 has a view depicted in Figure 2C, according to certain non-limiting embodiments of the present technology.

[0075] The first method 100 thus proceeds to step 140.

STEP 140: PARSING, BY THE PROCESSOR, THE AT LEAST ONE FUNCTION TO IDENTIFY THEREIN AT LEAST ONE FUNCTION COMMAND

[0076] Referring back to Figure 1A, at step 150, according to certain non-limiting embodiments of the present technology, the processor 501 can be configured to identify, within a given function associated with the given software, at least one function command. According to these embodiments, the at least one function command may be associated with a respective action of the given software. Further, the processor 501 may be configured to identify, within the at least one function command, at least one argument associated with the respective action. Moreover, in alternative non-limiting embodiments of the present technology, strings, consisting of one, two or three symbols, for example, can be excluded from further processing.

[0077] According to certain non-limiting embodiments of the present technology, a particular configuration of the at least one function command depends on the syntax of the programming language, in which the software source code under analysis has been written. Thus, in some non-limiting embodiments of the present technology, a beginning of the at least one function command can be a beginning of a given string, and an end of the at least one function command could thus be an end command symbol in the programming language. For example, in those embodiments where the programming language is the C programming language, the command ending symbol can comprise a semicolon (;). In alternative non-limiting embodiments of the present technology, a predetermined symbol sequence located on a separate string could be considered as the at least one function command. Thus, for example, with reference to Figure 2D, in an other portion 270 of the software source code each of strings thereof respectively numbered 271, 272, 273, 274, 275, and 276 can be considered to be the at least one function command.

[0078] According to certain non-limiting embodiments of the present technology, the processor 501 may be configured to identify the respective action associated with the at least one function command and at least one argument (if any) associated therewith by applying a command parser specifically preconfigured to be used with the programming language, that is based on the syntax thereof including a predetermined set of symbols for developing commands within the software source code.

[0079] The command parser can thus be configured for sequentially comparing strings within the software source code with a predetermined set of regular expressions corresponding to possible actions in the programming language. If none of the predetermined set of regular expressions corresponds to

symbols of a given string under analysis, the given string is ignored. If the given string corresponds to a predetermined action, the command parser determines, in the given string, a presence of the at least one argument associated with the predetermined action, and further – its argument value.

[0080] A format of the at least one argument associated with the predetermined action, such as a decimal number, a symbol string in quotes, etc., is also defined by the syntax of the programming language and determined based by the one of the predetermined set of regular expression corresponding to the predetermined action. Thus, a given pair "action-argument" including the predetermined action and the at least one argument associated therewith can be saved in the file with the results. If the predetermined action is associated with the at least one argument having an improper format, the at least one argument can be assigned a zero value.

[0081] With reference to Figure 2E, the following example pairs "action-argument" respectively numbered 281 - 282, 283 - 284, 285 - 286, and 287 - 288. In the depicted example, a given pair "action-argument" 289 includes an argument with the zero value. Further, as it may become apparent, a non-zero argument could include more than one value as, for example, arguments 282, 286, and 288.

[0082] The first method 100 thus advances to step 150.

STEP 150: GENERATING, BY THE PROCESSOR, FOR THE AT LEAST ONE FUNCTION, BASED ON THE RESPECTIVE ACTION AND THE AT LEAST ONE ARGUMENT THEREOF, A RESPECTIVE ONE OF A PLURALITY OF FUNCTION IDENTIFIERS ASSOCIATED WITH THE GIVEN SOFTWARE

[0083] At step 150, according to certain non-limiting embodiments of the present technology, the processor 501 can be configured to generate, based on the pairs “action-argument” associated with the given function, a respective function identifier. To that end, the processor 501 may be configured to convert each pair "action-argument" associated with the given function associated with the given software into a respective number sequence, and concatenate respective number sequences associated with each of the pairs “action-argument” associated with the given function. In various non-limiting embodiments of the present technology, the respective number sequence associated with a given pair “action-argument” can be represented as a decimal number, a hexadecimal number, a binary number, and the like.

[0084] According to certain non-limiting embodiments of the present technology, the generating the respective number sequence can include applying, by the processor 501, one or more hash functions to the given pair “action-argument”. In some non-limiting embodiments of the present technology, the processor 501 may be configured to apply a first hash function to the respective action and a second hash function to the at least one argument associated therewith. In other non-limiting embodiments of the

present technology, the first hash function can be different from the second hash function. Further, the number sequences obtained through the application of the one or more hash functions can be concatenated, thereby generating the respective number sequence associated with the given pair “action-argument”. If the at least one argument includes more than one value, the one or more hash functions are applied to a string representing concatenated strings of all values of the at least one argument.

[0085] For example, in the depicted embodiments of Figures 2B and 2D, the first function 251 includes a first command 271, a second command 272, a third command 273, a fourth command 274, and a fifth command 275 of the first function 251. Further, the third function 253 includes a first command 276 of the third function 253.

[0086] Further, with reference to Figure 2E, the first command 271 of the first function 251 includes a first given action 281 and a first associated argument 282; the second command 272 of the first function 251 includes a second given action 283 and a second associated argument 284; the third command 273 of the first function 251 includes a third given action 285 and a third associated argument 286; finally, the fourth command 274 of the first function 251 includes a fourth given action 287 and a fourth associated argument 288. The first command 276 of the third function 253 includes a fifth given action 289 and an associated argument having a zero value.

[0087] Further, as noted above, using the one or more hash functions, for example, the processor 501 can be configured to convert the first command 271 of the first function 251 into a first hexadecimal number:

21812f4d00522bd21c014228f948063b18de4875ffe533b830f08a0326348a9160afafc8ada44db,

where c21812f4d00522bd21c014228f948063b18de487 is a result of taking the one or more hash functions from the first given action 281, and 5ffe533b830f08a0326348a9160afafc8ada44db is a result of taking the one or more hash functions from the first associated argument 282.

[0088] Similarly, the second command 272 of the first function 251 can be converted into a second hexadecimal number:

70233cdf6b130df09a0fd4fd84f056e08b1cf51636c6f169e67d0cf02fc3433b2112402804d9a6a3;

the third command 273 of the first function 251 can be converted into a third hexadecimal number:

eef70649fc977965dabbe7516c55f7951a40ca1513be4d3d8b43d0d4ecf268ff7c16fa7944139e99;

the fourth command 274 of the first function can be converted into a fourth hexadecimal number:

eef70649fc977965dabbe7516c55f7951a40ca1591c305309bf156bb2da46a2fed619951c852dd45;

and the first command 276 of the third function 253 can be converted into a fifth hexadecimal number:

bd4ee045946f5de9bba8059a9304ae5539d6730d0b6589fc6ab0dc82cf12099d1c2d40ab994e8410c.

[0089] In alternative non-limiting embodiments of the present technology, other approaches to converting the given pair "action-argument" pairs into the respective number sequence may be used. For example, the processor 501 can be configured to apply a serialization process before applying the one or more has functions to the at least one function command, which may allow saving information on an initial data type associated therewith. In this example, before hashing, a value associated with a given argument can be represented, for example, as a string, whose first byte codes an argument data type (such as integer, floating-point number, or zero, i.e. the zero argument), the second byte stores information about a string total length of the given argument, then a string of ASCII symbols associated with the given argument follows. Needless to mention that a given action may similarly be converted using the serialization process before hashing. For example, first, the processor 501 can be configured to convert, using the serialization process, the third given action 285 into the following string:

033E534f4654574152455c5c4d6963726f736f66745c5c57696e646f77735c5c43757272656e7456657273696f6e5c5c506f6c69636965735c5c53797374656d.

[0090] Second, the processor 501 can be configured to apply the one or more has function to the string above, thereby generating the following number sequence:

cc3c144bb04455bdaabcd9e199b1cdc9c5ed3d93b37b0b533f5c9b18a4137bbd.

[0091] Accordingly, the processor 501 can further be configured to generate the following number sequence associated with the third associated argument 286, in a similar way:

ffa1ace8dba62cdbc33d8c09ad07e46ccb0da176eeaf4cc67b4860c7d2bb531b.

[0092] Finally, as noted above, the processor 501 can be configured to concatenate the first hexadecimal number, the second hexadecimal number, the third hexadecimal number, and the fourth hexadecimal number respectively associated with the first command 271, the second command 272, the third command 273, and the fourth command 274 of the first function 251, thereby generating a first function identifier associated therewith:

c21812f4d00522bd21c014228f948063b18de4875ffe533b830f08a0326348a9160afafc8ada44db,
70233cdf6b130df09a0fd4fd84f056e08b1cf51636c6f169e67d0cf02fc3433b2112402804d9a6a3,
eef70649fc977965dabbe7516c55f7951a40ca1513be4d3d8b43d0d4ecf268ff7c16fa7944139e99,
eef70649fc977965dabbe7516c55f7951a40ca1591c305309bf156bb2da46a2fed619951c852dd45.

[0093] Similarly, the processor 501 can be configured to generate a third function identifier associated with the third function 253:

bd4ee045946f5de9bba8059a9304ae5539d6730d0b6589fc6ab0dc82cf12099d1c2d40ab994e8410c.

[0094] Thus, in accordance with certain non-limiting embodiments of the present technology, each one of the first function 251 and the third function identified, by the processor 501, within the software source code associate with the given software, is respectively associated with the first function identifier and the third function identifier.

[0095] The first method 100 thus proceeds to step 160.

STEP 160: AGGREGATING, BY THE PROCESSOR, THE PLURALITY OF FUNCTION IDENTIFIERS ASSOCIATED WITH THE GIVEN SOFTWARE INTO AN AGGREGATED ARRAY OF FUNCTION IDENTIFIERS GIVEN AN ORDER OF OCCURRENCE OF ASSOCIATED FUNCTIONS WITHIN THE SOFTWARE SOURCE CODE

[0096] At step 160, in some non-limiting embodiments of the present technology, the processor 501 may be configured to aggregate the first function identifier and the third function identifier in an order of appearance of the first function 251 and the third function 253 within the list 250 associated with the software source code of the given software. To that end, each one of the first function identifier and the third function identifier may be represented as P_{ij}, where i indicates a sequential number of a given one of the first function 251 and the second function 253, in which a given "action-argument" pair has been identified, and j indicates a sequential number of the given "action-argument" pair within the given function, in which the given pair has been detected.

[0097] As it may become apparent, software source codes of the majority of modern software programs can include a considerable number of functions. Therefore, in certain non-limiting embodiments of the present technology, the software source code of the given software can be transformed into an aggregated array of function identifiers including n lines, wherein n is indicative of a total number of non-library functions having been identified during the above analysis:

F1 = P11, P12, P13,... P1j..., P1a

...

$$\begin{aligned}
F_i &= P_{i1}, P_{i2}, P_{i3}, \dots P_{ij}, \dots, P_{ib} \\
&\dots \\
F_n &= P_{n1}, P_{n2}, P_{n3}, \dots P_{nj}, \dots, P_{nc}
\end{aligned} \tag{1}$$

As it can be appreciated from the above, indices a, b, and c in Equation (1) are indicative of different numbers pairs "action-argument" within respective functions, in a general case.

[0098] The first method 100 thus advances to step 170.

STEP 170: APPLYING, BY THE PROCESSOR, AT LEAST ONE CLASSIFIER TO THE AGGREGATED ARRAY OF FUNCTION IDENTIFIERS ASSOCIATED WITH THE GIVEN SOFTWARE TO DETERMINE A LIKELIHOOD PARAMETER INDICATIVE OF THE GIVEN SOFTWARE BEING AFFILIATED TO A RESPECTIVE TARGET SOFTWARE

[0099] At step 170, according to certain non-limiting embodiments of the present technology, the processor 501 can be configured to analyze the aggregated array of function identifiers expressed by Equation (1) to determine the affiliation of the given software with the target software. To that end, the processor 501 can be configured to feed the aggregated array of function identifiers into one or more classifiers having been trained to determine the affiliation with the target software. How the one or more classifiers can be trained, in accordance with certain non-limiting embodiments of the present technology, will be described below with reference to Figure 4.

[0100] Further, in accordance with certain non-limiting embodiments of the present technology, the one or more classifiers, when applied to the aggregated array of function identifiers, may be configured to generate a likelihood parameter, which may be expressed, for example, as a numerical estimate of probability, that the given software is affiliated with the target software. The likelihood parameter can be constantly updated, i.e. reviewed at input of each portion of the software source code represented by Equation (1).

[0101] The first method 100 hence proceeds to step 180.

STEP 180: IN RESPONSE TO THE LIKELIHOOD PARAMETER BEING EQUAL TO OR GREATER THAN A PREDETERMINED LIKELIHOOD PARAMETER THRESHOLD: IDENTIFYING THE GIVEN SOFTWARE AS BEING AFFILIATED TO THE RESPECTIVE TARGET SOFTWARE; STORING DATA INDICATIVE OF THE GIVEN SOFTWARE IN A DATABASE OF AFFILIATED SOFTWARE; AND USING THE DATA INDICATIVE OF THE GIVEN SOFTWARE FOR FURTHER DETERMINING AFFILIATION TO THE GIVEN TARGET SOFTWARE

[0102] At step 180, in response to the likelihood parameter generated by the one or more classifiers being equal to or greater than a predetermined likelihood parameter threshold value, the processor 501 can be configured to identify the given software as being affiliated to the target software. As noted hereinabove, in some non-limiting embodiments of the present technology, the target software may include software of a predetermined software family and/or of a predetermined authorship.

[0103] Further, in certain non-limiting embodiments of the present technology, the processor 501 may be configured to store data indicative of the given software in a dedicated database for further use. For example, the processor 501 can be configured to use the data indicative of the given software to train the one or more classifiers to determine affiliation of other software to the target software.

[0104] In other non-limiting embodiments of the present technology, where the likelihood parameter is below the predetermined likelihood parameter threshold value, the processor 501 can be configured to determine that the given software is not affiliated to the target software; and thus, the processor 501 would not proceed to store the data indicative of the given software for further use.

[0105] The first method 100 thus terminates.

[0106] According to certain non-limiting embodiments of the present technology, the software source code of the given software can be obfuscated complicating the analysis thereof according to the first method 100. However, according to certain non-limiting embodiments of the present technology, the processor 501 may be configured to determine a specific obfuscation technique having been used for obfuscating the given software. To that end, for example, the processor 501 may be configured to apply (or otherwise have access to) a respective deobfuscator, i.e. a preliminarily prepared program intended to restore initial view of the software source code obfuscated by the specific obfuscation technique.

[0107] However, in accordance with other non-limiting embodiments of the present technology, the processor 501 may not be configured to determine the specific obfuscation technique having been used (for example, due to unavailability of the respective deobfuscator). Accordingly, the processor 501 may be configured to execute a second method 101 for determining the affiliation of the given software to the target software, whose flowchart diagram is depicted in Figure 1B and to which the reference is currently being made.

STEP 111: RECEIVING, BY THE PROCESSOR, A SOFTWARE SOURCE CODE OF THE GIVEN SOFTWARE

[0108] The second method 101 commences at step 111 where the processor 501 can be configured to receive the software source code associated with the given software. According to certain non-limiting

embodiments of the present technology, step 111 is substantially analogous to step 110 of the first method 100 described above.

[0109] The second method 101 further proceeds to step 121.

STEP 121: EXECUTING, BY THE PROCESSOR, THE SOFTWARE SOURCE CODE IN AN ISOLATED PROGRAM ENVIRONMENT TO IDENTIFY AT LEAST ONE OF A PLURALITY OF OUTGOING REQUESTS OF THE GIVEN SOFTWARE

[0110] Further, according to certain non-limiting embodiments of the present technology, at step 121, similar to step 120 of the first method 100 described above, the processor 501 can be configured to determine the reference data, including the program platform and the programming language associated with the software source code of the given software. Further, in certain non-limiting embodiments of the present technology, the processor 501 can be configured to execute the software source code in an isolated program environment including functionality for logging. As it can be appreciated, in these embodiments, the software source code may have been developed in one of interpreted programming languages. In various non-limiting embodiments of the present technology, the interpreted programming languages may include Python, Perl, PHP, Ruby, JavaScript, and the like. For example, if the software source code has been developed in PowerShell, the processor 501 can be configured to use built-in logging facilities of Windows 10 operating system; alternatively, the processor 501 may be configured to use specialized Log4j logging library.

[0111] Accordingly, by executing the software source code in the isolated program environment, the processor 501 may be configured to identify at least one of a plurality of outgoing requests generated by the software source code. For example, the processor 501 may be configured to receive (and thus further identify) the plurality of outgoing requests including a COM object and an ActiveX control element call to a given framework. Further, in some non-limiting embodiments of the present technology, the plurality of outgoing requests may include program interrupt signals and system function calls to a given platform. The so received and identified plurality of outgoing requests can further be saved by the processor 501, for example, as a text file.

[0112] With reference to Figure 2F, there is depicted a fragment of the plurality of outgoing requests 290 to the given framework and the given platform, in accordance with certain non-limiting embodiments of the present technology. In the depicted embodiments, the plurality of outgoing requests includes two outgoing requests: a first outgoing request 291 and a second outgoing request 296.

[0113] Further, in accordance with certain non-limiting embodiments of the present technology, the processor 501 can be configured to identify, within each one of the first outgoing request 291 and the second outgoing request 296, a respective pair “action-argument”. For example, with continued

reference to Figure 2F, each one of the first outgoing request 291 and the second outgoing request 296 respectively include the following elements: (1) a first record sequence number 292 and a second record sequence number 297; (2) a first request function 293 and a second request function 298 having been called upon the execution of the software source code; (3) request statuses 294; and (4) a first request argument 295 and a second request argument 299 respectively associated with the first request function 293 and the second request function 298. Therefore, in this example, the processor 501 can be configured to parse the plurality of outgoing requests 290, thereby identifying individual outgoing requests, that is, the first outgoing request 291 and the second outgoing request 296, as portions of records logged by the isolated program environment and enclosed within record sequence numbers – that is, the first record sequence number 292 and the second record sequence number 297, respectively.

[0114] Thus, for each of the first outgoing request 291 and the second outgoing request 296, the processor 501 can be configured to identify a first pair "action-argument" including the first request function 293 and the first request argument 295; and a second pair "action-argument" including the second request function 298 and the second request argument 299. Further, the processor 501 can be configured to save the pairs "action-argument" in a dedicated file. Similar to the above description in respect of the first method 100, if the processor 501 has identified a given action whose argument is absent or has an improper format, the processor 501 can be configured to generate a respective pair "action-argument" where the argument has a zero value.

[0115] The method thus proceeds to step 131.

STEP 131: GENERATING, FOR THE AT LEAST ONE OF THE PLURALITY OF OUTGOING REQUESTS, BASED ON THE RESPECTIVE ACTION AND THE AT LEAST ONE ARGUMENT, A RESPECTIVE ONE OF A PLURALITY OF IDENTIFIERS ASSOCIATED WITH THE GIVEN SOFTWARE

[0116] According to certain non-limiting embodiments of the present technology, at step 131, the processor 501 can be configured to generate, for each one of the plurality of outgoing requests 290, a respective request identifier associated with the respective pair "action-argument". For example, the processor 501 can be configured to convert the respective pair "action-argument" into a respective number sequence as described above with respect to step 150 of the first method 100, that is, using the one or more hash functions.

[0117] Thus, in some non-limiting embodiments of the present technology, the processor 501 can be configured to generate, for the first pair "action-argument", based on the first request function 293 and the first request argument 295, a first request hexadecimal number, thereby generating a first request identifier:

00b66f89be96d7fa40e584d7eb9728f9b43e29d241d0544c3f658c6b193d81f7e5e7201d0c02ef3a,

where 00b66f89be96d7fa40e584d7eb9728f9b43e29d2 is a result of taking the one or more hash functions from the first request function 293 (that is, a first action), and 41d0544c3f658c6b193d81f7e5e7201d0c02ef3a is a result of taking the one or more hash functions from the first request argument 295. Similarly, the processor 501 can be configured to convert the second pair “action-argument” comprising the first request function 295 (that is, a second action) and the second request argument 299 into a second request hexadecimal number, thereby generating a second request identifier:

18ede0448e4ad01e04175beb0c5216977ddf8b74a94d5f2d650020e4467fa02e979c295ab252f415.

[0118] Similar to the above-described method 100 with respect to the step 150, the processor 501 can be configured to apply more complicated conversion techniques to each one of the pairs “action-argument” into respective number sequences including serialization, as an example.

[0119] The second method 101 hence advances to step 141.

STEP 141: AGGREGATING, BY THE PROCESSOR, THE PLURALITY OF IDENTIFIERS ASSOCIATED WITH THE GIVEN SOFTWARE INTO AN AGGREGATED ARRAY OF IDENTIFIERS GIVEN A TIME ORDER OF IDENTIFYING ASSOCIATED OUTGOING REQUESTS

[0120] At step 141, akin to step 160 of the first method 100 described above, the processor 501 can be configured to aggregate request identifiers associated with respective pairs “action-arguments” in an order of appearance of each one of the plurality of outgoing requests 290 associated therewith, thereby generating an aggregated array of request identifiers.

[0121] The second method 101 thus proceeds to step 151.

STEP 151: APPLYING, BY THE PROCESSOR, AT LEAST ONE CLASSIFIER TO THE AGGREGATED ARRAY OF IDENTIFIERS TO DETERMINE A LIKELIHOOD PARAMETER INDICATIVE OF THE GIVEN SOFTWARE BEING AFFILIATED TO A RESPECTIVE TARGET SOFTWARE

[0122] At step 151, according to certain non-limiting embodiments of the present technology, the processor 501 can be configured to feed the aggregated array of request identifiers into one or more classifiers having been trained to determine the affiliation to the target software based on request identifiers generated as described above.

[0123] Accordingly, in certain non-limiting embodiments of the present technology, the one or more classifiers can be configured to generate a request likelihood parameter associated with the aggregated

array of request identifiers. Further, the processor 501 can be configured to determine, based on the request likelihood parameter, the affiliation of the given software to the target software.

[0124] The second method 101 thus proceeds to step 161.

STEP 161: IN RESPONSE TO THE LIKELIHOOD PARAMETER BEING EQUAL TO OR GREATER THAN A PREDETERMINED LIKELIHOOD PARAMETER THRESHOLD: IDENTIFYING THE GIVEN SOFTWARE AS BEING AFFILIATED TO THE RESPECTIVE TARGET SOFTWARE; STORING DATA INDICATIVE OF THE GIVEN SOFTWARE IN A DATABASE OF AFFILIATED SOFTWARE; AND USING THE DATA INDICATIVE OF THE GIVEN SOFTWARE FOR FURTHER DETERMINING AFFILIATION TO THE GIVEN TARGET SOFTWARE

[0125] At step 161, in response to the request likelihood parameter generated by the one or more classifiers being equal to or greater than a predetermined request likelihood parameter threshold value, the processor 501 can be configured to identify the given software as being affiliated to the target software. As noted hereinabove, in some non-limiting embodiments of the present technology, the target software may include software of a predetermined software family and/or of a predetermined authorship.

[0126] Further, in certain non-limiting embodiments of the present technology, the processor 501 may be configured to store data indicative of the given software in the dedicated database for further use. For example, the processor 501 can be configured to use the data indicative of the given software to train the one or more classifiers to determine affiliation of other software to the target software.

[0127] In other non-limiting embodiments of the present technology, where the request likelihood parameter is below the predetermined request likelihood parameter threshold value, the processor 501 can be configured to determine that the given software is not affiliated to the target software; and thus, the processor 501 would not proceed to store the data indicative of the given software for further use.

[0128] The second method 101 thus terminates.

[0129] In some non-limiting embodiments of the present technology, after executing steps 130 and 131 of the first method 100 and the second method 101, respectively, each of the methods may further comprise an additional step of identifying, within the software source code, and deleting therefrom source code portions inherently non-indicative of the affiliation to the target software. In the context of the present specification, source code portions can be inherently non-indicative of the affiliation to the target software, by virtue of the portions of the software source code of the given software not being specific enough for further analysis of the given software for the affiliation thereof to the target software (for example, due to the fact that they often occur in different software and, thus, are not indicative

enough of affiliation of any given software); and thus these source code portions can be omitted from the software source code without affecting the accuracy of such analysis. For example, inherently non-indicative source code portions may include function prologues of functions within the software source code.

[0130] To that end, the processor 501 can be configured to execute a third method 300, a flowchart diagram of which is depicted in Figure 3, and to which reference is currently being made.

STEP 310: RECEIVING A PLURALITY OF SOFTWARE SOURCE CODE SAMPLES

[0131] The third method 300 commences at step 310 with the processor 501 being configured to receive a plurality of software source code samples associated with the given software. In certain non-limiting embodiments of the present technology, the plurality of software source code samples may include, for example hundreds, thousands, or even hundreds of thousands of software source code samples associated with the given software, however, different in functionality and having been developed by different teams of developers.

[0132] The third method 300 further proceeds to step 320.

STEP 320: IDENTIFYING A LIST OF INHERENTLY NON-INDICATIVE SOURCE CODE PORTIONS AND DETERMINE FREQUENCY OF OCCURRENCE THEREOF WITHIN THE PLURALITY OF SOURCE CODE SAMPLES

[0133] At step 320, according to certain non-limiting embodiments of the present technology, the processor 501 can be configured to identify, within the software source code, inherently non-indicative source code portions repeatedly occurred therewithin. In some non-limiting embodiments of the present technology, the processor 501 can further be configured to determine a frequency of occurrence of each inherently non-indicative source code portion within the plurality of source code samples. For example, in some non-limiting embodiments of the present technology, the processor 501 can be preliminarily provided with a minimum sequence length value, e.g., 20 symbols, for identifying the inherently non-indicative source code portions within the plurality of software source code samples. In some non-limiting embodiments of the present technology, a maximum sequence length value may not be preset. Thus, the processor 501 can be configured to generate a list of inherently non-indicative source code portions with associated respective frequencies of occurrences thereof.

[0134] The third method 300 thus proceeds to step 330.

STEP 330: SELECTING A MOST FREQUENT INHERENTLY NON-INDICATIVE SOURCE CODE PORTION WITHIN THE LIST OF INHERENTLY NON-INDICATIVE SOURCE CODE PORTIONS

[0135] Further, at step 330, the processor 501 can be configured to select, from the list of inherently non-indicative source code portions generated at the previous steps, based on the associated respective frequencies of occurrences, a most frequent inherently non-indicative source code portion. For example, a given inherently non-indicative source code portion has been occurred once in each one of the plurality of software source code samples including, for example, 100 software source code samples, which amounts to 100 occurrences of the given inherently non-indicative source code portions therewithin.

[0136] Also, in some non-limiting embodiments of the present technology, the processor 501 can be configured to select a sub-plurality of most frequent inherently non-indicative source code portions based on a predetermined frequency threshold value, which can be determined, for example, based on a number of software source code samples within the plurality of software source code samples including the given inherently non-indicative source code portion.

[0137] The third method 300 thus advances to step 340.

STEP 340: DELETING THE MOST FREQUENT INHERENTLY NON-INDICATIVE SOURCE CODE PORTION FROM EACH ONE OF THE PLURALITY OF SOFTWARE SOURCE CODE SAMPLES

[0138] At step 340, according to certain non-limiting embodiments of the present technology, the processor 501 may be configured to delete the most frequent inherently non-indicative source code portion identified at step 330 from each one of the plurality of software source code samples associated with the given software.

[0139] Thus, by executing the third method 300, the processor 501 can be configured to delete the inherently non-indicative source code portions from the software source code associated with the given software. Accordingly, in some non-limiting embodiments of the present technology, upon completion of the third method 300, the processor 501 can be configured to proceed to steps 140 and 141 of the first method 100 and the second method 101, respectively.

[0140] The third method 300 thus terminates.

Training Classifier

[0141] As alluded to hereinabove, according to certain non-limiting embodiments of the present technology, the processor 501 can be configured to train the one or more classifier to determine the affiliation with respective target software, which may further be used in the first method 100 and method 101. With reference now to Figure 4, there is depicted a flowchart diagram of a fourth method 400 for training a classifier to determine the affiliation with a given target software, in accordance with certain non-limiting embodiments of the present technology.

STEP 410: RECEIVING, BY THE PROCESSOR, A PLURALITY OF TARGET SOFTWARE SOURCE CODES ASSOCIATED WITH THE GIVEN TARGET SOFTWARE

[0142] The fourth method 400 commences at step 410 with the processor 501 being configured to receive a plurality of target software source codes associated with the given target software. According to certain non-limiting embodiments of the present technology, a total number of software source codes in the plurality thereof can be predetermined and comprise, for example, without limitation around 30-70 software source codes of a predetermined software family to which the given target software belongs. Alternatively, the processor 501 can be configured to receive around 20-30 software source codes of a predetermined authorship associated with the given target software. Further, the processor 501 can be configured to analyze each one of the plurality of target software source codes, for example, sequentially.

[0143] The fourth method 400 hence advances to step 420.

STEP 420: DETERMINING, BY THE PROCESSOR, FOR EACH ONE OF THE PLURALITY OF TARGET SOFTWARE SOURCE CODES, RESPECTIVE REFERENCE DATA

[0144] At step 420, according to certain non-limiting embodiments of the present technology, the processor 501 can be configured to determine a program platform and a programming language of each one of the plurality of target software source codes associated with the given target software. In these embodiments, the processor 501 can be configured to execute step 420 similar to executing step 120 of the first method 100 described above.

[0145] The fourth method 400 thus proceeds to step 430.

STEP 430: IDENTIFYING, BY THE PROCESSOR, IN EACH ONE OF THE PLURALITY OF TARGET SOFTWARE SOURCE CODES, BASED ON THE RESPECTIVE REFERENCE DATA, AT LEAST ONE TARGET FUNCTION

[0146] At step 430, the processor 501 can be configured to parse each one of the plurality of target software source codes to identify therein at least one target function. In certain non-limiting embodiments of the present technology, the processor 501 can be configured to parse a given one of the plurality of target software source codes using the function parser configured for parsing a given target software source code based on reference data associated therewith – that is, a programming language and a program platform associated therewith. Thus, in these embodiments, the processor 501 can be configured to execute step 430 similar to executing step 130 of the method 110. The fourth method 400 thus proceeds to step 440.

STEP 440: PARSING, BY THE PROCESSOR, THE AT LEAST ONE TARGET FUNCTION TO IDENTIFY THEREIN AT LEAST ONE TARGET FUNCTION COMMAND

[0147] Akin to executing step 140 of the first method 100 described above, at step 440, the processor 501 can be configured to parse the at least one target function to identify therein at least one target function command. Accordingly, as described above, the at least one function command may further comprise at least one target pair “action-argument” including a given target action and a target argument associated therewith.

[0148] Further, in some non-limiting embodiments of the present technology, the processor 501 can be configured to identify, in the given one of the plurality of target software source codes, library functions and delete them therefrom, as described above with respect to step 130 of the first method 100.

[0149] In other non-limiting embodiments of the present technology, the processor 501 can further be configured to identify, in the given one of the plurality of target software source codes, inherently non-indicative source code portions and delete them therefrom, as described above with reference to Figure 3 in respect of the third method 300.

[0150] The fourth method 400 thus proceeds to step 450.

STEP 450: GENERATING, BY THE PROCESSOR, BASED ON EACH OF THE AT LEAST ONE TARGET FUNCTION COMMAND, A RESPECTIVE TARGET FUNCTION IDENTIFIER ASSOCIATED WITH THE AT LEAST ONE TARGET FUNCTION, THE RESPECTIVE TARGET FUNCTION IDENTIFIER COMPRISING AN ASSOCIATED NUMBER SEQUENCE

[0151] At step 450, according to certain non-limiting embodiments of the present technology, the processor 501 can be configured to generate for the at least one target pair “action-argument” a respective target function identifier. In some non-limiting embodiments of the present technology, the processor 501 can be configured to generate the respective target function identifier as a respective number sequence, similar to generating the respective function identifier as described above in respect of step 150 of the first method 100.

[0152] Further, the processor 501 can be configured to save the respective target function identifier associated with the at least one target pair “action-argument” for further use.

[0153] The fourth method 400 further advances to step 460.

STEP 460: AGGREGATING, BY THE PROCESSOR, NUMBER SEQUENCES ASSOCIATED WITH RESPECTIVE TARGET FUNCTIONS OVER THE PLURALITY OF TARGET SOFTWARE SOURCE CODES, THEREBY GENERATING A NUMBER ARRAY ASSOCIATED WITH THE RESPECTIVE TARGET SOFTWARE

[0154] At step 460, according to certain non-limiting embodiments of the present technology, the processor 501 can be configured to aggregate target function identifiers over the plurality of target software source codes to generate a target number array associated with the given target software. For example, the processor 501 can be configured to aggregate the target function identifiers in an order of occurrence of functions associated therewith as described above with respect to step 160 of the first method 100.

[0155] The method thus proceeds to step 470.

STEP 470: IDENTIFYING, BY THE PROCESSOR, IN THE NUMBER ARRAY ASSOCIATED WITH THE GIVEN TARGET SOFTWARE, AT LEAST ONE PATTERN

[0156] At step 470, according to certain non-limiting embodiments of the present technology, the processor 501 can be configured to identify, in the target number array, at least one pattern associated with the given target software. In some non-limiting embodiments of the present technology, the at least one pattern comprises a predetermined repetitive number sequence within the target number array. Thus, in these embodiments, the predetermined repetitive number sequence can be said to be indicative of a frequency of occurrence of the at least one target pair “action-argument” within the given target software.

[0157] According to certain non-limiting embodiments of the present technology, a length of the predetermined repetitive number sequence, i.e. a number of symbols therein, can be predetermined. Thus, in some non-limiting embodiments of the present technology, the length of the predetermined repetitive number sequence could be based on an interval, for example, from 4 to 10 symbols within the target number array or, alternatively, for example, from 60 to 80 symbols within the target number array. In other non-limiting embodiments of the present technology, the length of the predetermined repetitive number sequence could be predetermined as a constant number, e.g. 40 symbols within the target number array associated with the given target software.

[0158] In yet other non-limiting embodiments of the present technology, the length of the predetermined repetitive number sequence could be determined iteratively, based on a current number of such a predetermined repetitive number sequence within the aggregated array of target identifiers. In these embodiments, a search begins, for example, at an initial length of 8 symbols. Once a number of identified number sequences of the initial length exceeds a predetermined pattern threshold value (100, as an example), the initial length increases by one, and the search starts over omitting shorter number sequences detected before. Such cycle is repeated until the number of patterns of a maximum possible length less than the predetermined pattern threshold value is identified. Thus, the at least one pattern may further be part of a training set of data for training the classifier.

[0159] In some non-limiting embodiments of the present technology, the processor 501 can be configured to assign to the at least one pattern a respective weight value.

[0160] In some non-limiting embodiments of the present technology, the respective weight value can be determined based on types of commands and operations associated with the at least one pattern. For example, the respective weight value can be 2 times exceeding respective weight values of other patterns the at least one pattern is indicative of commands at least one math operation. In another example, the respective weight value can be, e.g. 3 times exceeding weight values of other patterns, if the at least one pattern is indicative of at least 2 string constants.

[0161] By contrast, the respective weight value can be decreased, e.g. can comprise 0.3 of weight values of other patterns, if the at least one pattern includes symbols indicative of neither commands with math operations nor string constants.

[0162] Further, in some non-limiting embodiments of the present technology, the processor 501 can be configured to determine a frequency of occurrence of the at least one pattern within the target number array. Broadly speaking, the frequency of occurrence of the at least one pattern can be a numeric value indicating how often the at least one pattern occurs in the plurality of target software source codes associated with the given target software, i.e. how often an associated set of commands occurs within the given target software.

[0163] In some non-limiting embodiments of the present technology, the frequency of occurrence of the at least one pattern can be determined according to the following equation:

$$\lambda = L/K, \quad (2)$$

where L is a frequency of occurrence of the at least one pattern within the target number array of target identifiers associated with the given target software, and K is a number of software source codes in the plurality of target software source codes associated with the given target software.

[0164] In some non-limiting embodiments of the present technology, the respective weight value to be assigned to the at least one pattern may be based on the frequency of occurrence thereof determined in accordance with Equation (2).

[0165] The fourth method 400 thus proceeds to step 480.

STEP 480: STORING THE AT LEAST ONE PATTERN, THEREBY GENERATING A TRAINING SET OF DATA

[0166] Further, at step 480, in some non-limiting embodiments of the present technology, the processor 501 can be configured to assign the at least one pattern with a label indicative of an association between

the at least one pattern and the given target software. Thus, the processor 501 can be configured to store the at least one pattern associated with the label and the respective weight value in the training set of data used for training the classifier.

[0167] The fourth method 400 finally advances to step 490.

STEP 490: TRAINING THE CLASSIFIER, BASED ON THE TRAINING SET OF DATA, TO DETERMINE THE AFFILIATION OF A GIVEN SOFTWARE TO THE GIVEN TARGET SOFTWARE

[0168] At step 490, the processor 501 can be configured to train the classifier, based on the so generated training set of data, to determine the affiliation to the given target software. It should be expressly understood that it is not limited as to how the classifier can be implemented, and in various non-limiting embodiments of the present technology, the classifier can be implemented, for example, as one of a probabilistic graph model (Random Forest) and as a SVM-classifier.

[0169] In specific non-limiting embodiments of the present technology, the processor 501 can be configured to train the classifier using one or more machine-learning techniques.

[0170] The fourth method 400 thus terminates.

[0171] As noted hereinabove, according to certain non-limiting embodiments of the present technology, the classifier trained in accordance with the fourth method 400 may thus be used at step 170 of the first method 100 for determining the affiliation of the given software to the target software.

[0172] It should be expressly understood that a similar method for training the classifier to be used in the second method 101, that is, based on outgoing requests generated by the given target software executed in the isolated program environment, and whose target software source codes are obfuscated, may be devised, *mutatis mutandis*, without departing from the scope of the present technology. Thus, the classifier trained in accordance with a such a method can be used at step 151 for determining the affiliation of the given software to the target software.

Computing Environment

[0173] With reference to Figure 5, there is depicted an example functional diagram of the computing device 500 configurable to implement certain non-limiting embodiments of the present technology including the first method 100, the second method 101, the third method 300, and the fourth method 400 described above.

[0174] In some non-limiting embodiments of the present technology, the computing device 500 may include: the processor 501 comprising one or more central processing units (CPUs), at least one non-

transitory computer-readable memory 502 (RAM), a storage 503, input/output interfaces 504, input/output means 505, data communication means 506.

[0175] According to some non-limiting embodiments of the present technology, the processor 501 may be configured to execute specific program instructions the computations as required for the computing device 500 to function properly or to ensure the functioning of one or more of its components. The processor 501 may further be configured to execute specific machine-readable instructions stored in the at least one non-transitory computer-readable memory 502, for example, those causing the computing device 500 to execute one of the first method 100, the second method 101, the third method 300, and the fourth method 400.

[0176] In some non-limiting embodiments of the present technology, the machine-readable instructions representative of software components of disclosed systems may be implemented using any programming language or scripts, such as C, C++, C#, Java, JavaScript, VBScript, Macromedia Cold Fusion, COBOL, Microsoft Active Server Pages, Assembly, Perl, PHP, AWK, Python, Visual Basic, SQL Stored Procedures, PL/SQL, any UNIX shell scripts or XML. Various algorithms are implemented with any combination of the data structures, objects, processes, procedures and other software elements.

[0177] The at least one non-transitory computer-readable memory 502 may be implemented as RAM and contains the necessary program logic to provide the requisite functionality.

[0178] The storage 503 may be implemented as at least one of an HDD drive, an SSD drive, a RAID array, a network storage, a flash memory, an optical drive (such as CD, DVD, MD, Blu-ray), etc. The storage 503 may be configured for long-term storage of various data, e.g., the aforementioned documents with user data sets, databases with the time intervals measured for each user, user IDs, etc.

[0179] The input/output interfaces 504 may comprise various interfaces, such as at least one of USB, RS232, RJ45, LPT, COM, HDMI, PS/2, Lightning, FireWire, etc.

[0180] The input/output means 505 may include at least one of a keyboard, joystick, (touchscreen) display, projector, touchpad, mouse, trackball, stylus, speakers, microphone, and the like. A communication link between each one of the input/output means 505 can be wired (for example, connecting the keyboard via a PS/2 or USB port on the chassis of the desktop PC) or wireless (for example, via a wireless link, e.g., radio link, to the base station which is directly connected to the PC, e.g., to a USB port).

[0181] The data communication means 506 may be selected based on a particular implementation of the network 120, and may comprise at least one of: an Ethernet card, a WLAN/Wi-Fi adapter, a

Bluetooth adapter, a BLE adapter, an NFC adapter, an IrDa, a RFID adapter, a GSM modem, and the like. As such, the connectivity hardware 404 may be configured for wired and wireless data transmission, via one of WAN, PAN, LAN, Intranet, Internet, WLAN, WMAN, or GSM networks.

[0182] These and other components of the computing device 500 may be linked together using a common data bus 510.

[0183] It should be expressly understood that not all technical effects mentioned herein need to be enjoyed in each and every embodiment of the present technology.

[0184] Modifications and improvements to the above-described implementations of the present technology may become apparent to those skilled in the art. The foregoing description is intended to be exemplary rather than limiting. The scope of the present technology is therefore intended to be limited solely by the scope of the appended claims.

CLAIMS

1. A method for determining an affiliation of a given software with a target software, the method being executable by a processor, the method comprising:

receiving, by the processor, a software source code of the given software;

determining, by the processor, reference data associated with the software source code;

parsing, by the processor, based on the reference data associated with the software source code, the software source code to identify therein at least one function,

the identifying comprising identifying, in the software source code, library functions and deleting the library function by applying, by the processor, a signature analysis to the software source code;

parsing, by the processor, the at least one function to identify therein at least one function command,

the at least one function command comprising a respective action and at least one argument associated with the respective action;

generating, by the processor, for the at least one function, based on the respective action and the at least one argument thereof, a respective one of a plurality of function identifiers associated with the given software,

the generating comprising applying, by the processor, a hash function to the respective action and to each value of the at least one argument associated therewith to generate respective hash function values; and

aggregating the respective hash function values;

aggregating, by the processor, the plurality of function identifiers associated with the given software into an aggregated array of function identifiers given an order of occurrence of associated functions within the software source code;

applying, by the processor, at least one classifier to the aggregated array of function identifiers associated with the given software to determine a likelihood parameter indicative of the given software being affiliated to a respective target software,

the at least one classifier having been trained to determine the affiliation with the respective target software;

in response to the likelihood parameter being equal to or greater than a predetermined likelihood parameter threshold:

identifying the given software as being affiliated to the respective target software;

storing data indicative of the given software in a database of affiliated software, wherein the data comprises the aggregated array of function identifiers; and

using the data indicative of the given software for further determining affiliation of an other software to the respective target software.

2. The method of claim 1, wherein the identifying the library functions further comprises identifying, in the software source code, source code portions inherently non-indicative of the affiliation to the target software, and deleting the source code portions inherently non-indicative of the affiliation to the target software from the software source code.

3. The method of claim 1, wherein the applying the hash function further comprises applying a first hash function to the respective action and applying a second hash function to the at least one argument, the second hash function being different from the first hash function.

4. The method of claim 1, wherein, the generating the respective one of the plurality of identifiers further comprises applying a serialization process to the respective action and the at least one argument associated therewith.

5. A method for training a classifier to determine an affiliation with a given target software, the method being executable by a processor, the method comprising:

receiving, by the processor, a plurality of target software source codes associated with the given target software;

determining, by the processor, for each one of the plurality of target software source codes, respective reference data;

identifying, by the processor, in each one of the plurality of target software source codes, based on the respective reference data, at least one target function;

parsing, by the processor, the at least one target function to identify therein at least one target function command;

generating, by the processor, based on each of the at least one target function command, a respective target function identifier associated with the at least one target function, the respective target function identifier comprising an associated number sequence;

aggregating, by the processor, number sequences associated with respective target functions over the plurality of target software source codes, thereby generating a number array associated with the given target software;

identifying, by the processor, in the number array associated with the given target software, at least one pattern, wherein:

the at least one pattern comprises a predetermined repetitive number sequence within the number array, and

the predetermined repetitive number sequence is indicative of a frequency of occurrence of at least one associated target function command within the given target software;

storing the at least one pattern, thereby generating a training set of data,

the training set of data comprising a plurality of patterns, a given pattern of the plurality of patterns including a respective label being indicative of an association between the given pattern and the given target software;

training the classifier, based on the training set of data, to determine the affiliation of a given software to the given target software.

6. The method of claim 5, wherein the identifying the at least one target function further comprises identifying, in each of the plurality of target software source codes, library functions and deleting the library functions.

7. The method of claim 6, wherein the identifying the library functions further comprises identifying, in each of the plurality of target software source codes, source code portions inherently non-indicative of the affiliation to the given target software, and deleting the source code portions inherently non-indicative of the affiliation to the given target software from each of the plurality of target software source codes.

8. The method of claim 5, wherein the at least one target function command comprises at least one action and at least one argument associated with the at least one action; and

the generating the respective target function identifier further comprises:

applying a hash function to the at least one action and to each value of the at least one argument to generate respective hash function values, each one of the respective hash function values being a respective number sequence; and

concatenating the respective hash function values to form the associated number sequence.

9. The method of claim 5, wherein the given target software is associated with at least one of: a predetermined software family, and a predetermined authorship.

10. The method of claim 5, further comprising determining a frequency occurrence value associated with the at least one pattern, the determining being according to the following formula:

$$\lambda = \frac{L}{K},$$

where L is a frequency of occurrence of the at least one pattern within the number array associated with the given target software, and K is a number of software source codes in the plurality of target software source codes associated with the given target software used for generating the training set of data.

11. The method of claim 5, further comprising assigning a weight value to the at least one pattern.

12. The method of claim 11, wherein the weight value is further determined based on the frequency of occurrence value associated with the at least one pattern.

13. A method for determining an affiliation of a given software with a target software, the method being executable by a processor, the method comprising:

receiving, by the processor, a software source code of the given software;

executing, by the processor, the software source code in an isolated program environment to identify at least one of a plurality of outgoing requests of the given software,

the at least one of the plurality of outgoing requests comprising a respective action and at least one argument associated with the respective action; and

generating, for the at least one of the plurality of outgoing requests, based on the respective action and the at least one argument, a respective one of a plurality of identifiers associated with the given software;

aggregating, by the processor, the plurality of identifiers associated with the given software into an aggregated array of identifiers given a time order of identifying associated outgoing requests;

applying, by the processor, at least one classifier to the aggregated array of identifiers to determine a likelihood parameter indicative of the given software being affiliated to a respective target software;

in response to the likelihood parameter being equal to or greater than a predetermined likelihood parameter threshold:

identifying the given software as being affiliated to the respective target software;
storing data indicative of the given software in a database of affiliated software,
wherein the data comprises the aggregated array of identifiers; and
using the data indicative of the given software for further determining affiliation
of an other software to the respective target software.

14. The method of claim 13, wherein the receiving the software source code further comprises identifying therein source code portions inherently non-indicative of the affiliation to the target software, and deleting the source code portions inherently non-indicative of the affiliation to the target software from the software source code.

15. The method of claim 13, wherein:

the generating the respective one of the plurality of identifiers further comprises:

applying a hash function to the respective action and to each value of the at least one argument associated with the at least one of the plurality of outgoing requests to generate respective hash function values, each one of the respective hash function values being a respective number sequence;

concatenating the respective hash function values.

16. The method of claim 15, wherein the applying the hash function further comprises applying a first hash function to the respective action and applying a second hash function to the at least one argument, the second hash function being different from the first hash function.

17. The method of claim 15, wherein, the generating the respective one of the plurality of identifiers further comprises applying a serialization process to the respective action and the at least one argument associated therewith.

18. The method of claim 13, wherein the respective target software is associated with at least one of: a predetermined software family, and a predetermined authorship.

19. The method of claim 13, wherein the at least one classifier has been trained to determine the affiliation to the respective target software based on a training set of data, and the method further comprising generating the training set of data, the generating comprising:

receiving, by the processor, a plurality of target software source codes associated with the respective target software;

executing, by the processor, a given one of the plurality of target software source codes in the isolated program environment to identify at least one of a respective plurality of target outgoing requests of the respective target software,

the at least one of the respective plurality of outgoing requests comprising a respective target action and at least one target argument associated with the respective target action;

generating, for the at least one of the respective plurality of target outgoing requests, based on the respective target action and the at least one target argument, a respective one of a respective plurality of target identifiers associated with the respective target software;

aggregating, by the processor, target identifiers associated with the respective target software over respective pluralities of target identifiers into an aggregated array of target identifiers given a time order of identifying associated target outgoing requests;

identifying, by the processor, in the aggregated array of target identifiers associated with the respective target software, at least one pattern, wherein:

the at least one pattern comprises a predetermined repetitive sequence of a given target identifier within the aggregated array of target identifiers, and

the predetermined repetitive sequence of the given target identifier is indicative of a frequency of the respective target software executing the respective target action in the isolated program environment;

storing the at least one pattern with a label indicative of an association between the at least one pattern and the respective target software for inclusion thereof into the training set of data.

20. The method of claim 19, wherein a length of the predetermined repetitive sequence is determined as a constant number.

21. The method of claim 20, wherein the length of the predetermined repetitive sequence is further determined iteratively, based on a current number thereof within the aggregated array of target identifiers.

22. The method of claim 19, further comprising determining a frequency occurrence value associated with the at least one pattern, the determining being according to the following formula:

$$\lambda = \frac{L}{K},$$

where L is a frequency of occurrence of the at least one pattern within the aggregated array of target identifiers associated with the respective target software, and K is a number of software source codes in the plurality of software source codes associated with the respective target software used for generating the training set of data.

23. The method of claim 19, further comprising assigning a weight value to the at least one pattern.
24. The method of claim 23, wherein the weight value is indicative of a type of the respective action associated with the at least one of the plurality of outgoing requests of the respective target software.
25. The method of claim 23, wherein the weight value is further determined based on the frequency of occurrence value associated with the at least one pattern.