(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
23 November 2017 (23.11.2017)

WIPO | PCT

(10) International Publication Number
WO 2017/197519 A1

(51) International Patent Classification:
G06F 21/56 (2013.01) *G06F 21/57* (2013.01)

(21) International Application Number:
PCT/CA2017/050599

(22) International Filing Date:
18 May 2017 (18.05.2017)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
62/338,423 18 May 2016 (18.05.2016) US

(71) Applicant: **THE GOVERNING COUNCIL OF THE UNIVERSITY OF TORONTO** [CA/CA]; Banting Institute, 100 College Street, Suite 413, Toronto, Ontario M5G 1L5 (CA).

(72) Inventors: **LIE, David**; 4 Breen Crescent, Toronto, Ontario M2P 1Z7 (CA). **MIYANI, Dhaval**; 22 Wishford Drive, Scarborough, Ontario M1B 2T3 (CA). **SKANDARANIYAM, Janahan**; 124 Riverwalk Drive, Markham, Ontario L6B 0E2 (CA). **THANOS, Daniel**; 2022 Quinte Street, Burlington, Ontario L7L 7B6 (CA).

(74) Agent: **NORTON ROSE FULBRIGHT CANADA LLP**; 1, Place Ville Marie, Suite 2500, Montreal, Québec H3B 1R1 (CA).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report (Art. 21(3))*

(54) Title: SYSTEM AND METHOD FOR DETERMINING CORRESPONDENCE AND ACCOUNTABILITY BETWEEN BINARY CODE AND SOURCE CODE

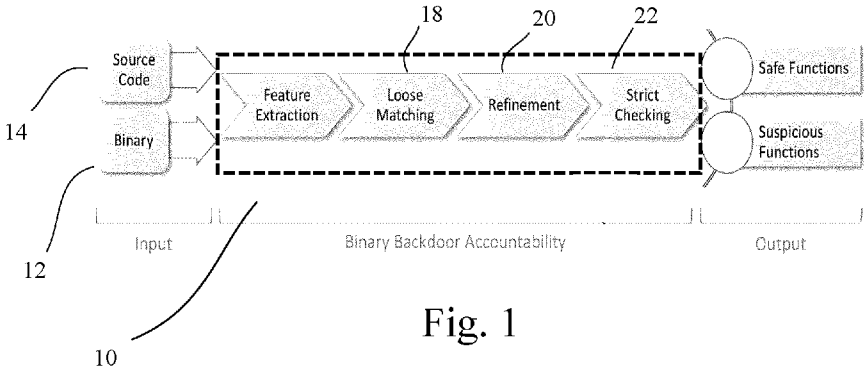


Fig. 1

(57) Abstract: A first set of code, for example source code, and a second code, for example binary code, are compared to find corresponding functions. A comparison of features can be used to find correspondences of functions. The comparison of functions can be iterated and can be refined and can be further used to carry out a further, stricter comparison of functions found to correspond to reduce the chance of falsely finding a function in the second code to be accountable in the first code.

WO 2017/197519 A1

SYSTEM AND METHOD FOR DETERMINING CORRESPONDENCE AND ACCOUNTABILITY BETWEEN BINARY CODE AND SOURCE CODE

CROSS REFERENCE TO RELATED APPLICATION

[0001] This application claims the benefit of and priority to U.S. Provisional Application No. 62/338,423 filed May 18, 2016 the contents of which is hereby incorporated by reference.

TECHNICAL FIELD

[0002] Embodiments relate to tools for detection of malicious code in software.

INTRODUCTION

[0003] Security sensitive organizations perform source code audits on software they use. However, after the audit is performed, they must still perform a binary code audit to ensure the binary provided to them matches the source code that was audited.

[0004] Example work on analysis and detection of malware and vulnerabilities in binary include: BinDiff, which matches binaries using a variant of graph-isomorphism; BinSlayer, which performs bipartite matching using the Hungarian algorithm; Blanket Execution, which matches functions in binaries, with the goal of either classifying malware or aiding automatic exploit generation; BinHunt and its successor iBinHunt, which redefine graph-based matching problem as maximum common induced subgraphs isomorphism problem; discovRe, which uses a loose matching algorithm to match binaries using structural and numeric features of call function graphs; BINJUICE, which normalized instructions of a basic block to extract relationships established by the block and then structurally compared the relationships; BINHASH, which models functions as a set of features that represent the input-output behaviour of a basic block; EXPOSÉ, which is a search engine for binary code that uses simple features such as the number of functions to identify a set of candidate function matches; TEDEM, which automatically identifies binary code regions that are similar to code regions containing a known bug; and backdoor detectors.

[0005] Source code can be developed by multiple parties and integrated to create software product. The trustworthiness of a software product is only as good as the least trustworthy principle who contributed to the product. When the number of parties that contribute to a software product increases, the trustworthiness decreases. To establish trust, organizations must thus perform their own auditing of a software product.

SUMMARY

[0006] In an aspect, embodiments provide a code auditor and related methods for classifying code elements in binary code into a first category accounted for in source code and into a second category as not accounted for in the source code. In an embodiment, the method comprises extracting, using a first extractor, a source code features from the source code, the source code features including source code functions; extracting, using a second extractor, a binary code features from the binary code, the binary code features including a second set of first set of functions; obtaining, in a matching block, a correspondence between the source code functions and the binary code functions by comparing a comparison selection of the source code features to a comparison selection of the binary code features; and classifying, in an accountability block, each function of the binary code functions into the first category or the second category according to a comparison of an accountability selection of features of the binary code features characterizing the function with corresponding features of the first set of features characterizing the respective function of the first set of functions estimated to correspond with the function. The apparatus may comprise the extractors, matching apparatus and accountability block.

[0007] Features can be preserved at compilation and accordingly are present in both the source code and binary code. Code auditor is operable to detect and match features present in both the source code and binary code. If a feature is present in the binary code and not the source code it may be flagged or classified as potentially suspicious.

[0008] The code auditor can implement the comparison selection using different matching processes such as a bi-partite match, for example. If a feature is present in both the source code and binary code then there can be a 1-1 match. The code auditor can also detect similar features in the source code and binary code and generate a measure of how good the match is. The code auditor can find an optimal set of 1-1 feature pairs in the source code and binary code that maximizes overall code level correspondence. The code auditor can use machine learning for example, to extract features and find the optimal set of 1-1 feature pairs in the source code and binary code. The code auditor can use machine learning to find which features pairs are more useful for matching. Key features can be parameters for the function call for graph edges and string literals, for example.

[0009] The code auditor can implement the accountability selection to characterize a feature that is present in the binary code and not the source code as potentially suspicious. Malicious code can be present in the binary code and not present in the source code and code auditor identifies and flags this malicious code. Key features for accountability can be the same features used for matching, for example.

[0010] In an aspect, embodiments provide a code auditor and related methods for classifying code elements in a second body of code into a first category accounted for in a first body of code and into a second category. In an embodiment, the method comprises extracting, using a first extractor, a first set of features from the first body of code, the first set of features including a first set of functions; extracting, using a second extractor, a second set of features from the second body of code, the second set of features including a second set of functions; obtaining, in a matching block, a correspondence between the first set of functions and the second set of functions by comparing a comparison selection of the first set of features to a comparison selection of the second set of features; and classifying, in an accountability block, each function of the second set of functions into the first category or the second category according to a comparison of an accountability selection of features of the second set of features characterizing the function with corresponding features of the first set of features characterizing the respective function of the first set of functions estimated to correspond with the function. The apparatus may comprise the extractors, matching apparatus and accountability block.

BRIEF DESCRIPTION OF THE FIGURES

[0011] Embodiments will now be described with reference to the figures, in which like reference characters denote like elements, by way of example, and in which:

[0012] Fig. 1 is shows operation of a code auditor;

[0013] Figs. 2A, 2B and 2C show respectively function call graphs of an exemplary binary code and source code, and a function call graph after execution of a code auditor's prediction phase, respectively;

[0014] Fig. 3 shows an exemplary weighted bipartite graph;

[0015] Fig. 4 shows an exemplary code auditor apparatus;

[0016] Fig. 5 shows results of applying an embodiment of the code auditor to a set of benign programs;

[0017] Fig. 6 shows code for an example binary with functionality inserted that was not present in the source code;

[0018] Fig. 7 is an example schematic of a system with the code auditor on a mobile device according to some embodiments;

[0019] Fig. 8 is an example schematic of a system with the code auditor on a server connected a mobile device according to some embodiments; and

[0020] Fig. 9 is an example schematic of a system with the code auditor on a server according to some embodiments.

DETAILED DESCRIPTION

[0021] Modifications may be made to the embodiments described here without departing from what is covered by the claims. In the claims, the word “comprising” is used in its inclusive sense and does not exclude other elements being present. The indefinite articles “a” and “an” before a claim feature do not exclude more than one of the feature being present. Each one of the individual features described here may be used in one or more embodiments and is not, by virtue only of being described here, to be construed as essential to all embodiments as defined by the claims.

[0022] Embodiments described herein relate to a code auditor that reduces the effort required to audit both the source code and the binary code. For Binary Backdoor Accountability the code auditor can identify the sections of the binary code whose provenance can be accounted for in the source code such that if the source code is free of backdoors, then the binary code is also free of backdoors. Accounted for code does not need to be inspected during the binary audit, saving manual effort by auditors. In some embodiments, the code auditor can compare a binary code that is produced by compiling a source code. In some embodiments, the code auditor may compare any two codes where one is a transformation of the other, and the codes each have a call graph.

[0023] Binaries can be stripped of symbols and compiled with a compiler that the customer does not have access to, which might arbitrarily apply a variety of compiler optimizations during

the process of transforming source code into a binary. As a result, a challenge for the code auditor is differentiating between differences in binaries and source code that are due to legitimate compiler optimizations and differences due to a backdoor that has been inserted in the binary after a source code audit. To overcome this, the code auditor identifies code features that are invariant under most compiler optimizations, but would be modified if a backdoor is inserted. For other optimizations that do alter these features, the code auditor uses machine learning to train to predict when these optimizations are likely to be applied to account for them.

[0024] Proving equivalence between two programs is equivalent to the halting problem, and can be undecidable. For the same reason, it can be difficult to prove that compilers produce binary code that is equivalent to the input source code. Rather than try to prove equivalence between a binary and source code, in some embodiments, the code auditor can detect another property, Binary Accountability. Binary Accountability aims to identify the functions in a binary that are accounted for by functions in some corresponding source code.

[0025] As an example, the call graph of binary B is accountable by the call graph of source code S if and only if every function in a non-empty set of functions b in B matches a non-empty set of functions s in S. The inverse is not true – functions in S do not have to match a function in B because source code can be present but not included into the final binary or the compiler may optimize some source code functions away.

[0026] According, binary code and source code can have a call graph. The code auditor can determine the accountability of binary code if the call graph of binary code is accountable to the call graph of source code when a set of binary code functions match a set of source code functions. The set of binary code functions can correspond to or be characterized by an extracted set of binary features. The set of source code functions can correspond to an extracted set of source code features.

[0027] In the above example formulation, *match* is a function that will return true if b is produced from s under benign compilation, but false if a backdoor is inserted into b that is not present in s. A backdoor inserted into the binary is malicious code that: 1) specifies malicious functionality that is not present in the source code and 2) that malicious functionality should be triggered only under very specific circumstances so as to remain stealthy. In some cases, the backdoor could be a large amount of functionality, such as making additional network and file system calls, features found in many remote access tools. These give a remote attacker access

to nearly unlimited functionality on the victim machine. In other cases, the additional functionality could be as little as an additional code path that allows an attacker to bypass authentication with a hard-coded password or secret keyword, such as that found in various well-documented backdoors. In this case, the attacker only has access to legitimate functionality in the binary, but in many cases this is still enough functionality to do a great deal of damage. While these two classes of backdoors are not exhaustive, a great majority of backdoors can fall into one of these cases. The code auditor can apply a backdoor threat model based on these two classes for new function calls that eventually lead to system calls are inserted, or for new code paths guarded by hard-coded strings or constants are inserted to make them stealthy.

[0028] The backdoor model can exclude backdoors that are built around vulnerabilities. For example, if an attacker inserts a memory corruption vulnerability that they can later exploit to inject new code or perform a return-oriented-programming attack, then the new functionality does not appear in the binary, but is instead injected at exploit time. Since binary accountability is a static analysis technique, it cannot analyze new dynamically injected functionality.

[0029] The code auditor can apply to binaries or other code from which a call graph can be determined. For a binary code, the binary code should be able to be reliably disassembled using a disassembler. The code auditor may not catch obfuscated backdoors that disassemble to normal-looking code (for example, by overlaying 2 different instruction streams over the same binary values). However, this requirement raises the level of difficulty for the attacker considerably. For source code, the source code can be parsed by the code auditor, for example if the source code is a readily available dialect of C or C++. Undefined or incorrectly defined symbols may be avoided, for example by access to values used in processor directives or by including a front end in the code auditor that ignores undefined symbols and accepts mismatched types.

[0030] A difficulty with determining binary accountability is the code transformations compilers apply to optimize the compiled binary. Fortunately, many optimizations are orthogonal to binary accountability as they do not introduce or remove system calls, nor do they affect hard-coded strings or constants used in a binary. The following are examples of code optimizations performed by compilers. These are examples only.

[0031] **Inlining.** This optimization moves a callee function into the body of a caller function, essentially removing the callee from the binary. In addition, a new function can be created that

will be a mix of the features of both the original callee and caller. Inlining makes binary accountability hard because two or more functions in the source code must be properly matched to a single function in the binary. The code auditor handles arbitrary inlining.

[0032] **Library call substitution.** Compilers may have standard library-specific optimizations that substitute library calls for a more efficient version under certain circumstances. In many of these cases, these library calls may make different system calls. Binary accountability may properly disambiguate these benign optimizations from modifications that would result from an addition of a backdoor in the binary.

[0033] **Local optimizations.** Compilers also optimize control-flow basic blocks to improve the performance of the execution. For example, optimizations such as loop unrolling may significantly change the structure of the control-flow graph. In addition, register allocation optimizations can obscure the number of parameters passed to a function at certain call sites if the compiler determines that the appropriate argument is already in the appropriate register.

[0034] **String modification.** Compilers may insert new string constants in the compilation process (i.e. standard pre-defined macros such as for example __FUNC__ or __FILE__), as well as modify existing ones (for example, perform statically-resolvable string substitutions for format string functions).

[0035] **Compiler-inserted functions.** Compiler will sometimes insert calls to helper functions. For example, GCC will insert calls to _stack_chk_fail to detect stack overflow. These functions appear in the binary but not in the source code. The code auditor will flag these functions as unaccounted for since they do not have a source code equivalent. However, they are easily recognizable and can be easily white listed by users.

[0036] In an embodiment, the code auditor computes binary accountability by comparing a function in the binary with a corresponding function in the source code. This comparison checks for differences in code features that could indicate the presence of a backdoor. Pairs (e.g. binary code feature and source code feature) that show no differences in these features are labeled as “safe”, and can be trusted to not contain a backdoor. Pairs where the code auditor detects differences may be different due to compiler optimizations. If the code auditor can determine that the difference is due to a benign optimization, the binary function is also labeled “safe”. The code auditor can conservatively mark all binary functions as “suspicious” when it could not determine if the differences are due to the compiler, and this set of functions must be

audited by a human to determine if a backdoor is really present or not. This comparison procedure can be referred to as Strict Checking.

[0037] However, before the code auditor can perform strict checking, it can first determine which functions in the binary correspond to which functions in the source code. The binary is stripped of symbols. The code auditor can determine this mapping. An approach might be to start at the entry points of both binaries (i.e. main()), and then perform a traversal of the call graph, but this might not work for two reasons. First, a function could have several callees, and there must be a way to disambiguate the callees. Second, edges in the call graph are often incomplete because of computed function pointers, whose target cannot be determined statically. As a result, the code auditor performs an iterative Loose Matching phase that uses various code features to determine the most likely source function that will match a binary function. In cases where loose matching is unable to find a good match for a binary function, these binary functions will be labeled as unmatched or multi-matched, and also require a human auditor to determine if they are new functions inserted as part of a backdoor or not.

[0038] Binary accountability is intended for scenarios where the user has very high security requirements, and has already committed to spending a large amount of human resources on code auditing. In such cases, the user is willing to tolerate a high false alarm rate because the current situation is that human auditors examine the entire binary anyway. Binary accountability is a cost saving tool whose purpose is to determine what parts of a binary need not be audited by a human. As a result, in an embodiment, the code auditor is designed to 1) take advantage of the fact that it has source code to perform accountability and 2) be very conservative in marking binary functions as safe as they will not be examined further by a human auditor.

[0039] A criterion for the utility of the code auditor is how many functions it can safely determine to be free of backdoors according to the backdoor model and thus save the human auditor from having to spend manual effort on and whether it can ever mistakenly mark a binary function that contains a backdoor as safe.

[0040] Fig. 1 shows the work-flow of a code auditor 10. The code auditor 10 configures a processor to execute instructions to implement the work flow. A binary code 12 and source code 14 are the input to the system 16, where the code auditor 10 extracts code features from them. Next, the code auditor uses the extracted features to perform loose matching 18 to identify corresponding function pairs in the source code and binary. A refinement phase 20 then helps

disambiguate functions that lose matching was not able to uniquely pair together. Finally, the strict matching phase 22 checks the function pairs to determine if there are any differences that cannot be accounted for by benign compiler optimizations.

[0041] The code auditor 10 can be implemented, for example, by programming a computer with a processor and memory. However, it could also be implemented as a hardwired structure for implementing the specific algorithms described. These implementations are equivalent, so that a general purpose computer programmed as described takes on a physical structure, e.g. by arrangement of electrons and holes within transistors, and distribution of charges creating voltages, that is a physical structure that implements the algorithms described. Fig. 4 shows an exemplary code auditor apparatus. An extractor is configured to accept as input a first set of code (e.g. source code) and extract from the code a first set of features including a first set of functions, and to accept as input a second set of code and extract a second set of features, including a second set of functions. The extractor may be, as shown in Fig. 1, separated into a first component 30 for extracting features from the first set of code and a second component 32 for extracting features from the second set of code. A matching block 34 is configured to obtain a correspondence between the first set of functions and the second set of functions by comparing a comparison selection of the first set of features to a comparison selection of the second set of features. An accountability block 36, which functions as a classifier, is configured to compare an accountability selection of features of the second set of features characterizing functions of the second set of functions with corresponding features of the first set of features characterizing the corresponding functions of the first set of functions to make a determination of the accountability of the second set of functions in terms of the first set of functions. These blocks may be components of a dedicated hardwired device. The blocks may also be implemented on a computer which may include one or more hardware processors and a computer recordable storage medium, or memory, coupled to the hardware processors. The extractors, matching block and accountability block may be programmed as computer executable program code, or as extractive logic, matching logic and accountability logic within the memory.

[0042] To help explain the operation of the code auditor, consider a running example, as exemplified by a toy game application with example code listed below. In this simplified example, the user inputs the name and the level he/she wants to play. Depending on the level users input, they play beginner or advance mode of the game. However, a backdoor is inserted in the binary of the game, which is shown in Fig. 6. If the user enters LegendaryHack name,

then he/she unlocks extreme level, which is hidden in the game play. Using this example, each of the code auditor's phases will be described in detail.

```
void advance(char *name, int level) {
        if (name == NULL) return;

}

void beginner(char *name, int level) {

}

  void start(char *name, int level) {
        if (level < 5) beginner(name , level);
        else advance(name , level);
  }

  inline static void init () {
        printf (" Initializing...\n");
  }

  int maxTen(int level) { return level %10; }

  void main (int argc , char ** argv) {
        init ();
        int level =0;
        printf ("Enter level: ");
        scanf ("%d", & level);
        level = maxTen(level);

        start (argv [1] , level);
  }
```

[0043] Both source code and binary are abstracted by extracting the same set of features, and comparing the features. Matching features are selected that are 1) mostly invariant under

compiler optimizations and 2) should change if a backdoor is inserted and 3) appear commonly enough in source code and binary to be useful for matching.

[0044] Predictive features are extracted that are only available in the source code, which they cannot be used for matching but may be used to help predict when function inlining will take place, which improves loose matching. Example features that can be used for matching and inlining predication are shown in Table 1.

Matching	String constants
	Integer constants
	Library Calls
	Function Call Graph
	Control Flow Graph
	# of function arguments
Predictive	Static Declaration
	Extern Declaration
	Virtual Declaration
	Nested Declaration
	Variadic Argument Declaration
	Recursion
	Computed Goto

Table 1: Features used for matching and inlining prediction.

[0045] To extract these features, the code auditor 10 extracts the call graph from both the binary and source code, and the features are extracted from each function in the call graph. Both the binary and the source code call graph may be missing edges due to function pointers.

[0046] **Matching Features.** The code auditor 10 extracts a set of references to string constants from each binary and source code function. Since string constants exist as string literals in the source code and references to the constants section of the binary, these are trivial to extract from both binary and source code. An instance is recorded for each use of the string

and the code auditor 10 conducts use-def analysis to ensure it detect the correct number of uses of a string even if it assigned to a variable first before being used. String constants are a feature that is invariant under many compiler optimizations, but tend to be used in guards of backdoors.

[0047] Similarly, integer constants are another feature that is suitable for binary accountability. Like string constants, these are extracted into a set that is associated with each function. Integer constants in a binary not only represent integers in the source code, but may also represent various other values such as character constants, enum values and addresses. Constant values such as 0, 1 and -1 may be ignored which are commonly used in functions, leaving other unique integers. Similar to constant strings, constant integers can serve to uniquely identify functions. In addition, an attacker may also break a hard-coded value into several constant integers, so this feature is also necessary to detect backdoors.

[0048] Library calls can also help identify and pair functions and are also extracted into a set of library calls that are associated with each function. Unlike string constants, library calls may not be uniquely identify functions, but at the same time it is difficult for the compiler to optimize this feature. Also, rather than call a system call directly, a backdoor may make a call to a library function in an example library like libc, so this feature should be included to ensure that backdoors are detected.

[0049] The number of arguments a function takes is also a feature that can help pair a binary function with its source code equivalent. While the number arguments is easily extracted from the source code, it is not always as easily extracted from the binary due to compiler optimizations.

[0050] Finally, the function call graph (FCG) and control-flow graph (CFG) of both binary and source code are also used as features. The FCG helps identify and pair functions by checking that paired functions have similar callers and callees. Since a backdoor can be implemented by inserting a call to a function that may lead to a system call, checking callees is also required to ensure that a function with a backdoor is not marked as safe. Using the extracted FCG, the code auditor records the set of caller and callee functions as features for each function.

[0051] Function CFGs are often heavily modified during compilation and thus are not a good feature for direct matching. However, the code auditor still uses them during the refinement

phase to help disambiguate non-unique matches that arise out of loose matching. However, because of the large changes CFGs undergo, they are not suitable for use during the strict matching phase as they would lead to too many benign differences marked as suspicious. The code auditor does not record the full CFG of each function, but only keeps the number of conditional branches in the CFG for each function. The ternary operator might not have its own conditional node when extracted from the source code. To compensate for this, code auditor 10 can increment the conditional branches in source code functions where there is a ternary operator.

[0052] **Predictive Features.** A difficulty for the next phase is dealing with function inlining. In the absence of inlining, a binary function will match exactly one source function. However, with inlining, inlined source functions must be combined before their matching features will match the corresponding binary function. Because the code auditor has access to source code, it is able to use code features from the source code to predict which functions the compiler is likely to inline.

[0053] Predictive features include features of the function declaration, such as whether it is static, extern virtual, nested or has variadic arguments, and may include features of the function body, such as whether it contains recursion (direct) or a computed goto.

[0054] To train a classifier, a set of applications and a compiler may be used. These do not need to be the same application or the same compiler for which the code auditor is trying to determine binary accountability. The intuition is that all compilers follow some common principles of when to inline, which are applied independently of the application being compiled. Thus, this training can be done once for all uses of the code auditor.

[0055] The predictor may be trained by building a corpus of inlined and non-inlined functions extracted from a variety of applications. In an example embodiment, this may be used to train an Alternating Decision Tree (ADTree) classifier. The trained classifier is then applied to the source code to be audited. Functions that are classified as likely to be inlined have their matching features copied to their parents. Edges may be added in the FCG from the parent to the children of the inlined function, but the inlined function is not removed from the FCG. The reason is that if the predictor is wrong, then the binary functions will still have a chance to match their corresponding inlined functions. However, if the predictor is correct, then the inlined source code function will simply be left as an unmatched function in the source code.

[0056] The code auditor extracts the predictive features from the source code and the binary. Example FCG derived from the binary and the source code are shown in Fig. 2A and Fig. 2B, respectively. As observed from the FCGs, init function is inlined in the binary. Using the predictive features, the code auditor correctly predicted init as inlined into main. As a result, the matching features of init (namely the library call to .printf) will be copied into main. However, as noted above, the init node is not removed from the FCG even though it is inlined. Fig. 2C shows the final FCG after the code auditor's prediction phase.

[0057] The goal of loose matching is to label every binary function as matched, unmatched or multi-matched. Multi-matched functions are binary functions where loose matching determines more than one potential match. These functions are further refined in the next phase to either matched or unmatched. Functions that are unmatched are labeled suspicious, if the code auditor was not able to find a corresponding function in the source code that it can be accounted to. Finally, all matched functions undergo strict matching to determine the final set of safe and suspicious functions.

[0058] After inlining prediction is applied, every function in the binary should uniquely match a single function in the source code. If inlining took place and was predicted correctly then the binary function should match the parent function where the function was inlined. Otherwise the un-inlined binary functions will match their respective source functions. As a result, loose matching is a classic bipartite matching problem.

[0059] The functions in the binary and source code can be mapped into a weighted bipartite graph, as shown in a small example in Fig. 3. A bipartite graph is a graph G where the nodes can be divided into two disjoint sets S and B, such that no two nodes within the same set are connected by an edge. Here, one set represents the functions in the source code and the other the functions in the binary. Weighted edges in G connect a node from S to B. An optimal bipartite assignment is determined between the nodes in B and S that minimizes the total weight of the edges between them. This problem is analogous to solving an assignment problem. An example way to solve such problems is to apply the Hungarian algorithm, which produces a sub-optimal solution in polynomial time, O(N3). The code auditor 10 can 1) compute the weight for each edge between B and S and 2) label the pairs of nodes in the resulting assignment as matched, multi-matched or unmatched. In the example embodiment, the code auditor can apply the Hungarian algorithm by the computation and label. Algorithm 1 presents the pseudocode for the matching based on bipartite graph.

Algorithm 1 Loose Matching

```
function LOOSE_MATCHING(BinGraph, SrcGraph)
    reGenerateCallerAndCalleeNodes(BinGraph)
    reGenerateCallerAndCalleeNodes(SrcGraph)
    repeat
        ▷ Create weighted bipartite graph
        weights ← 0
        pairs ← 0
        for b ∈ BinGraph do
            for s ∈ SrcGraph do
                weights ← compWeights(b, s)
            end for
        end for
        ▷ Run Hungarian and assign labels
        pairs ← HungarianAlgorithm(weights)
        assignLabels(binGraph, pairs)
    until convergence = true
end function
```

[0060] **Computing weights.** Weights are calculated for each pair of binary/source code functions. For each pair of nodes (an edge) in a bipartite graph, the code auditor 10 can use these set of features to assign a cost. The total weight is a weighted sum of the individual costs of each matching feature:

$$\sum_{i=1}^{N} w_i C_{f_i}$$

where N is the total number of features, w is the weight factor for a feature and C is cost for a feature.

[0061] The code auditor 10 can calculate the cost of each feature, C_f, on the scale of 0 and 1, inclusive. The cost of 0 means a pair of function is the most similar, whereas the cost of 1 means the pair is least similar. If a pair do not share any common features in an example

embodiment the code auditor 10 can assign predefined MAX_COST, otherwise the code auditor 10 can calculate the cost based on features. The way a cost for a feature is computed depends on whether the features is a set, such as for string constants, or whether it is a scalar value, such as for the number of function arguments.

[0062] For set features, the code auditor 10 can compute an similarity index between the two sets. An example is a modified Jaccard index. The Jaccard index is computed as follows:

$$J(B,S) = \frac{|B \cap S|}{|B \cup S|}$$

where B and S represent a set feature from a node in the binary graph and source graph, respectively. However, Jaccard index will calculate the same cost for cases where a string is missing in set B and a string is added in set B. Because backdoors are likely to introduce new strings, the code auditor 10 can assign a higher cost if a string is added in B. Thus, the code auditor 10 can use modified version of Jaccard index to calculate the cost:

$$C_f(B,S) = \begin{cases} \frac{|S-B|}{|S|} \; if \; |B-S| = 0 \\ \frac{|B-S|}{|B|} \qquad otherwise \end{cases}$$

[0063] For the number of function arguments, which is a scalar value, the code auditor 10 can assign a cost of 1 if the source code and binary function do not have the same value and a 0 if they have the same value.

[0064] Consider the game example where the code auditor 10 can calculate the cost of string constant feature, f, for the function main in the binary. Since there are only two functions that contain this feature, f is calculated for the two pairs: 1) main from the binary and init from the source code, and 2) main from the binary and main from the source code. The string constant feature contains three strings for the function main in the binary and one string for the function init. For pair 1 the cost C_{f_1} is 0.667, whereas for pair 2 the cost C_{f_2} is 0.333. As a result, the binary main is more likely to match the source code main because of the lower weight these lower costs would result in. A similar effect will occur due to the other features.

[0065] Each feature contributes a different amount to the likelihood of determining a correct match. To account for this, each feature's cost is multiplied with a weight factor before being combined into an overall edge weight. The code auditor 10 can determine the weight factor of each feature using Sequential Minimal Optimization (SMO) to train a Support Machine Vector (SMV) machine learning classifier in an example implementation. Once the training is completed, the code auditor 10 can obtain the weight factors of each of the features according to their importance determined by the SVM.

[0066] One challenge is that while nodes in both binary and source code functions contain a set of callers and callees in the FCG feature, the contribution of components towards the total edge weight cannot be computed initially because there is no mapping between callers and callees. That is, functions in the source code are identified by function names, but functions in the binary are identified by their addresses, and it is the mapping between those that code auditor 10 can try to compute. The code auditor can work around this problem by iteratively computing the edge weights, performing assignment and labeling the functions. The code auditor can perform Hungarian assignment, for example. On each iteration, the code auditor 10 can match some number of binary/source code functions, and these matches are then used to compute costs in the FCG feature for the next iteration. These iterations are then performed until the resultant changes converge.

[0067] **Labeling functions.** Once weights are assigned to each edge in the bipartite graph, the code auditor 10 can determine an assignment that minimizes the weights of the edges. This may be using the Hungarian process, for example. Each pair of functions in the resultant assignment can have an edge weight that is either unique, equal to some other pair or MAX COST. A unique weight means that the binary function has no other edge between it and another source function with the same weight. Such binary functions are labeled as matched. In other cases, there can several edges whose weights are equal to the weight of the edge between the binary and the source function that the Hungarian algorithm assigns. In these cases, the used features cannot definitively match a single function so these will be resolve during refinement. The code auditor 10 can label the binary function as multi-matched and note the other source functions that have equal weight. Finally, a pair that has MAX COST is labeled unmatched because the binary function has no features in common with the source code function.

[0068] Recall that after labeling, The code auditor will update the edge weights of the FCG caller and callee features based on the new labels. After each iteration, more functions will be marked as matched, and these matched functions affect the number of elements that intersect when computing the modified Jaccard Index. As a result, this increases the contribution that the FCG caller and callee features make during loose matching for each iteration, allowing more matches to be identified.

[0069] For example, let us consider functions main and start in the binary, which would be labeled as matched after the first iteration due to its features. Once this match is found, it allows the code auditor to learn the corresponding functions in source and binary, and enable it to disambiguate advance and beginner from maxTen in the next iteration since start calls advance and beginner while main calls maxTen.

[0070] **Refinement.** At this point all binary functions have been labeled as matched, multi-matched or unmatched. As an optional step, refinement uses CFG features to disambiguate the multi-matched binary and source code functions that have the same edge weights between them. To do this, the code auditor 10 can perform a procedure similar to loose matching on these functions. However, unlike in the case of loose matching, the edge weights in this bipartite graph are determined solely from the number of conditional branches in the CFG feature (recall that this feature is not used in loose matching in the primary embodiment described). The cost of CFG feature is the difference between the number of conditional branches:

$$C_{cfg} = |N_b - N_s|$$

where N_b and N_s is the number of conditional branch feature for the binary node CFG and source node CFG respectively. The code auditor 10 can again run the Hungarian algorithm on the bipartite graph and re-label the nodes in this smaller bipartite graph based on whether the resultant assignments have a unique, non-unique or MAX COST weight.

[0071] For example, the function advance in the toy game contains one conditional branch in both the binary and the source code. This function will thus be labeled as matched.

[0072] **Strict Checking.** At this point, the code auditor has determined the best source code function that it can find for the matched binary functions. The code auditor classifies each of the binary functions as accountable in terms of the source code function or as not accountable (including unknown).Every binary function that is still unmatched or multi-matched is marked as

suspicious (classified as unaccountable) and will be manually audited. However, functions that are labeled matched are marked as safe (classified as accountable) and won't be audited. As a result, to ensure they are truly safe, the code auditor performs strict checking to determine if there are differences between their features that could indicate the presence of a backdoor.

[0073] The code auditor checks the string constant, integer constant, callee and library call features of every pair of matched functions for strict equivalence. An inserted backdoor is likely to change one of these features. The string constant, integer constant, callee and library call features are used during loose matching so one might wonder why the code auditor 10 can allow functions that differ in these features to be matched at all during loose matching. The reason is that when the code auditor 10 can be strict during the matching phase, this can result in very few matches being found at all due to compiler optimizations. Since very few matches could be found, loose matching could not "bootstrap" itself and find callees and callers during the iterations during the matching phase. As a result the callee and caller features are not as helpful and many functions that would have been matched and pass strict checking were never matched in the first place. Thus, it is better to have some functions that are slightly different due to compiler optimizations be matched so that their callers or callees can, which might be exactly the same can also be matched during loose matching.

[0074] Considering the game example, the functions main, start, advance and beginner can be matched respective to its source code functions in the Loose Matching and Refinement stages. In the Strict Checking phase, these matched functions are compared, for instance advance from the binary will be compared with advance from the source code. After comparison the advance can contain additional string constant in the binary than the source code and, thus the code auditor flags the function as suspicious for manual checking.

[0075] One possible reason that equivalent functions might not pass during strict checking is due to an instance where function inlining occurred in the binary, but was not predicted to happen during loose matching. As a result, the inlined features from the inlined callees will appear in the binary function, but not in the matched source function. To prevent this misprediction from causing correct matches to be labeled as suspicious, strict checking accounts for inlining by recursively searching the children of a mismatched source code function for any additional features found in the binary function. To avoid searching to the entire sub-graph, the code auditor only checks for inlined feature at a maximum depth of two levels in the embodiment described. This means that the code auditor only check the callees and their

children functions. Since compiler optimizations may add or remove integer constants, the code auditor uses a threshold and treats differences in integer constants less than the threshold as the same. In our code auditor prototype, an example threshold of 8 can be used.

[0076] While the code auditor is able to correctly match many binary functions with the correct source code function and determine the absence of a backdoor, there are still anywhere from 10-35% of functions where it is unable to do so. The main reason for this is benign compiler optimizations. Since static compilers cannot use dynamic profiles to guide when to apply optimizations, they often must rely on complex algorithms to determine optimizations should be applied. With the exception of inlining, it may be ineffective to try to predict when these other optimizations will occur.

[0077] Two design features in the code auditor enable it to achieve matches despite these optimizations. First, the code auditor allows functions with some minor differences to still match during loose matching. Second, the code auditor only uses features that a backdoor would affect during strict checking. For example, a commonly applied optimization that the code auditor does not try to predict is loop unrolling, which will increase the number of conditional branches used in the CFG feature. However, this optimization has only a minor effect on the final result of the code auditor since it is only used during the refinement phase.

[0078] Another optimization is register allocation and spills to the stack. On the x86-64 processor architecture, there are 6 dedicated registers used for passing arguments in a function call (rdi, rsi, rdx, rcx, r8, r9). The code auditor 10 can use these registers to calculate the number of arguments feature for binary functions. However, due optimizations definitions and uses of these registers may be removed making it difficult to tell which registers are live at a function call site. As a result, machine learning can place a low weight factor on this feature, meaning that it only came into play when all the other features between two functions were very similar.

[0079] Compilers may also perform library call substitutions that replace some standard library calls with more efficient versions if the call-site arguments permit it. While these differ widely from compiler to compiler, and thus are a poor target for training unless one has access to the compiler itself (something the code auditor does not assume), they are fairly easily recognizable by a human auditor, as they are almost universally applied to standard library functions and a human who understands the semantics of the library calls would be able to

easily determine their equivalence (i.e. replacing printf with puts or vsprintf). Thus, the code auditor can be configured with a white list of such substitutions. Once the auditor notices a compiler making such substitutions, they can add the substitution to the code auditor's white list and run the code auditor again, which prevents it from incorrectly flagging these substitutions as suspicious. A similar white list also exists for compiler-inserted functions.

[0080] Finally, the string constant and function caller-callee relationships can be rarely modified by compiler optimizations, making them reliable for both loose matching and strict checking. In an example embodiment, these three features can be given the heaviest weight factors by the machine learning rules used by the code auditor 10. The major case where compilers insert strings are with pre-processor defined strings. Another more obscure case was where format strings that contained a substitution that could be statically resolved (i.e. sprintf(str, "%d", 5)) might be performed at compilation time. While some pre-processor inserted stings such as __LINE__ or __FILE__ can be resolved since the code auditor theoretically knows the file and line where the macro appears, in general, these string substitutions are difficult to white list as the inserted string can be very dependent on the environment where the binary was compiled (i.e. the date or compiler version). As a result, the current prototype code auditor does not white list any compiler-generated string constants.

[0081] **Extracting and comparing features.** The code auditor 10 can extract features from source code. The code auditor can extract features from binaries. The code auditor implements binary accountability on the two sets of extracted features. The source code feature extraction component of the code auditor can be implemented by extending the ROSE compiler framework with 1907 LOC. An example code auditor can access the build scripts of an application (i.e. Makefiles) to extract compiler directives so that preprocessor macros and include files resolve properly. The dependency may be eliminated by removing type checking, permitting the use of undefined variables and searching the source tree to resolve include files. The source code extraction feature processes each source file individually and outputs each function as a description. The functions are then linked together along with the other source features, scripts and libraries.

[0082] Binary features can be extracted using different programs. To extract the number of arguments, code auditor can extract the necessary binary features to compute the number of arguments. The code auditor can take into account compiler idioms such as xor r8, r8, which appear to be reading register r8, but are in fact just initializing it to zero. Unfortunately, due to

control-flow imprecision, statically determining dynamic instruction ordering is not always reliable. As a result, the code auditor can extract the wrong number of arguments.

[0083] The example matching phase described is implemented in 12,110 lines of JAVA™ code in an embodiment. The example implementation can be used with an example Graph Matching Toolkit framework, which provides an implementation of the Hungarian algorithm. Most of the code in the framework has been modified and updated to include the loose matching, and refinement and strict matching phases.

[0084] **Machine learning.** In some embodiments, code auditor 10 can train machine learning classifiers for inlining prediction during feature extraction and to compute the optimal weights for edge weight calculation during loose matching. The likelihood of inlining and the relative importance of the features can be dependent on when the compiler applies optimizations, which can change depending on the optimization level that is passed to the compiler. For example, there can be four standard optimization levels in GNU GCC compiler. The GCC flag -O0 will turn off all optimizations, whereas -O1 will turn on 31 different optimizations. The flag -O2 will turn on another 26 optimizations and -O3 additional 9. The code auditor 10 may not know the optimization level the binary to be audited has been compiled with, or even whether it was compiled with GCC or not. As a result, the code auditor 10 can train with a mix the -O2 and -O3 optimizations levels, as example optimization levels for production code. Other training levels can be used in various embodiments.

[0085] To train the inlining prediction classifier, the code auditor 10 can use a corpus of training applications. In an example embodiment, each application can be compiled twice, once with -O2 and again with -O3 compiler optimization levels. The code auditor 10 can use debugging symbols within the compiled binaries to divide the functions into a set that were inlined and a set that were not inlined. For each application, the code auditor 10 then selected the smaller of the two sets (usually the set that was inlined) and randomly select an equal size set of functions from the other set making a set of functions that is 50% inlined and 50% not-inlined. For each function, the code auditor 10 extracts a feature vector containing the predictive features for the function. The code auditor 10 can then aggregate these functions and feature vectors across all applications and train an inlining predictor. The resulting predictor can be used by the code auditor 10 to predict whether a function will be inlined or not using predictive features.

[0086] The edge weights for loose matching are computed by the code auditor 10 using a similar procedure. Again, the code auditor 10 can use a corpus of applications and compile each application using GCC, but only at the -O2 optimization level so that only functions that are very likely to be inlined get inlined. The code auditor 10 can then exclude all the inlined functions. Using the ground truth for which source function matches which binary function, the code auditor 10 can then create a set of function pairs for each application composed of 50% correct matches and 50% incorrect matches. Inlined functions are excluded because the code auditor 10 cannot always combine the callee and caller features correctly, and these errors pollute the training set. The code auditor 10 can then train the weights so that the SVM classifier is able to maximally classify these two sets correctly across all applications in the training set. During the evaluation, the code auditor 10 performs a 5-fold cross evaluation across our 10 applications. The code auditor 10 tabulates the average computed weights from this 5-fold evaluation. Example results in Table 2, which shows that FCG callee and callers, as well as string constants are the main features used for matching as they have the heaviest weights. This is an illustrative example and non-limiting.

Feature No.	Feature Name	Weight
1	string constant	1.469
2	integer constant	0.6315
3	library calls	0.2828
4	caller functions	2.9293
5	callee functions	2.9293
6	# of arguments	0.9296

Table 2: Average weights of the features used in Loose Matching phase

[0087] **Evaluation.** Example embodiments for the code auditor can be on virtual machines running Ubuntu™ 14.04 on Intel™ Core i7-2600 CPUs (4 cores @ 3.4 GHz) with 16 GB of memory. An example study can be a set of 10 applications: Proftpd 1.3.4b, Busy-box 1.23.2, Apache (HTTP) 2.4.12, Bind 9.10.2, Mongoose 2.1.0, OpenSSH 7.1p2, Dropbear 2015.71, OpenVPN 2.3.10, Transmission 2.84 and Lighttpd 1.4.39. Applications can be compiled with GCC 4.8.2 with optimization level -O2 and -O3, and the ICC Intel™ compiler 15.0.3 with the highest optimization level -O3.

[0088] The code auditor can mark binary functions with various backdoors inserted into them as suspicious.

In the absence of backdoors, what percentage of binary functions does the code auditor correctly mark safe and consequently how much binary auditing effort does the code auditor save.

[0089] **Identifying backdoors.** The code auditor can flas backdoors as suspicious when they are inserted in binaries and accounted against source code that is free of backdoors. An example can be backdoors in ProFTP, Apache, Dropbear and Juniper™'s ScreenOS™. In cases where there is not access to the actual backdoor binaries, it can be implemented based on the descriptions of the backdoors. Examples may relate to source code for ScreenOS™ and its backdoor can be implemented in Dropbear. The ProFTP, DropBear and ScreenOS™ backdoors all use hard-coded strings to access hidden functionality. Since the code auditor can flag binary functions with extra string constants, it can flag all the binary functions containing these backdoors as suspicious. The Apache backdoor is more sophisticated in that a GET request with a special string to decrypt to a particular format when decrypted with the requester's IP address. In this case, the code auditor can detect an additional function call within the binary function with the backdoor, which is used for comparing the input string with the encoded string.

[0090] In some embodiments, these real backdoors are flagged as suspicious by the code auditor 10 because they do not hide their presence from static analysis tools. To further evaluate the resilience of the code auditor 10 to evasion, consider an example of an adversary who is aware of how the code auditor 10 works, but is still constrained by the backdoor model. There are different ways an adversary could insert a backdoor. Example backdoors on Mongoose, Proftpd and Dropbear are as follows:

[0091] - **Strings as individual characters.** Instead of encoding a hard-coded string as a string constant, the attacker embeds the string check directly in the code as a series of comparisons against individual characters of the string. This avoids having a string constant. However, it increases the number of conditional branches and integer constants because each comparison requires an integer constant and a conditional branch causing the code auditor 10 to classify it as suspicious.

[0092] - **Compute a hash of the string and compare the resulting value.** Instead of comparing the string directly, the adversary hashes the string using a hash and compares the hash value to reduce the number of conditional branches and integer constants. Comparing an MD5 hash only requires 4 comparisons with 4 integer constants, which is below the code auditor's threshold. However, calling the MD5 function adds a function call, which causes the code auditor 10 to mark the function as unmatched and label it suspicious. If the adversary inlines the MD5 hash function to avoid calling a function, then The code auditor 10 can detect an increase in the number of integer constants and branches due to the MD5 code and marks the function as suspicious.

[0093] - **Calling an intermediate function.** Instead of directly calling a library function for string comparison, code auditor 10 can call a local wrapper function, which calls a library function. In this case, the code auditor 10 no longer matches the binary function no longer to its source code function and the binary function is labeled unmatched, causing the code auditor to eventually label it as unmatched. This eventually causes the code auditor 10 to label the function as suspicious.

[0094] - **Calling an existing function.** The adversary can find a function that already makes a call to the MD5 function and put the backdoor in that function. However, the code auditor tracks the number of calls to each function in its callee list, causing the binary function with the backdoor to not match its source function, resulting in it being labeled as suspicious.

[0095] - **Use an existing string constant in the function**. Instead of introducing a new string constant or a string constant from another function, which the code auditor 10 can flag, the attacker could try to re-use an existing string constant that the function is already accessing in their backdoor. However, because the code auditor counts the number of string constant accesses, this still fails strict checking and the function is labeled as suspicious by code auditor 10.

[0096] - **Assign an existing string constant to a local variable.** Instead of using the string constant twice, the adversary modifies the function to assign the string constant to a temporary variable and modifies the original use to use that variable. A temporary variable can be used in the backdoor. However, the code auditor 10 can take use-def chains into account when computing the number of string constant references, so this backdoor can also be caught during strict checking.

[0097] - **Portknocking.** Instead of preventing accidental discovery of the backdoor with a long string constant, the attacker uses port-knocking to hide the presence of the backdoor. However, port-knocking introduces new library calls to listen to the various ports in the sequence, which can cause the code auditor 10 to flag the function as suspicious.

[0098] For example, an adversary could find one or a series of existing checks for obscure errors (an out of memory error for example) set a global variable if it is triggered. The global variable then triggers some existing functionality that is useful to the attacker. However, this increases the difficulty of using this backdoor as the attacker must find error paths that are not likely to be triggered by accident, yet must be easy for the attacker to trigger from an external interface. A code auditor can detect backdoors placed on obscure paths and triggered indirectly via a variable or other dependency would be able to detect such an attacker.

[0099] In another instance, the attacker could use a very short string (less than 7 characters long) as the hard-coded string that triggers the backdoor. Due to local optimizations, in some embodiments, the code auditor can allow differences in conditional branches and integer constants below some threshold and a small enough increase in these may not cause the code auditor to mark a function as suspicious. However, using a short string increases the chances that the backdoor might be triggered accidentally or during blackbox testing. In some embodiments the code auditor 10 can prevent any type of backdoor from being marked as accounted for.

[0100] An example report can evaluate how much binary auditing effort the code auditor can save by evaluating the percentage of functions marked safe by the code auditor when performing binary accountability on benign binaries. An illustrative example can involve a 5-fold cross validation for training the inlining predictor and matching weights when running the code auditor on 10 example applications. The code auditor can find white-list the standard library function substitutions. The code auditor 10 runs again with the extracted white-list and example results are shown in Figure 5. In this example, the code auditor is able to match an average of 76.5% of the binary functions across the 10 applications and 3 compiler-optimization combinations, reducing the binary auditing effort by more than 1/4 on average.

[0101] Different embodiments can enable the code auditor to match case with very few matching features, different functions that have similar features and various compiler optimizations that confuse relationship between corresponding functions.

[0102] In some embodiments, even though both the inlining predictor and matching weights are trained with GCC, comparable matching results can be achieved on completely different compilers, such as on the Intel™'s ICC compiler. While compiler implementation may vary wildly, the types of optimizations they make can be similar "textbook" type optimizations.

[0103] To further understand how the code auditor achieves robust matching across compilers, consider the performance of the inlining predictor as an example. A 5-fold cross evaluation can be implemented across 10 applications. As an example, the predictor can be trained on the on GCC as described previously and then evaluated on the Intel™ ICC compiler. On average, the training set can include 12040 feature vectors and the testing set of 735 feature vectors. The cross-compiler performance of the inline predictor can enable the code auditor to achieve high percentage of accountability in some example embodiments, even across compilers.

[0104] The embodiments of the devices, systems and methods described herein may be implemented in a combination of both hardware and software. These embodiments may be implemented on programmable computers, each computer including at least one processor, a data storage system (including volatile memory or non-volatile memory or other data storage elements or a combination thereof), and at least one communication interface.

[0105] Fig. 7 is an example schematic of a system with the code auditor 10 on a mobile device 700 according to some embodiments. Mobile device 700 can install code auditor 10 to process applications with source code 14 and binary code 12 to detect malicious software. Code auditor 10 can process application at time of download or before execution, for example. Code auditor 10 can continuously monitor applications installed on mobile device 700 to detect malicious code.

[0106] The code auditor 10 can accept as input source code 14 and extract source code features including source code functions. The code auditor 10 can accept as input binary code and extract binary features, including binary functions.

[0107] The code auditor 10 can match the source code functions and the binary code functions by comparing a comparison selection of the source code features to a comparison selection of the binary code features.

[0108] The code auditor 10 can compare an accountability selection of features of the binary code features characterizing functions of the binary code functions with corresponding features of the source code features characterizing the corresponding functions of the source code functions to make a determination of the accountability of the binary code functions in terms of the source code functions.

[0109] The code auditor 10 can use machine learning for example, to extract features and find the optimal set of 1-1 feature pairs in the source code and binary code for the comparison selection. The code auditor can use machine learning to find which features pairs are more useful for matching. Key features can be parameters for the function call for graph edges and string literals, for example.

[0110] The code auditor 10 can output classifications 702 of safe functions 704 and suspicious functions 706 based on the accountability of the binary code functions in terms of the source code functions, for example. The mobile device 700 can use the classifications 702 for automatic decision making based on a security profile or policy. The security profile or policy may indicate one or more appropriate action items when malicious software is detected. For example, the mobile device 700 may uninstall or delete an application flagged with suspicious functions 706.

[0111] Fig. 8 is an example schematic of a system with the code auditor 10 on server 800 connected a mobile device 700 according to some embodiments. In this example, mobile device 700 can send source code 14 and binary code 12 to server 800 with code auditor 10 for processing in order to detect malicious software. Code auditor 10 can generate the file of classifications 702 for transmission by server 800 to mobile device 700. Mobile device 700 uses the classifications 702 to implement appropriate action items when malicious code is detected.

[0112] The server 800 connects to other components in various ways including directly coupled and indirectly coupled via the network. Network (or multiple networks) is capable of carrying data and can involve wired connections, wireless connections, or a combination thereof. Network may involve different network communication technologies, standards and protocols.

[0113] Fig. 9 is an example schematic of a system with the code auditor 10 on a server 800 according to some embodiments. The server 800 can provide Software as a Service or cloud service to process binary code 12 and source code 14 and output classifications 702. For

example, a user device can connect to server 800 to transmit binary code 12 and source code 14 for processing. In response, the user device receives the classifications 702 to implement appropriate action items when malicious code is detected.

[0114] Mobile device 700 or server 800 includes at least one processor, memory, at least one I/O interface, and at least one network interface.

[0115] The processor may be, for example, any type of general-purpose microprocessor or microcontroller, a digital signal processing (DSP) processor, an integrated circuit, a field programmable gate array (FPGA), a reconfigurable processor, or any combination thereof.

[0116] Memory may include a suitable combination of any type of computer memory that is located either internally or externally such as, for example, random-access memory (RAM), read-only memory (ROM), compact disc read-only memory (CDROM), electro-optical memory, magneto-optical memory, erasable programmable read-only memory (EPROM), and electrically-erasable programmable read-only memory (EEPROM), Ferroelectric RAM (FRAM) or the like.

[0117] The I/O interface enables mobile device 700 or server 800 to interconnect with one or more input devices, such as a keyboard, mouse, camera, touch screen and a microphone, or with one or more output devices such as a display screen and a speaker.

[0118] Each network interface enables mobile device 700 or server 800 to communicate with other components, to exchange data with other components, to access and connect to network resources, to serve applications, and perform other computing applications by connecting to a network (or multiple networks) capable of carrying data.

[0119] Mobile device 700 or server 800 is operable to register and authenticate users (using a login, unique identifier, and password for example) prior to providing access to code auditor 10.

[0120] Program code is applied to input data to perform the functions described herein and to generate output information. The output information is applied to one or more output devices. In some embodiments, the communication interface may be a network communication interface. In embodiments in which elements may be combined, the communication interface may be a software communication interface, such as those for inter-process communication. In still other

embodiments, there may be a combination of communication interfaces implemented as hardware, software, and combination thereof.

[0121] Throughout the foregoing discussion, numerous references will be made regarding servers, services, interfaces, portals, platforms, or other systems formed from computing devices. It should be appreciated that the use of such terms is deemed to represent one or more computing devices having at least one processor configured to execute software instructions stored on a computer readable tangible, non-transitory medium. For example, a server can include one or more computers operating as a web server, database server, or other type of computer server in a manner to fulfill described roles, responsibilities, or functions.

[0122] One should appreciate that the systems and methods described herein may improve computing functionality by providing an innovative way to flag malicious software.

[0123] Various example embodiments are described herein. Although each embodiment represents a single combination of inventive elements, all possible combinations of the disclosed elements include the inventive subject matter. Thus if one embodiment comprises elements A, B, and C, and a second embodiment comprises elements B and D, then the inventive subject matter is also considered to include other remaining combinations of A, B, C, or D, even if not explicitly disclosed.

[0124] The term “connected” or "coupled to" may include both direct coupling (in which two elements that are coupled to each other contact each other) and indirect coupling (in which at least one additional element is located between the two elements).

[0125] The technical solution of embodiments may be in the form of a software product. The software product may be stored in a non-volatile or non-transitory storage medium, which can be a compact disk read-only memory (CD-ROM), a USB flash disk, or a removable hard disk. The software product includes a number of instructions that enable a computer device (personal computer, server, or network device) to execute the methods provided by the embodiments.

[0126] The embodiments described herein are implemented by physical computer hardware, including computing devices, servers, receivers, transmitters, processors, memory, displays, and networks. The embodiments described herein provide useful physical machines and particularly configured computer hardware arrangements. The embodiments described herein are directed to electronic machines and methods implemented by electronic machines

adapted for processing and transforming electromagnetic signals which represent various types of information.

[0127] Although the embodiments have been described in detail, it should be understood that various changes, substitutions and alterations can be made herein without departing from the scope as defined by the appended claims.

[0128] Moreover, the scope of the present application is not intended to be limited to the particular embodiments of the process, machine, manufacture, composition of matter, means, methods and steps described in the specification. As one of ordinary skill in the art will readily appreciate from the disclosure of the present invention, processes, machines, manufacture, compositions of matter, means, methods, or steps, presently existing or later to be developed, that perform substantially the same function or achieve substantially the same result as the corresponding embodiments described herein may be utilized. Accordingly, the appended claims are intended to include within their scope such processes, machines, manufacture, compositions of matter, means, methods, or steps.

[0129] As can be understood, the features described above and illustrated are intended to be examples.

WHAT IS CLAIMED IS:

1. A code auditor comprising:

an extractor configured to accept as input source code and extract source code features including source code functions, and to accept as input binary code and extract binary features, including binary functions;

a matching block configured to obtain a correspondence between the source code functions and the binary code functions by comparing a comparison selection of the source code features to a comparison selection of the binary code features; and

an accountability block configured to compare an accountability selection of features of the binary code features characterizing functions of the binary code functions with corresponding features of the source code features characterizing the corresponding functions of the source code functions to make a determination of the accountability of the binary code functions in terms of the source code functions,

the source code auditor to output a classification file of safe functions and suspicious functions based on the accountability of the binary code functions in terms of the source code functions.

2. The code auditor of claim 1 wherein the code auditor is configured to use machine learning to extract the source code features and the binary features and obtain the correspondence by determining an optimal set of 1-1 feature pairs in the source code and binary code.

3. The code auditor of claim 1 wherein the comparison selection implements a matching process based on a bi-partite match.

4. The code auditor of claim 1 wherein the safe functions are present in the source code functions and the binary code functions, wherein the suspicious functions are not present in the source code functions but are present in the binary code functions.

5. The code auditor of claim 1 wherein the comparison selection of the binary code features is different from the accountability selection of the features of the binary code features characterizing the function.

6. The code auditor of claim 1 wherein the comparison selection of the second set of features is identical to the accountability selection of the features of the binary code features characterizing the function.

7. The code auditor of claim 1 wherein the accountability selection is more strict than the comparing in the comparison selection.

8. The code auditor of claim 1 wherein the comparison selection of the source code features and the comparison selection of the binary code features each comprise function call graphs.

9. The code auditor of claim 1 wherein the comparison selection between the source code functions and the binary code functions is carried out iteratively, and the correspondence obtained at each non-final iterative step are used in the comparison of features in a subsequent iteration.

10. The code auditor of claim 1 wherein the comparison selection involves refinement in which estimated correspondence is refined by comparing the number of conditional branches of the first code body functions to the number of conditional branches of the second code body functions.

11. The code auditor of claim 1 wherein the accountability selection includes string constants.

12. The code auditor of claim 1 wherein the accountability selection includes integer constants.

13. The code auditor of claim 1 wherein differences in integer constants less than a threshold are treated as the same.

14. The code auditor of claim 1 wherein the binary code is a transformation of the source code, wherein the source code and the object code each have a call graph.

15. The code auditor of claim 1 wherein the accountability selection includes callees.

16. The code auditor of claim 1 wherein the accountability selection includes library calls.

17. The code auditor of claim 1 wherein the comparison selection, for each pair composed of a function of the source code functions and a function of the binary code functions, a value is computed representing a measure of a difference between the comparison selection of the features characterizing the function of the pair from the source code functions and the comparison selection of the features characterizing the function of the pair from the binary code functions, and the estimation of the correspondence is carried out based on minimizing the computed values for pairs whose members are estimated to correspond.

18. The code auditor of claim 1 wherein the computed value is a weighted average.

19. The code auditor of claim 1 wherein the weights are determined using machine learning.

20. The code auditor of claim 17 wherein minimizing the computed values for pairs whose members are estimated to correspond comprises assigning the computed values as edge weights on a bipartite graph.

21. The code auditor of claim 1 wherein the matching block implements machine learning rules to determine an optimal set of feature pairs.

22. A system for identifying suspicious features in code comprising:

one or more hardware processors; and

a memory coupled to the one or more hardware processors, the memory comprising:

extractive logic that, when executed by the one or more hardware processors, receives as input source code and binary code, and extracts a first set of code features and a second set of code features, including a first set of functions and a second set of functions,

matching logic that, when executed by the one or more hardware processors, estimates a correspondence between the first set of functions and the second set of functions by comparing a comparison selection of the first set of features to a comparison selection of the second set of features, and

accountability logic that, when executed by the one or more hardware processors, compares an accountability selection of features of the second set of features characterizing functions of the second set of functions with corresponding

features of the first set of features characterizing the corresponding functions of the first set of functions to make a determination of the accountability of functions of the second set of functions in terms of the first set of functions.

23. A computer program product for identifying suspicious features in binary code, comprising:

a computer recordable storage medium having stored therein computer executable program code, which when executed by a computer hardware system, causes the computer hardware system to perform:

extracting, using a first extractor, source code features from the source code, the source code features including source code functions extracted from the source code and source code features characterizing the source code functions;

extracting, using a second extractor, binary code features from the binary code, the binary code features including binary code functions extracted from the binary code and binary code features characterizing the binary code functions;

obtaining, in a matching block, a correspondence between the source code functions and the binary code functions by comparing a comparison selection of the binary code features characterizing the binary code functions to a comparison selection of the source code features characterizing the source code functions; and

classifying, in an accountability blocks, functions in the binary code functions into either a first category or a second category according to a comparison of an accountability selection of the binary code features characterizing the binary code functions with corresponding source code features characterizing the respective source code functions estimated to correspond to the binary code functions.

24. A method of classifying code elements in binary code into a first category accounted for in source code and a second category, the method comprising:

extracting, using a first extractor, a first set of features from the source code, the first set of features including a first set of functions;

extracting, using a second extractor, a second set of features from the binary code, the second set of features including a second set of functions;

obtaining, in a matching block, a correspondence between the first set of functions and the second set of functions by comparing a comparison selection of the first set of features to a comparison selection of the second set of features; and

classifying, in an accountability block, each function of the second set of functions into the first category or the second category according to a comparison of an accountability selection of features of the second set of features characterizing the function with corresponding features of the first set of features characterizing the respective function of the first set of functions estimated to correspond with the function.

25. The method of claim 24 in which the comparison selection of the second set of features is different from the accountability selection of the features of the second set of features characterizing the function.

26. The method of claim 24 in which the comparison selection of the second set of features is identical to the accountability selection of the features of the second set of features characterizing the function.

27. The method of claim 26 in which the comparison in the step of classifying each function of the second set of functions is more strict than the comparing in the step of estimating the correspondence.

28. The method of claim 24 in which the comparison selection of the first set of features and the comparison selection of the second set of features each comprise function call graphs.

29. The method of claim 24 in which the step of estimating a correspondence between the first set of functions and the second set of functions is carried out iteratively, and the correspondence obtained at each non-final iterative step are used in the comparison of features in a subsequent iterative step.

30. The method of claim 24 in which the step of estimating a correspondence between the first set of functions and the second set of functions comprises a refinement step in which the estimated correspondence is refined by comparing the number of conditional branches of the

first code body functions to the number of conditional branches of the second code body functions.

31. The method of claim 24 in which the accountability selection includes string constants.

32. The method of claim 24 in which the accountability selection includes integer constants.

33. The method of claim 32 in which differences in integer constants less than a threshold are treated as the same.

34. The method of claim 24 wherein the binary code is a transformation of the source code, the source code and the object code each having a call graph.

35. The method of claim 24 in which the accountability selection includes callees.

36. The method of claim 24 in which the accountability selection includes library calls.

37. The method of claim 24 in which in the step of estimating a correspondence between the first set of functions and the second set of functions, for each pair composed of a function of the first set of functions and a function of the second set of functions, a value is computed representing a measure of a difference between the comparison selection of the features characterizing the function of the pair from the first set of functions and the comparison selection of the features characterizing the function of the pair from the second set of functions, and the estimation of the correspondence is carried out based on minimizing the computed values for pairs whose members are estimated to correspond.

38. The method of claim 37 in which the computed value is a weighted average.

39. The method of claim 38 in which the weights are determined using machine learning.

40. The method of claim 37 in which minimizing the computed values for pairs whose members are estimated to correspond comprises assigning the computed values as edge weights on a bipartite graph.

41. A code auditor comprising:

an extractor configured to accept as input a first body of code and extract a first set of features including a first set of functions, and to accept as input a second body of code and

extract a second set of features, including a second set of functions, the second body of code being a transformation of the first body of code, wherein the first body of code and the second body of code each have a call graph;

a matching block configured to obtain a correspondence between the first set of functions and the second set of functions by comparing a comparison selection of the first set of features to a comparison selection of the second set of features; and

an accountability block configured to compare an accountability selection of features of the second set of features characterizing functions of the second set of functions with corresponding features of the first set of features characterizing the corresponding functions of the first set of functions to make a determination of the accountability of the second set of functions in terms of the first set of functions.

42. A system for identifying suspicious features in code comprising:

one or more hardware processors; and

a memory coupled to the one or more hardware processors, the memory comprising:

extractive logic that, when executed by the one or more hardware processors, receives as input a first body of code and a second body of code, and extracts a first set of code features and a second set of code features, including a first set of functions and a second set of functions,

matching logic that, when executed by the one or more hardware processors, estimates a correspondence between the first set of functions and the second set of functions by comparing a comparison selection of the first set of features to a comparison selection of the second set of features, and

accountability logic that, when executed by the one or more hardware processors, compares an accountability selection of features of the second set of features characterizing functions of the second set of functions with corresponding features of the first set of features characterizing the corresponding functions of the first set of functions to make a determination of the accountability of functions of the second set of functions in terms of the first set of functions.

43. A computer program product for identifying suspicious features in binary code, comprising:

a computer recordable storage medium having stored therein computer executable program code, which when executed by a computer hardware system, causes the computer hardware system to perform:

extracting, using a first extractor, source code features from the source code, the source code features including source code functions extracted from the source code and source code features characterizing the source code functions;

extracting, using a second extractor, binary code features from the binary code, the binary code features including binary code functions extracted from the binary code and binary code features characterizing the binary code functions;

obtaining, in a matching block, a correspondence between the source code functions and the binary code functions by comparing a comparison selection of the binary code features characterizing the binary code functions to a comparison selection of the source code features characterizing the source code functions; and

classifying, in an accountability blocks, functions in the binary code functions into either a first category or a second category according to a comparison of an accountability selection of the binary code features characterizing the binary code functions with corresponding source code features characterizing the respective source code functions estimated to correspond to the binary code functions.

44. A method of classifying code elements in a second body of code into a first category accounted for in a first body of code and a second category, the method comprising:

extracting, using a first extractor, a first set of features from the first body of code, the first set of features including a first set of functions;

extracting, using a second extractor, a second set of features from the second body of code, the second set of features including a second set of functions;

obtaining, in a matching block, a correspondence between the first set of functions and the second set of functions by comparing a comparison selection of the first set of features to a comparison selection of the second set of features; and

classifying, in an accountability block, each function of the second set of functions into the first category or the second category according to a comparison of an accountability selection of features of the second set of features characterizing the function with corresponding features of the first set of features characterizing the respective function of the first set of functions estimated to correspond with the function.

45. The method of claim 44 in which the comparison selection of the second set of features is different from the accountability selection of the features of the second set of features characterizing the function.

46. The method of claim 44 in which the comparison selection of the second set of features is identical to the accountability selection of the features of the second set of features characterizing the function.

47. The method of claim 46 in which the comparison in the step of classifying each function of the second set of functions is more strict than the comparing in the step of estimating the correspondence.

48. The method of claim 44 in which the comparison selection of the first set of features and the comparison selection of the second set of features each comprise function call graphs.

49. The method of claim 44 in which the step of estimating a correspondence between the first set of functions and the second set of functions is carried out iteratively, and the correspondence obtained at each non-final iterative step are used in the comparison of features in a subsequent iterative step.

50. The method of claim 44 in which the step of estimating a correspondence between the first set of functions and the second set of functions comprises a refinement step in which the estimated correspondence is refined by comparing the number of conditional branches of the first code body functions to the number of conditional branches of the second code body functions.

51. The method of claim 44 in which the accountability selection includes string constants.

52. The method of claim 44 in which the accountability selection includes integer constants.

53. The method of claim 52 in which differences in integer constants less than a threshold are treated as the same.

54. The method of claim 53 in which the threshold is 8.

55. The method of claim 54 in which the accountability selection includes callees.

56. The method of claim 54 in which the accountability selection includes library calls.

57. The method of claim 54 in which in the step of estimating a correspondence between the first set of functions and the second set of functions, for each pair composed of a function of the first set of functions and a function of the second set of functions, a value is computed representing a measure of a difference between the comparison selection of the features characterizing the function of the pair from the first set of functions and the comparison selection of the features characterizing the function of the pair from the second set of functions, and the estimation of the correspondence is carried out based on minimizing the computed values for pairs whose members are estimated to correspond.

58. The method of claim 57 in which the computed value is a weighted average.

59. The method of claim 58 in which the weights are determined using machine learning.

60. The method of claim 57 in which minimizing the computed values for pairs whose members are estimated to correspond comprises assigning the computed values as edge weights on a bipartite graph.

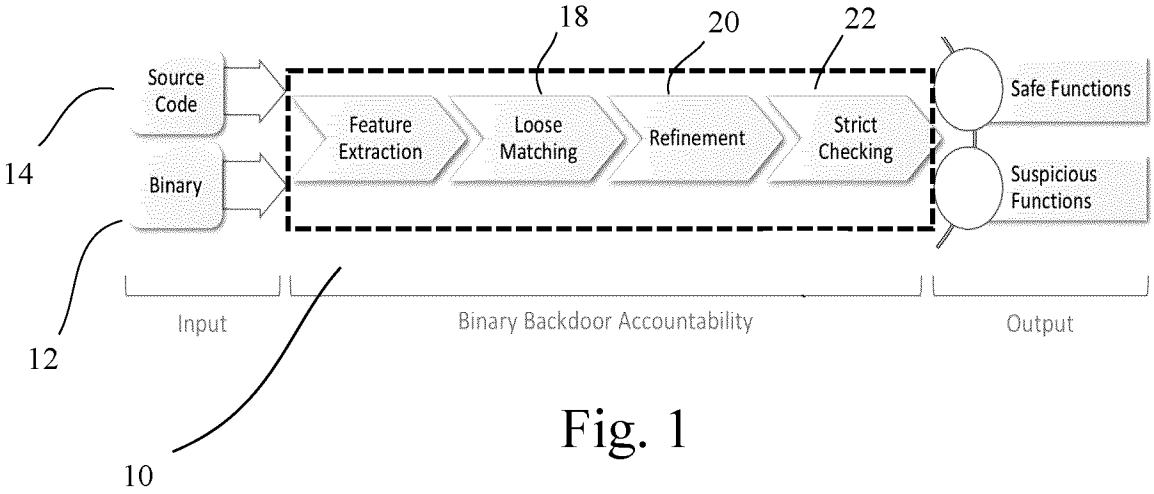

18
20
22
Source Code
Binary
Feature Extraction
Loose Matching
Refinement
Strict Checking
Safe Functions
Suspicious Functions
14
12
10
Input
Binary Backdoor Accountability
Output

Fig. 1

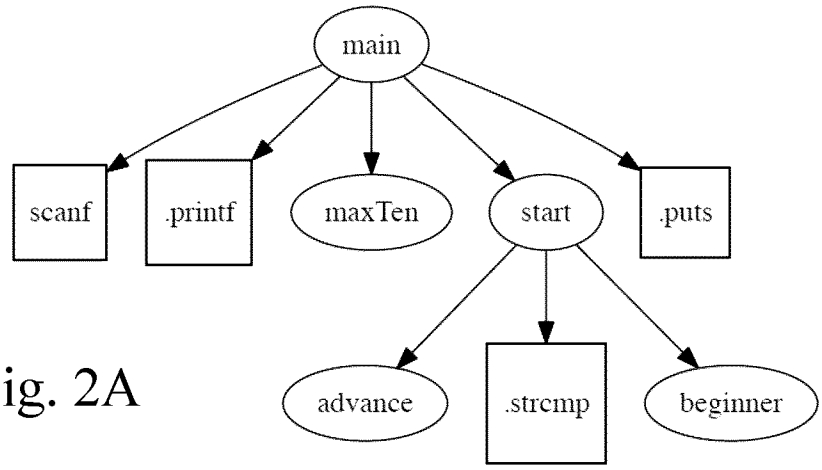

main
scanf
.printf
maxTen
start
.puts
advance
.strcmp
beginner

Fig. 2A

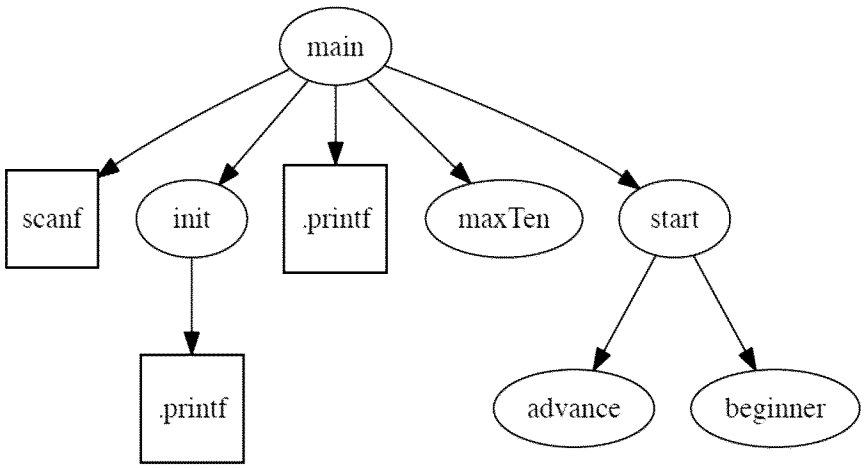

main
scanf
init
.printf
maxTen
start
.printf
advance
beginner

Fig. 2B

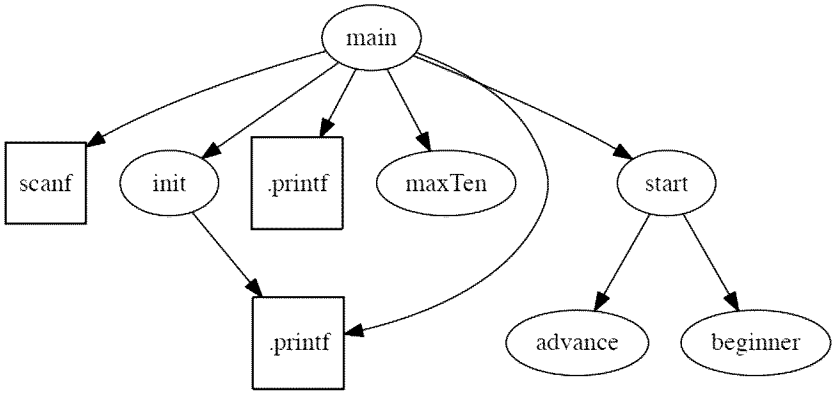

main
scanf
init
.printf
maxTen
start
.printf
advance
beginner

Fig. 2C

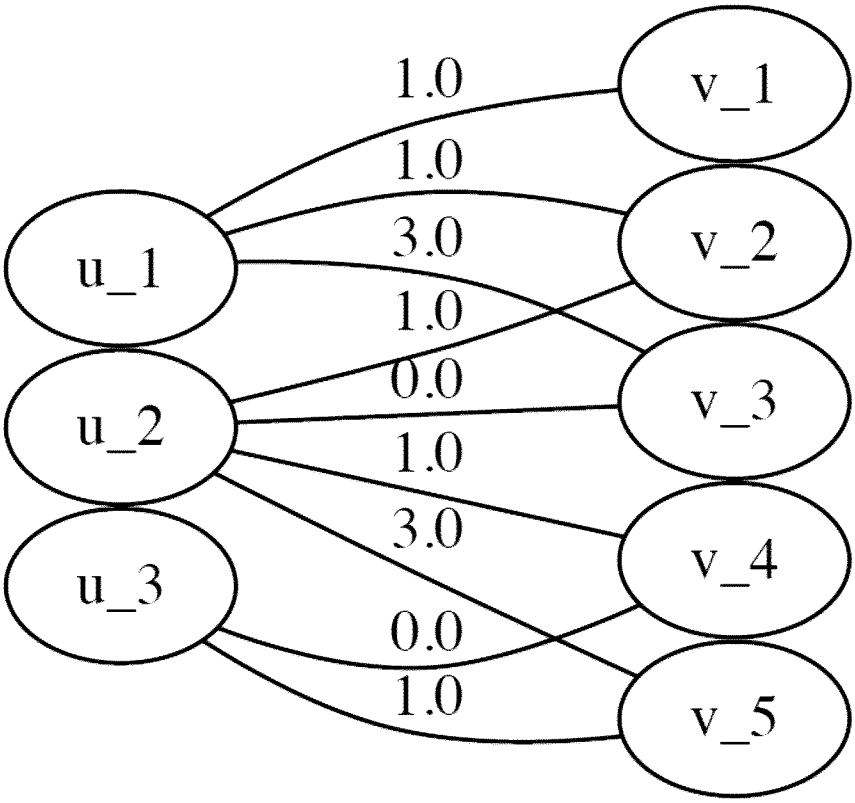


Fig. 3

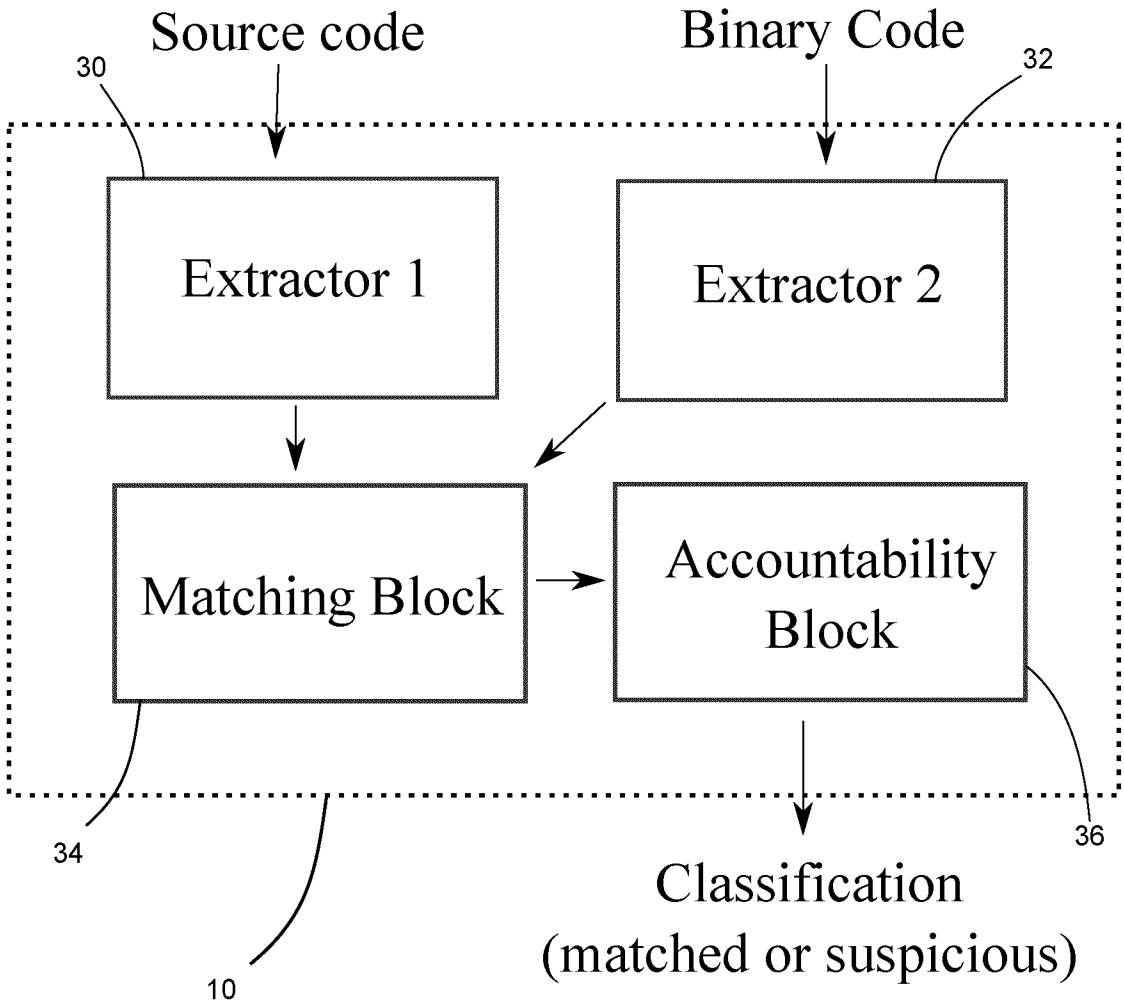

Source code
Binary Code
30
32
Extractor 1
Extractor 2
Matching Block
Accountability Block
34
36
10
Classification
(matched or suspicious)

Fig. 4

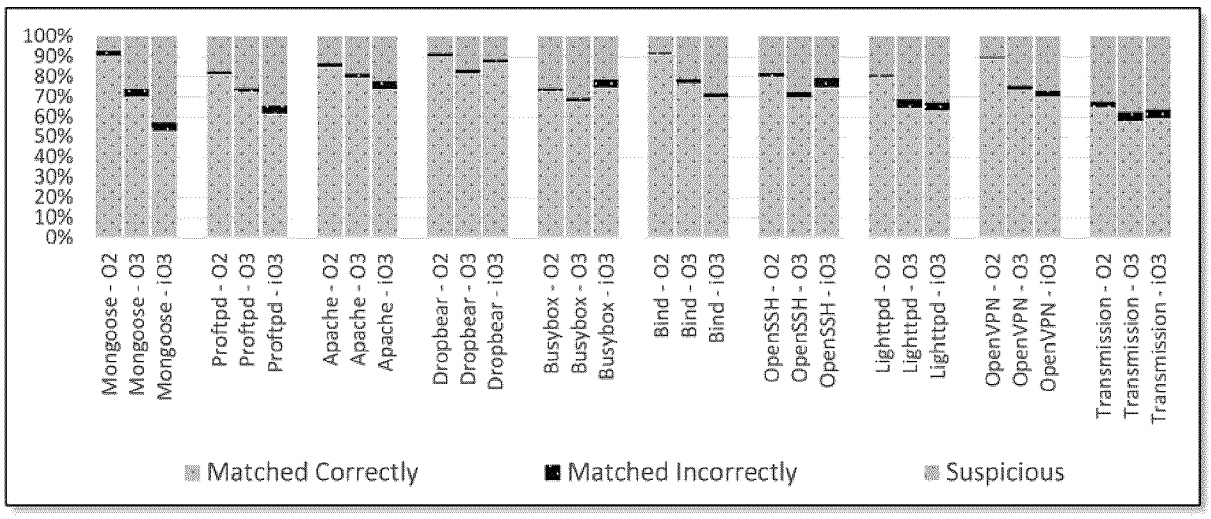

100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
Mongoose - O2
Mongoose - O3
Mongoose - iO3
Proftpd - O2
Proftpd - O3
Proftpd - iO3
Apache - O2
Apache - O3
Apache - iO3
Dropbear - O2
Dropbear - O3
Dropbear - iO3
Busybox - O2
Busybox - O3
Busybox - iO3
Bind - O2
Bind - O3
Bind - iO3
OpenSSH - O2
OpenSSH - O3
OpenSSH - iO3
Lighttpd - O2
Lighttpd - O3
Lighttpd - iO3
OpenVPN - O2
OpenVPN - O3
OpenVPN - iO3
Transmission - O2
Transmission - O3
Transmission - iO3
Matched Correctly
Matched Incorrectly
Suspicious

Fig. 5

```
; void __cdecl start(char *name, int level)
                public start
start           proc near               ; CODE XREF: main+65↓p

level           = dword ptr -0Ch
name            = qword ptr -8

                push    rbp
                mov     rbp, rsp
                sub     rsp, 10h
                mov     [rbp+name], rdi
                mov     [rbp+level], esi
                mov     rax, [rbp+name]
                mov     esi, offset s2  ; "LegendaryHack"
                mov     rdi, rax        ; s1
                call    _strcmp
                test    eax, eax
                jnz     short locret_4006CD
                cmp     [rbp+level], 4
                jg      short loc_4006BC
                mov     edx, [rbp+level]
                mov     rax, [rbp+name]
                mov     esi, edx        ; level
                mov     rdi, rax        ; name
                call    beginner
                jmp     short locret_4006CD
; ---------------------------------------------------------------------------

loc_4006BC:                             ; CODE XREF: start+28↑j
                mov     edx, [rbp+level]
                mov     rax, [rbp+name]
                mov     esi, edx        ; level
                mov     rdi, rax        ; name
                call    advance

locret_4006CD:                          ; CODE XREF: start+22↑j
```

Backdoor Injected

Fig. 6

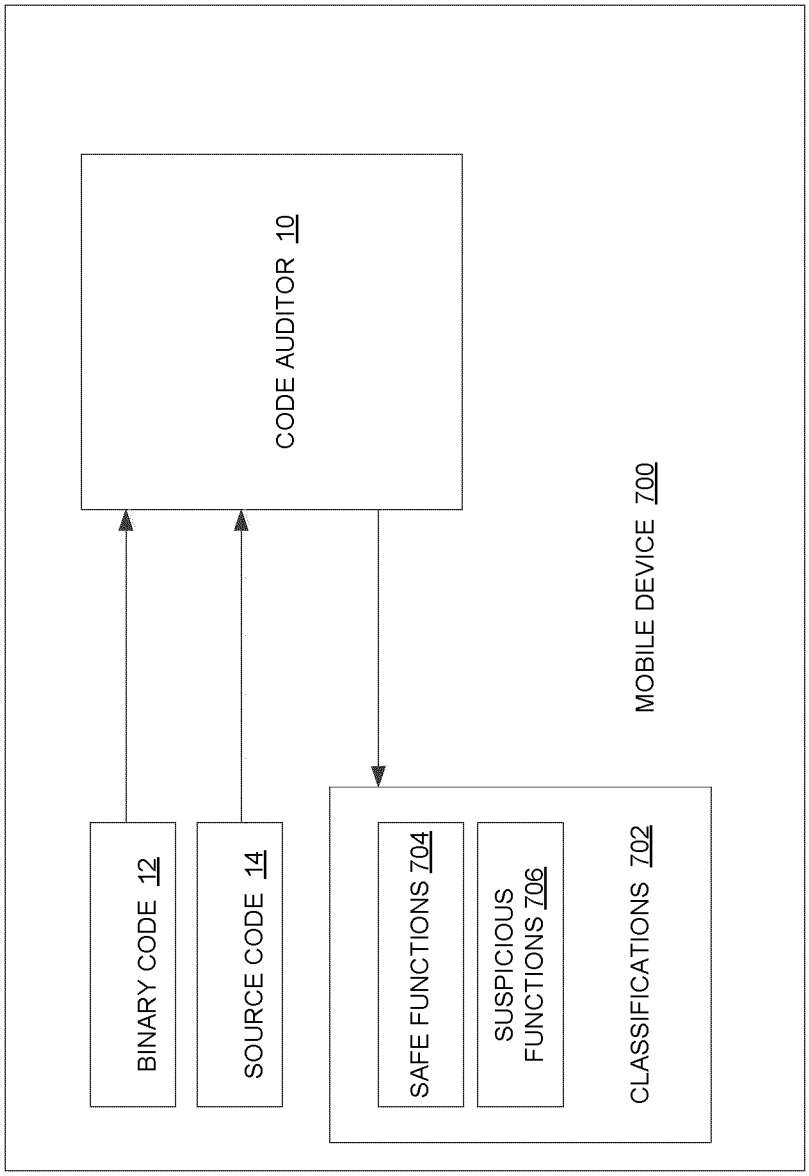

MOBILE DEVICE 700
CODE AUDITOR 10
BINARY CODE 12
SOURCE CODE 14
CLASSIFICATIONS 702
SAFE FUNCTIONS 704
SUSPICIOUS FUNCTIONS 706

FIG. 7

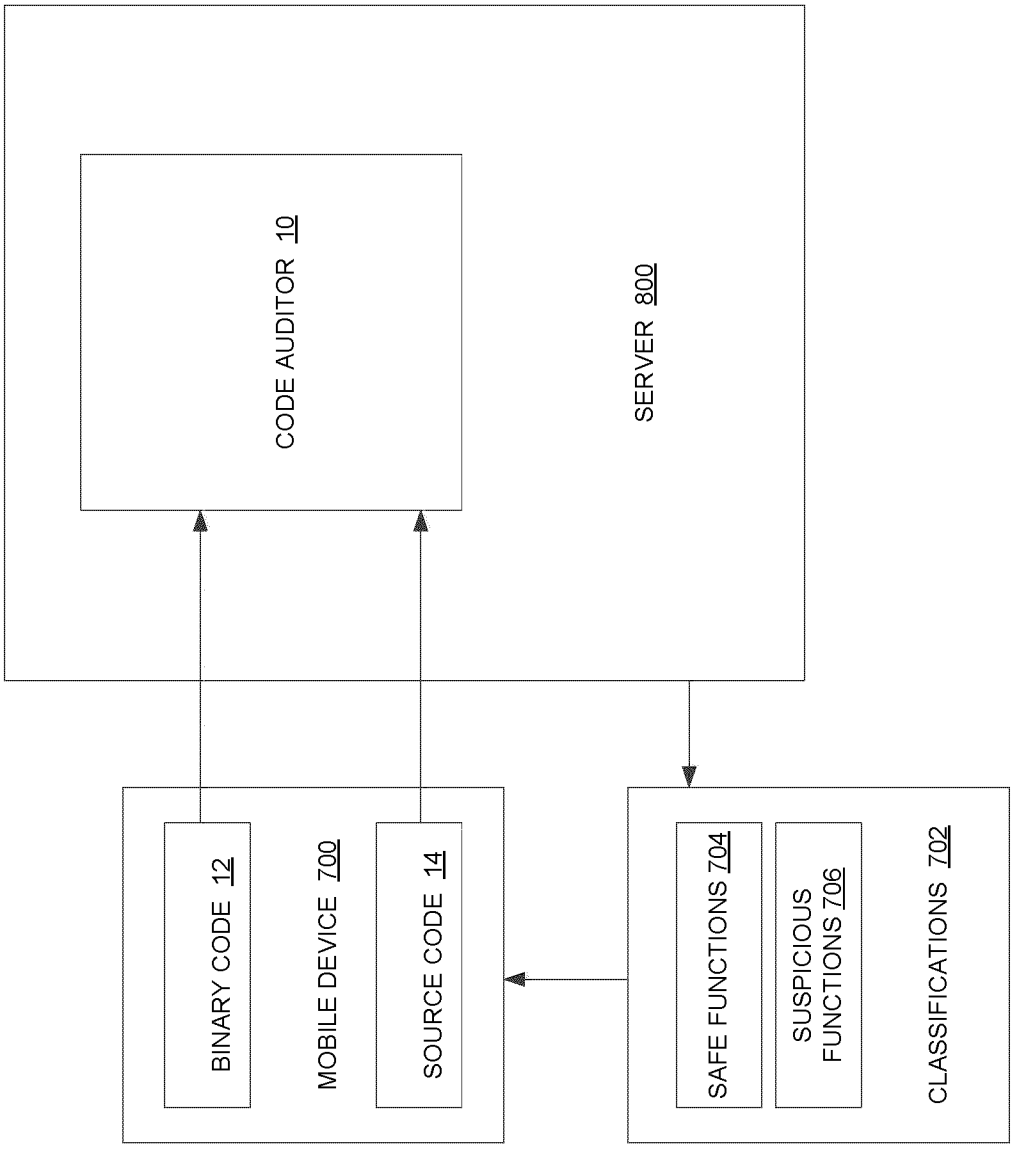

SERVER 800
CODE AUDITOR 10
BINARY CODE 12
MOBILE DEVICE 700
SOURCE CODE 14
SAFE FUNCTIONS 704
SUSPICIOUS FUNCTIONS 706
CLASSIFICATIONS 702

FIG. 8

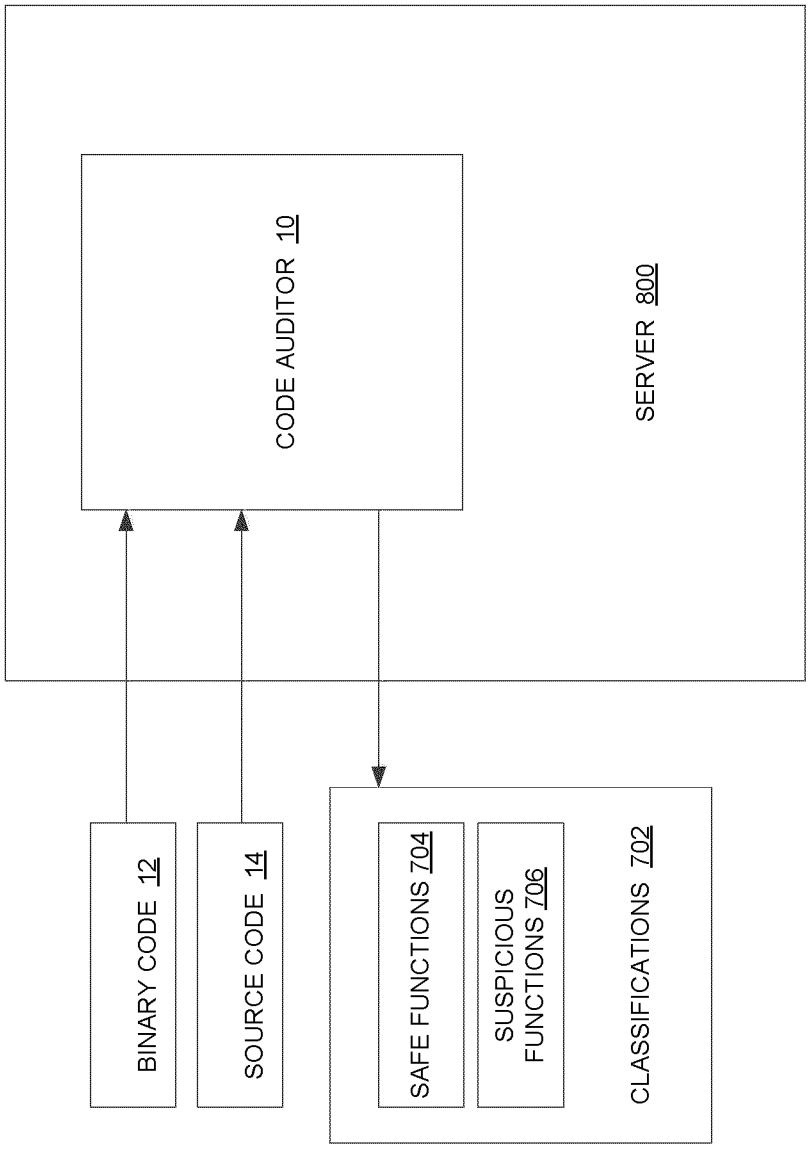

SERVER 800
CODE AUDITOR 10
BINARY CODE 12
SOURCE CODE 14
CLASSIFICATIONS 702
SAFE FUNCTIONS 704
SUSPICIOUS FUNCTIONS 706

FIG. 9

INTERNATIONAL SEARCH REPORT

International application No.
PCT/CA2017/050599

A. CLASSIFICATION OF SUBJECT MATTER
IPC: ***G06F 21/56*** (2013.01) , ***G06F 21/57*** (2013.01)

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F* (2013.01)(all subclasses in combination with keywords)

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic database(s) consulted during the international search (name of database(s) and, where practicable, search terms used)
Questel Orbit, Google, Canadian Patent Database (keywords: source code, binary code, code audit, suspect, correspondence, matching, malicious, backdoor)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	Eschweiler et al. : "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code". NDSS '16, 21-24 February 2016, San Diego, CA, USA. Copyright 2016 Internet Society, ISBN 1-891562-41-X, http://dx.doi.org/10.14722/ndss.2016.23185. • See entire document	22, 23, 42, 43
A	Wysopal et al.:" Static detection of application backdoors". Datenschutz und Datensicherheit – DuD. March 2010, Volume 34, Issue 3, pp 149–155. • See entire document	22, 23, 42, 43
A	Ming et al.: "MalwareHunt: semantics-based malware diffing speedup by normalized basic block memorization". Journal of Computer Virology and Hacking Techniques pp 1–12, First online: 17 May 2016. • See entire document	22, 23, 42, 43

☑ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier application or patent but published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search 14 July 2017 (14-07-2017)	Date of mailing of the international search report 19 July 2017 (19-07-2017)
Name and mailing address of the ISA/CA Canadian Intellectual Property Office Place du Portage I, C114 - 1st Floor, Box PCT 50 Victoria Street Gatineau, Quebec K1A 0C9 Facsimile No.: 819-953-2476	Authorized officer Saadia Khan (819) 576-2161

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	Bourquin et al.: "BinSlayer: accurate comparison of binary executables", Proceeding PPREW '13 Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, Article No. 4, Rome, Italy — January 26 - 26, 2013. ACM New York, NY, USA ©2013. ISBN: 978-1-4503-1857-0, doi>10.1145/2430553.2430557 • See entire document	22, 23, 42, 43

INTERNATIONAL SEARCH REPORT

International application No.
PCT/CA2017/050599

Box No. II Observations where certain claims were found unsearchable (Continuation of item 2 of the first sheet)

This international search report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. ☑ Claim Nos.: 1-21, 24-41, 44-60
because they relate to subject matter not required to be searched by this Authority, namely:

Claims 1-21, 24-41, 44-60 are directed to a scientific and/or mathematical theory, which the International Searching Authority is not required to search under PCT Rule 39.1(i). Claims 1-21, 24-41, 44-60 are of an abstract character or do not provide a practical application having a useful, concrete, and tangible result. Claims 1-21, 24-41, 44-60 are directed to a scheme, rule or a method of doing business, which the International Searching Authority is not required to search under PCT Rule 39.1(iii). The claimed subject matter, when viewed as a whole, is of abstract character or does not provide a practical application having a useful, concrete, and tangible result.

2. ☐ Claim Nos.:
because they relate to parts of the international application that do not comply with the prescribed requirements to such an extent that no meaningful international search can be carried out, specifically:

3. ☐ Claim Nos.:
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

Box No. III Observations where unity of invention is lacking (Continuation of item 3 of first sheet)

This International Searching Authority found multiple inventions in this international application, as follows:

1. ☐ As all required additional search fees were timely paid by the applicant, this international search report covers all searchable claims.

2. ☐ As all searchable claims could be searched without effort justifying additional fees, this Authority did not invite payment of additional fees.

3. ☐ As only some of the required additional search fees were timely paid by the applicant, this international search report covers only those claims for which fees were paid, specifically claim Nos.:

4. ☐ No required additional search fees were timely paid by the applicant. Consequently, this international search report is restricted to the invention first mentioned in the claims; it is covered by claim Nos.:

Remark on Protest

☐ The additional search fees were accompanied by the applicant's protest and, where applicable, the payment of a protest fee.

☐ The additional search fees were accompanied by the applicant's protest but the applicable protest fee was not paid within the time limit specified in the invitation.

☐ No protest accompanied the payment of additional search fees.