(54) Title: SYSTEM AND METHOD FOR RANDOM ALGORITHM SELECTION TO DYNAMICALLY CONCEAL THE OPERATION OF SOFTWARE

(74) Agents: SMITH, Dallas, F. et al; Gowing Lafleur Henderson LLP., 160 Elgin Street, Suite 2600, Ottawa, Ontario KIP 1C3 (CA).


(84) Designated States (unless otherwise indicated, for every kind of regional protection available): ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (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, SE, SI, SK, SM,.... [Continued on next page]

FIGURE 2
Declarations under Rule 4.17:
— as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(H))

Published:
— with international search report (Art. 21(3))
FIELD OF THE INVENTION

The present invention relates generally to software that is resistant to unauthorized analysis. More particularly, the present invention relates to systems and methods for the production of software code that randomizes usage of functionally equivalent algorithms such that analysis of the code either during run-time or during an attempt of reverse engineering is made more difficult.

BACKGROUND OF THE INVENTION

In the field of computing, software typically exhibits modular characteristics rather than being monolithic. Moreover, there are oftentimes a number of separate and distinct algorithms employed within any given piece of software. Such disparate algorithms combine in such a manner so as to provide services (i.e., functionalities) that are needed by the software. It is often the case that for one particular service, many different algorithms are available. Generally speaking, an algorithm in this scenario is a sequence of computational steps that carries out a task or a set of tasks. An algorithm can have various sizes. It can be very large, or it can be as small as a set of a few instructions. An algorithm can contain smaller algorithms, which in turn can contain even smaller algorithms. This hierarchy may have an} number of levels.

It is well understood that software can be reverse engineered. Moreover, such reverse engineering is undesirable in many commercial applications. To combat such undesirable reverse engineering, technologies exist to transform software programs into formats that are difficult to understand and to reverse engineer. Such transformations often result in much more complicated, and therefore disadvantageous, implementations of the original definition in the source code of the application.
Although the use of software transformations makes it difficult to reverse engineer the application, an attacker may further derive valuable information from the execution of the application. Indeed, such run time observations can provide a great amount of information. During the execution, an attacker may use a myriad of known techniques in order to observe the usage pattern. Such techniques include known methods of static code analysis or dynamic code analysis including debugging attacks, side channel analysis, or other similar mechanisms. Applying such techniques to the same algorithm produces information that forms a good basis for further attacks.

It is, therefore, desirable to provide more resistance to software against unauthorized analysis.

**SUMMARY OF THE INVENTION**

It is an object of the present invention to obviate or mitigate at least one disadvantage of previous attempts to provide software that is resistant to unauthorized analysis. To that end, the present invention provides a system and method for the production of software code that randomizes usage of functionally equivalent algorithms in order to deter an useful analysis of the code either during run-time or during an attempt of reverse engineering.

In a first aspect, the present invention provides a method of concealing computer software source code including: identifying at least one sequence of computational steps embodied in a computer software source code of a computer program; establishing a set of alternative implementations of the at least one sequence of computational steps; obtaining one alternative implementation randomly selected from the set of alternative implementations; and replacing the at least one sequence with the one alternative implementation to form an alternative embodiment of the computer software source code; wherein the computer software source code and the alternative embodiment of the computer software source code are diverse instances of functionally equivalent code.

In a further aspect, the present invention provides a system for concealing computer software source code, the system including: a set of machine executable code segments operable to produce software code that randomizes usage of functionally equivalent sequences of computational steps contained in the computer software source...
code, the machine executable code executable to perform the steps of: identifying at least one sequence of computational steps embodied in a computer software source code of a computer program; establishing a set of alternative implementations of the at least one sequence of computational steps; obtaining one alternative implementation randomly selected from the set of alternative implementations; and replacing the at least one sequence with the one alternative implementation to form an alternative embodiment of the computer software source code; wherein the computer software source code and the alternative embodiment of the computer software source code are diverse instances of functionally equivalent code.

In another aspect, the present invention provides an apparatus for concealing computer software source code including: means for identifying at least one sequence of computational steps embodied in a computer software source code of a computer program; means for establishing a set of alternative implementations of the at least one sequence of computational steps; means for obtaining one alternative implementation randomly selected from the set of alternative implementations; and means for replacing the at least one sequence with the one alternative implementation to form an alternative embodiment of the computer software source code; wherein the computer software source code and the alternative embodiment of the computer software source code are diverse instances of functionally equivalent code.

In still another aspect, the present invention provides a computer readable memory medium storing computer software code for concealing computer software source code, the computer software code executable to perform the steps of: identifying at least one sequence of computational steps embodied in a computer software source code of a computer program; establishing a set of alternative implementations of the at least one sequence of computational steps; obtaining one alternative implementation randomly selected from the set of alternative implementations; and replacing the at least one sequence with the one alternative implementation to form an alternative embodiment of the computer software source code; wherein the computer software source code and the alternative embodiment of the computer software source code are diverse instances of functionally equivalent code.
Other aspects and features of the present invention will become apparent to those ordinarily skilled in the art upon review of the following description of specific embodiments of the invention in conjunction with the accompanying figures.

BRIEF DESCRIPTION OF THE DRAWINGS

5 Embodiments of the present invention will now be described, by way of example only, with reference to the attached Figures.

FIGURE 1 is illustrates a known computer system in which the present invention may be embodied.

FIGURE 2 is a flow chart illustrating one embodiment of random algorithm selection at build time in accordance with the present invention.

FIGURE 2A is a flow chart illustrating another embodiment of random algorithm selection at build time in accordance with the present invention.

FIGURE 3 is a flow chart illustrating generation of alternative algorithms used by the random algorithm selection in accordance with the present invention.

FIGURE 4 is a flow chart illustrating random algorithm selection at run time in accordance with the present invention.

FIGURE 5 is a block diagram illustrating the dispatching of alternative algorithms in accordance with the present invention.

FIGURE 6 is an illustration of an example of alternative algorithm arrangement in memory in accordance with the present invention.

FIGURE 7 is an illustration of an example of multi-level random algorithm selection in accordance with the present invention.

FIGURE 8 is a block diagram illustrating an alternative dispatching mechanism for the dispatching of alternative algorithms in accordance with the present invention.

DETAILED DESCRIPTION

Generally, the present invention provides a method and system for producing software code that randomizes usage of functionally equivalent algorithms contained in the code. This effectively deters any useful analysis of the code by an attacker either during run-time or during an attempt of reverse engineering.
An example of a computer system upon which the invention may be performed is presented as a block diagram in FIGURE 1. This computer system 110 includes a display 112, keyboard 114, computer 116 and external devices 118.

The computer 116 may contain one or more processors or microprocessors, such as a central processing unit (CPU) 120. The CPU 120 performs arithmetic calculations and control functions to execute software stored in an internal memory 122, preferably random access memory (RAM) and/or read only memory (ROM), and possibly additional memory 124. The additional memory 124 may include, for example, mass memory storage, hard disk drives, floppy disk drives, magnetic tape drives, compact disk drives, program cartridges and cartridge interfaces such as those found in video game devices, removable memory chips such as EPROM or PROM, or similar storage media as known in the art. This additional memory 124 may be physically internal to the computer 116, or external as in FIGURE 1.

The computer system 110 may also include other similar means for allowing computer programs or other instructions to be loaded. Such means can include, for example, a communications interface 126 which allows software and data to be transferred between the computer system 110 and external systems. Examples of communications interface 126 can include a modem, a network interface such as an Ethernet card, a serial or parallel communications port. Software and data transferred via communications interface 126 are in the form of signals which can be electronic, electromagnetic, and optical or other signals capable of being received by communications interface 126. Multiple interfaces, of course, can be provided on a single computer system 110.

Input and output to and from the computer 116 is administered by the input/output (I/O) interface 128. This I/O interface 128 administers control of the display 112, keyboard 114, external devices 118 and other such components of the computer system 110.

The invention is described in these terms for convenience purposes only. It would be clear to one skilled in the art that the invention may be applied to other computer or control systems 110. Such systems would include all manner of appliances having computer or processor control including telephones, cellular telephones, televisions.
television set top units, point of sale computers, automatic banking machines, lap top computers, servers, personal digital assistants and automobiles.

In the preferred embodiment, the invention is implemented in terms of an intermediate compiler program running on a computer system 110. Standard compiler techniques are well known in the art, and will not be reviewed in detail herein. Two standard references which may provide necessary background are "Compilers Principles, Techniques, and Tools" 1988 by Alfred Aho, Ravi Sethi and Jeffrey Ullman (ISBN 0-201-1008-6), and "Advanced Compiler Design & Implementation" 1997 by Steven Muchnick (ISBN 1-55860-320-4).

Generally, a software compiler is divided into three components, described as the front end, the middle, and the back end. The front end is responsible for language dependent analysis, while the back end handles the machine-dependent parts of code generation. Optionally, a middle component may be included to perform optimizations that are independent of language and machine. Typically, each compiler family will have only one middle, with a front end for each high-level language and a back end for each machine-level language. All of the components in a compiler family can generally communicate in a common intermediate language so that they are easily interchangeable. This intermediate language is generally in a form which exposes both control- and data-flow so that the form are easily manipulated. Such an intermediate form may be referred to as flow-exposed form. In the preferred embodiment of the invention, it is the intermediate code that will be manipulated to make the desired areas of the input software tamper-resistant.

The invention can most easily be applied to software code in Static Single Assignment (SSA) form. SSA is a well-known, popular and efficient flow-exposed form used by software compilers as a code representation for performing analyses and optimizations involving scalar variables. Effective algorithms based on SSA have been developed to address constant propagation, redundant computation detection, dead code elimination, induction variable elimination, and other requirements. Of course, the method of the invention could be applied to flow-exposed forms other than SSA, where these provide similar levels of semantic information, as in that provided in Gnu CC. Gnu CC software is currently available at no cost from the Free Software Foundation. Similarly, the method of the invention could be applied to software in its high level or low level
forms, if such forms were augmented with the requisite control-flow and data-flow information. This flexibility will become clear from the description of the encoding techniques described hereinafter.

As mentioned above, an algorithm is generally a sequence of computational steps that carries out a task or a set of tasks. In the present invention, the definition of algorithm should be understood to also encompass the implementations of algorithms. Therefore, an algorithm can be a set of computer instructions or a piece of high level software programming that carries out a task or a set of tasks on a computing device. For a given task in a piece of software in accordance with the present invention, the software normally selects one out of a set of N algorithms to use. All of the N algorithms are functionally equivalent, though no two algorithms are the same in terms of their specific implementation. Each algorithm in this set of N algorithms is called an alternative algorithm.

Random algorithm selection in accordance with the present invention can be realized for any segment of the software as long as there is a set of alternative algorithms for this segment of the software to choose from. It should therefore be understood that a segment could be the entire software if there are alternative algorithms (i.e., alternative implementations) of this software. Likewise, the segment could be a simple arithmetic operation if alternative algorithms (i.e., alternative operations) are available.

In accordance with the present invention, the random algorithm selection can happen in multiple levels. In any given known software implementation, hierarchy exists such that it is not uncommon that an algorithm uses other algorithms in its implementation, and the lower level algorithms in turn use other algorithms. Random algorithm selection in accordance with the present invention can be activated at any one or all levels in this hierarchy. As a result, a single algorithm can be diversified into many different instances based on the diversity of its sub-algorithms. Moreover, the number of diverse instances of the algorithm can be combinatorial! large. A simplified example of this would be that a given algorithm has three alternative algorithms, uses four sub-algorithms itself, and each of the sub-algorithms has five alternative algorithms. In such a scenario, the total number of diverse instances of this given algorithm is 3 * 4 * 5 = 60.
One feature of the present invention is that each alternative algorithm can be chosen from the set of alternative algorithms uniformly at random and independent from one another. An alternative feature may be that the probability distribution is not necessarily uniform. Rather, the probability of an alternative algorithm being chosen can be affected by external activities (i.e., actions applied externally to the software) and/or internal activities (i.e., actions generated within the software execution). For example, a user of the present invention can assign a weight to each alternative algorithm where the weight affects the frequency of each alternative algorithm being chosen. Therefore, based on the weight, certain alternative algorithms may be chosen more frequently than others.

This configurability enables the user to manage the trade-off between performance and security. Such trade-off is due to certain alternative algorithms being more efficient than others, and the efficiency of an alternative algorithm may change from platform to platform.

In accordance with the present invention, there can also be dependency between different sets of alternative algorithms. For instance, the result of choosing one alternative algorithm from a set “A” of alternative algorithms may affect the weight distribution for another set “B” of alternative algorithms and possibly other sets. This dependency among sets of alternative algorithms provides further ability for fine-tuning the performance of software. It should be understood however, that a weighting distribution that is readily apparent to one of ordinary skill in the programming art may result in a compromise in security. This could, of course, be considerable if this dynamic weight changing mechanism can be tampered with by the attacker. However, the present invention provided randomness at various granularities which helps compensate for usage patterns based on weight/dependency.

In accordance with the present invention, random algorithm selection can be static and/or dynamic. In other words, the selection from among alternative algorithms can be performed during the compilation and build cycle and/or is performed during the execution of the application at run-time. The former method produces diverse instances of the software statically. The latter, which is dynamic random algorithm selection, enables the software to be different from one run to another. These two methods can be combined.
which produces statically different copies of the software, and each cop} runs differently
from one execution to another, while all the executions are functionally equivalent.

In the case of dynamic random algorithm selection, the program contains all the
alternative algorithms. It is therefore desirable to make the software resistant to static code
analysis. That is to say, an attacker studying the binary of the executable without
executing the program should not give the attacker meaningful information regarding the
internal working of the software. M} known techniques exist to make software less
susceptible to static code analysis and can be utilized in a manner consistent with known
programming techniques. One such example includes the branch obfuscation technique of
“Tamper Resistant Software Control-flow Encoding” as disclosed in United States Patent
No. 6,799,114 issued to Chow et al. on August 17, 2004.

The details of such known technique of Chow et al. are not central to the present
invention and, as such, are not further described in detail herein. In general however, this
known technique obfuscates the branch instructions in the executable such that the} do not
jump to the correct addresses. The address resolves to the correct one only at run time
when the branch instruction is about to be processed. This technique also thwarts static
code analysis because the attacker cannot learn the correct control flow of the program
without executing the program. The attacker is thus forced to use dynamic analysis in
order to study the program. However, the result of each analysis will be different when
dynamic random algorithm selection is implemented in accordance with the present
invention. Using the inventive techniques including alternative algorithms in different
activations of the software during run time changes the usage pattern of a single algorithm
into a number of usage patterns to the alternative algorithms. Hence, the attacker no longer
can use the usage pattern to mount an attack on the service.

In accordance with the present invention and given the above description, it should
therefore be readily apparent that for a number of functional elements in a piece of
software (including the software itself) there are different algorithms and different
implementations of the algorithm. The present invention involves randomly selecting one
of these alternative algorithms for execution, at build time as well as at run time. Details of
such embodiments of the present invention will now be described with specific regard to
the figures.
FIGURE 2 illustrates a block flow diagram of the random algorithm selection system and method in accordance with a first embodiment of the present invention. In particular, this diagram shows one such random algorithm selection 100 during build time.

At build time, a set of algorithms A1, A2, ...AN; where N is an integer greater than 1, in the program, P, are chosen to be registered algorithms as seen by block 1. It should be understood that the registered algorithms together form the given implementation of the software program P. A registered algorithm is an algorithm with which the present invention seeks to achieve diversity. For each of the registered algorithms, there is a corresponding set of alternative algorithms as seen in block 2. Alternative algorithms can be pre-written or generated at build time.

Alternative algorithms can be generated through mix-and-match sub alternative algorithms. For example, if a registered algorithm A has a few sub registered algorithms, then the different combinations of the sub alternative algorithms make different alternative algorithms for A. This idea can be further generalized. If some instructions and/or groups of instructions of a registered algorithm can be replaced by functionally equivalent but different instructions and/or groups of instructions, then the alternative algorithms for this registered algorithm can be generated by each time picking a different alternative from all of the valid choices for these instructions and/or groups of instructions.

Alternative algorithms can also be generated through compiler-related technologies, such as pattern matching and pattern selection illustrated by way of FIGURE 3. It should of course be understood that compiler-related match and select operations as illustrated in FIGURE 3 are but one manner by which the set of alternative algorithms of block 2 in FIGURE 2 can be generated, and other such generation mechanisms are possible without straying from the intended scope of the present invention. In pattern matching and selection, operations may be replaced by equivalent, yet structurally different operations. For instance, for a mathematical operation in the algorithm, there can be a number of functionally equivalent but presentation-wise different operations. To generate alternative algorithms for a registered algorithm 20, a pattern set 23 and the registered algorithm 20 are inputted into the compiler 21, whereby the compiler's pattern matcher chooses a corresponding pattern for the current operation that is under evaluation. The pattern set 23 can also include multi-node operations ~ i.e., for a
group of operations there can be a functionally equivalent, but different, group of operations. The selection process covers the algorithm graph with a consistent set of patterns from the alternatives found in matching. Selection is a constraint-solving problem which maximizes performance while balancing the security and size constraints. Each run of the compiler on the registered algorithm will generate a different alternative algorithm 22a, 22b, 22c (i.e., alternative algorithms 1, 2, ...M; where M is an integer greater than 1 that may differ from N) because of the different selections of the operation patterns in each run.

With continued reference to FIGURE 2, the random selection engine 7 chooses at random (in block 3) an algorithm from each set of alternative algorithms (from block 2) for each registered algorithm (from block 1). After this step, the program (as shown in block 4) has each of its registered algorithms filled with a randomly selected alternative algorithm. This program from 4 is then compiled and linked (at block 5), and the result is the final executable (as at block 6). This process produces diverse instances of the executable which are functionally equivalent. In this particular illustration, it should be understood that the given executable is however statically diverse.

The random selection engine 7 has a pseudorandom number generator (PRNG) which acts in a known manner in order to generate randomness used in selecting alternative algorithms. The random selection engine 7 needs to gather entropy in order to seed the PRNG. It can use simple but less secure methods such as calling the "rand" function in a standard C library to gather entropy, or many use more involved and much securer methods such as requesting entropy from a trusted hardware random number generator and letting this piece of hardware send the entropy back via secure channels. Such details of the PRNG are well within the common knowledge in the programming art and are not further described herein.

An alternative build time embodiment of the present invention is shown by way of random selection engine 100a illustrated in FIGURE 2A. The process illustrated here is similar to the operation of the random selection engine 110 of FIGURE 2. Here however, the random selection engine 7 and the sets of alternative algorithms 2 combine with program P with branch obfuscation (as earlier discussed by way of example to Chow et al. or an similar technique) to produce the final executable code. This process not only
produces diverse instances of the executable with static diversity, but the executable is also produced to be capable of achieving dynamic diversity at run time.

More specifically, the process of **FIGURE 2A** compiles program P to an intermediate form at block 5. The intermediate form is compiled in an}1 known manner. For instance, it should be understood that in either **FIGURES 2** or 2A, compiling ma}2 be accomplished by known mechanisms such as, but not limited to, MSC (Microsoft C cross compilers), GCC (GNU Compiler Collection), or an}3 suitable compiler system without straying from the intended scope of the present invention. The intermediate form in block 5 is then subjected to a protection step. In particular, block 8 shows the step performing branch obfuscation, though other protection mechanisms are possible as mentioned above.

In block 9, there is performed final object code generation whereby the output executable code at block 10 contains an obfuscated form of program P along with the random selection engine and the sets of alternative algorithms.

It should be understood that **FIGURES 2** and 2A differ from one another in that **FIGURE 2** has a conventional compile-link process, whereas the process in **FIGURE 2A** is extended to provide a stage at which branch obfuscation can be performed on intermediate code. Specifically, the conventional compile-link process (at block 5) of **FIGURE 2** takes in only the program P, and thus the final executable there only represents P.

It should further be noted that branch protection as shown in **FIGURE 2A** can be applied to the output from the compile/link step, or it can be applied somewhere within the compile/link step itself. For example, if the compile/link step encompasses a transcoder (not shown), then branch protection can be applied during such transcoder stage, and then the protected code is outputted from the transcoder stage to a compile mechanism (e.g., GCC or MSC). However, while it is possible to perform branch obfuscation on the final executable machine code, it should be understood that this ma}4 be less advantageous due to difficulty because the symbolic intermediate information has been lost at that point.

With regard to **FIGURE 4**, random algorithm selection at run time in accordance with the present invention is illustrated by a high-level, simplified block flowchart. During the run time steps 300 as illustrated, when a registered algorithm needs to run at block 30, the random selection engine is invoked at block 31 to choose an algorithm from the
corresponding set of alternative algorithms. Once an alternative algorithm is chosen, a
dispatcher dispatched at block 32 the chosen alternative algorithm for the registered
algorithm which then runs at block 33. The dispatcher is a coding mechanism by which an
alternative algorithm is identified and thereby obtained.

With regard to FIGURE 5, there may be one dispatcher in the program 40 for each
registered algorithm (A1, A2, ... AN). Each dispatcher is able to dispatch any algorithm in
the set of alternative algorithms 40a, 40b, 40c that corresponds to the given registered
algorithm (A1, A2, ... AN). The dispatcher uses the random selection engine 41 which
itself is seeded by a source of entropy 42 as discussed previously, to randomly select from
among the alternative algorithm sets 40a, 40b, 40c. While the alternative algorithm sets
40a, 40b, 40c are shown arranged in sequential order in memory, it should be readily
apparent that such arrangement may to some extent compromise security due.
Accordingly, alternative algorithms for a registered algorithm do not need to be laid out
next to each other in sequential order in memory. Indeed, further alternative arrangements
are possible without straying from the intended scope of the present invention.

It may be preferred that alternative algorithms for the same registered algorithm
reside apart from each other in memory as illustrated by program 50 as shown in FIGURE
6. As well, random algorithm selection can occur on multiple levels. It is possible to have
registered algorithms contained within alternative algorithms. Therefore, there can be
dispatchers, such as illustrated in FIGURE 7, in alternative algorithms 60 that further
dispatch alternative algorithms in a lower level 61, and so forth 62. It is also possible to
implement just one dispatcher that handles all the dispatching jobs for all the registered
algorithms. The dispatcher mechanism can also be implemented in such a way that there is
one dispatcher for each level of registered algorithm in the hierarchy.

In terms of specific implementation, it should be readily apparent to one of
ordinary skill in the art that the dispatcher can be implemented in many ways without
straying from the intended scope of the present invention. For example, the dispatcher can
contain a table of function pointers (e.g., a call table) whereby each algorithm in the
corresponding set of alternative algorithms is a function. In such instance, there is a one-to-one correspondence relation between the table of function pointers and the set of
alternative algorithms. Once the random selection engine chooses an alternative algorithm, the dispatcher calls the corresponding function pointer.

As illustrated in FIGURE 8, the dispatcher may also be implemented in the instruction level. In this case, the dispatcher 70 contains a table 70a of starting addresses of each alternative algorithm (e.g., a jump table). Once the random selection engine chooses an alternative algorithm, the dispatcher branches to the starting address of the selected algorithm with a corresponding jump to the alternative algorithms 71. At the completion of each alternative algorithm, a branch instruction returns to the instruction immediately following the registered algorithm. Such approach does not incur performance penalty of creating activation records on the system stack as may occur by the function pointer approach. This approach of FIGURE 8 is similar to conditional branches, thus the dispatcher can be implemented using conditional branches. There may be one conditional branching instruction per alternative algorithm. The branch address is the starting address of the alternative algorithm. Here, the branch is taken only if the evaluation of the condition is true. Otherwise, the branch is not taken. The random selection engine chooses an alternative algorithm, and sets the condition based on the choice. The conditional branching instructions then evaluate the condition and decide whether or not to branch to the designated addresses.

In terms of strengthening security of software, the present invention will now be discussed by way of an example applying the inventive dynamic random algorithm selection to strengthen public-key cryptography software such as RSA (Rivest, Shamir and Adleman) private key exponentiation. In RSA private key exponentiation, Montgomery multiplication is used over and over again at least a few hundred times - i.e., the RSA private key exponentiation is implemented as a series of Montgomery multiplication executions. It is common practice that all of the Montgomery multiplications are implemented in the same manner. As such, it is not typically difficult for an attacker to identify RSA private key exponentiation from a transcript of an RSA execution. This is true even if the Montgomery multiplication is protected by code obfuscation because the transcript will display that a certain algorithm is used a massive amount of times and the only place such a pattern can occur is during RSA private key exponentiation. After this.
the attacker might be able to isolate each Montgomery multiplication in an effort to extract the RSA private key.

There are many algorithms that implement Montgomery multiplication. Examples include: Coarsely Integrated Operand Scanning (CIOS), Separate Operand Scanning (SOS), Finely Integrated Product Scanning (FIPS), and others. If dynamic random algorithm selection is employed in accordance with the present invention, then for each Montgomery multiplication, one of these alternative algorithms is selected at random during run time. As such, the RSA private key exponentiation will be a chain of different algorithms. Advantageously, this randomness makes less obvious from a transcript that a service (i.e., RSA private key exponentiation) is used over and over again.

It may further be desirable that different weights are assigned to different Montgomery multiplication implementations based on the execution environment. This would enable more efficient alternative algorithms to be chosen more frequently than others. Still further, the present invention can further diversify Montgomery multiplication in order to make RSA private key exponentiation less prone to pattern based attacks. For example, the majority of the arithmetic operations in Montgomery multiplication are additions. Here, the addition operation can be considered as an algorithm. It is readily understood that there are many alternative algorithms for addition. A particular example of such may be \(x + y\) is mathematically equivalent to an of the following:

\[
\begin{align*}
(1) \quad & (x \cdot y) + 2(x \& y) \\
(2) \quad & (x \mid y) + (x \& y) \\
(3) \quad & 2(x \mid y) - (x \cdot y)
\end{align*}
\]

In terms of the present invention, dynamic random algorithm selection may be applied to the addition operation so that each addition operation in the Montgomery multiplication is dynamically chosen from a set of alternative algorithms at run time. This further diversifies an alternative algorithm of Montgomery multiplication, such as CIOS, into many different instances.

The method steps of the invention may be embodied in sets of executable machine code stored in a variety of formats such as object code or source code. Such code has been described generically herein as algorithms, alternative algorithms, programming code, or a computer program for simplification. Clearly, the executable machine code may be
integrated with the code of other programs, implemented as subroutines, by external program calls or by other techniques as known in the art.

The embodiments of the invention may be executed by a computer processor or similar device programmed in the manner of method steps, or may be executed by an electronic system which is provided with means for executing these steps. Similarly, an electronic memory means such computer diskettes, CD-ROMs, Random Access Memory (RAM), Read Only Memory (ROM) or similar computer software storage media known in the art, may be programmed to execute such method steps. As well, electronic signals representing these method steps may also be transmitted via a communication network.

It would also be clear to one skilled in the art that this invention need not be limited to the existing scope of computers and computer systems. Credit, debit, bank, and smart cards could be encoded to apply the invention to their respective applications. An electronic commerce system in a manner of the invention could for example, be applied to parking meters, vending machines, telephones, inventors' control or rental cars and using magnetic strips or electronic circuits to store the software and passwords. Again, such implementations would be clear to one skilled in the art, and do not take away from the invention. The above-described embodiments of the present invention are intended to be examples only. It should be equally apparent that different software can benefit from strengthened security by way of the present invention. Moreover, alterations, modifications, and variations may be effected to the particular embodiments by those of skill in the art without departing from the scope of the invention, which is defined solely by the claims appended hereto.
What is claimed is:

1. A method of concealing computer software source code comprising:
   identifying at least one sequence of computational steps embodied in a computer software source code of a computer program;
   establishing a set of alternative implementations of said at least one sequence of computational steps;
   obtaining one alternative implementation randomly selected from said set of alternative implementations; and
   replacing said at least one sequence with said one alternative implementation to form an alternative embodiment of said computer software source code;
   wherein said computer software source code and said alternative embodiment of said computer software source code are diverse instances of functionally equivalent code.

2. The method as claimed in Claim 1, wherein said at least one sequence of computational steps includes a set of computer instructions.

3. The method as claimed in Claim 1, wherein said at least one sequence of computational steps includes a piece of high level software programming that carries out a task on a computing device

4. The method as claimed in Claim 1, wherein said at least one sequence of computational steps includes a piece of high level software programming that carries out a set of tasks on a computing device
5. The method as claimed in Claim 1, wherein each alternative implementation within one given said set of alternative implementations is functionally equivalent to a corresponding one of said at least one sequence of computational steps.

6. The method as claimed in Claim 1, wherein

   said identifying step includes more than one said sequence of computational steps,

   said establishing step includes multiple sets of alternative implementations, and

   each said set of alternative implementations are functionally equivalent to a corresponding one of said multiple sets of alternative implementations.

7. The method as claimed in Claim 6, wherein said obtaining step is accomplished by choosing from said multiple sets of alternative implementations uniformly at random and independently from one another.

8. The method as claimed in Claim 6, wherein said obtaining step is accomplished by choosing from said multiple sets of alternative implementations non-uniformly at random and independently from one another where a probability of an alternative implementation begin chosen is affected by activities external to said computer software source code.

9. The method as claimed in Claim 6, wherein said obtaining step is accomplished by choosing from said multiple sets of alternative implementations non-uniformly at random and independently from one another where a probability of an alternative implementation being chosen is affected by activities internal to said computer software source code.
10. The method as claimed in Claim 6, wherein said obtaining step is accomplished by choosing from said multiple sets of alternative implementations non-uniformly at random and independently from one another where a probability of any alternative implementation being chosen is affected by weightings assigned to each alternative implementation within each of said multiple sets of alternative implementations.

11. The method as claimed in Claim 6, wherein said obtaining step is accomplished by choosing from said multiple sets of alternative implementations non-uniformly at random where a probability of any alternative implementation being chosen is affected by weightings assigned to each alternative implementation within each of said multiple sets of alternative implementations, said weightings further providing dependency between different sets of said multiple set of alternative implementations.

12. The method as claimed in Claim 1, wherein said obtaining step is performed during a compilation and build cycle of said computer software source code.

13. The method as claimed in Claim 1, wherein said obtaining step is performed during an execution and run cycle of said computer software source code.

14. The method as claimed in Claim 1, wherein said obtaining step is performed during both a compilation and build cycle and an execution and run cycle of said computer software source code.
15. The method as claimed in Claim 1, wherein said computer program includes all said alternative implementations and said method further includes an anti-tampering step to protect an executable form of said alternative embodiment of said computer software code.

16. The method as claimed in Claim 1, wherein said anti-tampering step includes obfuscating branch instructions of said executable form of said alternative embodiment of said computer software code.

17. The method as claimed in Claim 1, wherein said establishing step includes generating said set of alternative algorithms by compiler match and select operations.

18. The method as claimed in Claim 17, wherein said compiler match and select operations include multi-node operations.

19. The method as claimed in Claim 1, wherein said establishing step includes generating said set of alternative algorithms by mix-and-match sub alternative operations.

20. The method as claimed in Claim 1, wherein said obtaining step operates in conjunction with a random selection engine to determine said one alternative implementation.

21. The method as claimed in Claim 20, wherein each of said at least one sequence of computational steps embodied in said computer software code of said computer program
includes a dispatcher, said dispatcher being responsive to said random selection engine so as to identify said one alternative implementation during said obtaining step.

22. The method as claimed in Claim 21, wherein said dispatcher is formed by way of a call table.

23. The method as claimed in Claim 21, wherein said dispatcher is formed by way of a jump table.

24. The method as claimed in Claim 21, wherein said dispatcher is formed by way of conditional branching.

25. The method as claimed in Claim 1, wherein said set of alternative implementations are stored in memory sequentially.

26. The method as claimed in Claim 1, wherein said set of alternative implementations are stored in memory randomly.

27. The method as claimed in Claim 1, wherein said method occurs in multiple levels.

28. The method as claimed in Claim 27, wherein said multiple levels form hierarchies at both a compilation and build cycle and an execution and run cycle of said computer software source code.
29. The method as claimed in Claim 21, wherein said method occurs in multiple levels, said multiple levels forming hierarchies at both a compilation and build cycle and an execution and run cycle of said computer software source code, and more than one said dispatcher is provided and arranged in multiple levels based upon said hierarchies.

30. A system for concealing computer software source code, said system comprising:

- a set of machine executable code segments operable to produce software code that randomizes usage of functionally equivalent sequences of computational steps contained in said computer software source code, said machine executable code executable to perform the steps of:
  - identifying at least one sequence of computational steps embodied in a computer software source code of a computer program;
  - establishing a set of alternative implementations of said at least one sequence of computational steps;
  - obtaining one alternative implementation randomly selected from said set of alternative implementations; and
  - replacing said at least one sequence with said one alternative implementation to form an alternative embodiment of said computer software source code;

wherein said computer software source code and said alternative embodiment of said computer software source code are diverse instances of functionally equivalent code.

31. The system as claimed in Claim 30, wherein said at least one sequence of computational steps includes a set of computer instructions.
32. The system as claimed in Claim 30, wherein said at least one sequence of computational steps includes a piece of high level software programming that carries out a task on a computing device

5

33. The system as claimed in Claim 30, wherein said at least one sequence of computational steps includes a piece of high level software programming that carries out said set of tasks on a computing device

10 34. The system as claimed in Claim 30, wherein each alternative implementation within one given said set of alternative implementations is functionally equivalent to a corresponding one of said at least one sequence of computational steps.

15 35. The system as claimed in Claim 30, wherein

said identifying step includes more than one said sequence of computational steps,

said establishing step includes multiple sets of alternative implementations, and

each said set of alternative implementations are functionally equivalent to a corresponding one of said multiple sets of alternative implementations.

20 36. The system as claimed in Claim 35, wherein said obtaining step is accomplished by choosing from said multiple sets of alternative implementations uniformly at random and independent!} from one another.
37. The system as claimed in Claim 35, wherein said obtaining step is accomplished by choosing from said multiple sets of alternative implementations non-uniformly at random and independently from one another where a probability of any alternative implementation being chosen is affected by activities external to said computer software source code.

38. The system as claimed in Claim 35, wherein said obtaining step is accomplished by choosing from said multiple sets of alternative implementations non-uniformly at random and independently from one another where a probability of any alternative implementation being chosen is affected by activities internal to said computer software source code.

39. The system as claimed in Claim 35, wherein said obtaining step is accomplished by choosing from said multiple sets of alternative implementations non-uniformly at random and independently from one another where a probability of any alternative implementation being chosen is affected by weightings assigned to each alternative implementation within each of said multiple sets of alternative implementations, said weightings further providing dependency between different sets of said multiple set of alternative implementations.

40. The system as claimed in Claim 35, wherein said obtaining step is accomplished by choosing from said multiple sets of alternative implementations non-uniformly at random where a probability of any alternative implementation being chosen is affected by weightings assigned to each alternative implementation within each of said multiple sets of alternative implementations, said weightings further providing dependency between different sets of said multiple set of alternative implementations.
41. The system as claimed in Claim 30, wherein said obtaining step is performed during a compilation and build cycle of said computer software source code.

42. The system as claimed in Claim 30, wherein said obtaining step is performed during an execution and run cycle of said computer software source code.

43. The system as claimed in Claim 30, wherein said obtaining step is performed during both a compilation and build cycle and an execution and run cycle of said computer software source code.

44. The system as claimed in Claim 30, wherein said computer program includes all said alternative implementations and said method further includes an anti-tampering step to protect an executable form of said alternative embodiment of said computer software code.

45. The system as claimed in Claim 30, wherein said anti-tampering step includes obfuscating branch instructions of said executable form of said alternative embodiment of said computer software code.

46. The system as claimed in Claim 30, wherein said establishing step includes generating said set of alternative algorithms by compiler match and select operations.

47. The system as claimed in Claim 46, wherein said compiler match and select operations include multi-node operations.
48. The system as claimed in Claim 30, wherein said establishing step includes generating said set of alternative algorithms by mix-and-match sub alternative operations.

49. The system as claimed in Claim 30, wherein said obtaining step operates in conjunction with a random selection engine to determine said one alternative implementation.

50. The system as claimed in Claim 49, wherein each of said at least one sequence of computational steps embodied in said computer software code of said computer program includes a dispatcher, said dispatcher being responsive to said random selection engine so as to identify said one alternative implementation during said obtaining step.

51. The system as claimed in Claim 50, wherein said dispatcher is formed by way of a call table.

52. The system as claimed in Claim 50, wherein said dispatcher is formed by way of a jump table.

53. The system as claimed in Claim 50, wherein said dispatcher is formed by way of conditional branching.
54. The system as claimed in Claim 30, wherein said set of alternative implementations are stored in memory sequentially.

55. The system as claimed in Claim 30, wherein said set of alternative implementations are stored in memory randomly.

56. The system as claimed in Claim 30, wherein said method occurs in multiple levels.

57. The system as claimed in Claim 56, wherein said multiple levels form hierarchies at both a compilation and build cycle and an execution and run cycle of said computer software source code.

58. The system as claimed in Claim 50, wherein said method occurs in multiple levels, said multiple levels forming hierarchies at both a compilation and build cycle and an execution and run cycle of said computer software source code, and more than one said dispatcher is provided and arranged in multiple levels based upon said hierarchies.

59. An apparatus for concealing computer software source code comprising:

   means for identifying at least one sequence of computational steps embodied in a computer software source code of a computer program;

   means for establishing a set of alternative implementations of said at least one sequence of computational steps;
means for obtaining one alternative implementation randomly selected from said set of alternative implementations; and

means for replacing said at least one sequence with said one alternative implementation to form an alternative embodiment of said computer software source code;

wherein said computer software source code and said alternative embodiment of said computer software source code are diverse instances of functionally equivalent code.

60. A computer readable memory medium storing computer software code for concealing computer software source code, said computer software code executable to perform the steps of:

identifying at least one sequence of computational steps embodied in a computer software source code of a computer program;

establishing a set of alternative implementations of said at least one sequence of computational steps;

obtaining one alternative implementation randomly selected from said set of alternative implementations; and

replacing said at least one sequence with said one alternative implementation to form an alternative embodiment of said computer software source code;

wherein said computer software source code and said alternative embodiment of said computer software source code are diverse instances of functionally equivalent code.
FIGURE 1
(Prior Art)
Random Selection Engine

1. Program P with a list of registered algorithms:
   - A1
   - A2
   - ...
   - AN

2. Set of alternative algorithms for A1
   Set of alternative algorithms for A2
   ... Set of alternative algorithms for AN

3. Select randomly an algorithm from each set of alternative algorithms for each registered algorithm in P

4. P with each registered algorithm randomly selected

5. Compile and link

6. Statically Diverse Executable

FIGURE 2
Random Selection Engine

Program \( P \) with a list of registered algorithms:
- A 1
- A 2
- ...
- A \( N \)

Select randomly an algorithm from each set of alternative algorithms for each registered algorithm in \( P \)

\( P \) with each registered algorithm randomly selected

Compile to intermediate form

Apply branch obfuscation

Generate and link final object code

Statically Diverse Executable Capable of Dynamic Diversity

FIGURE 2A
300

Registered algorithm starts to run

30

Invoke random selection engine to randomly select an alternative algorithm for the registered algorithm

31

The dispatcher for the algorithm dispatches the selected alternative algorithm

32

The registered algorithm ends

33

FIGURE 4
Program

- A 1 Dispatcher
- A 2 Dispatcher
- A N Dispatcher
- Alternative algorithm 1 for A 1
- Alternative algorithm 3 for A 2
- Alternative algorithm M for A 7
- Alternative algorithm 2 for A 1
- Alternative algorithm 1 for A 2

FIGURE 6
# INTERNATIONAL SEARCH REPORT

**International application No.**

PCT/CA2010/000393

**A. CLASSIFICATION OF SUBJECT MATTER**

**IPC:** G06F 21/22 (2006.01)

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** (2006.01)

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)

**Databases:** EPOQUE (EPDOC), Canadian Patents Database, Google Patents, IEEE XPLORER

**Keywords:** software, code, diversity, diversification, conceal*, random*, alternate*, implementation, obfuscate*, secure*

**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>* Abstract; paragraphs [0009], [0010], [0022] - [0029]; Figure 1 *</td>
<td>13 - 15, 28 - 29, 42 - 44, 57 - 58</td>
</tr>
<tr>
<td>Y</td>
<td>* Abstract; Figures 2 and 6; col 4, line 54 - col 9, line 24 *</td>
<td>13 - 14, 28 - 29, 42 - 43, 57 - 58</td>
</tr>
</tbody>
</table>

[X] See patent family annex.

[X] Further documents are listed in the continuation of Box C.

| * | Special categories of cited documents: |
| "A" | document defining the general state of the art which is not considered to be of particular relevance |
| "E" | earlier application or patent but published on or after the international filing date |
| "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) |
| "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 |

[X] 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

[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

[X] 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

[X] document member of the same patent family

**Date of the actual completion of the international search**

26 November 2010 (26-11-2010)

**Date of mailing of the international search report**

10 December 2010 (10-12-2010)

**Name and mailing address of the ISA/CA**

Canadian Intellectual Property Office

Place du Portage I, C14 - 1st Floor, Box PCT

50 Victoria Street

Gatineau, Quebec K1A 0C9

Facsimile No.: 001-819-953-2476

**Authorized officer**

Raghid Shreih (819) 994-2694

Form PCT/ISA/210 (second sheet) (July 2009)
<table>
<thead>
<tr>
<th>Patent Document Cited in Search Report</th>
<th>Publication Date</th>
<th>Patent Family Member(s)</th>
<th>Publication Date</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>US7512936B2</td>
<td>31 March 2009 (31-03-2009)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CA2662558A1</td>
<td>25 September 2008 (25-09-2008)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CN101529436A</td>
<td>09 September 2009 (09-09-2009)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EP2076863A2</td>
<td>08 July 2009 (08-07-2009)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IL197361 D0</td>
<td>24 December 2009 (24-12-2009)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JP2010508583T</td>
<td>18 March 2010 (18-03-2010)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MX2009003342A</td>
<td>12 May 2009 (12-05-2009)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WO20081 15279A2</td>
<td>25 September 2008 (25-09-2008)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WO20081 15279A3</td>
<td>19 February 2009 (19-02-2009)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CA2293650A1</td>
<td>14 January 1999 (14-01-1999)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CN1260055A</td>
<td>12 July 2000 (12-07-2000)</td>
</tr>
</tbody>
</table>