DEVICE AND METHOD FOR DYNAMICALLY MODIFYING INTERMEDIATE REPRESENTATION FILES DURING JUST-IN-TIME COMPILATIONS

Field of the Invention

This invention relates to a device and method for a device to dynamically modify an intermediate representation file during a just-in-time (JIT) compilation of the intermediate representation file.

Summary of the Prior Art

A common challenge faced by software developers is the protection of sensitive portions of programming code in software applications from being copied, modified, disabled, or reverse engineered. As an example, it is in the software developer's interest to ensure that portions of code in the software application that relate to implementation of digital rights management or electronic licensing may not be copied, modified or disabled. If these portions of code were to be modified or disabled, this would result in the unauthorized distribution or use of the software application by unlicensed users.

A software developer may introduce protective measures directly to the software application as the application is being coded however, such an approach is quite time consuming and arduous for the developer hence, such a method is not widely adopted.

Another approach that has been proposed by those skilled in the art involves the obfuscation of compiled binary files after the files have been generated by a compiler. However, such an approach is undesirable as the obfuscation of compiled binaries results in the unstable execution of programs when these obfuscated binaries are executed. Hence, this approach limits the types of obfuscation techniques that may be safely applied to the compiled binaries. As only limited types of obfuscation techniques can be applied, this makes it easier for hackers to reverse engineer the obfuscated code using either static code analysis or dynamic code analysis techniques.

When static code analysis is applied to a software application, the software's code will be examined offline without executing the software application. This process of understanding the code structure and the functions of the code provides a reader insight into the inner workings of the software application. As for dynamic code analysis, this analysis method is applied to the software application when the application is executed. As the application is

running, the application will then be evaluated and analysed in real time to determine the flow structure of the functions contained within the application.

Another approach that has been proposed by those skilled in the art involves the modification of code within a compiler to produce one or more modifications that obfuscate the code as part of the compilation process. As a result of this static obfuscation process, the final compiled version of the code would have incorporated the modifications that were produced within the compiler.

A downside of this approach is that as only static modifications are applied to the original code, the obfuscated code is still vulnerable to dynamic analysis techniques. In particular, this means that if a hacker were to repeatedly execute the obfuscated code, the hacker would eventually be able to reverse engineer the code.

Further, the fixed nature of the obfuscation method means that the attacker may compare different instances of the obfuscated code to guess and hypothesize details about the obfuscation techniques applied. For example, the hacker may modify the code slightly and then monitor the obfuscated output to obtain a hypothesis of the obfuscation code applied. Further, such an approach is not portable as the obfuscation techniques applied in this approach are programming language specific.

For the above reasons, those skilled in the art are constantly striving to come up with a device and method to protect sensitive portions of code in a software application from unauthorized users.

Summary of the Invention

Devices and methods to improve the protection of sensitive portions of code in a software application from unauthorized users are set out below in accordance with embodiments of the invention.

A first improvement proposed by embodiments of devices and methods in accordance with the invention is that binary code of interest is continuously dynamically modified in a random manner during runtime of the binary code. As the binary code is dynamically modified during runtime, this means that the control flow of the program will be dynamically altered as well but in such a manner that it does not compromise the performance of the program.

A second improvement proposed by embodiments of devices and methods in accordance with the invention is that traditional source-to-source obfuscation techniques are now achievable at the binary level.

A third improvement proposed by embodiments of devices and methods in accordance with the invention is that the invention does not make changes to the code during a compiling process, i.e. in the compiler environment, instead, the random and dynamic transformations are applied to blocks of binary code prior to the compilation of the binary code.

A fourth improvement proposed by embodiments of devices and methods in accordance with the invention is that the dynamic and random transformations applied to the binary code takes place on a thread that is separate from the program's main thread thereby ensuring that the dynamic versioning of the binary code does not become overly burdensome for the execution of the program's main thread.

According to a first aspect of the invention, a device for dynamically modifying a just-in-time (JIT) compiled intermediate representation file during an execution of the JIT compiled intermediate representation file is provided, the device comprising: a processor; and a non-transitory media readable by the processor, the non-transitory media storing instructions that when executed by the processor, cause the processor to: receive the JIT compiled intermediate representation file, wherein the JIT compiled intermediate representation file includes a main thread and is associated with a global mapping table having entries that each correspond to a function in the main thread; spawn a child thread from the main thread, and direct the child thread to: randomly retrieve a first function from the main thread and apply a dynamic versioning algorithm to the first function to generate a unique version of the first function such that the unique version produces an outcome that is the same as the first function, compile the unique version of the first function, and replace a first entry associated with the first function in the global mapping table with an address of the compiled unique version of the first function such that when the main thread calls the first function, the main thread will execute the compiled unique first function, wherein the main thread and the child thread are run in parallel.

With reference to the first aspect, in accordance with embodiments of the invention, the dynamic versioning algorithm comprises: instructions for directing the processor to: generate a version of the first function; change a control flow of code execution in the version of the first function; and obfuscate the code in the version of the first function.

With reference to the first aspect, in accordance with embodiments of the invention, the changing the control flow of code execution in the version of the first function comprises at least one of: inserting benign code, adding of jump instructions, modifying entry points into the code, call redirection, rearranging of code blocks or modules, creating a replica function, applying obfuscation algorithms, changing symbol attributes, segmenting blocks of code, replacing calls or encrypting code portions.

With reference to the first aspect, in accordance with embodiments of the invention, the obfuscating the code in the version of the first function comprises at least one of: inserting benign code, adding of jump instructions, modifying entry points into the code, call redirection, rearranging of code blocks or modules, creating a replica function, applying obfuscation algorithms, changing symbol attributes, segmenting blocks of code, replacing calls or encrypting code portions.

With reference to the first aspect, in accordance with embodiments of the invention, the device according to claim 1 further comprises: instructions for directing the processor to direct the child thread to: randomly retrieve a second function from the main thread and apply the dynamic versioning algorithm to the second function to generate a unique version of the second function such that the unique version of the second function produces an outcome that is the same as the second function; compile the unique version of the second function; replace a second entry associated with the second function in the global mapping table with an address of the compiled unique version of the second function such that when the main thread calls the second function, the main thread will execute the compiled unique second function.

With reference to the first aspect, in accordance with embodiments of the invention, the second function comprises the first function.

With reference to the first aspect, in accordance with embodiments of the invention, the dynamic versioning algorithm comprises: instructions for directing the processor to: generate a version of the second function; change a control flow execution of code in the version of the second function; and obfuscate the code in the version of the second function.

With reference to the first aspect, in accordance with embodiments of the invention, the intermediate representation file comprises a bitcode file or a bytecode file.

With reference to the first aspect, in accordance with embodiments of the invention, the device further comprises: instructions for directing the processor to: encode machine codes associated with compiled versions executed by the main thread using a secret key.

With reference to the first aspect, in accordance with embodiments of the invention, the first function comprises a function that has not been executed by the main thread.

According to a second aspect of the invention, a method for using a computer system to dynamically modify a just-in-time (JIT) compiled intermediate representation file during an execution of the JIT compiled intermediate representation file is provided, the method comprises: receiving the JIT compiled intermediate representation file, wherein the JIT compiled intermediate representation file includes a main thread and is associated with a global mapping table having entries that each correspond to a function in the main thread; spawning a child thread from the main thread, and directing the child thread to: randomly retrieving a first function from the main thread and applying a dynamic versioning algorithm to the first function to generate a unique version of the first function such that the unique version produces an outcome that is the same as the first function, compiling the unique version of the first function, and replacing a first entry associated with the first function in the global mapping table with an address of the compiled unique version of the first function such that when the main thread calls the first function, the main thread will execute the compiled unique first function, wherein the main thread and the child thread are run in parallel.

With reference to the second aspect, in accordance with embodiments of the invention, the dynamic versioning algorithm comprises the steps of: generating a version of the first function; changing a control flow of code execution in the version of the first function; and obfuscating the code in the version of the first function.

With reference to the second aspect, in accordance with embodiments of the invention, the changing the control flow of code execution in the version of the first function comprises at least one of: inserting benign code, adding of jump instructions, modifying entry points into the code, call redirection, rearranging of code blocks or modules, creating a replica function, applying obfuscation algorithms, changing symbol attributes, segmenting blocks of code, replacing calls or encrypting code portions.

With reference to the second aspect, in accordance with embodiments of the invention, the obfuscating the code in the version of the first function comprises at least one of: inserting

benign code, adding of jump instructions, modifying entry points into the code, call redirection, rearranging of code blocks or modules, creating a replica function, applying obfuscation algorithms, changing symbol attributes, segmenting blocks of code, replacing calls or encrypting code portions.

With reference to the second aspect, in accordance with embodiments of the invention, the method further comprises: randomly retrieving a second function from the main thread and applying the dynamic versioning algorithm to the second function to generate a unique version of the second function such that the unique version of the second function produces an outcome that is the same as the second function; compiling the unique version of the second function; and replacing a second entry associated with the second function in the global mapping table with an address of the compiled unique version of the second function such that when the main thread calls the second function, the main thread will execute the compiled unique second function.

With reference to the second aspect, in accordance with embodiments of the invention, the second function comprises the first function.

With reference to the second aspect, in accordance with embodiments of the invention, the dynamic versioning algorithm comprises the step of: generating a version of the second function; changing a control flow execution of code in the version of the second function; and obfuscating the code in the version of the second function.

With reference to the second aspect, in accordance with embodiments of the invention, the intermediate representation file comprises a bitcode file or a bytecode file.

With reference to the second aspect, in accordance with embodiments of the invention, the method further comprises: encoding machine codes associated with compiled versions executed by the main thread using a secret key.

With reference to the second aspect, in accordance with embodiments of the invention, the first function comprises a function that has not been executed by the main thread.

Brief Description of the Drawings

The above advantages and features in accordance with this invention are described in the following detailed description and are shown in the following drawings:

Figure 1 illustrating an exemplary block diagram of a network to support dynamic modification of intermediate representation files during just-in-time (JIT) compilations in accordance with embodiments of the invention;

Figure 2 illustrating a block diagram representative of components in an electronic device for implementing embodiments in accordance with embodiments of the invention;

Figure 3 illustrating a block diagram of a virtual machine in accordance with embodiments of the invention together with a flow diagram showing the interaction of external modules with internal modules contained within the virtual machine;

Figure 4 illustrating a flow diagram showing the application of the dynamic versioning process to a function having two iterations in accordance with embodiments of the invention; and

Figure 5 illustrating a flow diagram showing the process flow of dynamically modifying an intermediate representation file during a just-in-time compilation in accordance with embodiments of the invention.

Detailed Description

This invention relates to a system and method for a device to dynamically modify an intermediate representation file during a just-in-time (JIT) compilation of the intermediate representation file.

In particular, when the main thread is initiated and begins the execution of the intermediate representation file, the invention will spawn a child thread which is configured to randomly generate one or more unique versions of each function contained within the main thread of the intermediate representation file. It should be noted that each new unique version of a function will produce a same output as that produced by the original function. As each new unique version is generated, the new unique version is JIT compiled. The address of the JIT compiled unique function will then subsequently be used to update a corresponding entry in a global mapping table such that when the original function is called within the main thread, instead of JIT compiling and executing the original version, the compiled version referenced in the global mapping table will instead be executed by the main thread. It should be noted that the term program and software application may be used interchangeably throughout the description without departing from the invention.

One skilled in the art will recognize that many functional units in this description have been labelled as modules throughout the specification. The person skilled in the art will also recognize that a module may be implemented as electronic circuits, logic chips or any combination of electrical and/or electronic discrete components within a device. Further, one skilled in the art will also recognize that a module may be implemented in software which may then be executed by a variety of processors. In embodiments of the invention, a module may also comprise computer instructions or executable code that may instruct a computer processor to carry out a sequence of events based on instructions received. The choice of the implementation of the modules is left as a design choice to a person skilled in the art and does not limit the scope of this invention in any way.

As understood by those skilled in the art, just-in-time (JIT) compilation refers to a compilation process that is carried out during an execution of a software application. This means that the code of the software application is only compiled at runtime rather than prior to the execution of the software application. When the code is compiled, this causes the code to be translated to machine code which is then executed directly by a processor or a virtual machine.

For example, in a commonly known bytecode-compiled system, the system's programming code will first be converted to an intermediate representation file format known as bytecode or bit-code. These formats are widely adopted because they are not specific to any particular type of processor or device and as such, and are portable among various computer architectures. The bytecode may then be interpreted by a device's processor, or run on a virtual machine. The virtual machine comprises an emulation of a computer system and provides the functionality of a physical computer. The detailed functioning and architecture of the virtual machine are omitted for brevity as such details are known to those skilled in the art.

The JIT compiler will then read the bytecodes and compile the read bytecodes dynamically and sequentially into machine language. These processes may be done on a per-file basis, a per-function basis or even on any arbitrary code fragment. In short, the read bytecode will be compiled just before it is about to be executed (hence the name "just-in-time"), and then cached and executed later without needing to be recompiled. Hence, bytecodes that have been compiled need not be recompiled again as its associated machine code would have been stored at an assigned address in the device's memory. Further, a lookup table would also

be updated with the assigned addresses of the associated machine codes as the bytecodes are compiled.

Figure 1 illustrates an exemplary block diagram of a network to support dynamic modification of intermediate representation files during just-in-time (JIT) compilations in accordance with embodiments of the invention. Network 100 includes a device 105 and a resource server 110. Device 105 may comprise, but is not limited to, any device that is able to carry out computing functions either wirelessly or through wired connections such as servers, computers, tablet computers, mobile computers, or netbooks or any type of computing devices that may be configured to receive and execute instructions. These devices are also provided with a variety of modules for carrying out the various computing and processing functions.

For example, device 105 may be provided with a development module that is operable to provide an integrated development environment (IDE) for development of software applications. The development module may be configured to support the design, development, testing, and/or deployment of software applications within device 105 or server 110. The development module is also configured to support different programming languages and to integrate different modules to simplify the development of software applications. Further, the development module is also configured such that a developer may create and edit source code for a project and JIT compile the source code to execute a software application.

Device 105 and resource server 110 may both be configured to wirelessly communicate with each other through the Internet or wireless networks 120 such as, but are not limited to, cellular networks, satellite networks, telecommunication networks, or Wide Area Networks (WAN). Communications between these two devices may also take place via wired means. Through this wireless communication, users of resource server 110 may remotely access modules contained within device 105 to carry out development of software applications or to securely run software applications contained within device 105.

Figure 2 illustrates a block diagram representative of components of module 200 that may be provided within device 105 and server 110 for implementing embodiments in accordance with embodiments of the invention. One skilled in the art will recognize that the exact configuration of each wireless device provided within the devices or the access point may be different and the exact configuration of module 200 may vary and Figure 2 is provided by way of example only.

In embodiments of the invention, module 200 comprises controller 201 and user interface 202. User interface 202 is arranged to enable manual interactions between a user and module 200 and for this purpose includes the input/output components required for the user to enter instructions to control module 200. A person skilled in the art will recognize that components of user interface 202 may vary from embodiment to embodiment but will typically include one or more of display 240, keyboard 235 and track-pad 236.

Controller 201 is in data communication with user interface 202 via bus 215 and includes memory 220, processor 205 mounted on a circuit board that processes instructions and data for performing the method of this embodiment, an operating system 206, an input/output (I/O) interface 230 for communicating with user interface 202 and a communications interface, in this embodiment in the form of a network card 250. Network card 250 may, for example, be utilized to send data from electronic device 200 via a wired or wireless network to other processing devices or to receive data via the wired or wireless network. Wireless networks that may be utilized by network card 250 include, but are not limited to, Wireless-Fidelity (Wi-Fi), Bluetooth, Near Field Communication (NFC), cellular networks, satellite networks, telecommunication networks, Wide Area Networks (WAN), etc.

Memory 220 and operating system 206 are in data communication with CPU 205 via bus 210. The memory components include both volatile and non-volatile memory and more than one of each type of memory, including Random Access Memory (RAM) 220, Read Only Memory (ROM) 225 and a mass storage device 245, the last comprising one or more solid-state drives (SSDs) or any other types of drives. Memory 220 also includes secure storage 246 for securely storing secret keys, or private keys. It should be noted that the contents within secure storage 246 might only be accessible by a super-user or administrator of module 200 and might not be accessed by any user of module 200. One skilled in the art will recognize that the memory components described above comprise non-transitory computer-readable media and shall be taken to comprise all computer-readable media except for a transitory, propagating signal. Typically, the instructions are stored as program code in the memory components but can also be hardwired. Memory 220 may include a kernel and/or programming modules such as a software application that may be stored in either volatile or non-volatile memory.

Herein the term “processor” is used to refer generically to any device or component that can process such instructions and may include: a microprocessor, microcontroller, programmable logic device or other computational device. That is, processor 205 may be

provided by any suitable logic circuitry for receiving inputs, processing them in accordance with instructions stored in memory and generating outputs (for example to the memory components or on display 240). In this embodiment, processor 205 may be a single core or multi-core processor with memory addressable space. In one example, processor 205 may be multi-core, comprising—for example—an 8 core CPU.

Figure 3 illustrates a block diagram of various modules contained within virtual machine 301 in accordance with embodiments of the invention. In addition to the above, Figure 3 also illustrates a flow diagram showing the interaction of source code 305 with translator module 310 before a resulting intermediate representation file 312 is provided to virtual machine 301. In embodiments of the invention, virtual machine 301 may be provided within a memory of device 105 or virtual machine 301 may be configured as module within device 105's memory.

In operation, a source code that is to be JIT compiled and executed within a virtual machine will first be provided to a translator to translate the source code to an intermediate representation file format before the intermediate representation file is provided to the virtual machine. A sample of source code 305 is illustrated below. One skilled in the art will recognize that this source code is shown as an example only and does not limit the invention in any way.

```
void function A();
void function B();

int main( )
        {
        int i;


        A();
        B();
        }
```

Source code 305 is then provided to translator 310. Translator 310 may comprise any software or computing module, such as a Low Level Virtual Machine (LLVM) compiler or a Clang compiler that is configured to translate or convert source code 305 into an intermediate representation file format.

In the example above, source code 305 comprises code in the “C Programming” language and this code will be converted by translator module 310 into an intermediate representation file 312 comprising a bytecode or bit-code format.

When bit-code 312 is loaded for execution by virtual machine 301, bit-code 312 will be JIT compiled by JIT compiler 335 and as a result, an associated global mapping table 315 will be generated. As each function contained within intermediate representation file 312 is JIT-compiled, an address associated with each JIT-compiled function will then be added to global mapping table 315 such that when a processor or program intends to execute a function contained within intermediate representation file 312, the program will just have to look up the compiled function’s address in global mapping table 315 to execute the function. In other words, each JIT-compiled function within intermediate representation file 312 will have its own associated address in global mapping table 315.

In the example above, this means that once it has been JIT-compiled, “*function A( )*” in bytecode may be assigned address “0x7f00b9c” in global mapping table 315 while “*function B( )*” in bytecode may be assigned address “0x7f11b9d” in global mapping table 315. Hence, when a virtual machine wishes to call the bytecode of “*function B( )*”, the virtual machine will refer to the address assigned to “*function B( )*”, i.e. “0x7f11b9d”, to call “*function B( )*’s” bytecode.

During its execution, if the content pointed to by an entry in global mapping table 315 comprises a bit-code or a byte-code, a JIT compiler 335 in virtual machine 301 will be invoked to generate a corresponding assembly code for this content. However, if the entry in the global mapping table 315 points to a compiled bit-code or byte-code, i.e. the corresponding assembly code, the corresponding assembly code will be directly executed by main thread module 320.

In another embodiment of the invention, translator module 310 may be provided within virtual machine 301. This means that the process of converting source code 305 into intermediate representation file 312 may take place within a translator module that is integrated within virtual machine 301. In this embodiment, once intermediate representation file 312 has been created, virtual machine 301 will then load file 312 into module 320.

Once main thread module 320 has been loaded with file 312, and once file 312 has been JIT-compiled and corresponding entries have been made to global mapping table 315, main thread module 320 will begin to execute file 312. As soon as main thread module 320 has initiated the execution of file 312, virtual machine 301 will then cause child thread module 325 to

be spawned from main thread module 320. Once spawned, child thread module 325 will randomly obtain bit-code or byte-code functions from file 312. These obtained bit-code or byte-code functions are then provided to dynamic versioning module 330. Dynamic versioning module 330 will then generate a unique version of each provided function whereby the outcome produced by each unique function is the same as that produced by the original function. As each unique version of each function is generated, the function will be JIT-compiled by compiler 335 and dynamic version module 330 will use the addresses of these unique compiled versions to replace its corresponding version in global mapping table 315. Once this is done, this means that the address in global mapping table 315 will point to the unique version generated by dynamic version module 330 instead.

When the main thread module 320 attempts to call the original function contained in file 312 by calling the address provided in global mapping table 315, module 320 will instead be redirected to call the address of the unique version thereby invoking the new version instead. To summarize, this means that during the updating of the address of a compiled version into global mapping table 315, the address of the compiled version will be used to replace the original address associated with the original function in table 315. Further, as the new version produces an identical output as that of the original function with a given input, the outcome of file 312 remains unchanged.

In the example described above, once child thread module 325 has spawned from main thread module 320, child thread module 325 will begin by randomly selecting a function from file 312. In this example, it is assumed that child thread module 325 will randomly select and extract function A( ) from file 312. Function A( ) will then be provided to dynamic versioning module 330 so that module 330 may generate unique versions of function A( ) by changing the control flow of code execution of function A( ) in each unique version. This means that each unique version of function A( ) will have its own unique flow of code execution. Further, in embodiments of the invention, the resulting control flow of the code in each unique version may then be further obfuscated using existing known obfuscation techniques that are compatible with compiler 335.

In embodiments of the invention, the changing of the control flow or the obfuscation techniques that may be utilized by dynamic versioning module 330 includes, but is not limited to, at least one of: addition of extra predicates, inserting benign code, adding of jump instructions, modifying entry points into the code, call redirection, rearranging of code blocks or modules,

creating replica functions, applying obfuscation algorithms, changing symbol attributes, segmenting blocks of code, replacing calls and/or encrypting code portions.

When function A( ) is provided to dynamic versioning module 330, module 330 will first generate a new version of function A( ) which is function A_1( ). Function A_1( ) will produce a outcome that is the same as that of function A( ) but function A_1( ) will have a different code execution flow as compared to the code execution flow of function A. The code execution flow in function A_1( ) may be differentiated from the code execution flow in function A( ) by applying source to source obfuscation methods which change the control flow of the execution of code in function A_1( ).

Dynamic versioning module 330 will then cause compiler 335 to JIT compile function A_1() and then update global mapping table 315 by replacing the address of function A( ) in table 315 with the address of compiled function A_1( ). This means that when function A( ) is eventually called by main thread module 320, the address in global mapping table 315 will point to function A_1( ) instead as the address of function A( ) has been updated and replaced with the address of function A_1( ) during the updating step. As a result, function A_1( ) will be executed.

After dynamic versioning module 330 has generated and updated the address of function A_1( ) into global mapping table 315, (under the assumption that module 330 has been randomly provided with function A( ) again) module 330 will then proceed to generate another new version of function A( ) which is function A_2( ). Function A_2 ( ) will produce an outcome that is the same as that of functions A ( ) and A_1( ) but function A_2( ) will have a different code execution flow as compared to the code execution flow of functions A( ) and A_1( ). Function A_2 ( ) is JIT compiled by compiler 335 and then its address is provided to global mapping table 315. Global mapping table 315 will then update the corresponding address by replacing the address of function A_1( ) in table 315 with the address of compiled function A_2( ). This means that when the original function A( ) is again eventually called by main thread module 320, the compiled function A_2( ) will instead be executed as the address of function A( ) which has been updated with the address of function A_1( ) has been further replaced with the address of function A_2( ) during the most recent updating step.

In embodiments of the invention, for the sake of performance, dynamic versioning module 330 may be configured to generate a fixed or random number of versions for each function. In the embodiment where a fixed number of versions for a function is generated, this

means that the generation of the multiple new functions of function A( ) and the updating of the global mapping table 315 with the address of the latest version of function A( ) continuously repeats until n-number of functions A( ) have been generated from dynamic versioning module 330 where n may comprise a number determined by a person skilled in the art, i.e. function $A_{1...N}$. It should be noted that this process does not necessarily occur in a back to back manner as child thread 325 selects functions to obfuscate in a random manner. So there is the possibility that child thread 325 may return to generate another version of function A( ) only after it has completed the processing of other functions in main thread 320.

In yet another embodiment of the invention, there is the possibility that original function A( ) may be called by main thread 320 before a new version of function A( ) is generated, JIT compiled and its address updated into global mapping table 315. In such a situation, main thread 320 will use the original address of function A( ) in table 315 to execute function A( ).

Regardless of the embodiment adopted, once main thread 320 calls function A( ) in file 312, main thread module 320 will check global mapping table 315 to obtain the latest address associated with function A( ). In this example, assuming that the address of function A( ) in table 315 has been replaced with the address of the most recently generated version of function A( ) which is function A_2( ), main thread 320 will retrieve the address associated with compiled function A_2( ) and main thread module 320 will execute compiled function A_2( ) instead of function A( ).

In embodiments of the invention, while main thread 320 is executing function A_2( ), child thread 325 may have randomly selected and extracted another function, such as function B( ) from file 312 (as previously loaded in main thread 320). Function B( ) will then be provided to dynamic versioning module 330 so that module 330 may generate unique versions of function B( ) by changing the control flow of code execution of function B( ) in each unique version using the techniques described above. In other embodiments of the invention, the child thread may be configured to obfuscate the control flow of only selected functions in file 312.

Nevertheless, when function B( ) is provided to dynamic versioning module 330, module 330 will similarly first generate a new version of function B( ) which is function B_1( ). Function B_1( ) will produce an outcome the same as that of function B( ) but function B_1( ) will have a different code execution flow as compared to the code execution flow of function B( ). The code execution flow in function B_1( ) may be differentiated from the code execution flow in function B( )

by applying source to source obfuscation methods which change the control flow of the execution of code in function $B_1( )$.

Dynamic versioning module 330 will then JIT compile function $B_1( )$ and update global mapping table 315 by replacing the address of function B( ) in table 315 with the address of compiled function $B_1( )$. This means that when function B( ) is eventually called by main thread module 320, the compiled function $B_1( )$ will instead be executed as the address of function B( ) in global mapping table 315 has been updated with the address of function $B_1( )$ during the updating step.

It is important to note that in embodiments of the invention, the child thread randomly selects the functions that are to be provided to dynamic versioning module 330. Further, child thread 325 does not necessarily cause dynamic versioning module 330 to generate a fixed number of versions for a selected function at one go. Rather, it randomly picks a function from file 312, carries out the obfuscation of the selected function's control flow using dynamic versioning module 330 and then moves on to randomly select another function to be similarly processed. After a random number of functions have been selected, child thread 325 may randomly reselect the same function again to be processed by dynamic versioning module 330.

In other embodiments of the invention, child thread 325 may be configured to only randomly select functions that have yet to be executed by main thread 320. This means that functions that have been executed or called by main thread 320 will no longer be available for selection by child thread 325. This is to ensure that child threat 325 may focus its efforts on obfuscating functions that have yet to be called by main thread 320.

Returning to the example above, when function B( ) is called by main thread 320, main thread 320 will then check global mapping table 315 to obtain the latest address associated with function B( ). Under the assumption that the address of function B( ) in table 315 has been replaced with the address of the most recently compiled version of function B( ), e.g. function $B_5( )$, main thread 320 will retrieve the address associated with compiled function $B_5( )$ and main thread module 320 will execute compiled function $B_5( )$ instead of function B( ). One skilled in the art will recognize that function $B_5( )$ is selected as a random function in this example.

The processes above then repeat themselves throughout the execution of file 312 by main thread 320 until main thread 320 has completed the running of file 312. Main thread 320 then terminates child thread 325 before main thread 320 ends.

Figure 4 illustrates a flow diagram showing the application of the dynamic versioning process to a function in a source code whereby the function is provided within a “for-loop” having “i" iterations in accordance with embodiments of the invention. In operation, source code 405 that is to be JIT compiled and executed within a virtual machine will first be provided to translator 410 to translate the source code to an intermediate representation file format before the intermediate representation file is provided to the virtual machine (not shown in Figure 4). A sample of source code 405 is illustrated below. One skilled in the art will recognize that this source code is shown as an example only and does not limit the invention in any way.

```
int main( )
        {
        int i;
        for (i=1; … ; i++)
        {
                A();
                B();
        }
        return ()
```

Source code 405 is then provided to translator 410 and this code will be converted by translator module 410 into an intermediate representation file. When the intermediate representation file is loaded during execution, the file will be JIT-compiled and addresses of functions in the JIT-compiled file will populate a global mapping table that is instantiated when the file is loaded. As mentioned in the previous example, every function in the intermediate representation file will be associated with an entry in this table.

Through a fixed mapping strategy, the location of function A( ) for all the iterations of “*i*" in the global mapping table may be obtained from the intermediate representation file. Initially, the default bit-code for function A( ) will be copied to a newly allocated block of memory and the beginning address of that block will be recorded in the entry for function A( ) in the global mapping table. This means that the first time that the main thread module 420 executes function A( ),i.e. when “*i = 1*”, it will retrieve the address of function A ( ) from the global mapping table.

Also, at the beginning of the application’s execution, the virtual machine will spawn a child thread module 440. This child thread module will initiate its processes by randomly selecting an entry in the global mapping table. Under the assumption that the entry

corresponding to function A( ) is randomly selected by child thread 440, child thread 440 will then obtain the function referenced by the entry in the global mapping table, i.e. function A( ). This function will then be provided to a dynamic versioning module to generate a function having a different control flow but having a same outcome, for example function A_1( ), by using various obfuscation techniques at its disposal. Once the dynamic versioning module has generated function A_1( ), it is JIT compiled and it will then alter the entry for function A( ) in the global mapping table to point to A_1( ). This means that for the next iteration of “i", when function A( ) is called, A_1( ) will be executed instead.

After the global mapping table has been updated with the address of function A_1( ), completely in parallel and in total disregard for what the main thread 420 is doing, child thread 440 will then randomly select another entry in the global mapping table and repeat the process of generating another version of the function that corresponds to the selected entry. Hence under the assumptions that the different versions of function A( ) are generated as function A_1( ), function A_2( ), etc. and the different versions of function B( ) are generated as function B_1( ), function B_2( ), etc. then interleaving of the actual execution (including multiple repeats of the functions) of the iterations of “*i*" are possible. For example, for iteration i = 1, 2, 3, 4, 5, the functions JIT-compiled and executed in each iteration may comprise A_1( );B_1( ), A_2( );B_2( ), A_2( );B_2( ), A_3( );B_2( ), A_3( );B_3( ) … or A( );B( ), A_2( );B( ), A_2( );B_2( ), A_2( );B_2( ), A_6( );B_3( ) etc. It should be noted that the various combinations of functions in each iteration will randomly vary from one run to the next.

In accordance with embodiments of the invention, a method for a device to dynamically modify a just-in-time (JIT) compiled intermediate representation file during an execution of the JIT compiled intermediate representation file comprises the device carrying out the following steps:

Step 1, receive the JIT compiled intermediate representation file, wherein the JIT compiled intermediate representation file includes a main thread and is associated with a global mapping table having entries that each correspond to a function in the main thread;

Step 2, spawn a child thread from the main thread, and direct the child thread to:

randomly retrieve a first function from the main thread and apply a dynamic versioning algorithm to the first function to generate a unique version of the first function such that the unique version produces an outcome that is the same as the first function;

Step 3, compile the unique version of the first function; and

Step 4, replace a first address associated with the first function in the global mapping table with an address of the compiled unique version of the first function such that when the main thread calls the first function, the main thread will execute the compiled unique first function, wherein the main thread and the child thread are run in parallel

In order to provide such a method, a process is needed for configuring the device to dynamically modify an intermediate representation file during a JIT compilation and execution of the intermediate representation file. The following description and Figure 5 describes embodiments of processes that provide the necessary steps in accordance with this invention.

Figure 5 illustrates process 500 that is performed by a device to dynamically modify an intermediate representation file during execution of the intermediate representation file. Process 500 begins at step 505 with the device receiving the intermediate representation file. Process 500 then JIT-compiles the intermediate representation file and populates a global mapping table with addresses of compiled functions from the file.

Process 500 then proceeds to spawn child thread 515 from main thread 540 of the intermediate representation file at step 510. As main thread 540 is executing commands contained in the intermediate representation file, child thread 515 will randomly retrieve a function from the intermediate representation file being executed by main thread 540 at step 520. Process 500 then applies a dynamic versioning algorithm to the retrieved function at step 525 to generate a unique version of the retrieved function such that the unique version produces a same outcome but comprises a different control flow from the original function. Process 500 then JIT-compiles the new unique version of the function at step 533 and uses its address to replace a corresponding entry in the global mapping table at step 535. This means that when main thread 540 calls the original function by referring to its corresponding entry in the global mapping table, main thread 540 will call the newly generated unique function instead. The new version of the retrieved function will then be executed. Process 500 then returns to step 520 to randomly select another function from the intermediate representation file that's being executed

by main thread 540. Process 500 then repeats steps 520-535 until main thread 540 has completed the running of the intermediate representation file.

Meanwhile, as process 500 is carrying out steps 520-535, process 500 will be utilizing main thread 540 to continuously execute the code contained in the intermediate representation file until process 500 encounters a function at step 542. Process 500 then utilizes an entry in the global mapping table that corresponds to the encountered function to call the function. Process 500 then JIT compiles and executes the function referred to in the global mapping table's entry at step 545. Process 500 then proceeds to step 548 whereby it determines whether there is a next function to be executed in the intermediate representation file. If there is a next function, process 500 returns to step 542 where process 500 continues the execution of the intermediate representation file until it encounters the next function in the file. Process 500 then repeats steps 542 to 548 until all the functions have been executed. Process 500 then ends when main thread 540 has completed the running of the intermediate representation file.

The above is a description of embodiments of a device and process in accordance with the present invention as set forth in the following claims. It is envisioned that others may and will design alternatives that fall within the scope of the following claims.

CLAIMS:

1. A device for dynamically modifying a just-in-time (JIT) compiled intermediate representation file during an execution of the JIT compiled intermediate representation file comprising:

a processor; and

a non-transitory media readable by the processor, the non-transitory media storing instructions that when executed by the processor, cause the processor to:

receive the JIT compiled intermediate representation file, wherein the JIT compiled intermediate representation file includes a main thread and is associated with a global mapping table having entries that each correspond to a function in the main thread;

spawn a child thread from the main thread, and direct the child thread to:

randomly retrieve a first function from the main thread and apply a dynamic versioning algorithm to the first function to generate a unique version of the first function such that the unique version produces an outcome that is the same as the first function,

compile the unique version of the first function, and

replace a first entry associated with the first function in the global mapping table with an address of the compiled unique version of the first function such that when the main thread calls the first function, the main thread will execute the compiled unique first function,

wherein the main thread and the child thread are run in parallel.

2. The device according to claim 1 wherein the dynamic versioning algorithm executed by the processor, cause the processor to:

generate a version of the first function;

change a control flow of code execution in the version of the first function; and

obfuscate the code in the version of the first function.

3. The device according to claim 2 wherein the step of changing the control flow of code execution in the version of the first function comprises at least one of: inserting benign code, adding of jump instructions, modifying entry points into the code, call redirection, rearranging of code blocks or modules, creating a replica function, applying obfuscation algorithms,

changing symbol attributes, segmenting blocks of code, replacing calls or encrypting code portions.

4. The device according to claims 2 or 3 wherein the obfuscating the code in the version of the first function comprises at least one of: inserting benign code, adding of jump instructions, modifying entry points into the code, call redirection, rearranging of code blocks or modules, creating a replica function, applying obfuscation algorithms, changing symbol attributes, segmenting blocks of code, replacing calls or encrypting code portions.

5. The device according to anyone of claims 1 to 4 further comprising:
instructions for directing the processor to direct the child thread to:
randomly retrieve a second function from the main thread and apply the dynamic versioning algorithm to the second function to generate a unique version of the second function such that the unique version of the second function produces an outcome that is the same as the second function;
compile the unique version of the second function;
replace a second entry associated with the second function in the global mapping table with an address of the compiled unique version of the second function such that when the main thread calls the second function, the main thread will execute the compiled unique second function.

6. The device according to claim 5 wherein the second function comprises the first function.

7. The device according to claim 5 or 6 wherein the dynamic versioning algorithm executed by the processor, cause the processor to:
generate a version of the second function;
change a control flow execution of code in the version of the second function; and
obfuscate the code in the version of the second function.

8. The device according to any one of claims 1 to 7 wherein the intermediate representation file comprises a bitcode file or a bytecode file.

9. The device according to anyone of claims 1 to 8 further comprising:

instructions for directing the processor to:

encode machine codes associated with compiled versions executed by the main thread using a secret key.

10. The device according to anyone of claims 1 to 9 wherein the first function comprises a function that has not been executed by the main thread.

11. A method for using a computer system to dynamically modify a just-in-time (JIT) compiled intermediate representation file during an execution of the JIT compiled intermediate representation file, the method comprising:

receiving the JIT compiled intermediate representation file, wherein the JIT compiled intermediate representation file includes a main thread and is associated with a global mapping table having entries that each correspond to a function in the main thread;

spawning a child thread from the main thread, and directing the child thread to:

randomly retrieving a first function from the main thread and applying a dynamic versioning algorithm to the first function to generate a unique version of the first function such that the unique version produces an outcome that is the same as the first function,

compiling the unique version of the first function, and

replacing a first entry associated with the first function in the global mapping table with an address of the compiled unique version of the first function such that when the main thread calls the first function, the main thread will execute the compiled unique first function,

wherein the main thread and the child thread are run in parallel.

12. The method according to claim 11 wherein the step of applying a dynamic versioning algorithm to the first function to generate a unique version of the first function comprises the steps of:

generating a version of the first function;

changing a control flow of code execution in the version of the first function; and

obfuscating the code in the version of the first function.

13. The method according to claim 12 wherein the changing the control flow of code execution in the version of the first function comprises at least one of: inserting benign code, adding of

jump instructions, modifying entry points into the code, call redirection, rearranging of code blocks or modules, creating a replica function, applying obfuscation algorithms, changing symbol attributes, segmenting blocks of code, replacing calls or encrypting code portions.

14. The method according to claims 12 or 13 wherein the obfuscating the code in the version of the first function comprises at least one of: inserting benign code, adding of jump instructions, modifying entry points into the code, call redirection, rearranging of code blocks or modules, creating a replica function, applying obfuscation algorithms, changing symbol attributes, segmenting blocks of code, replacing calls or encrypting code portions.

15. The method according to anyone of claims 11 to 14 further comprising:

randomly retrieving a second function from the main thread and applying the dynamic versioning algorithm to the second function to generate a unique version of the second function such that the unique version of the second function produces an outcome that is the same as the second function;

compiling the unique version of the second function; and

replacing a second entry associated with the second function in the global mapping table with an address of the compiled unique version of the second function such that when the main thread calls the second function, the main thread will execute the compiled unique second function.

16. The method according to claim 15 wherein the second function comprises the first function.

17. The method according to claim 15 or 16 wherein the step of applying the dynamic versioning algorithm to the second function to generate a unique version of the second function comprises the step of:

generating a version of the second function;

changing a control flow execution of code in the version of the second function; and

obfuscating the code in the version of the second function.

18. The method according to any one of claims 11 to 17 wherein the intermediate representation file comprises a bitcode file or a bytecode file.

19. The method according to anyone of claims 11 to 18 further comprising:

encoding machine codes associated with compiled versions executed by the main thread using a secret key.

20. The method according to anyone of claims 11 to 19 wherein the first function comprises a function that has not been executed by the main thread.