US 20170124322A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2017/0124322 A1**

GUILLEY et al. (43) **Pub. Date: May 4, 2017**

(54) **A COMPUTER IMPLEMENTED METHOD AND A SYSTEM FOR CONTROLLING DYNAMICALLY THE EXECUTION OF A CODE**

(71) Applicant: **SECURE-IC SAS**, Cesson-Sévigné (FR)

(72) Inventors: **Sylvain GUILLEY**, PARIS (FR); **Thibault PORTEBOEUF**, PARIS (FR)

(73) Assignee: **SECURE-IC SAS**, Cesson-Sévigné (FR)

(21) Appl. No.: **15/317,325**

(22) PCT Filed: **Jun. 19, 2015**

(86) PCT No.: **PCT/EP2015/063880**

§ 371 (c)(1),
(2) Date: **Dec. 8, 2016**

(30) **Foreign Application Priority Data**

Jun. 20, 2014 (EP) 14305954.1

Publication Classification

(51) **Int. Cl.**

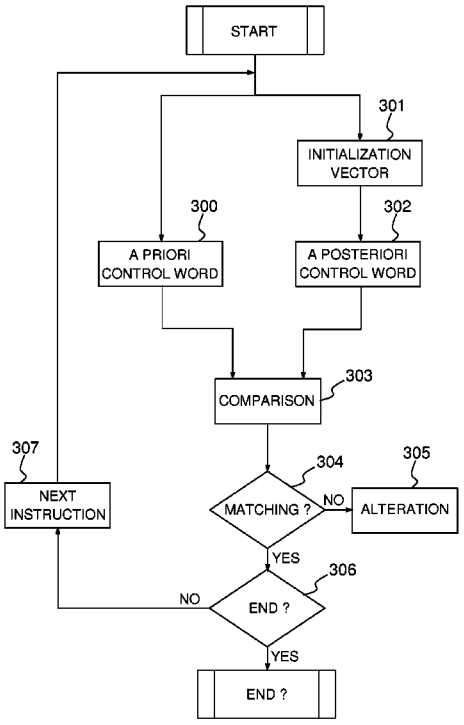
G06F 21/54	(2006.01)
G06F 9/44	(2006.01)
G06F 21/56	(2006.01)

(52) **U.S. Cl.**

CPC ***G06F 21/54*** (2013.01); ***G06F 21/566*** (2013.01); ***G06F 8/75*** (2013.01); *G06F 2221/2125* (2013.01); *G06F 2221/034* (2013.01)

(57) **ABSTRACT**

According to the invention, there is provided a computer implemented method for controlling dynamically the execution of a code by a processing system, said execution being described by a control flow graph comprising a plurality of basic blocks composed of at least an input node and an output node, a transition in the control flow graph corresponding to a link between an output node of origin belonging to a first basic block and an input node of a second basic block, a plurality of initialization vectors being associated to the output nodes at the time of generating the code, an a priori control word being associated to each input node which is linked to the same output node of origin according the control flow graph, said a priori control word being precomputed at the time of generating the code by applying a predefined deterministic function F to the initialization vector associated to its output node of origin, the following steps being applied once the execution of the output node belonging to a first basic block is terminated and at the time of executing the input node of a second basic block: providing (**300**) the a priori control word associated to the input node of the second basic block; providing (**301**) the initialization vector associated to the output node of the first basic block; determining (**302**) an a posteriori control word by applying to the provided initialization vector the same function F which has been used for generating the a priori control word; determining (**303**, **304**) if the a priori control word matches with the a posteriori control word, a forbidden transition in respect to the control flow graph being otherwise detected (**305**).



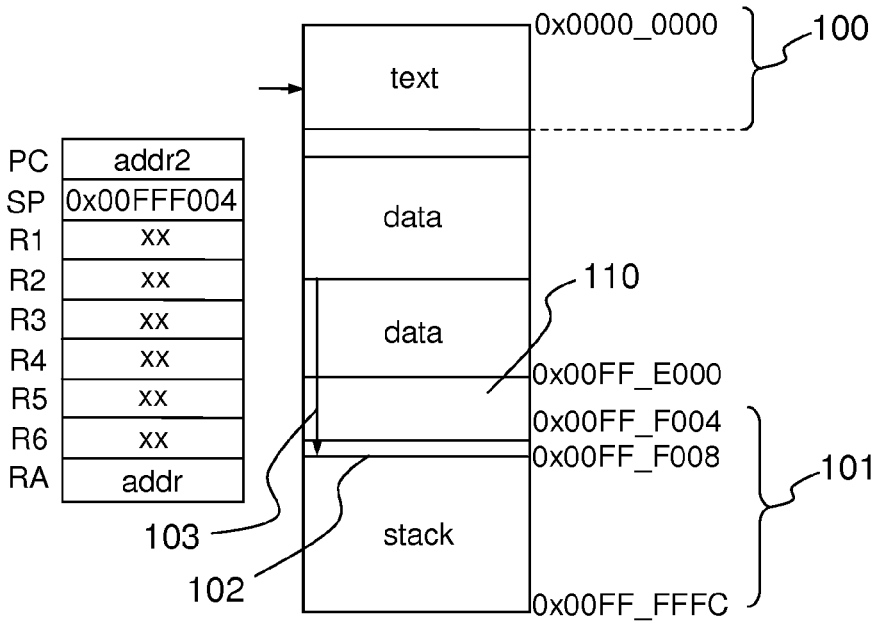

PC
addr2
SP
0x00FFF004
R1
xx
R2
xx
R3
xx
R4
xx
R5
xx
R6
xx
RA
addr
text
data
data
stack
0x0000_0000
100
110
0x00FF_E000
0x00FF_F004
0x00FF_F008
101
103
102
0x00FF_FFFC

FIG.1A

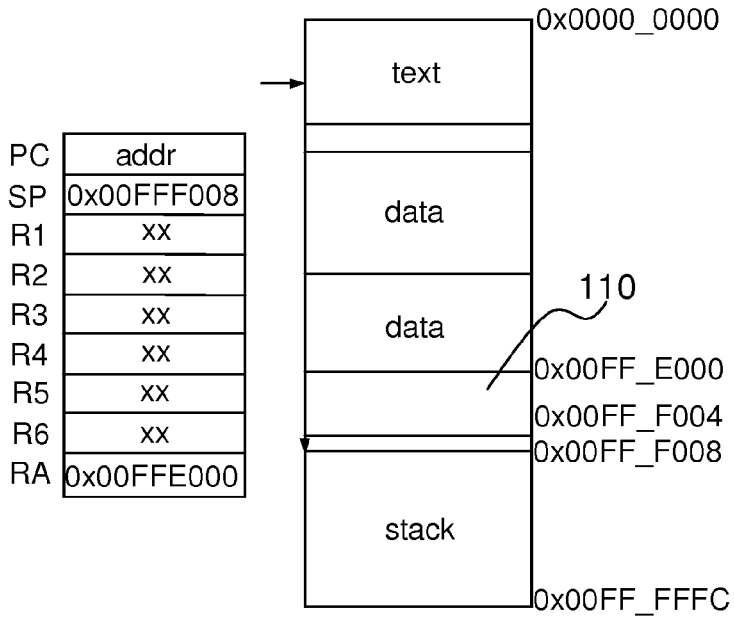

PC
addr
SP
0x00FFF008
R1
xx
R2
xx
R3
xx
R4
xx
R5
xx
R6
xx
RA
0x00FFE000
text
data
data
stack
0x0000_0000
110
0x00FF_E000
0x00FF_F004
0x00FF_F008
0x00FF_FFFC

FIG.1B

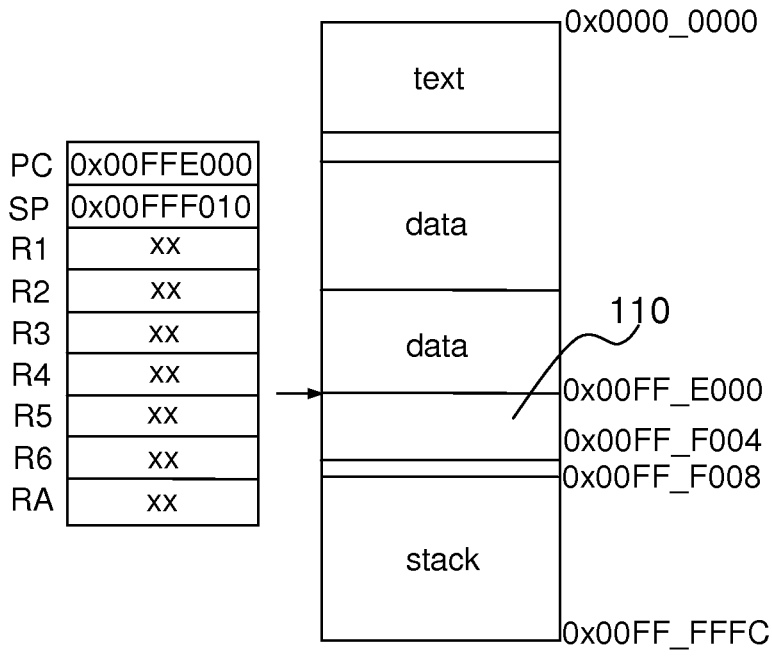

PC
0x00FFE000
SP
0x00FFF010
R1
xx
R2
xx
R3
xx
R4
xx
R5
xx
R6
xx
RA
xx
text
data
data
stack
0x0000_0000
110
0x00FF_E000
0x00FF_F004
0x00FF_F008
0x00FF_FFFC

FIG.1C

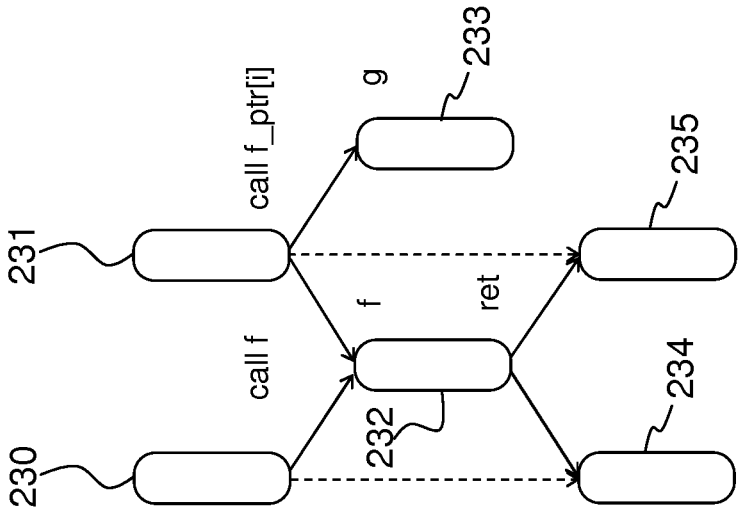

231
call f_ptr[i]
g
233
235
f
ret
call f
234
232
230

FIG.2B

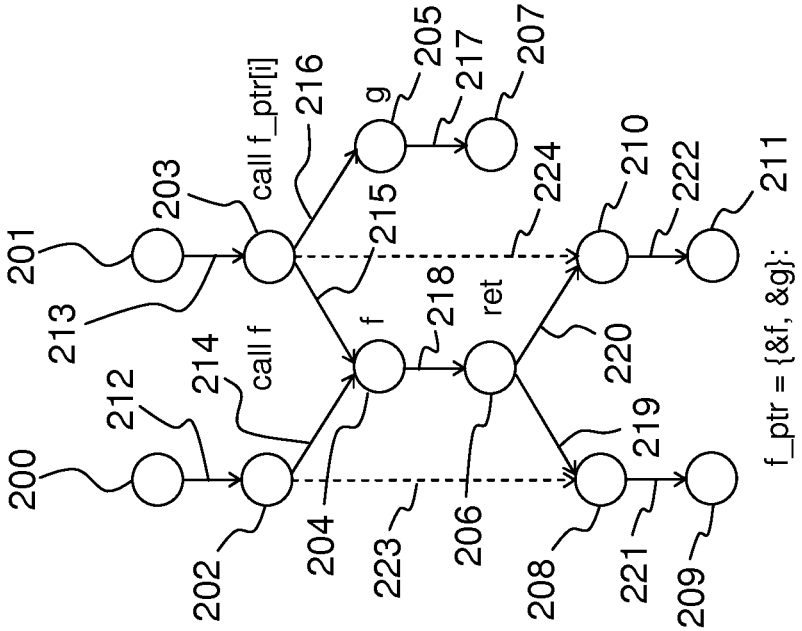

201
203
call f_ptr[i]
216
g
205
217
207
213
215
224
210
222
211
f
218
ret
220
call f
214
212
200
219
f_ptr = {&f, &g};
202
204
223
206
208
221
209

FIG.2A

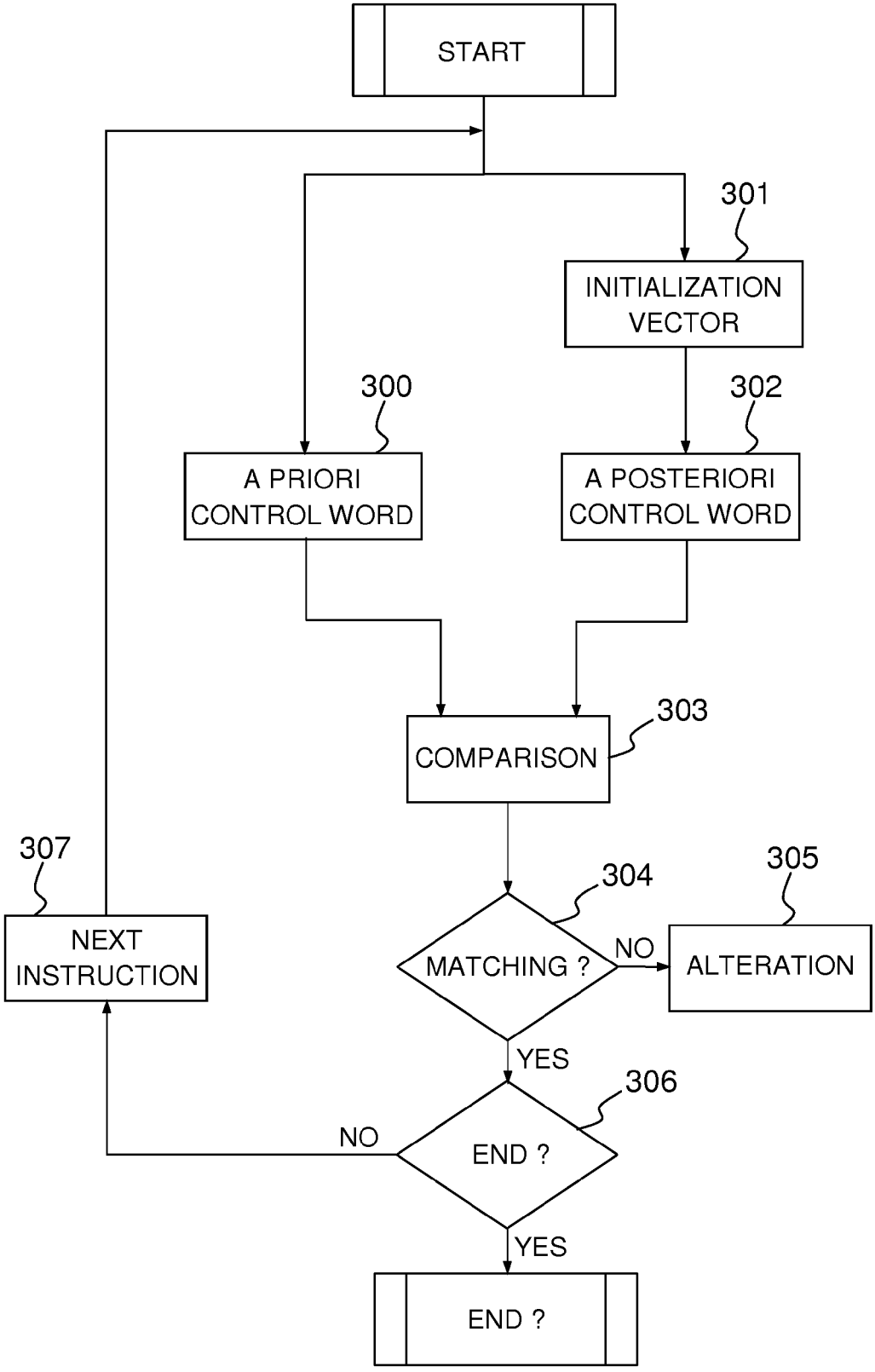

START
301
INITIALIZATION VECTOR
300
A PRIORI CONTROL WORD
302
A POSTERIORI CONTROL WORD
303
COMPARISON
307
NEXT INSTRUCTION
304
MATCHING ?
NO
305
ALTERATION
YES
306
NO
END ?
YES
END ?

FIG.3

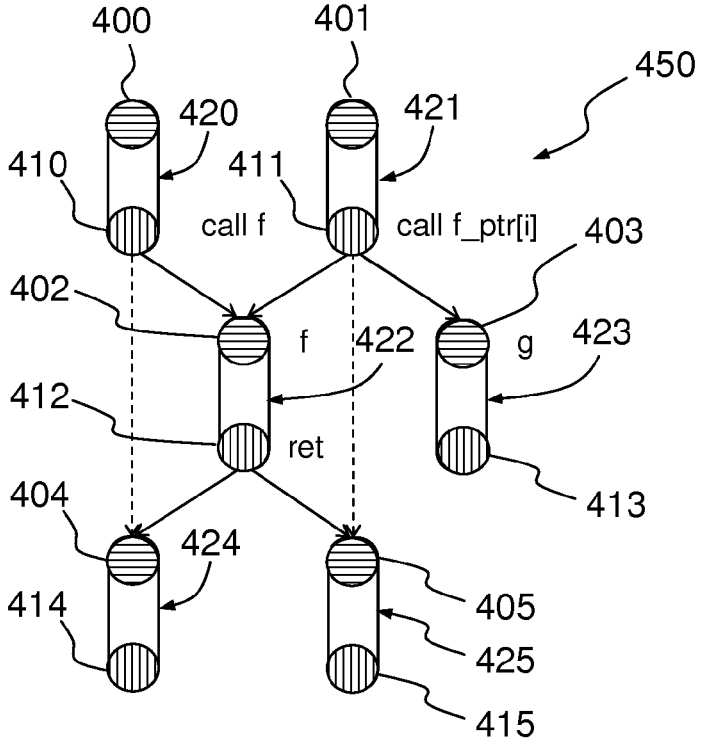

400
401
450
420
421
410
411
call f
call f_ptr[i]
403
402
f
422
g
423
412
ret
413
404
424
405
414
425
415

FIG.4

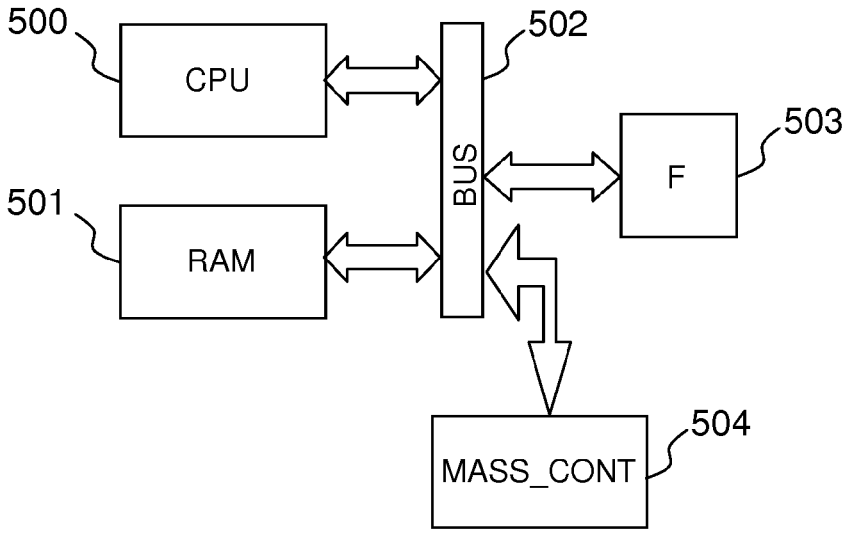

500
CPU
502
BUS
503
F
501
RAM
504
MASS_CONT

FIG.5

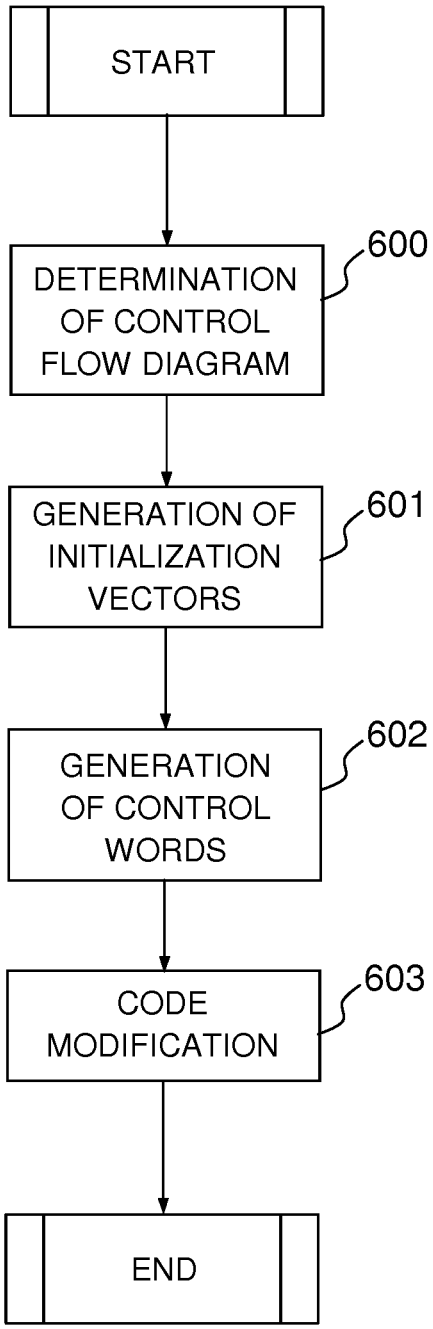

START
DETERMINATION OF CONTROL FLOW DIAGRAM
600
GENERATION OF INITIALIZATION VECTORS
601
GENERATION OF CONTROL WORDS
602
CODE MODIFICATION
603
END

FIG.6

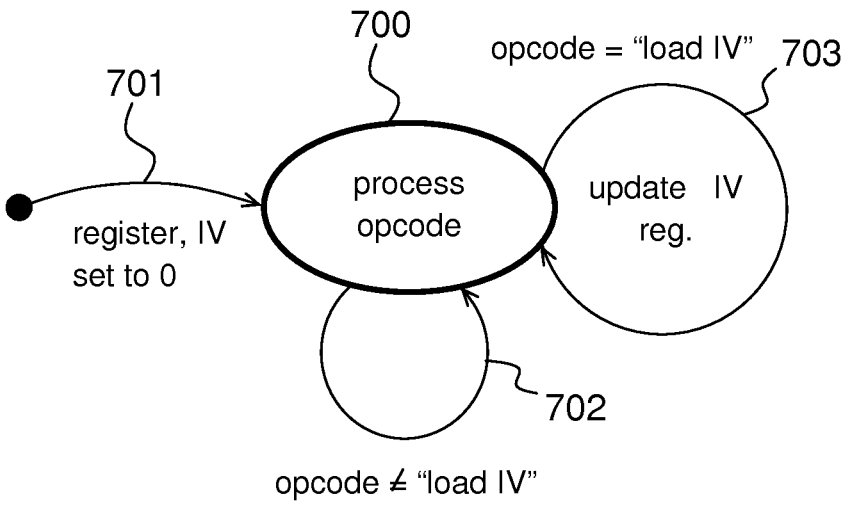

700
opcode = "load IV"
703
701
process opcode
update IV reg.
register, IV set to 0
702
opcode ≠ "load IV"

FIG.7

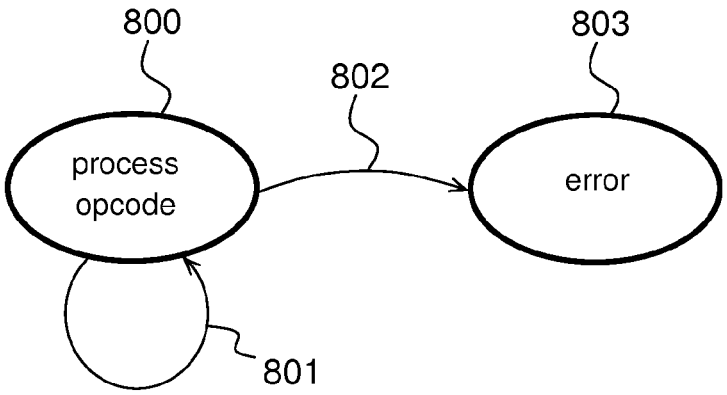


FIG.8

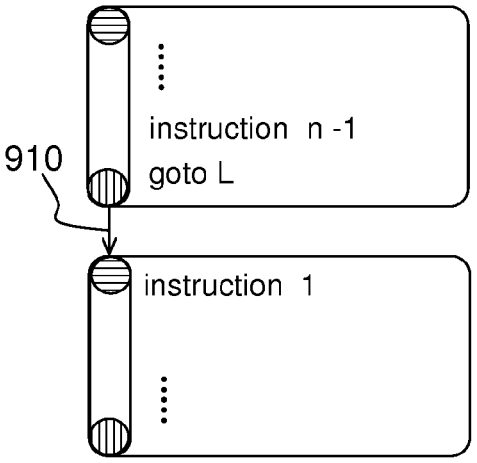


FIG.9A

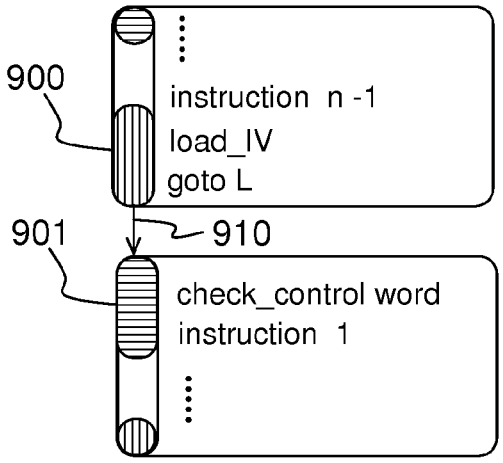


FIG.9B

A COMPUTER IMPLEMENTED METHOD AND A SYSTEM FOR CONTROLLING DYNAMICALLY THE EXECUTION OF A CODE

[0001] This invention relates to a computer implemented method for controlling dynamically the execution of a code, a processing system and a method for generating a code which can be executed by said processing system. The invention is particularly, but not exclusively, applicable to secured embedded systems.

[0002] Cyber attacks consist into running a software code in a way that has not been anticipated at design time.

[0003] One possibility to do so results from obvious errors (weak passwords, tricking the user into doing something bad for him, and similar techniques also called as social engineering) or poor quality software (protection with holes or backdoors for example). Protections against those vulnerabilities are often non-technical.

[0004] An alternative which can be considered to implement cyber attacks consists in bringing the program in a non-specified state by sending it crafted data. This means that the program is designed to follow a certain amount of paths, but that malevolent inputs are able to abuse the programme. This results in a hijacking of the execution flow and leads to a remote takeover. It is customary to make the distinction between the crafted user-provided data that allows deflecting the program from its intended behaviour (also referred to as the trigger) and the malicious behaviour (also referred to as the payload).

[0005] Briefly, the intimate reasons why these exploits are possible are due to a two-factor reason, namely a combination of programming language weaknesses and execution permissivity.

[0006] One typical example is the stack smashing thanks to a buffer overflow, for instance. Listing **1** is a simple C program which is used hereinafter to illustrate the principle of stack smashing.

Listing 1

```
#include <stdio.h>
void dummy( )
{
        printf("Should not be called\n");
}
void get_data(char* data)
{
        printf("Input your data:\n");
        scanf("%s",data);
}
int main( )
{
        char data[10];
        get_data(data);
        return 0;
}
```

[0007] With a specially crafted string data input by the user from outside of the program, it is possible to call function dummy( ) that is otherwise not callable from the functional control flow graph. Data will certainly contain some binary non printable characters, but this is not checked in this example.

[0008] The exploit is sketched in FIGS. **1**A, **1**B and **1**C wherein a physical memory is represented, said memory being used as a buffer containing two distinct areas **100** and **101**.

[0009] In this example, the stack **101** grows from address 0x00FF_FFFC downwards, and contains the stacking of multiple frames, a frame comprising local variables, arguments and a return address. Namely, the stack depth is equal to the depth of the functions call tree.

[0010] The program is itself in a .text segment **100**, from address 0x0000_0000 upwards.

[0011] If the attacker is able to write some data in the buffer which is intended to be used by the program, then some data **103** can be written into the stack **101**. In particular, the return address of a called function can be overwritten and replaced by a different address **102** chosen by the attacker (see FIG. **1**A).

[0012] When the current function returns which also means that the current frame is quitted, then the processor pops the crafted return function (FIG. **1**B) and branches on it (see FIG. **1**C). Then, the program is executing the payload **110** of the attacker, and not the original program.

[0013] The stack is deliberately a memory area which can be freely read from and written to. Indeed, some processors are able to forbid some accesses to some given memory areas. For instance, when a MMU (Memory Management Unit) is implemented in a processing system, large portions of the RAM memory (Random Access Memory) can be set in read-only mode. This prevents for instance a program from inadvertently or malevolently overwriting constants. The .text segment **100** can also be set to read-only mode when executing it.

[0014] The stack is nevertheless a general purpose memory chunk where any access is possible, for a better convenience of the execution. Therefore, the attacker is able to corrupt the stack by illegally writing out a buffer boundaries.

[0015] Besides, the attacker can write code in the stack and then to jump on it. This strategy would even be more straightforward. However, this attack technique is easy to counteract by using state-of-the-art techniques, such as the NX bit technology. NX bit technology is described in the document entitled "Data Execution Prevention", Hewlett Packard, 2005.

[0016] Therefore, with most of the state-of-the-art processors, the attacker must inject his payload by writing only data and not machine code. As depicted before, an interesting approach for the attacker is to overwrite the return address memorized in the stack (FIGS. **1**A-**1**C). In that case and as already explained, the program is re-routed elsewhere which means that the control flow is diverted.

[0017] Even if the execution of arbitrary code stays complex, several existing techniques allow taking advantage of this situation. For instance, return-oriented programming technique (ROP) makes possible to build a program by borrowing chunks of code from different places, especially in the legacy libraries that are linked with the program (e.g., the C library aka libc, where handy and hardly avoidable functions such as malloc are implemented).

[0018] As a summary, these state-of-the-art attacks alter the execution graph by replacing jump addresses with a forged data that will be erroneously interpreted by the processor as addresses.

[0019] Several existing protections can be used depending of the context.

[0020] Virus can be detected and then quarantined or removed by « anti-viruses » programs, that either check statistically their source code (against some portions of the binary that are renowned to be evil) or dynamically analyze their behaviors. However, anti-viruses act too late, since they detect the virus once it is already inserted in the execution system.

[0021] Other techniques allow to proactively catch the exploit when it is triggered, so as to block it « just-in-time » (JIT) that is to say red-handed. For that purpose, two strategies are usually employed.

[0022] The first strategy is called randomization. The ASLR technique (Address Space Layout Randomization) is one of them. The memory locations of program functions and data are chosen differently at each execution, and thus their addresses are not predictable. However, due to some limitations (e.g. finite length of the addresses), the ASLR can be bypassed. Also, some advanced attacks manage to execute the virus within the ASLR. Hence, this protection can be considered as weak.

[0023] A second strategy is hardening and CFI (Control Flow Integrity) is one example. CFI can be used with a static pre-processing, or dynamically (what we called « JIT »). But this solution is pure software and has therefore several shortcomings. It considerably slows down the execution of the program. Furthermore, it is itself attackable as it is a software only solution. For example, it can be bypassed if there is an exploitable bug in it.

[0024] It would be desirable to address the above issues, to develop a solution for controlling the execution of a code by a processing system.

[0025] According to the invention, there is provided a computer implemented method for controlling dynamically the execution of a code by a processing system, said execution being described by a control flow graph comprising a plurality of basic blocks composed of at least an input node and an output node, a transition in the control flow graph corresponding to a link between an output node of origin belonging to a first basic block and an input node of a second basic block, a plurality of initialization vectors being associated to the output nodes at the time of generating the code, an a priori control word being associated to each input node which is linked to the same output node of origin according the control flow graph, said a priori control word being precomputed at the time of generating the code by applying a predefined deterministic function F to the initialization vector associated to its output node of origin, the following steps being applied once the execution of the output node belonging to a first basic block is terminated and at the time of executing the input node of a second basic block:

[0026] providing the a priori control word associated to the input node of the second basic block;

[0027] providing the initialization vector associated to the output node of the first basic block;

[0028] determining an a posteriori control word by applying to the provided initialization vector the same function F which has been used for generating the a priori control word;

[0029] determining if the a priori control word matches with the a posteriori control word, a forbidden transition in respect to the control flow graph being otherwise detected.

[0030] According to one aspect of the invention, the code execution is interrupted when an output control word and an input control word belonging to two subsequent basic blocs are not identical.

[0031] For example, the second basic block is enciphered at the time of generating the code by using its associated a priori control word as a ciphering key, said method comprising the step of deciphering the second basic block by using its associated a posteriori control word as a deciphering key at the time of executing the input node of said second basic block.

[0032] According to the invention, there is also provided a processing system for executing a code comprising a processor, said system comprising also:

[0033] a memory area configured to store the code to be executed, said code being associated to a control flow graph comprising a plurality of basic blocks composed of at least an input node and an output node, a transition in the control flow graph corresponding to a link between an output node of origin belonging to a first basic block and an input node of a second basic block, said memory area being also configured to store a plurality of initialization vectors associated to the output nodes at the time of generating the code, to store a plurality of a priori control words, an a priori control word being associated to each input node which is linked to the same output node of origin according the control flow graph, said a priori control word being precomputed at the time of generating the code by applying a predefined deterministic function F to the initialization vector associated to its output node of origin;

[0034] an hardware implemented module configured to generate a posteriori control words, an a posteriori control word being generated for a given input node by applying to the initialization vector the same function F which has been used for generating the a priori control associated to the same input node;

[0035] a module configured for determining if an a posteriori control and an a priori control word which are associated to the same input node are matching, a forbidden transition in respect to the control flow graph being otherwise detected.

[0036] As an example, the initialization vectors encode the nature of the jumps implementing allowed transitions in the control flow graph.

[0037] According to one aspect of the invention, the initialization vectors are memorized in the processing system using a set of dedicated registers. According to another aspect of the invention, function F can be adapted to take into account an additional input which plays the role of an activation key.

[0038] As an example, the activation key is unique per device.

[0039] Alternatively, the activation key can be unique per program.

[0040] In one embodiment, the basic blocks are enciphered at the time of generating the code by using their associated a priori control words as a ciphering key, said system comprising a module to decipher said basic blocks at

the time of executing their input node by using their associated a posteriori control word as a deciphering key.

[0041] According to the invention, there is also provided a computer implemented method for generating an improved version of an initial code intended to be executed on the processing system as described before, comprising the steps of:

[0042] determining a control flow graph representative of an unaltered execution of the code, said control flow graph comprising a plurality of basic blocks composed of at least an input node and an output node, a transition in the control flow graph corresponding to a link between an output node of origin belonging to a first basic block and an input node of a second basic block in the control flow graph;

[0043] generating a plurality of initialization vectors, an initialization vector being associated to each output node at the time of generating the code;

[0044] for each input node, determining an a priori control word associated to each input node which is linked to the same output node of origin according the control flow graph, said a priori control word being precomputed at the time of generating the code by applying a predefined deterministic function F to the initialization vector associated to its output node of origin;

[0045] modifying the initial code by inserting the a priori control words in line with their corresponding instructions.

[0046] The control flow diagram is determined for example through a static analysis of an initial code, said initial code being a source code.

[0047] For example, the control flow diagram is determined through a static analysis of an initial code, said initial code being an assembly code.

[0048] For example, the control flow diagram is determined through a static analysis of an initial code, said initial code being a binary code.

[0049] For example, the initialization vectors are attributed randomly.

[0050] For example, the value of an a priori control word is chosen as a function F of at least a destination address which is the address where the instruction corresponding to an input node following an output node of origin according to the control flow graph is located.

[0051] For example, the a priori control words are inserted inside the code to be executed.

[0052] For example, the a priori control words are inserted in line with the instructions corresponding to their associated input node.

[0053] For example, the initialization vectors are inserted in line with the instructions corresponding to their associated output nodes.

[0054] According to the invention, there is also provided a computer program product, stored on a computer readable medium comprising code means for causing a computer to implement the method for generating an improved version of an initial code as described before.

[0055] A better understanding of the embodiments of the present invention can be obtained from the following detailed description in conjunction with the following drawings, in which:

[0056] FIGS. **1**A, **1**B and **1**C give an example of stack smashing thanks to a buffer overflow;

[0057] FIG. **2**A is an example of control flow diagram;

[0058] FIG. **2**B provides an example of a control flow diagram which is abstracted as an oriented graph containing only basic blocks;

[0059] FIG. **3** illustrates a method for controlling dynamically the execution of a code;

[0060] FIG. **4** gives an example of a representation of a control flow graph including transformed input and output nodes;

[0061] FIG. **5** illustrates an example of a processing system comprising a mechanism to control the execution of a code;

[0062] FIG. **6** provides an example of a method for generating an improved version of a code which is executable by the processing system according to the invention;

[0063] FIG. **7** shows the operation of an opcode designed to set a new initialization vector in the form of a finite state machine;

[0064] FIG. **8** illustrates the upgraded operation of the processing system in the form of a new finite state machine;

[0065] FIG. **9**A provides an example of a simple control flow graph where two nodes are connected with an edge;

[0066] FIG. **9**B gives an example of insertion of the initialization vector at the output node and the a priori control word and its check instruction at the other end of the edge, that is to say the input node.

[0067] In the following description, a basic block designates linear portions of code, that is to say a sequence of instructions without deviations from a straight execution.

[0068] Additionally, divergences or convergences are designating locations in the code which are corresponding respectively to the beginning and the end of a basic block. A basic block is composed of at least an input node and an output node which are representing respectively a convergence and a divergence in the control flow graph.

[0069] Moreover, the jump from the output node of a first basic block towards the input node of a second basic block is designated as a transition.

[0070] Further, the word instruction refers to an assembly line of code.

[0071] A source code can be analyzed in order to produce an oriented graph, customarily referred as the CFG (Control Flow Graph). The CFG is an oriented graph wherein each instruction is a node (or vertex) and possible sequences of instructions are indicated by the presence of an oriented edge from one node to the other.

[0072] Basic blocks correspond to linear portions of a control flow graph. In such a block, and without special instruction, the program implicitly continues to the next instruction. This means the register in the processor, which is often referred as the Program Counter (PC), is by default incremented by the size of an instruction after every instruction which is non special (i.e. non-jump).

[0073] The end of a basic block corresponds to an instruction implementing a divergence, for example:

[0074] a conditional jump, like in if, switch, while and goto constructs, "||" and "&&" binary operators, "?:" ternary operator, and also calls in function arrays;

[0075] a function call/a function return.

[0076] The difference between these two is that a function call/return it implies, in addition to the « sequence break », the saving of some variables on the stack (referred to as a push) for a function call, and the restoration of variables on the stack (referred to as a pop). In assembly language, they

also belong to kinds of opcodes: « JUMPs » and « CALLs « / » RETs » . For the sake of simplicity, we refer to both sequence breaks as « jumps » .

[0077] In the scope of this invention, it is important to make a difference between statically determined jump destinations and destinations which are discovered dynamically. Several examples are provided below:

[0078] static jumps are gotos to fixed labels or function calls,

[0079] dynamic jumps are all the others.

[0080] There is a characteristic which allows differentiating dynamic jumps into two families.

[0081] A first family comprises the dynamic jumps whose possible destinations are known to belong to a finite state when the program is analyzed. Those are called direct jumps. Direct jumps are:

[0082] if, switch (at least with few cases), while, ||, &&, call of functions via an array of functions pointers;

[0083] goto to built labels (because they are necessarily within the scope of one function).

[0084] A second family comprises the other dynamic jumps which are called indirect jumps. For these jumps, the number of destinations is unknown at the compilation. Indirect jumps are:

[0085] switch (with many cases, usually >3);

[0086] return from functions that are exported, or

[0087] function calls via a register (virtual functions in C++ for example).

[0088] In this description, the destination of a jump is called a label and is noted L. The destination of a function call is called as the function address and is noted &f, f being the function. It can also be noted f like in the C language. The destination of a return has no special name, but it is implicitly saved by the processor. It is a destination address.

[0089] The entrance of a basic block does not correspond to a particular assembly location. But, when compiling a program, those are known:

[0090] for conditional jumps, as the next instruction that follows the jump instruction;

[0091] for function calls, as the beginning of the function, and for function returns, as the instruction that directly follows the call.

[0092] A special case is for longjumps in C and exceptions in C++; in this case, the stack management is « exceptional » , as the execution flow.

[0093] A program can be associated with a control flow graph which describes all of its instructions.

[0094] FIG. **2**A is an example of control flow graph CFG. Each circle **200-211** stands for one instruction, that is to say one line in the assembly dump of the code. More precisely, the CFG is depicted by the full arrows **212-222**. The dashed arrows **223**, **224** represent the next instruction of a given instruction **202**, **203**, that allow deriving the return address in function calls.

[0095] FIG. **2**B provides an example of a control flow graph which is abstracted as an oriented graph containing only basic blocks **230-235**. As already mentioned, a basic block is made of one or more instructions jumping from one to the following without divergences or convergences, except for the first and the last ones. In FIG. **2**B, the array of function pointers f_ptr contains the addresses of two functions, namely f and g. This representation illustrates function calls/returns but is also applicable to jumps.

[0096] Some programs are more complex since they are not monolithic, but use dynamic libraries as for example .so object files under GNU/Linux and .dll under Windows. In this case, the call between basic blocks from the main program to a dynamic library needs to pass through dedicated functions, but that can still be seen as basic blocks.

[0097] The construction of a control flow diagram can be achieved through a static analysis of the source code. It is also possible to recover the structure of a binary code. The recovery might be partial, all the more so as obfuscation techniques are employed to obscure the binary. But still, tools like IDA Pro perform quite well in this functionality and one can write its own disassembly tool.

[0098] FIG. **3** illustrates a method for controlling dynamically the execution of a code and FIG. **4** gives an example of a representation of a control flow graph including input and output nodes across which the control is enforced.

[0099] This method is computer-implemented. This means that the steps (or substantially all the steps) of the method are executed by at least one processor.

[0100] The method comprises a sequence of several steps which are applied during the code execution each time a basic block execution terminates, that is to say before the execution of a subsequent instruction.

[0101] The unaltered execution of the code can be described by a control flow graph comprising a plurality of basic blocks **420-425** and edges defining allowed transitions between basic blocks. A basic block **420-425** is composed of at least an input node **400-405** and an output node **410-415** which are representing respectively a convergence and a divergence in the control flow graph. As already explained, a node corresponds to an instruction of the code to be executed such as a machine code. According to the invention, a plurality of initialization vectors is attributed to the output nodes at the time of generating the code, for example at the time of compiling a source code or at the time of analyzing an assembly code. In one embodiment, a distinct initialization vector is generated for each output node identified in the control flow graph.

[0102] The initialization vectors can be memorized into the processing system. In a preferred embodiment, the initialization vectors are memorized in the processing system using a set of dedicated registers.

[0103] The method according to the invention carries out several steps once the execution of an output node belonging to a first basic block is terminated and at the time of executing the input node of a second basic block.

[0104] A step **300** aims at providing an a priori control word associated to the input node of the second basic block. This a priori control word is precomputed at the time of generating the code by applying a predefined function F to an initialization vector, said initialization vector being associated to the output node of origin. Said differently, an allowed transition according to the control flow graph can be defined by an output node of origin which belongs to a first basic block and an input node of destination which belongs to a second basic block. Thus, an initialization vector is associated to the output node of origin and an a priori control word is associated to the input node of destination. More precisely, the a priori control word is deduced of the initialization vector associated to the output node of origin by applying function F to it.

[0105] In one embodiment, the a priori control word is inserted inside the code to be executed, for example in line with the instruction corresponding to the input node of the basic blocks.

[0106] Another step **301** aims at providing the initialization vector associated to the output node of the first basic block. As an example, this initialization vector is memorized in a dedicated register, providing this vector means reading its value for a future use.

[0107] In one embodiment, the initialization vector is inserted inside the code to be executed, for example in line with the instruction corresponding to the output node of the basic blocks.

[0108] In one embodiment, a different initialization vector is randomly attributed to output nodes for every authorized transitions or edges in the control flow graph.

[0109] In another step **302**, an a posteriori control word is determined by applying the predefined function F to the provided initialization vector. According to an essential aspect of the invention, the same deterministic function F should be used for computing the a priori and the a posteriori control words associated to a given transition in the control flow graph.

[0110] An example of a deterministic function F taking an initialization vector is a SHA (Secured Hash Algorithm) cryptographic hash function of said initialization vector.

[0111] The method also comprises a step of determining **303**, **304** if one of the a priori control words matches with the a posteriori control word.

[0112] If the a priori and the a posteriori control words do not match i.e. are different, an alteration **305** of the code execution is detected. More precisely, this means that a forbidden transition in respect to the control flow graph being is detected. In that case, the code execution may be interrupted.

[0113] In one embodiment, after detecting a forbidden transition in respect to the control flow graph, a security policy is enforced by triggering a hardware and/or software function.

[0114] The method according to the invention enables a hardware-assisted protection of a program from attacks that aim at re-routing the execution flow.

[0115] Advantageously, the protection is efficient in terms of security and performance-wise.

[0116] Additionally, some hardware resources such as key registers can be hidden to the attacker. Further, the verification of the control flow graph integrity cannot be bypassed as it can be encoded in a finite state machine, and the function F itself can be hardware implemented or hidden from access of a purported attacker. Another advantage is that the control can be done in parallel with the code execution which minimizes the impact on the execution speed.

[0117] FIG. **5** illustrates a processing system comprising a mechanism to control the execution of a code.

[0118] The processing system comprises a central processing unit (CPU) **500** connected to an internal communication BUS **501**, a random access memory (RAM) **502** also connected to the BUS. The processing system further comprises a mass storage device controller **504** managing accesses to a mass memory device, such as hard drive. Mass memory devices suitable for tangibly embodying computer program instructions and data include all forms of non-volatile memory, including by way of example semiconductor memory devices, such as EPROM, EEPROM, and flash memory devices; magnetic disks such as internal hard disks and removable disks; magneto-optical disks; and CD-ROM disks. Any of the foregoing may be supplemented by, or incorporated in, specially designed ASICs (application-specific integrated circuits).

[0119] In one embodiment, the processing system comprises a stack stored inside the random access memory **501**.

[0120] As an example, a memory area located for example in the random access memory **502** stores the code to be executed. Alternatively, a memory which is located in the mass memory device **504** can be used to store the code to be executed.

[0121] This code execution can be described by a control flow graph which comprises a plurality of basic blocks. As already mentioned, a basic block is composed of at least an input node and an output node which are representing respectively a convergence and a divergence in the control flow graph. A plurality of initialization vectors is attributed to the output nodes at the time of generating said code which means before its execution by the processing system.

[0122] The processing system also comprises a memory area wherein said initialization vectors are stored. This memory area can be a memory area which belongs to the stack of the system. Alternatively, this memory area can be implemented by adding a set of dedicated registers to the CPU **500**.

[0123] The processing system also comprises a memory area wherein a plurality of a priori control words is stored. As already explained, a priori control words are precomputed at the time of generating the code. For that purpose, a predefined function F is applied to the aforementioned initialization vectors.

[0124] In one embodiment, the a priori control word is inserted inside the code to be executed, for example in line with the instruction corresponding to the input node.

[0125] Further, the processing system comprises a hardware implemented module **503** for generating a posteriori control words. An a posteriori control word is generated by applying the function F to the initialization vector which is associated to a given output node at the time of executing an instruction which follows the execution of this output node.

[0126] Additionally, the processing system comprises a module for determining if an a posteriori control word which has been calculated at the time of leaving a basic block matches an a priori control word. An alteration of the code execution is detected if the two control words do not match. This module can be either software or hardware implemented. For example, this module can be implemented by the processor **500**.

[0127] FIG. **6** provides an example of a method for generating an improved version of a code which is executable by the processing system according to the invention.

[0128] The improved code can be generated thanks to a computer implemented method which uses an initial code as an input.

[0129] A step **600** determines a control flow graph representative of an unaltered execution of the initial code. As already explained, the control flow graph comprises a plurality of basic blocks, a basic block being composed of at least an input node and an output node which are representing respectively a convergence and a divergence in the control flow graph.

[0130] The method also comprises a step **601** for generating a plurality of initialization vectors, a given initialization vector being allocated to a given output node.

[0131] Further, for each input node linked in the control flow graph to an output node associated to an initialization vector, a step **602** determines an a priori control word by applying a predefined function F to said initialization vector.

[0132] In one embodiment, for example if the code generation platform is similar to the code execution platform depicted in FIG. **5**, the function F is hardware accelerated.

[0133] Then, the initial code is modified **603** in order to generate an improved version of the code by inserting the a priori control words in line with their corresponding instructions, that is to say the instructions corresponding to their associated input nodes.

[0134] In one embodiment, the modification **603** of the code also embeds initialization vectors in the improved version of the code in line with their corresponding instructions, that is to say the instructions corresponding to their associated output nodes.

[0135] In this description, a program is said to execute with an unaltered control flow diagram if it dynamically upon execution travels through the edges and nodes previously identified statically during the link and/or dynamic link processes, for example at compilation. A processing system such as a processor can typically identify at runtime divergences (that correspond for example to « jump » instructions) but has no means to identify which instruction is a licit destination. This comes from the fact that this information is not present statically in a binary code because the notion of convergence has been semantically removed after compilation.

[0136] This information is nonetheless present while compiling, and can be extracted albeit with non-trivial efforts from a dynamic analysis of the binary. The method according to the invention verifies at runtime that the CFG is unaltered.

[0137] According to the invention, binary information called control word can be added directly to a set of chosen instructions or between instructions in order to secure the association between intended departures and arrival points. This means that the .text segment **100** will slightly grow. Alternatively, the control words may be calculated on the fly.

[0138] The verification of unaltered execution ensures that no new edge is created in the control flow graph, and therefore that an attacker cannot inject new nodes. But, of course, such verification cannot check that the correct (licite) selection of vertices is made. For example, in an "if then else" statement, a program executes with an unaltered control flow graph if after the test, the program counter points either to the beginning of the "then" or of the "else" statement, and not anywhere else.

[0139] In order to keep the control flow graph execution unaltered and to prevent a drastic slow-down in the execution, a combination of new hardware and new information in the software code is implemented. Some values can be added in the binary and these values are called control words in this description. A control word is a precomputed value which is inserted statically in the binary at compile time.

[0140] Besides, the hardware is augmented to check the control words dynamically by recomputing them. This recomputation can be made mandatory, which avoids "forgetting" the verification of "warranty of legality" of one jump in the program.

[0141] The control flow graph alteration is detected unambiguously if the control words do not match across a jump, for example if they differ.

[0142] Different ways of computing control words are provided hereafter by means of examples. They are obtained by the evaluation of a function F, that can have one or more inputs, depending on the targeted security level.

[0143] A minima, it can be checked that the nature of the jump is respected, for example that it is illegal to jump at the beginning of a function from a conditional test such as an if statement. Reciprocally, it is not allowed to jump at the beginning of a basic block from a function call or return. Technically speaking, this means that the nature of the jump can be taken as an input of F while computing a control word.

[0144] However, more advanced verifications can be done. For instance, in case of a jump destination instruction wherein all the possible sources are identified, the source point can be constrained to be within the list of possible origins leading to a given destination instruction. Note that in this description, a jump destination refers to the first instruction of a basic block. Additionally, the jump destination instructions that are considered in this case include conditional jumps and even dynamic jumps, except as discussed before instructions such as return from exported function or calls to virtual functions.

[0145] This implies a notion of classes of compatible source and destination pairs, that can be enforced by initialization vectors that creates classes of matching divergences and convergences. Concretely, this means that the initialization vector is an input to the function F in charge of creating control words. The previous minimal example (the nature of jump shall impact the control word) can be seen as a special case of an initialization vector wherein the nature of the jump is implicitly converted as an initialization vector. For example:

- **[0146]** in case of an if instruction, the initialization vector will be equal to 0x00;
- **[0147]** in case of a call instruction, the initialization vector will be equal to 0x01;
- **[0148]** and so on for other kind of jumps which may be encountered in the code.

[0149] According to one aspect of the invention, the value of a control word can be chosen as a function of the destination address that should be reached after a jump instruction. The advantage is that reuse and displacement of control words are impossible.

[0150] In another embodiment, the control words may depend on a key concealed in the processor. The advantage of this embodiment is that it prevents the dynamic forgery of the control words.

[0151] In another embodiment, the control words may depend on a key concealed in a segment of the code that is not readable from outside. The advantage of this embodiment is also that it prevents the dynamic forgery of the control words.

[0152] A combination of these embodiments may also be considered, which means that the F function may have two or three inputs in addition to the mandatory initialization vector.

[0153] In a processing system implementing the invention, the binary .text section **100** must be upgraded in order to embed the control words and/or the initialization vectors. They can be placed after the opcodes in line. This means that

the words must be longer or that a second memory must be used as a « padding » of the first one. Alternatively, specific instructions may be used, said instructions being adapted such that:

[0154] initialization vectors are loaded before any "jump" operation;

[0155] a control word check computed knowing the current initialization vector is requested, along with the expected (a priori) control word value.

[0156] According to the invention, the initial vectors can be computed and allocated using different techniques. For instance, every parent vertex (also called output nodes) corresponding to any licit input node has identical initialization vectors attributed, so that an initialization vector depends only on the destination. This allows a graph traversal with « coloring » of vertices.

[0157] In an alternative embodiment, the parent node can dynamically compute the initialization vector or select the initialization vector in a precomputed table depending on the computed or selected destination. This implementation has the advantage of increasing the number of possible initialization vectors and thus decreasing the possibility of undetected malicious control flow hijacking.

[0158] In addition to classical registers that contain the current opcode and the current address and usually called PC (Program Counter), at least one register is added to the system. This additional register comprises the initialization vector.

[0159] In one embodiment, this initial vector can be set and reset by a specific instructions which is added to the instruction set of the processing system. We recall that the initialization vector is a piece of information that is required for the a priori and a posteriori control words (over a « jump ») to be compatible by association.

[0160] A deterministic function F is used and designed to compute the control words. This function F is implemented in the processing system. F takes as input at least the initialization vector, which for instance encodes the nature of the opcode, that is to say: jump or not, and if jump, several sub-categories can be defined (classes of matching end points).

[0161] As previously stated, an example of a deterministic function of an initialization vector is a SHA cryptographic hash of said initialization vector.

[0162] The function F also aims at recomputing dynamically and just in time (JIT) a control word during the program execution. It is in particular automatically reevaluated if the current instruction is a jump (this can be achieved trivially in the pipeline of a processor, and is indeed most of the time already implemented, let alone to know whether the PC must be incremented [no jump] or loaded from an external value [jump]).

[0163] The result of applying the F function is a control word, that is compared with a Boolean test to the statically and read-only declared a priori control word, which can be found in the binary.

[0164] The function F can advantageously be adapted to take into account an additional input which plays the role of a key. As already mentioned, this key can be unique per device or unique per program. Alternatively, it can be unique per process, a process being an instance of a program. Using a key has the advantage to associate one binary code to one device or program or process, thereby further reducing the possibility of an attacker to fraud the protection. In that case, a second additional register is required to host the key.

[0165] So, in general, for flexibility considerations, the function F might well depends on only a subset of these arguments, depending on the expected level of verification. For example, if the association between the code and the processing system is not a requirement, the « key » input can be ignored.

[0166] The F function can be chosen as a compression function. Indeed, since the output must fit on a limited amount of bits which will generally correspond to the word size used by the system (for example 32 bits) minus the possible opcode length when the control word is introduced by an instruction in the code. This value must be large enough to avoid accidental control words equality, which happens with probability about $2^{-\#bits}$, where #bits is the control word bitwidth.

[0167] The F function can be advantageously chosen so that it will be collision resistant.

[0168] Additionally, the F function can be one-way (at least for the key). That way, it will not possible to recover its arguments by knowing its output. This will advantageously protect the processing system against the recovery of the key.

[0169] Then the F function can be chosen such that it will be fast to compute, ideally in one clock cycle or with the number of clock cycles required to execute one instruction on the processing system, so as not to impede the latency.

[0170] It is also possible to add optional new opcodes. An opcode can be added to initialize/set a new initialization vector, which can be for example equal to zero by default.

[0171] FIG. **7** shows the operation of an opcode designed to set a new initialization vector in the form of a finite state machine with one state **700** and three transitions **701-703**.

[0172] An opcode can be added to request the verification of a control word, in the case this is not done by default at each clock cycle. This implicit behavior does not slow down the execution, because the control word computation and check is done in parallel with the nominal program execution.

[0173] FIG. **8** illustrates the upgraded operation of the processing system in the form of a new finite state machine with two states **800**, **803** and two transitions **801**, **802**. The minimal condition for the control word to be checked is twofold:

[0174] the current opcode is a jump (in the general sense, i.e. a conditional branch, a function call or return), and

[0175] if it is a conditional branch, the jump is effective (opposed to « continue to the next instruction »).

[0176] In case the a priori and a posteriori control words do not match after a jump, the new finite state machine enters an error state **803** which means that the control flow graph has been corrupted. In that case, several actions can be taken, for example halt the program execution and/or active some defensive countermeasures (erase some secrets from memory).

[0177] In another embodiment, the opcodes can be enciphered by a transformation function that depends on the control words. This does not impede the normal execution of the code, as the verification of the control word is necessary before executing the sequel of the code: hence the control word is readily available to decipher in real-time the arriving enciphered opcodes.

[0178] The advantage of this technique is that code injection or reuse elsewhere is rendered very chancy, if not impossible. Furthermore, if for some reason the code happens to leak out of the processing system, then it will be unintelligible, and thus impossible to reverse-engineer, for instance so as to find vulnerabilities in it.

[0179] In one embodiment, encoding the opcodes is done using a block cipher, the key being equal or derived from the control word, for example by using a hash function.

[0180] In one embodiment the block cipher is used in conjunction with cryptographic mode of operations such as ECB (Electronic Code Book), CBC (Cipher Block Chaining), PCBC (Propagating Cipher Block Chaining), CFB (Cipher Feedback), OFB (Output Feedback) and CTR (CounTeR).

[0181] In one embodiment, encoding the opcodes is done by means of a stream cipher, the key being equal or derived from the control word, for example by using a hash function.

[0182] FIG. **9**A provides an example of a simple control flow graph where two nodes are connected with a vertice **910**. FIG. **9**B gives an example of insertion of the a priori control word and its check at the other end of the vertice **910**.

[0183] In order to forbid an attacker from derouting a program by overwriting the return addresses, the control flow graph is made more robust by modifying the ends of the basic blocks, so that:

[0184] on leaving a basic block, some value based on the possible destinations is computed. It is called a set of a priori control words;

[0185] on entering a basic block, the precomputed control word fitting this location is checked against the control word from the incoming vertex. By design, those two values differ only if a new edge, either to an existing node or to a newly created (forged) node, has been created in the control flow graph.

[0186] The transmission of the control word can be done on the stack. The modified final instruction of the basic block **900** and the modified first instruction of the basic bloc **901** are both represented hatched subset of instructions.

[0187] For functions, a similar mechanism can be used, with in addition the push/pop of the a priori control words. This is depicted in listing **2** which shows on an example wherein transitions are verified along function calls /returns in a control flow graph.

Listing 2

Without protection	With protection
Ret (equivalent to: pop %eax jmp %eax+4) call f (equivalent to: push return_address jmp f)	pop IV (deciphered) ret push IV (enciphered) call f

[0188] In one embodiment the initialization vector (IV) is not pushed in the clear on the stack. Instead, it is encrypted by some function that depend on an exposed key (to avoid their retrieval) and on their address (namely % esp, to avoid replay).

[0189] In mirror, the state machine of the hardware must be upgraded, for the verification to be done automatically. This will prevent jumping at an unplausible address, that is to say inside a basic block.

[0190] The described embodiments thus allows to reactively fight cyber-attacks, i.e. malicious modifications of the computer state by the abuse of bugs in the program it runs. For instance, a cyber-attacker might guide the program into unexpected states, that is undocumented by the specification and/or unanticipated by the developer, through various mechanisms.

[0191] One example of such cyber-attacks is to have the program reach a state with corner-case arguments (e.g., negative values when the program semantics would expect only positive values). In this example, the code is too permissive, and the cyber-attacker takes advantage of this weakness. Another example is to have the program fall into a bug uncarefully left by the developer. Such bug can represent an overflow in size when reading into a buffer of characters or an overflow of values (two integers, at least one of which is provided externally, and whose sum, computed by the program, overflows to maximum value for an integer, say 232-1 or 264-1 or 0xffff . . . ff in hexadecimal notation). It should be noted that the invention also applies to many other program state corruption techniques not described herein for the sake of conciseness.

[0192] Depending on the nature of the vulnerability, the attacker can also modify the program memory, including control-flow related information with variable degree of flexibility.

[0193] This kind of threat may be addressed by the use of a specially crafted cryptographic function F, which is configured so that it cannot be forged by an attacker, according to certain embodiments. Indeed, a cyber-attacker is capable of running the program multiple times with different arguments and observe the way it reacts (legal output, crash, or not etc.). This means the cyber-attacker is adaptative, hence the properties of the above described cryptographic function F. It should be noted that in the embodiments where the cryptographic function F is hardcoded (in hardware), less computational effort is required. Further, by implementing the cryptographic function F in hardware, the attacker is deprived of the possibility of manipulating this function F. Similarly, security verifications using the function F may be implemented in hardware in order to obtain a similar advantage: the security checks cannot be tampered, disabled, modified or by-passed. It should be noted that the invention may be also applied for protection against faults other than from a "cyber" origin, such as bugs in the program (e.g., caused by problems in the compiler, which would generate an incorrect control flow graph), or even physical faults induced by a perturbation of the environment (for example, low voltage, electromagnetic injection due to bad "electromagnetic compatibility" shield, overclocking, etc.), whether natural or triggered by an attacker.

[0194] It should be noted that the cryptographic function may be implemented according to different techniques. For example, the cryptographic function F may be implemented using a HMAC (Keyed Hash Message Authentication Code) computation of the initialization vector and a secret key stored in a hardware configuration register. The HMAC input may also comprise the jump class (call, jump, etc). Additionally, the HMAC input may also comprise the destination address.

[0195] Alternatively, the cryptographic function F may be implemented using a block cipher taking as plaintext input the initialization vector and a secret key stored in a hardware configuration register. The block cipher input may comprise the jump class (call, jump, etc), and/or the destination address. Examples of block ciphers comprise with no limitation different types of ciphering algorithms such as AES (Advanced Encryption Standard) and 3DES (DES stands for Data Encryption Standard). In particular, the block cipher may be chosen to be lightweight and fast to compute in hardware. Examples of such block ciphers comprise SMALL PRESENT and SIMON.

[0196] The function F may be also implemented by using a CBC-MAC (Cipher Block Chaining Message Authentication Code) computation of the initialization vector and a secret key stored in a hardware configuration register.

[0197] In another embodiment, the function F may be implemented using a stream cipher taking as plaintext input the initialization vector and a secret key stored in a hardware configuration register. The stream cipher input may also comprise the jump class (call, jump, etc) and/or the destination address. The stream ciphers may comprise any type of algorithm such as TRIVIUM or chained block ciphers such as AES-CBC. The stream cipher may be also chosen to be lightweight and fast to compute in hardware (like stream ciphers comprising TRIVIUM).

[0198] In still another embodiment, the function F may be implemented using a asymmetric cryptography signature of the initialization vector and a secret key stored in a hardware configuration register. The signed data may also comprise the jump class (call, jump, etc) and/or the signed data also comprises the destination address.

[0199] The processing system, methods and configurations as described above and in the drawings are for ease of description only and are not meant to restrict the apparatus or methods to a particular arrangement or process in use.

1. A computer implemented method for controlling dynamically the execution of a code by a processing system, said execution being described by a control flow graph comprising a plurality of basic blocks composed of at least an input node and an output node, a transition in the control flow graph corresponding to a link between an output node of origin belonging to a first basic block and an input node of a second basic block, a plurality of initialization vectors being associated to the output nodes at the time of generating the code, an a priori control word being associated to each input node which is linked to the same output node of origin according the control flow graph, said a priori control word being precomputed at the time of generating the code by applying a predefined deterministic function F to the initialization vector associated to its output node of origin, the following steps being applied once the execution of the output node belonging to a first basic block is terminated and at the time of executing the input node of a second basic block:

providing the a priori control word associated to the input node of the second basic block;

providing the initialization vector associated to the output node of the first basic block;

determining an a posteriori control word by applying to the provided initialization vector the same function F which has been used for generating the a priori control word;

determining if the a priori control word matches with the a posteriori control word, a forbidden transition in respect to the control flow graph being otherwise detected.

2. A method according to claim **1** wherein the code execution is interrupted when an output control word and an input control word belonging to two subsequent basic blocs are not identical.

3. A method according to claim **1** wherein the second basic block is enciphered at the time of generating the code by using its associated a priori control word as a ciphering key, said method comprising the step of deciphering the second basic block by using its associated a posteriori control word as a deciphering key at the time of executing the input node of said second basic block.

4. A processing system for executing a code comprising a processor, said system comprising also:

a memory area configured to store the code to be executed, said code being associated to a control flow graph comprising a plurality of basic blocks composed of at least an input node and an output node, a transition in the control flow graph corresponding to a link between an output node of origin belonging to a first basic block and an input node of a second basic block, said memory area being also configured to store a plurality of initialization vectors associated to the output nodes at the time of generating the code, to store a plurality of a priori control words, an a priori control word being associated to each input node which is linked to the same output node of origin according the control flow graph, said a priori control word being precomputed at the time of generating the code by applying a predefined deterministic function F to the initialization vector associated to its output node of origin;

an hardware implemented module configured to generate a posteriori control words, an a posteriori control word being generated for a given input node by applying to the initialization vector the same function F which has been used for generating the a priori control associated to the same input node;

a module configured for determining if an a posteriori control and an a priori control word which are associated to the same input node are matching, a forbidden transition in respect to the control flow graph being otherwise detected.

5. A processing system according to claim **4** wherein the initialization vectors encode the nature of the jumps implementing allowed transitions in the control flow graph.

6. A processing system according to claim **4** wherein the initialization vectors are memorized in the processing system using a set of dedicated registers.

7. A processing system according to claim **4** wherein function F is adapted to take into account an additional input which plays the role of an activation key.

8. A processing system according to claim **4** wherein the activation key is unique per device.

9. A processing system according to claim **4** wherein the activation key is unique per program.

10. A processing system to claim **4** wherein the basic blocks are enciphered at the time of generating the code by using their associated a priori control words as a ciphering key, said system comprising a module to decipher said basic

blocks at the time of executing their input node by using their associated a posteriori control word as a deciphering key.

11. A computer implemented method for generating an improved version of an initial code intended to be executed on the processing system according to the claim **3**, comprising the steps of:

determining a control flow graph representative of an unaltered execution of the code, said control flow graph comprising a plurality of basic blocks composed of at least an input node and an output node, a transition in the control flow graph corresponding to a link between an output node of origin belonging to a first basic block and an input node of a second basic block in the control flow graph;

generating a plurality of initialization vectors, an initialization vector being associated to each output node at the time of generating the code;

for each input node, determining an a priori control word associated to each input node which is linked to the same output node of origin according the control flow graph, said a priori control word being precomputed at the time of generating the code by applying a predefined deterministic function F to the initialization vector associated to its output node of origin;

modifying the initial code by inserting the a priori control words in line with their corresponding instructions.

12. A method according to claim **11** wherein the control flow diagram is determined through a static analysis of an initial code, said initial code being a source code.

13. A method according to claim **11** wherein the control flow diagram is determined through a static analysis of an initial code, said initial code being an assembly code.

14. A method according to claim **11** wherein the control flow diagram is determined through a static analysis of an initial code, said initial code being a binary code

15. A method according to claim **11** wherein the initialization vectors are attributed randomly.

16. A method according to claim **11** wherein the value of an a priori control word is chosen as a function F of at least a destination address which is the address where the instruction corresponding to an input node following an output node of origin according to the control flow graph is located.

17. A method according to claim **11** wherein the a priori control words are inserted inside the code to be executed.

18. A method according to claim **17** wherein the a priori control words are inserted in line with the instructions corresponding to their associated input node.

19. A method according to claim **11** wherein the initialization vectors are inserted in line with the instructions corresponding to their associated output nodes.

20. A computer program product, stored on a non transitory computer readable medium comprising code for causing a computer to implement the method according to claim **1**.

* * * * *