(54) PROGRAM EXECUTION INTEGRITY VERIFICATION FOR A COMPUTER SYSTEM

(75) Inventor: Xinyuan Wang, Clifton, VA (US)

(73) Assignee: George Mason Intellectual Properties, Inc., Fairfax, VA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 10 days.

(21) Appl. No.: 13/278,814

(22) Filed: Oct. 21, 2011

(65) Prior Publication Data

Related U.S. Application Data
Provisional application No. 61/405,663, filed on Oct. 22, 2010.

(51) Int. Cl.
G06F 11/00 (2006.01)

(52) U.S. Cl.
USPC 726/14; 726/26; 726/22; 713/176

(58) Field of Classification Search
CPC 726/24; 726/26; 726/22; 713/176

See application file for complete search history.

References Cited

U.S. PATENT DOCUMENTS

OTHER PUBLICATIONS
Relevant Sections: Chapter 5-System Calls and Appendix B—Kernel Random Number Generator.

ABSTRACT
A computer system may be employed to verify program execution integrity by receiving a request to launch a program that has been instrumented to include at least one integrity marker, instantiating the program with an integrity marker value, and verifying the execution integrity of the program based on the integrity marker value and information received from the program during execution. A computer system may also be employed for program instrumentation by modifying the program to include at least one instruction for passing an integrity marker value to an operating system kernel during execution of the instruction.

16 Claims, 9 Drawing Sheets
References Cited

OTHER PUBLICATIONS


References Cited

OTHER PUBLICATIONS


* cited by examiner
FIG. 1
FIG. 2
Modify program to include at least one instruction for passing integrity marker value to the operating system kernel

Further modify program to accept a random or pseudorandom number from the operating system kernel to use as the integrity marker value upon execution of the instruction

Recompile modified program to produce instrumented program

FIG. 3
FIG. 4
<table>
<thead>
<tr>
<th></th>
<th>Unused Bits</th>
<th>System Call Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resv</td>
<td>522</td>
<td>System Call No. 512</td>
</tr>
<tr>
<td>Integrity Marker</td>
<td>524</td>
<td></td>
</tr>
<tr>
<td>23 bits</td>
<td></td>
<td>9 bits</td>
</tr>
</tbody>
</table>

FIG. 5
Shared Library

```c
fopen(...M_A)  open(...M_B)
```

```
... mov $0x5, %eax
    int $0x80
...
```
Receive request to launch a program that has been instrumented to include at least one integrity marker

Generate integrity marker value for the program at run-time in response to the request

Instantiate the program with integrity marker value

Verify execution integrity of the program based on integrity marker value and information received from the program during execution

Post-verification activity

FIG. 7
FIG. 8

Instantiate Program

Create process or thread for program

Assign integrity marker value to process or thread

Provide integrity marker to process or thread via its run-time environment
Verify Program Execution Integrity

Check system call information received from program for integrity marker value

Integrity marker value received?

Yes

Compare received integrity marker value to integrity marker value with which the program was instantiated

No

Program verified

Match?

PVA enabled?

No

Yes

Stop program

Monitor malicious activity?

Yes

Allow Compromised Program to Continue

Declare program to be compromised by malware

FIG. 9
This invention was made with government support under grant number CNS-0845042 funded by the National Science Foundation. The government has certain rights in this invention.

CROSS-REFERENCE TO RELATED APPLICATIONS

This application claims the benefit of U.S. Provisional Application No. 61/405,663, filed on Oct. 22, 2010, entitled “Malware Immunization” which is hereby incorporated by reference in its entirety.

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWINGS

FIG. 1 is a system diagram illustrating an example computer system according to one or more embodiments of the present invention.

FIG. 2 is a flow diagram providing a basic overview of an example program execution integrity verification procedure using a malware immunization tool and a malware immunization infrastructure according to one or more embodiments.

FIG. 3 is a flow diagram illustrating an example procedure for a malware immunization tool to instrument a program from an original program according to one or more embodiments.

FIG. 4 illustrates an example modification of an original program at the assembly code level to instrument a program according to one or more embodiments.

FIG. 5 illustrates an example modification of a system call number according to one or more embodiments.

FIG. 6 illustrates the immunization of multiple instances of a given program that share a common library of function calls according to one or more embodiments.

FIG. 7 is a flow diagram illustrating an example procedure for a malware immunization infrastructure to verify program execution integrity according to one or more embodiments.

FIG. 8 illustrates an example procedure for instantiating the program with the integrity marker value according to one or more embodiments.

FIG. 9 illustrates an example procedure for verifying the execution integrity of the program based on the integrity marker value and information received from the program during execution, and for performing various post-verification activity according to one or more embodiments.

DETAILED DESCRIPTION OF EMBODIMENTS

Various embodiments of the present invention are directed to the verification of program execution integrity to protect against malware. Example embodiments are described below with reference to the drawings, wherein like reference numerals are used to refer to like elements throughout. The term “embodiments of the invention” does not require that all embodiments of the invention include the discussed feature or mode of operation. Alternate embodiments may be devised without departing from the scope of the invention, and well-known elements of the invention may not be described in detail or may be omitted so as not to obscure the relevant details.

In addition, the terminology used herein is for the purpose of describing particular embodiments only and is not intended to be limiting of the invention. For example, as used herein, the singular forms “a,” “an,” and “the” are intended to include the plural forms as well, unless the context clearly indicates otherwise. It will be further understood that the terms “comprises,” “comprising,” “includes,” and/or “including,” when used herein, specify the presence of stated features, integers, steps, operations, elements, and/or components, but do not preclude the presence or addition of one or more other features, integers, steps, operations, elements, components, and/or groups thereof.

System Overview

FIG. 1 is a system diagram illustrating an example computer system according to one or more embodiments of the present invention. As shown, the computer system 100 may include user-space programs 110, an operating system kernel 120, and various hardware components, such as a central processing unit 140, memory 150, and input/output (I/O) devices 160. It will be appreciated that the CPU 140 is shown as a single unit for illustration purposes only, and may include one or multiple CPUs. Further, each CPU may be a single-core processor or a multi-core processor that combines two or more independent processing cores. Example I/O devices 160 may include hard disks, keyboards, computer mice, printers, displays, etc.

The operating system kernel 120 generally refers to the component of many computer operating systems that acts as a bridge between software programs and the physical data processing performed at the hardware level. Responsibilities of the operating system kernel 120 may include managing the computer system’s 100 resources as well as the communication between hardware and software. For example, the operating system kernel 120 may manage the user-space programs’ 110 access to the CPU 140, memory 150, and I/O devices 160 through various functions and/or system calls.

The memory 150 may be used to store both program instructions and data, and may be divided by the operating system kernel 120 into both “kernel space” and “user space.” Kernel space includes memory that may be used for running the operating system kernel 120, kernel extensions, and/or some device drivers, but is not usually made accessible to the user-space programs 110. User space includes memory that may be set up by the operating system kernel 120 for operating a user’s context of execution, processes, threads, etc.

The user-space programs 110 may include various types of programs, such as applications 112 and libraries 114. The libraries 114 may include a variety of function calls, for example, which may invoke one or more system calls and may be shared among the applications 112. This sharing may allow multiple processes or threads from the applications 112 to use a common, single copy of a given library function during execution to avoid unnecessary duplication of code stored in the memory 150 at run-time.

As is further illustrated in FIG. 1, the computer system 100 may be exposed to one or more types of malware 50, such as viruses, worms, botnets, rootkits, trojans, spyware, keyloggers, etc., that may attempt to access the computer system 100 through the user-space programs 110. To protect against malicious actions of the malware 50, the user-space programs 110 may be specially instrumented, according to various techniques disclosed herein, by a malware immunization tool 60 such that a malware immunization infrastructure 130 operated by the operating system kernel 120 may verify the execution integrity of a given user-space program 110 before granting it control of the CPU 140, memory 150, I/O devices 160, etc.
For illustration purposes, the malware immunization infrastructure 130 is shown as a component of the operating system kernel 120. However, it will be appreciated that the malware immunization infrastructure 130 may be implemented in other embodiments as an entirely separate entity from the operating system kernel 120. In still other embodiments, the operating system kernel 120 may entirely subsume the operations of the malware immunization infrastructure 130. For example, the operating system kernel 120 may be specially instrumented to perform the operations otherwise attributed herein to the malware immunization infrastructure 130.

Similarly, the malware immunization tool 60 is shown for illustration purposes as being separate from the computer system 100. However, in general, the malware immunization tool 60 may be separate from or integral to the computer system 100, and may be operated independently or at the direction of the operating system kernel 120, as shown by the dashed line in FIG. 1. In some embodiments, the malware immunization tool 60 may be located remotely from the computer system 100 and be implemented as, or operated by, a separate computer system (not shown). In some embodiments, the malware immunization tool 60 may be implemented as, or otherwise include, a user level utility, such as a compiler.

FIG. 2 is a flow diagram providing a basic overview of an example program execution integrity verification procedure using the malware immunization tool 60 and the malware immunization infrastructure 130 according to one or more embodiments. In the illustrated example, the malware immunization tool 60 produces an instrumented program 220 from an original program 210 that is modified, as described herein, to protect it against various control flow hijacking attacks that may be mounted by the malware 50, such as buffer overflow attacks, return-to-libc attacks, return-oriented exploits, etc., in an attempt to gain access to the CPU 140, memory 150, I/O devices 160, etc. The instrumented program 220 may then be stored among the user-space programs 110 for operation on the computer system 100.

As will be described below in more detail, the malware immunization infrastructure 130 may subsequently exchange one or more integrity marker values 230, 240 with the instrumented program 220 during execution to verify its integrity before granting it control of the CPU 140, memory 150, I/O devices 160, etc.

Program Instrumentation

FIG. 3 is a flow diagram illustrating an example procedure 300 for the malware immunization tool 60 to instrument a program from an original program according to one or more embodiments. In this example, the malware immunization tool 60 modifies an original program (e.g., the original program 210) to include at least one instruction for passing an integrity marker value to the operating system kernel 120 during execution of the instruction (block 310) for run-time verification of program execution integrity by the malware immunization infrastructure 130. Modifying the program may include modifying at least one existing instruction of the program or adding at least one new instruction to the program. The modified or added instruction may be, for example, made part of a system call requesting access to one or more resources of the computer system 100, such as control of the CPU 140, memory 150, I/O devices 160, etc.

The malware immunization tool 60 may further modify the original program to accept a given value (e.g., a random or pseudorandom number) from the operating system kernel 120 to use as the integrity marker value (block 320). In some embodiments, these modifications may be performed by a binary rewriting of the original program at the assembly code level. In other embodiments, these modifications may be performed at the source code level, and may require the program to be recompiled to produce an instrumented program (e.g., the instrumented program 220) (block 330). It will be appreciated that any suitable mechanism for modifying the original program may be employed.

FIG. 4 illustrates an example modification of an original program at the assembly code level to instrument a program according to one or more embodiments. As shown, the original program (e.g., the original program 210) includes original assembly code 420 that corresponds to original source code 410. The original assembly code 420 includes various instructions that may correspond to various functions in the original source code 410. In the illustrated example, the original source code 410 includes an example function ("funcA") nested inside a parent function ("funcB") and accepting one or more parameters.

The example function funcA may in some embodiments correspond to a system call, for example, such as a "write" function accepting a given value (e.g., '100') desired to be written to the memory 150 at a desired location identified by a location pointer ("ptr"). Other example system calls may include 'open' functions, 'read' functions, 'close' functions, 'wait' functions, 'exec' functions, 'fork' functions, 'exit' functions, functions, etc. When funcA is called, various supporting instructions in the original assembly code 420 (e.g., 'push' and/or 'add' instructions) may be executed for pushing corresponding values to the top of the program's stack (identified by the pointer 'esp') and otherwise executing the function.

The original assembly code 420 is instrumented by the malware immunization tool 60 to produce immunized assembly code 430. As shown, the immunized assembly code 430 may include one or more additional or modified instructions (e.g., 'lea', 'push', and/or 'add' instructions) for locating an integrity marker value ('eax') provided by the operating system kernel 120 and passing it back to the operating system kernel 120 when funcA is called. In this way, the integrity marker value may be passed to the operating system kernel 120 as if it was an additional parameter of funcA, for example, as shown in the corresponding immunized source code 440.

It will be appreciated, however, that modifications to the actual source code are not required. In some designs, the original source code 410 may not be accessible to the malware immunization tool 60. Regardless, instrumentation of the original program 210 may be performed at the binary level without access to the original source code 410. The instrumented program 220 may therefore operate as if its source code has been modified to include the integrity marker value as shown in the immunized source code 440, even if access to the original source code 410 is not available or otherwise feasible.

The above-described instrumentation may accordingly be made effectively transparent to the user. For example, in the C programming language, the integrity marker value may be implicitly passed as the rightmost parameter of the example function funcA, in addition to any other existing parameters. Because the default C language function calling convention passes parameters from right to left, which inherently supports a variable number of parameters, passing the integrity marker value as an extra implicit rightmost parameter to C language function calls does not change the original semantics of the function.
However, in some embodiments, it may be desirable to provide an additional indicator to the operating system kernel 120 to specifically identify the location of the integrity marker value. For example, for functions that natively accept and pass a variable number of parameters (e.g., the 'printf' function), a canary may be used as an indicator of the location of the integrity marker value among the original parameters. The canary word may be randomly chosen by the malware instrumentation tool 60, for example, and different programs may be instrumented with different canary words.

It will of course be appreciated that the example code illustrated in FIG. 4 is presented in the C programming language by way of example only, and that other programming languages may be used in other embodiments.

Further, it will also be appreciated that the specific mechanism in the example of FIG. 4 for passing the integrity marker value to the operating system kernel 120 as an additional parameter is also provided merely as an example, and that any mechanism for passing the integrity marker value to the operating system kernel 120 may be used. For example, the system call number itself may be modified to include the integrity marker value in some embodiments.

FIG. 5 illustrates an example modification of a system call number according to one or more embodiments. Although many operating systems provide hundreds of system calls, the relatively large 32-bit or even 64-bit number schemes used by modern system architectures provide a number of unused bits in the system call number itself that may be used to pass the integrity marker value to the operating system kernel 120. For example, the Linux operating systems typically have just over 300 different system calls. Thus, in a 32-bit system, for example, a 9-bit system call number field would accommodate 512 unique system calls. This is more than enough to cover the 300 system calls, and still leaves space for a 23-bit integrity marker value with more than 8 million possible values.

In the illustrated example of FIG. 5, a modified system call number 500 for an example 32-bit operating system includes a 9-bit wide system call number field 510 and a 23-bit wide collection of previously unused bits 520. The previously unused bits 520 may be used to carry a new integrity marker field 524. The previously unused bits 520 may also retain one or more reserved bits 522 as well, if the desired length of the integrity marker field 524 is selected to be shorter than the total number of unused bits 520.

Accordingly, the integrity marker field 524 may be passed to the operating system kernel 120 as part of the system call number 500 when the call is invoked, as discussed above, and may be configured to carry the particular integrity marker value assigned to the invoking program. It will be appreciated that the number and arrangement of bits shown in FIG. 5 is for illustration purposes only, and may take any other form acceptable to the particular operating system or computer architecture being employed.

It will also be appreciated that in some embodiments, programs may not invoke system calls directly. Instead, libraries of functions may be used, such as an implementation of the C library (libc), to avoid loading redundant copies of the same library function code into memory at run-time. For example, with reference back to FIG. 1, some of the applications 112 may invoke certain functions, including system calls, through function calls provided by one or more of the libraries 114. Although convenient and common, this design may allow malware to access operating system services (e.g., open a shell) under certain circumstances without actually injecting any malicious code. This is sometimes referred to as a "return-to-libc" attack.

FIG. 6 illustrates the immunization of multiple instances of a given program that share a common library of function calls according to one or more embodiments. In this example, a first instance 610 ("Process A") of the instrumented program 220 and a second instance 620 ("Process B") of the instrumented program 220 share a common library 630 of function calls. Among other functions, the first instance 610 of the instrumented program 220 may execute a file open function ('open') and the second instance 620 of the instrumented program 220 may execute a separate open function ('open'). The shared library 630 may include shared library functions open() and open() such as those for system call number '5' in Linux.

As shown, each instance 610, 620 of the instrumented program 220 may be instantiated with its own integrity marker value. In the illustrated example, the first instance 610 of the instrumented program 220 is instantiated with a corresponding integrity marker value 'Marker_A' while the second instance 620 of the instrumented program 220 is instantiated with a corresponding integrity marker value 'Marker_B'. Upon execution of their respective functions, each instance 610, 620 of the instrumented program 220 may insert its corresponding integrity marker value into its called function. This allows the malware immunization infrastructure 130 to reliably determine if the initiating entity of the function call is a valid program or is instead malware, even if the function call is invoked via shared library functions.

Program Verification

Once a program has been instrumented according to one or more of the various techniques described herein, its execution may be subsequently verified by the malware immunization infrastructure 130 as being valid or trustworthy, rather than compromised by malware.

FIG. 7 is a flow diagram illustrating an example procedure 700 for the malware immunization infrastructure 130 to verify program execution integrity according to one or more embodiments. Process 700 begins at the point where the operating system kernel 120 receives a request to launch a program that has been instrumented to include at least one integrity marker, such as the instrumented program 220 discussed above (block 710).

In some embodiments, the operating system kernel 120 may specifically generate an integrity marker value for the program at run-time in response to the request (block 720). For example, the operating system kernel 120 may generate the integrity marker value by generating a random or pseudorandom number to uniquely identify each instance (e.g., process or thread) of the program. Any known technique for generating the random or pseudorandom number may be used, such as a Naur-Reingold pseudorandom function, a Park-Miller random number generator, linear feedback shift registers, or others known in the art. However, in other embodiments, a static value may be used for integrity marker value. In either case, the integrity marker value may be stored (e.g., at a given location in the memory 150) for future reference by the malware immunization infrastructure 130.

The operating system kernel 120 may instantiate the program with the integrity marker value (block 730), and verify the execution integrity of the program using the malware immunization infrastructure 130 based on the integrity marker value and any information received from the program during execution (block 740). Based on whether the program's integrity is verified or not, and whether subsequent processing is enabled, the operating system kernel 120 may perform various post-verification activity (block 750), such
as “malware forensics” operations. Such malware forensics capabilities may allow the malware immunization infrastructure 130 to accurately locate and/or monitor certain malicious activity of the malware 50.

FIG. 8 illustrates an example procedure for instantiating the program with the integrity marker value according to one or more embodiments. In this example, instantiating the program (block 730) may include creating a process or thread for the program (block 810), assigning the integrity marker value to the process or thread (820), and providing the integrity marker value to the process or thread, such as by placing it into the process’ run-time environment (830). It will be appreciated, however, that any suitable mechanism for launching the program and identifying the appropriate integrity marker value for the program to use may be employed. In some embodiments, the appropriate integrity marker value for the program to use may be pre-programmed such that the program and the malware immunization infrastructure 130 know a priori which value will be used for a given process or thread. FIG. 9 illustrates an example procedure for verifying the execution integrity of the program based on the integrity marker value and information received from the program during execution (block 740), and for performing various post-verification activity (block 750) according to one or more embodiments. In this example, the operating system kernel 120 uses the malware immunization infrastructure 130 to check the information received from the program for an integrity marker value that matches the integrity marker value with which the program was instantiated (block 910). Checking the information may include extracting the received information from a system call or other function executed by the program, for example.

If the information received from the program includes an integrity marker value (‘yes’ at decision 920), the malware immunization infrastructure 130 may compare the received integrity marker value to the integrity marker value with which the program was instantiated (block 930). If the information received from the program includes an integrity marker value that matches the integrity marker value with which the program was instantiated (‘yes’ at decision 940), the malware immunization infrastructure 130 may declare that the program is verified (block 950). It will be appreciated that a “match” may not be identical to the original or generated value. For example, the original integrity marker value with which the program was instantiated may be further processed in an agreed upon manner by both the operating system kernel 120 and the instrumented program, as a further security measure.

Otherwise, if the information received from the program does not include an integrity marker value (‘no’ at decision 920), or it includes an integrity marker value that does not match the integrity marker value with which the program was instantiated (‘no’ at decision 940), the malware immunization infrastructure 130 may refrain from declaring that the program is verified and instead determine whether any post-verification activity (PVA) is warranted (decision 960).

In some situations, it may be desirable for the malware immunization infrastructure 130 to perform one or more post-verification operations, while in other situations it may be desirable for the malware immunization infrastructure 130 to disable such post-verification operations, at least for a period of time. For example, it has been found that a newly instantiated process or thread may in some situations not be able to immediately locate the appropriate integrity marker value placed into the program’s run-time environment and use it for first few system calls. Thus, in some embodiments, determining whether any post-verification activity is warranted (decision 960) may include checking a flag maintained by the malware immunization infrastructure 130 (e.g., at a given memory location within the memory 150) to indicate whether post-verification activity is yet appropriate. The flag may be set, for example, based on whether the information received from the program in the past has included, at least one integrity marker value that matched the integrity marker value with which the program was instantiated (i.e., at least one ‘yes’ at decision 940). The reception of at least one matching integrity marker value may act as a trigger to enable the flag, thereby providing a grace period for the program to locate the appropriate integrity marker value and prevent the malware immunization infrastructure 130 from declaring the program to be compromised by malware until it is clear that the program has located the appropriate integrity marker value, such that false positives may be reduced or eliminated.

If post-verification activity is not enabled (‘no’ at decision 960), no further action is taken and the malware immunization infrastructure 130 returns to checking subsequent system call information (block 910). If post-verification activity is in fact enabled (‘yes’ at decision 960), and the current information received from the program does not include an integrity marker value (‘no’ at decision 920) or includes an integrity marker value that does not match the integrity marker value with which the program was instantiated (‘no’ at decision 940), the malware immunization infrastructure 130 may declare the program to be compromised by malware (block 1010).

The malware immunization infrastructure 130 may decide whether to continue the compromised program and monitor it for any malicious activity (decision 1020). This may be referred to as “malware forensics,” and may be useful in determining how the identified malware operates. If the malware immunization infrastructure 130 does not desire to monitor the compromised program for subsequent malicious activity (‘no’ at decision 1020), the operating system kernel 120 may simply stop the compromised program (block 1030).

Otherwise, if the malware immunization infrastructure 130 does desire to monitor the compromised program for subsequent malicious activity (‘yes’ at decision 1020), the operating system kernel 120 may allow the compromised program to continue as long as desired (block 1040). In some embodiments, a user level utility may be employed to allow a user of the computer system 100 to inform the malware immunization infrastructure 130 about which processes or threads it should check for the integrity marker value, and what steps should be taken once a compromised program is detected.

Many of the elements described in the disclosed embodiments may be implemented as modules. A module is defined here as an isolatable element that performs a defined function and has a defined interface to other elements. The modules described in this disclosure may be implemented in hardware, a combination of hardware and software, firmware, wetware (i.e., hardware with a biological element) or a combination thereof, all of which are behaviorally equivalent. For example, modules may be implemented using computer hardware in combination with software routine(s) written in a computer language (such as C, C++, Fortran, Java, Basic, Matlab or the like) or a modeling/simulation program such as Simulink, Stateflow, GNU Octave, or LabVIEW MathScript. The software routine(s) may be initially stored in a computer-readable medium, and loaded and executed by a processor. Additionally, it may be possible to implement modules using physical hardware that incorporates discrete or programmable analog, digital and/or quantum hardware. Examples of programmable hardware include: computers, microcontrollers, microprocessors, application-specific integrated circuits.
The computer system generating the integrity marker value for the program at run-time in response to the request.

The method of claim 2, wherein the generating of the integrity marker value comprises generating a random or pseudorandom number to uniquely identify an instance of the program execution.

The method of claim 1, wherein the instantiating of the program comprises:

- the computer system creating a process or thread for the program;
- the computer system assigning the unique and dynamically generated integrity marker value to the process or thread;
- and the computer system providing the integrity marker value to the process or thread in a run-time environment.

The method of claim 1, further comprising the computer system extracting the received information from a system call executed by the program.

The method of claim 1, wherein the verifying comprises checking the information received from the program execution for an integrity marker value that matches the integrity marker value with which the program was instantiated.

The method of claim 1, further comprising the computer system monitoring subsequent malicious activity of the program when the verifying fails.

The method of claim 1, wherein the program is declared to be compromised by malware only after at least one integrity marker value is received from the program that does not match the unique and dynamically generated integrity marker value with which the program was instantiated.

The method of claim 1, wherein the verifying comprises checking the information received from the program execution for an integrity marker value that matches a further processed value of the integrity marker value with which the program was instantiated.

The method of claim 1, wherein using the kernel to instantiate the program with an integrity marker value comprises assigning a new integrity marker value each time the program is instantiated.

The method of claim 1, wherein the program comprises a system call.

An apparatus for verifying program execution integrity, the apparatus comprising:

- at least one processor configured to:
  - execute an operating system kernel,
  - receive a request to launch a program that has been instrumented to include at least one integrity marker that is independent of other portions of the program;
  - the computer system using the kernel, and not a compiler outside of the kernel, to instantiate the program with a unique and dynamically generated integrity marker value;
  - and the computer system verifying the execution integrity of the program based on the unique and dynamically generated integrity marker value and information received from the program during execution.

The apparatus of claim 12, wherein the instantiating of the program comprises:

- creating a process or thread for the program;
- assigning the unique and dynamically generated integrity marker value to the process or thread;
- providing the integrity marker value to the process or thread via a run-time environment.

The apparatus of claim 12, wherein the verifying comprises checking the information received from the program execution for an integrity marker value that matches a further processed value of the integrity marker value with which the program was instantiated.
15. The apparatus of claim 12, wherein the processor is configured to use the kernel to instantiate the program with an integrity marker value by assigning a new integrity marker value each time the program is instantiated.

16. The apparatus of claim 12, wherein the program comprises a system call.