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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
5 June 2008 (05.06.2008)

PCT

(10) International Publication Number
WO 2008/067329 A1

(51) International Patent Classification:
G06F 9/45 (2006.01) *G06F 12/02* (2006.01)

(21) International Application Number:
PCT/US2007/085664

(22) International Filing Date:
27 November 2007 (27.11.2007)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
11/564,249 28 November 2006 (28.11.2006) US

(71) Applicant (*for all designated States except US*): **MICROSOFT CORPORATION** [US/US]; One Microsoft Way, Redmond, Washington 98052-6399 (US).

(72) Inventors: **WRIGHTON, David Charles**; One Microsoft Way, Redmond, Washington 98052-6399 (US). **UNOKI, Robert Sadao**; One Microsoft Way, Redmond, Washington 98052-6399 (US).

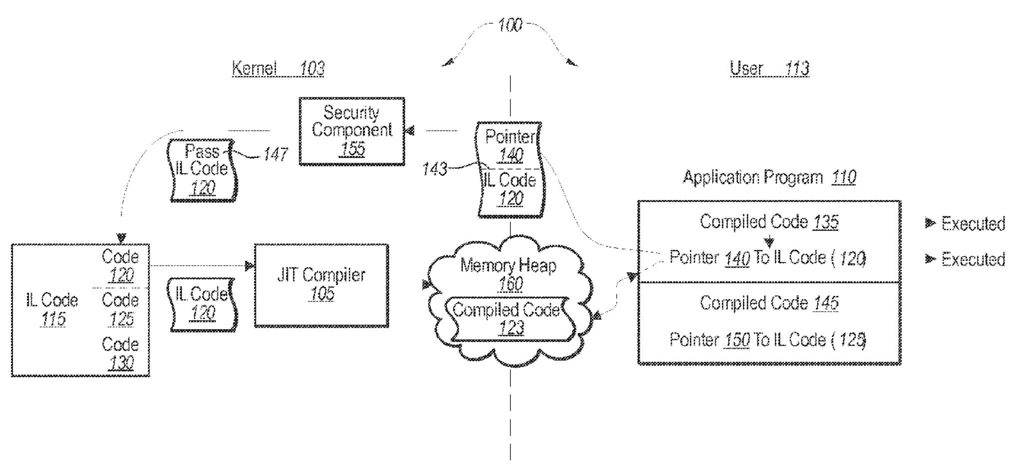
(81) Designated States (*unless otherwise indicated, for every kind of national protection available*): AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN,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

(84) Designated States (*unless otherwise indicated, for every kind of regional protection available*): ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, MT, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:
— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*
— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

Published:
— *with international search report*

(54) Title: COMPILING EXECUTABLE CODE INTO A LESS-TRUSTED ADDRESS SPACE



(57) Abstract: Unsafe application programs that implement managed code can be executed in a secure fashion. In particular, an operating system can be configured to execute an application program in user mode, but handle managed code compilation through a type-safe JIT compiler operating in kernel mode. The operating system can also designate a single memory location to be accessed through multiple address spaces with different permission sets. An application program operating in user mode can be executed in the read/execute address space, while the JIT compiler operates in a read/write address space. When encountering one or more pointers to intermediate language code, the application runtime can send one or more compilation requests to a kernel mode security component, which validates the requests. If validated, the JIT compiler will compile the requested intermediate language code, and the application program can access the compiled code from a shared memory heap.

WO 2008/067329 A1

COMPILING EXECUTABLE CODE INTO A LESS-TRUSTED ADDRESS SPACE

BACKGROUND

[0001] As computerized systems have increased in popularity, so have the various application programs used on the computerized systems. In particular, there are now a wide range of applications programs configured for any number of purposes, whether to function as complex operating systems, databases, and so forth, or as a simple calculator. In many cases, software developers will write new application programs with a particular operating system in mind, using any number of appropriate languages. Once the software is complete, the developer will compile the application into machine-executable code, which can then be installed on a computer system with the appropriate operating system.

[0002] One will appreciate, therefore, that there are a number of considerations that often must considered by developers of operating systems as well as of the individual application programs. Many of these interests may even be competing. For example, many application program developers may have interests related to quick and fast operation, while many operating system developers may have interests related to security and stability. In some cases, the security and stability requirements can cause some application programs to have slower execution and/or lower-performance.

[0003] For example, the operating system may be configured to have application programs run in a less-trusted "user" level, but have other system components run in a trusted "kernel" level. As a result, an application program running in a user level might only be able to perform certain types of functions by requesting the given function through an intermediary, trusted component. The intermediate component can then validate the request and then pass the request for the function to a kernel level component, which can then execute the request.

[0004] Other ways of managing security are to limit the various applications and components to specific readable, writable, and/or executable permission spaces. For example, an operating system might allow certain application programs to run only in a read/execute address space. This might allow the application programs to

execute any existing instructions, but would prohibit the application from performing any write operations. By contrast, the operating system might allow other sensitive system components to operate only in a read/write address space. This might allow the sensitive components to make new writes, but would prohibit those writes from being executed.

[0005] In still other cases, an operating system might allow only certain types of application programs conforming to certain code standards to run in a space that is readable, writable, and executable. For example, the operating system might only allow "type-safe" applications to run in a read/write/execute address space. One example of a type-safety rule might be to require an integer value to be added only to other integer values, rather than to floating point values. A type-safe compiler could then be used to compile only that executable program code that is type-safe, and thus trusted by the operating system.

[0006] Unfortunately, some recent trends in application program developing complicates various aspects of the above-mentioned security management approaches. For example, a wide range of application developers are now creating video game application programs using "managed code." In general, managed code includes executable program code, as well as intermediate language code that can be compiled on an as-needed basis. For example, a developer of an application program might include one or more references (in the compiled, executable code) to intermediate code. Thus, when the executable code comes to a point where it needs to use a function that is available only in intermediate language code, a JIT (just-in-time) compiler is used to compile certain intermediate language code into executable instructions.

[0007] One can appreciate, therefore, that operating systems will sometimes limit the use of managed code to type-safe applications. In particular, since the JIT compiler will need to write, and since the application will need to execute, and further since the application program will need to access the compiled code written by the JIT compiler, the JIT compiler and the executing application program will typically operate in the same address space, which is readable, writable, and executable. Thus, if the intermediate language code were not type-safe (or

conforming to some other program code restrictions), a malicious party could trick the JIT compiler into generating harmful instructions that are executed.

[0008] Unfortunately, program code restrictions such as type-safety are often believed to conflict with speed and performance considerations. This can be particularly problematic for video game applications, where speed and performance considerations are placed at a premium. In some cases, therefore, the developers of video game applications may find it better or more efficient to ignore specific code specifications, such as type-safety.

BRIEF SUMMARY

[0009] Implementations of the present invention provide systems, methods, and computer program products configured to allow for the use of managed code in an operating system, where the managed code may not necessarily conform to any particular code standard. In one implementation, for example, an operating system provides access to a memory location in two different address spaces, and sets the permissions in the address spaces, such that the memory location is accessible with different permissions from the two different address spaces. In one implementation, a JIT compiler operating in one address space passes compiled code into a shared memory heap. Executable program code, in turn, accesses the compiled code from the memory heap, and executes it in the other memory address space.

[0010] For example, a method of executing managed code so that untrusted program code can be compiled and executed in a manner that does not threaten or otherwise compromise system security can involve executing an application program in a first address space of a memory location. The method can also involve receiving one or more requests from the application program to compile one or more sets of intermediate language instructions. In addition, the method can involve compiling the one or more sets of intermediate language instructions into newly compiled code using a JIT compiler running in a second address space of the memory location. Furthermore, the method can involve passing the newly compiled code to a shared memory heap. The application program can then

retrieve the newly compiled code from the shared memory heap into the first address space.

[0011] Similarly, another method of generating computer executable program code in a manner that uses JIT compilation while avoiding security violations can involve receiving application program code that includes executable code and code to be compiled. The method can also involve executing the executable code in a lower-privilege mode and in a first address space. In addition, the method can involve identifying one or more pointers in the executable code for at least some code to be compiled. Furthermore, the method can involve switching to a higher-privilege mode. Still further, the method can involve compiling the at least some code in a different address space using a compiler operating in the higher-privilege mode.

[0012] This Summary is provided to introduce a selection of concepts in a simplified form that are further described below in the Detailed Description. This Summary is not intended to identify key features or essential features of the claimed subject matter, nor is it intended to be used as an aid in determining the scope of the claimed subject matter.

[0013] Additional features and advantages of the invention will be set forth in the description which follows, and in part will be obvious from the description, or may be learned by the practice of the invention. The features and advantages of the invention may be realized and obtained by means of the instruments and combinations particularly pointed out in the appended claims. These and other features of the present invention will become more fully apparent from the following description and appended claims, or may be learned by the practice of the invention as set forth hereinafter.

BRIEF DESCRIPTION OF THE DRAWINGS

[0014] In order to describe the manner in which the above-recited and other advantages and features of the invention can be obtained, a more particular description of the invention briefly described above will be rendered by reference to specific embodiments thereof which are illustrated in the appended drawings. Understanding that these drawings depict only typical embodiments of the

invention and are not therefore to be considered to be limiting of its scope, the invention will be described and explained with additional specificity and detail through the use of the accompanying drawings in which:

[0015] Figure 1A illustrates an overview schematic diagram of an implementation in accordance with the present invention in which an application program running in a less trusted security mode invokes managed code, which is compiled by a JIT compiler in a trusted security mode;

[0016] Figure 1B illustrates a schematic diagram in which a memory location managed by the operating system is accessible by components in two different address spaces, which have different permissions for accessing the memory location;

[0017] Figure 2 illustrates a flowchart of a sequence of acts in accordance with an implementation of the present invention in which a JIT compiler receives and handles one or more requests for intermediate language instructions; and

[0018] Figure 3 illustrates a flowchart of an overview sequence of acts in which an operating system receives an application program that includes one or more references to managed code, and executes the application program in accordance with one or more security mechanisms.

DETAILED DESCRIPTION

[0019] Implementations of the present invention extend to systems, methods, and computer program products configured to allow for the use of managed code in an operating system, where the managed code may not necessarily conform to any particular code standard. In one implementation, for example, an operating system provides access to a memory location in two different address spaces, and sets the permissions in the address spaces, such that the memory location is accessible with different permissions from the two different address spaces. In one implementation, a JIT compiler operating in one address space passes compiled code into a shared memory heap. Executable program code, in turn, accesses the compiled code from the memory heap, and executes it in the other memory address space.

[0020] As will be understood more fully herein, implementations of the present invention can provide a secure system without necessarily needing to verify that the generated code does not violate the security constraints of the system. This can be done at least partly by "sandboxing" the compiled code, as well as any other code that is being executed. In particular, implementations of the present invention can define a "sandbox," which is essentially a predefined set of boundaries in which any type of code can be executed. Specifically, the sandbox boundaries described herein will result in malicious request(s) made by the executing code being either denied by the operating system (as coming from a user mode component), or limited to actions or functions only within the predefined permissions (e.g., denying a write to a read/execute address space).

[0021] As a result, code that is compiled by a JIT compiler (e.g., 105), or even the application program (e.g., 110) ultimately invoking the JIT compiler, can be executed within the sandbox without necessarily being "type-safe," or conforming to some other security consideration. One will appreciate that this can free a given developer to write application program code in a manner that is potentially less constrained, and potentially faster and performance driven than previously possible.

[0022] In addition to ensuring that code is executed properly, implementations of the present invention also provide mechanisms that ensure that the JIT compiler, itself, cannot be "hijacked," such as when receiving and compiling intermediate language code. In particular, implementations of the present invention include a JIT compiler that is configured for type-safe execution, rather than necessarily checking incoming code for type-safety or compiling only type-safe code. As such, the JIT compiler in accordance with implementations of the present invention can be protected against requests that would cause the JIT compiler itself to violate safety definitions (e.g., type-safe definitions).

[0023] In one implementation, for example, the JIT compiler can be configured with type-safety definitions that restrict the JIT compiler from reaching outside of its own data structures, or the data structures that are defined as part of the system 100 runtime. For example, the JIT compiler can be configured to perform a series of checks to ensure that only valid casts are performed whenever performing casts

from one type to another. Similarly, the JIT compiler can be configured so that, whenever asked to read out of arrays, the JIT compiler performs one or more boundary checks to ensure that the JIT compiler is within the bounds of the array. With respect to use within the C programming language, for example, the JIT compiler can also be configured to ensure that whenever using a "union," the JIT compiler reads or writes to the proper part of the union. Furthermore, the JIT compiler can be configured to ensure the JIT compiler never overflows or underflows while reading or writing the type-stack (the type-stack within the JIT compiler).

[0024] In general, the JIT compiler's type-stack is an internal data structure that is generally important to maintain correctness, etc. For example, intermediate language code is typically a stack-based system in which the JIT compiler operates on objects in a stack in order, and places results back into the stack in order. The JIT compiler in accordance with implementations of the present invention is thus configured to simulate a stack to ensure that the JIT compiler is operating as expected. For example, the JIT compiler can perform stack simulation while compiling intermediate language code. If the simulated stack deviates significantly from what the JIT compiler is being fed, the JIT compiler can quit compilation or generate an error. This helps the JIT compiler ensure that it is operating within prescribed boundaries, and thus protected from violating one or more security rules.

[0025] Figure 1A illustrates an overview schematic diagram of a computerized system 100 (e.g., a video game operating system) in which an application program (i.e., 110) is being executed. In one implementation, application program 110 is a video game application, though one will appreciate that application program 110 can be any type of executable program code. In any event, Figure 1A also shows that application program 110 comprises one or more sets of executable instructions, such as compiled code 135, which includes a pointer 140 to intermediate language ("IL") code 120. Similarly, Figure 1A shows that application program 110 comprises compiled code 145, which includes pointer 150 to intermediate language code 125. Intermediate language code 125, in turn, comprises several different

components or modules, such as code 120, 125 and 130, which need further compilation before they can be executed.

[0026] There are any number of different ways that application program 110 will or can be executed in computer system 100. For example, a user might load a storage device onto another device on which the system 100 is installed. The storage device may include binary executable code for application program 110, as well as managed code in the form of intermediate language code 115. Both the executable code and intermediate language code of application program 110 could then be loaded into computerized system 100. In other cases, a user, such as a developer, may upload the application program 110, including intermediate language code 115 through a network connection. In such a case, the user might be executing application program 110 for testing newly developed application programs (e.g., 110).

[0027] In any event, Figure 1A also illustrates that application program 110 is being executed in a lower-privilege mode (e.g., "user" mode), while JIT compiler 105 is operating in a higher-privilege mode (e.g., "kernel" mode). For example, Figure 1A shows that application program 110 is operating in user mode 113 with user privileges, while JIT compiler 105 is operating in kernel mode 103 with corresponding kernel privileges. In addition, Figure 1A shows that intermediate language code 115 is accessed by one or more components with kernel 103 level privileges. Conversely, and as will be understood more fully herein, executable code will only be executed by components operating with user 113 levels of privileges.

[0028] Accordingly, as the runtime for application program 110 executes each of the compiled instructions 135, 145 in user 113 mode, the runtime will come across any of one or more pointers to intermediate language code. For example, during execution, the runtime for application program 110 comes across pointer 140 to intermediate language code 120. Since pointer 140 references code that can only be accessed in kernel 103 mode, the runtime will break out of user mode and system 100 will switch to kernel 103 mode.

[0029] The request 143 will then be handled by security component 155, which operates in kernel 103 mode. In general, security component 155 can comprise any number or type of components or modules configured to receive a user mode 113 component request (e.g., 143), and then validate whether the request is appropriate. This is done since user mode 113 is untrusted, and since application program 110 may or may not represent (or otherwise include) dangerous or malicious code.

[0030] Thus, to ensure that requests from user mode 113 execution will not damage system 100, security component 155 can perform any number or type of validation functions. For example, security component 155 can review message 143 for any number of handles, tokens, or the like. Furthermore, security component 155 can review request 143 for application instructions that could be used to compromise system 100, such as specific memory address requests, or requests that could result in a buffer overrun, etc. Upon validating request 143, security component 155 can initiate JIT compiler 105 in kernel mode.

[0031] Once operating in kernel mode, JIT compiler can then be fed the requested code (i.e., 120) and begin compilation. For example, Figure 1A shows that security component 155 executes one or more requests 147 that cause JIT compiler 105 to receive and compile intermediate language code 120. After compiling code 120 into executable binary instructions (i.e., compiled code 123), Figure 1A also shows that JIT compiler 105 can then pass code 123 into memory heap 160.

[0032] As will be understood more fully with respect to Figure 1B, memory heap 160 straddles the boundary between user mode 113 and kernel mode 103 operations. In effect, memory heap 160 acts as a cross-permission / cross-boundary store that is accessible by components operating in kernel mode 103 and/or in user mode 113. Once compilation is completed, system 100 can switch back to user mode and continue execution of the application program 110. In particular, application 110 – operating in user mode – can pull the compiled code 123 as soon as it is available, and begin executing it in user mode 113. One will appreciate, therefore, that memory heap 160 can be used to help maintain the security boundaries between the two security layers by allowing JIT compiler 105 and user

113 to function independently, in different privilege modes, without direct communication.

[0033] Figure 1B illustrates additional details on how the security boundary between the JIT compiler 105 and application program 110 can be accomplished or otherwise maintained. In particular, Figure 1B illustrates an implementation in which JIT compiler 105 and application program 110 operate with respect to a particular same memory location, albeit with different permission sets. In particular, Figure 1B illustrates an implementation in which the same memory location can be accessed by components in one address space with one set of permissions in one address space, and accessed by different components in another address space with a different set of permissions. For example, Figure 1B shows that memory location 160 is available in an address space 170 with read/write permissions, and an address space 165 with read/execute permission.

[0034] In general, one or more kernel layer 103 components of operating system 100 will maintain a memory page table 180 for any given address location and corresponding address spaces. For example, Figure 1B shows that memory page table 180 is maintained in kernel 103 layer (i.e., one or more kernel mode components) of system 100. One reason this is maintained by a kernel 103 mode component is to ensure that an untrusted application program (i.e., operating in user mode) cannot access or otherwise improperly manipulate the page table.

[0035] In any event, Figure 1B shows that page table 180 correlates memory locations 160 and 165 with address spaces 170, 175, 190, and 195. For example, memory location 160 is the shared memory heap, while memory location 165 is a location in which application program 110 is loaded for execution. In addition, page table 180 maps the access permissions of memory location 160 and 165, such that address spaces 170 and 190 have "read/write" access to locations 160 or 165, respectively. Similarly, page table 180 maps the permissions of memory location 160 and 165 for address spaces 175 or 195 as "read/execute," respectively. Accordingly, when security component 155 (Figure 1A) receives a request (e.g., 143) from a user mode 113 component, security component 155 can correlate the

address spaces of the component originating the request (e.g., 143) with the address space for JIT compiler output (e.g., 123).

[0036] As previously mentioned, one of the ways that system 100 can enforce the permission and security layer boundaries is through memory heap 160, which straddles the described security/permission boundaries. In general, a "memory heap" comprises a set of memory addresses set aside by system 100 during or just prior to runtime. In this particular example, system 100 can allocate and configure memory heap 160 so that only kernel layer components (e.g., JIT compiler 105) can write to memory heap 160 (e.g., via page table 180), while user layer components can only read from memory heap 160. As a result, application program 110 cannot execute any compiled code from JIT compiler 105 in memory heap 160, but, rather, must do so only in address space 175.

[0037] One will appreciate, therefore, that a "sandbox" can be set by requiring operation of an application only in user mode, and by requiring the application and JIT compiler to access certain components or data structures from a memory address associated with different permission sets. Accordingly, Figures 1A-1B and the corresponding text illustrate a number of different architectural components that can be used to access and/or execute virtually any type of executable code, including managed code, in a secure fashion. In particular, Figures 1A-1B and the corresponding text illustrate how an application can execute in a user 113 mode, and access a memory heap with only read or read/execute permissions for the JIT compiled code. In addition, the Figures and corresponding text illustrate how the application can invoke one or more kernel-layer components in different address space 170, which has read/write permissions for memory heap 160, and can thus compile and pass managed code to memory heap 160 but not execute it.

[0038] As previously mentioned, this type of distributed address space configuration can provide a number of different benefits to program execution and development. At the outset, for example, an application program developer can write virtually any type of code without worrying about safety considerations (e.g., type-safety) In addition, an operating system developer need not speed exhaustive

resources developing the runtime verification code that would force all executing program code to be safe (e.g., type-safe).

[0039] In addition to the foregoing, implementations of the present invention can also be described in terms of flow charts having one or more acts in a method for accomplishing a particular result. In particular, Figures 2 and 3, and the corresponding text, illustrates flow charts one or more acts for executing managed code so that safe and unsafe application program code can be executed without threatening or compromising security. The methods illustrated in Figures 2 and 3 are described below with reference to the components and diagrams of Figures 1A-1B.

[0040] Accordingly, Figure 2 shows that a method from the perspective client computer system can comprise act 200 of executing an application in a first address space. Act 200 includes executing an application program in a first address space of a memory location. For example, Figure 1B shows that application program 110 is executing from address space 175, which has read/execute permissions for accessing memory location 160 (i.e., where the JIT compiled code will be placed and thus designated as read/execute).

[0041] Figure 2 also shows that the method can comprise an act 210 of receiving a request from the application for intermediate language instructions. Act 210 can include receiving one or more requests from the application program to compile one or more sets of intermediate language instructions. For example, the runtime for application program 110 comes across pointer 140 to intermediate language code 120, which can only be accessed in kernel 103 mode. As such, the runtime passes the pointer 120 as message 143 to security component 155, which processes the request in kernel mode.

[0042] In addition, Figure 2 shows that the method can comprise an act 220 of compiling the intermediate language instructions in a second address space. Act 220 includes compiling one or more sets of intermediate language instructions into newly compiled code using a JIT compiler running in a second address space. For example, upon validating request 143, security component 155 prepares and executes one or more requests 147 to pass the requested intermediate language code

to JIT compiler 105. JIT compiler 105 then compiles the intermediate language code 120 in the second address space 170, which in this illustration is provided with read/write permissions to the shared memory heap 160.

[0043] Furthermore, Figure 2 shows that the method can comprise an act 230 of passing the compiled code to a shared memory heap. Act 230 includes passing the newly compiled code to a shared memory heap, wherein the application program can retrieve the newly compiled code into the first address space. For example, Figures 1A and 1B shows that JIT compiler 105, as well as application program 110, have access to memory heap 160. In particular, JIT compiler 105 can write to (but not execute in) memory heap 160, while application program 110 can only read and execute from memory heap 160. Thus, when JIT compiler 105 compiles and creates code 123, the runtime for application program 110 can retrieve compiled code 123 into address space 175, and execute the code in user mode.

[0044] In addition to the foregoing, Figure 3 shows that a method in accordance with an implementation of the present invention of generating computer-executable program code for a computer system in a manner that uses JIT compilation while avoiding security violations can comprise an act 300 of receiving executable code and code to be compiled. Act 300 includes receiving program code that includes executable code and code to be compiled. For example, operating system 100 receives one or more storage media, and/or receives a network-based upload of application program 110. Application program 110 includes executable program code, as well as intermediate language code 115, which is accessed separately by one or more kernel layer 103 components.

[0045] Figure 3 also shows that the method can comprise an act 310 of executing the executable code in a lower-privilege mode. Act 310 includes executing the executable code in a lower-privilege mode and in a first address space. For example, Figure 1A shows that the executable portion of application program 110 is accessed or otherwise executed only in user mode 113, whereas the intermediate language code 115 is only accessed by kernel mode components.

[0046] In addition, Figure 3 shows that the method can comprise an act 310 of receiving a pointer for code to be compiled. Act 310 includes receiving one or

more pointers in the executable code for at least some code to be compiled. For example, Figures 1A-1B shows application program 110, which is operating in user mode 113 and in/from address space 175, comprises compiled code 135, pointer 140 to intermediate language code 120, compiled code 145, and pointer 150 to intermediate language code 125. While executing application program 110 in user mode, the pointers 140 and/or 150 will be identified in turn.

[0047] Furthermore, Figure 3 shows that the method can comprise an act 330 of switching to a higher-privileged mode. For example, the runtime for application program 110 identifies pointer 140 during execution, and identifies that JIT compiler 105 will need to be initiated. Since JIT compiler 105 will need to operate in kernel mode, system 100 momentarily pauses execution of application 110, switches from user mode to kernel mode, and then initiates JIT compiler 105 as a kernel mode 103 component. A message 143, which includes pointer 140, is then passed to a kernel mode 103 security component 155. Security component 155, operating in kernel mode, then evaluates the request to ensure the request 143 is properly formed, and/or includes the appropriate handles, security identifiers, etc.

[0048] Still further, Figure 3 shows that the method can comprise an act 340 of compiling the requested code in a higher-privilege mode. Act 340 includes compiling the requested code in a different address space using a compiler operating in the higher-privilege mode. For example, Figures 1A and 1B show that JIT compiler 105, which is operating in the higher-privilege kernel layer 103, can compile code 120 in one address space (address space 170), and further pass compiled code 123 to memory heap 160, where the JIT compiler has read/write access. Upon switching back to user mode, application program 110 can then access the compiled code 123 and execute this code from an different address space (address space 175) which has read/execute permissions for the memory heap 160.

[0049] As such, Figures 1A-2 and the corresponding text provide a number of components, modules, and mechanisms that can be used to execute untrusted code, including managed code, without sacrificing important security guarantees. As previously described, this can be accomplished at least in part by separating compilation of intermediate language code and execution of binary code in separate

address spaces for the same program. In addition, this can be accomplished with a type-safe JIT compiler, which compiles intermediate code and passes the compiled code into a shared memory heap. The type-safe JIT compiler is configured so that, while it can accept and compile code that is not type-safe, the JIT compiler, itself, is constrained from operating outside of certain prescribed type-safety boundaries. Still further, this can be accomplished by ensuring that executable code is only accessed by components operating in user mode, and that intermediate language code is only accessed by components operating in kernel mode in a read/write address space.

[0050] The embodiments of the present invention may comprise a special purpose or general-purpose computer including various computer hardware, as discussed in greater detail below. Embodiments within the scope of the present invention also include computer-readable media for carrying or having computer-executable instructions or data structures stored thereon. Such computer-readable media can be any available media that can be accessed by a general purpose or special purpose computer.

[0051] By way of example, and not limitation, such computer-readable media can comprise RAM, ROM, EEPROM, CD-ROM or other optical disk storage, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to carry or store desired program code means in the form of computer-executable instructions or data structures and which can be accessed by a general purpose or special purpose computer. When information is transferred or provided over a network or another communications connection (either hardwired, wireless, or a combination of hardwired or wireless) to a computer, the computer properly views the connection as a computer-readable medium. Thus, any such connection is properly termed a computer-readable medium. Combinations of the above should also be included within the scope of computer-readable media.

[0052] Computer-executable instructions comprise, for example, instructions and data which cause a general purpose computer, special purpose computer, or special purpose processing device to perform a certain function or group of functions. Although the subject matter has been described in language specific to

structural features and/or methodological acts, it is to be understood that the subject matter defined in the appended claims is not necessarily limited to the specific features or acts described above. Rather, the specific features and acts described above are disclosed as example forms of implementing the claims.

5 **[0053]** The present invention may be embodied in other specific forms without departing from its spirit or essential characteristics. The described embodiments are to be considered in all respects only as illustrative and not restrictive. The scope of the invention is, therefore, indicated by the appended claims rather than by the foregoing description. All changes which come within the meaning and range

10 of equivalency of the claims are to be embraced within their scope.

CLAIMS

We claim:

1. In a computerized environment comprising a memory, as well as a JIT compiler and one or more application programs loaded in the memory, a method of executing managed code so that untrusted program code can be compiled and executed in a manner that does not threaten or otherwise compromise system security, comprising:

executing an application program from a first address space set with a first set of permissions for accessing a shared memory heap;

receiving one or more requests from the application program to compile one or more sets of intermediate language instructions;

compiling the one or more sets of intermediate language instructions into newly compiled code using a JIT compiler running in a second address space that has a second set of permissions for accessing the shared memory heap; and

passing the newly compiled code to the shared memory heap, wherein the application program can retrieve and execute the newly compiled code from the first address space.

2. The method as recited in claim 1, further comprising an act of, upon receiving an indication that the newly compiled code has been passed to the shared memory heap, switching from a kernel mode to a user mode level of operation.

3. The method as recited in claim 2, further comprising an act of the application program retrieving the compiled code, and executing the compiled code from the first address space.

4. The method as recited in claim 1, wherein the first address space is configured with read/execute permissions with respect to accessing the shared memory heap, such that no component operating in the first address space can write to the shared memory heap.

5. The method as recited in claim 1, wherein the second address space is configured to access the memory heap with read/write permissions, such that no

component operating in the second address space can execute code in the memory heap.

6. The method as recited in claim 1, wherein the JIT compiler is operating in a higher-privilege mode, and the application program is running in a lower-privilege mode.

7. The method as recited in claim 1, wherein the JIT compiler is constrained to execute within one or more type-safety restraints, but configured to accept and compile intermediate language code that is not type-safe.

8. The method as recited in claim 7, wherein the JIT compiler performs the acts of:

receiving one or more requests to perform a function that violates a security restraint for the JIT compiler; and

rejecting the one or more requests to perform the function, or discontinuing compiling the one or more sets of intermediate language instructions.

9. The method as recited in claim 1, further comprising an act of, upon receiving the one or more requests from the application program, activating a kernel mode level of operation.

10. The method as recited in claim 9, wherein the act of activating a kernel mode level of operation includes an act of initiating a kernel mode security component.

11. The method as recited in claim 10, wherein the one or more requests from the application program are received by a kernel mode security component.

12. The method as recited in claim 11, further comprising an act of the kernel mode security component validating the one or more requests from the application program.

13. The method as recited in claim 12, wherein the act of validating the one or more requests comprises an act of determining whether a handle included in the one or more requests is valid.

14. In a computerized environment comprising a storage, a JIT compiler, and one or more application programs loaded in memory, a method of generating

computer executable program code in a manner that uses JIT compilation while avoiding security violations, comprising:

receiving application program code that includes executable code and code to be compiled;

5 executing the executable code in a lower-privilege mode and in a first address space;

identifying one or more pointers in the executable code for at least some code to be compiled;

switching to a higher-privilege mode; and

10 compiling the at least some code in a different address space using a compiler operating in the higher-privilege mode.

15. The method as recited in claim 14, wherein the application program code comprises part of a video game application that is received from the storage into a video game operating system.

15 16. The method as recited in claim 14, wherein the compiler is a type-safe JIT compiler configured to handle only type-safe requests, but otherwise configured to compile type-safe or non-type-safe intermediate language code.

17. The method as recited in claim 14, wherein the higher-privilege mode is a kernel mode level of operation, and the lower-privilege mode is a user level of

20 operation.

18. The method as recited in claim 14, wherein the first address space is configured to access a memory heap with read/execute permissions, and the second address space is configured to access the memory heap with read/write permissions.

19. The method as recited in claim 14, further comprising the acts of:

25 switching to the lower-privilege mode upon identifying that the at least some code has been compiled; and

executing the compiled at least some code in the first address space.

20. In a computerized environment comprising a memory, a JIT compiler, and one or more application programs loaded in the memory, a computer program

30 storage product having computer executable instructions stored thereon that, when executed, cause one or more processors to perform a method comprising:

executing an application program from a first address space set with a first set of permissions for accessing a shared memory heap;

receiving one or more requests from the application program to compile one or more sets of intermediate language instructions;

5 compiling the one or more sets of intermediate language instructions into newly compiled code using a JIT compiler running in a second address space that has a second set of permissions for accessing the shared memory heap; and

passing the newly compiled code to the shared memory heap, wherein 10 the application program can retrieve and execute the newly compiled code from the first address space.

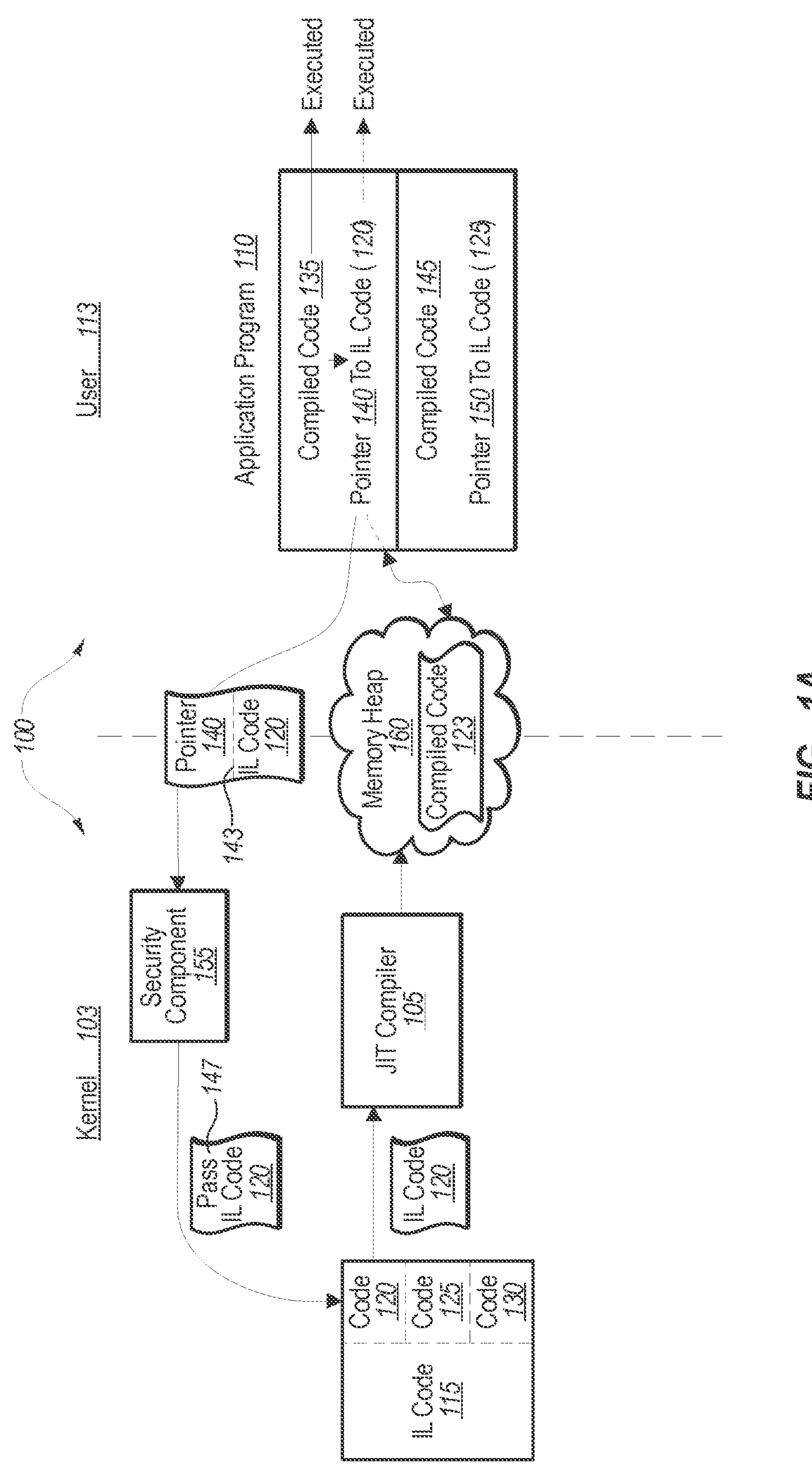


FIG. 1A

Page Table 180

Memory Location	Address Space 170	Address Space 175	Address Space 190	Address Space 195
160	Read / Write	Read / Execute	No Access	No Access
165	Read / Write	Read Only	Read / Write	Read / Execute

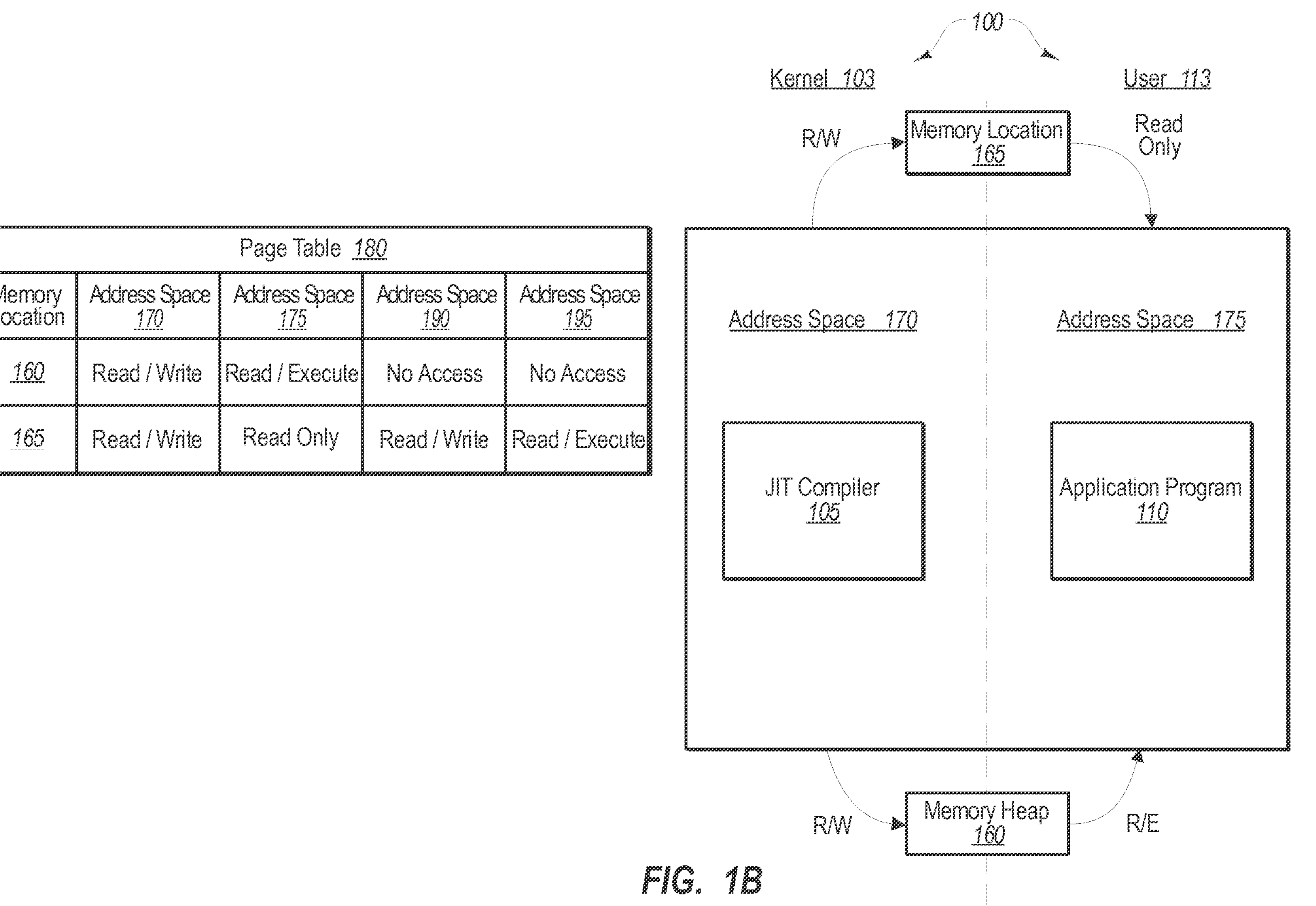


FIG. 1B

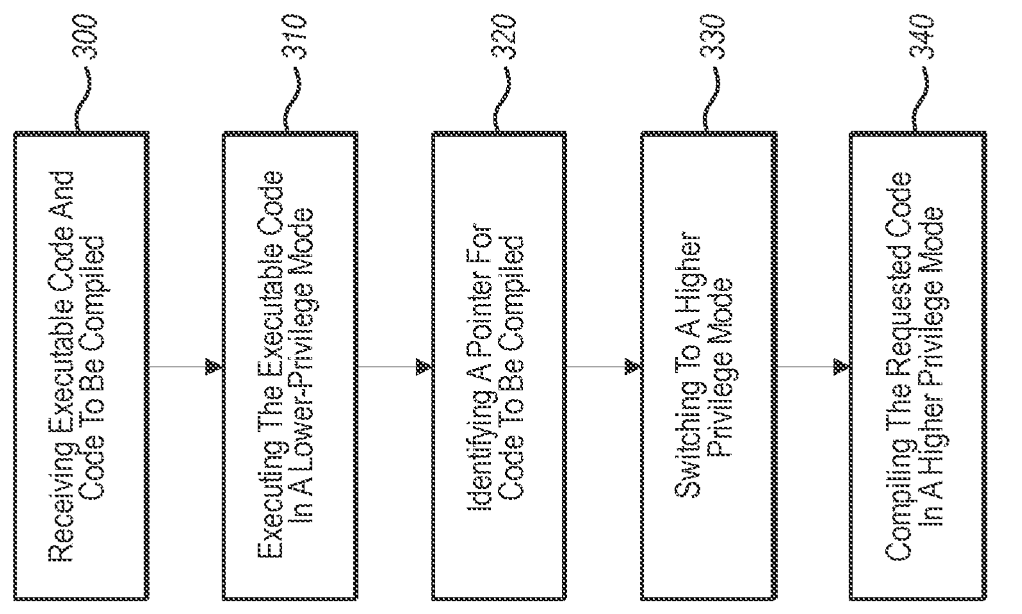


FIG. 3

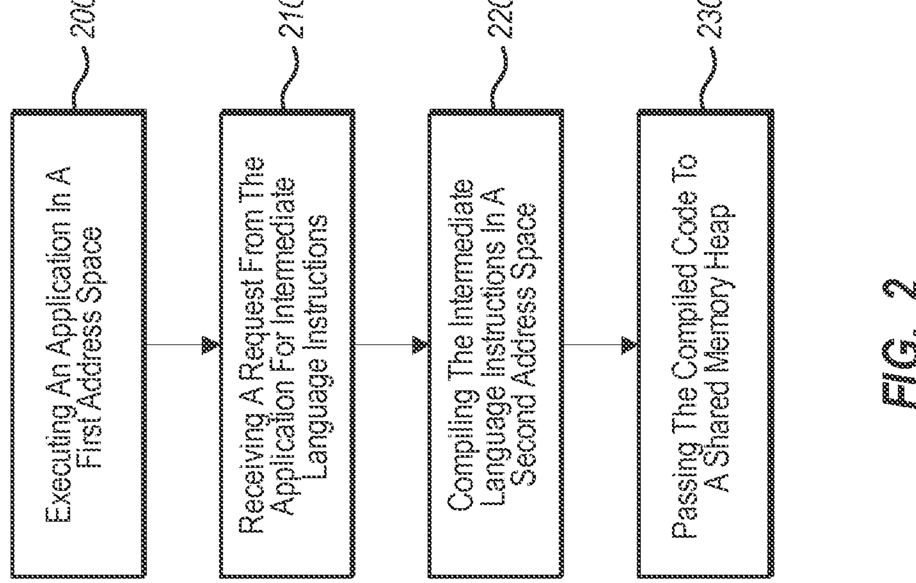


FIG. 2

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2007/085664

A. CLASSIFICATION OF SUBJECT MATTER

G06F 9/45(2006.01)i, G06F 12/02(2006.01)i

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 8 : G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
Korean utility models and applications for utility models since 1975
Japanese utility models and applications for utility models since 1975

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKIPASS(Kipo Internal), Google, YesKisti
keywords:security, trust, privilege, kernel, system, mode, level, compile, JIT, Intermediate, heap, execution, run

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US2004/0255268 A1 (MEIJER, E. et al.) 16 DECEMBER 2004 See figures 1,3; paragraphs [10][25][54]~[57].	1-20
A	US2006/0123403 A1 (BRUECKLMAYR, F. J. et al.) 08 JUNE 2006 See figures 1-3; paragraphs [25][64][65].	1-20
A	US2003/0236986 A1 (CRONCE, P. A. et al.) 25 DECEMBER 2003 See figures 3,4; paragraphs [10][11][40][41].	1-20
A	US2005/0172286 A1 (BRUMME, C. W. et al.) 04 AUGUST 2005 See figures 2,5; paragraphs [19][25][37].	1-20
PA	HUNT, G.C and LARUS, J.R. Singularity: Rethinking the Software Stack. ACM SIGOPS Operating Systems Review April 2007, Vol.41 Issue 2, pages 37-49. See section 2.1, 3.1.1, 3.2.1 and 3.6	1-20

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

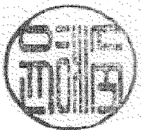
* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which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

Date of the actual completion of the international search	Date of mailing of the international search report
11 MARCH 2008 (11.03.2008)	**11 MARCH 2008 (11.03.2008)**

Name and mailing address of the ISA/KR

Korean Intellectual Property Office
Government Complex-Daejeon, 139 Seonsa-ro, Seo-gu, Daejeon 302-701, Republic of Korea
Facsimile No. 82-42-472-7140

Authorized officer
YOON, Hye Sook



Telephone No. 82-42-481-8370

INTERNATIONAL SEARCH REPORT
Information on patent family members

International application No.
PCT/US2007/085664

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US20040255268A1	16.12.2004	NONE	
US20060123403A1	08.06.2006	NONE	
US20030236986A1	25.12.2003	NONE	
US20050172286A1	04.08.2005	NONE	