

(19) World Intellectual Property Organization
International Bureau



(43) International Publication Date
13 December 2007 (13.12.2007)

PCT

(10) International Publication Number
WO 2007/143012 A2

(51) International Patent Classification:

G06F 11/26 (2006.01)

(21) International Application Number:

PCT/US2007/012812

(22) International Filing Date: 31 May 2007 (31.05.2007)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

60/809,936 31 May 2006 (31.05.2006) US

(71) Applicant (for all designated States except US): **THE TRUSTEES OF COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK** [US/US]; 412 Low Memorial Library, 535 West 116th Street, New York, NY 10027 (US).

(72) Inventors; and

(75) Inventors/Applicants (for US only): **KEROMYTIS, Angelos, D.** [GR/US]; 423 W. 120th Street, #98, New York, NY 10027 (US). **SIDIROGLOU, Stelios** [US/US]; 540 W. 122nd Street, Apt. 36, New York, NY 10027 (US).

(74) Agents: **BAIK, Peter, W.** et al.; Wilmer Cutler Pickering Hale And Dorr LLP, 399 Park Avenue, New York, NY 10022 (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.

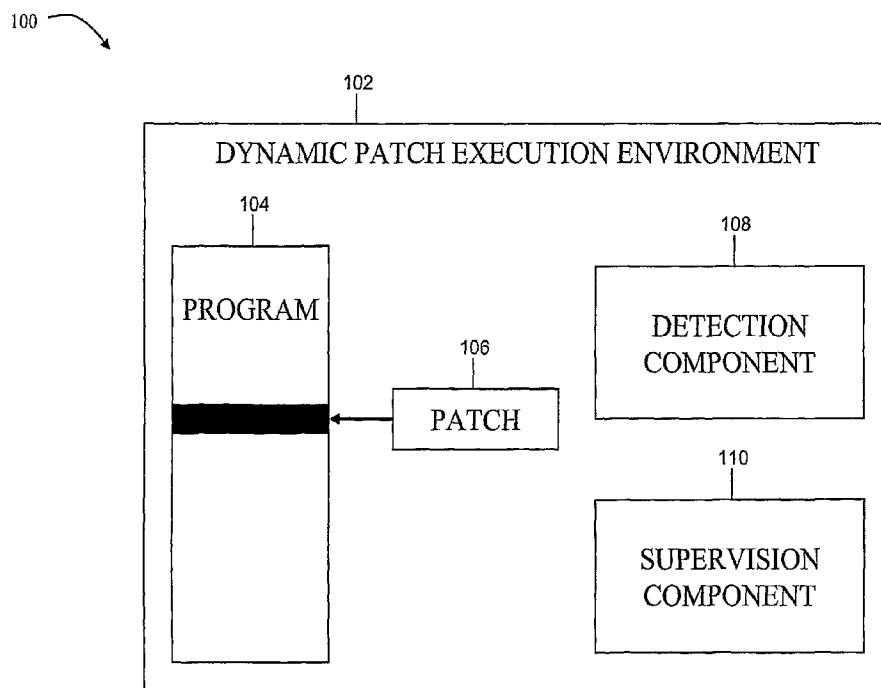
(84) Designated States (unless otherwise indicated, for every kind of regional protection available): ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, 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).

Published:

— without international search report and to be republished upon receipt of that report

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: SYSTEMS, METHODS, AND MEDIA FOR TESTING SOFTWARE PATCHES



(57) Abstract: Systems, methods, and media for testing software patches are provided. The methods include: injecting a software patch into a program; running multiple instances of a part of the program containing the software patch; obtaining outcomes of the multiple instances; and determining, using the outcomes, whether the software patch is executed correctly.

WO 2007/143012 A2

SYSTEMS, METHODS, AND MEDIA FOR TESTING SOFTWARE PATCHES

Cross-Reference to Related Applications

[0001] This application claims the benefit of United States Provisional Patent Application No. 60/809,936, filed May 31, 2006, which is hereby incorporated by reference herein in its entirety.

Technical Field

[0002] The disclosed subject matter relates to systems, methods, and media for testing software patches

Background

[0003] Despite decades of research and development efforts for, and the resulting improvement in, software reliability, software defects still account for many system failures. Moreover, methods for retroactively dealing with software defects, such as software patches, can often introduce latent and/or new defects.

[0004] The problem is further exacerbated by the burgeoning popularity of service oriented computing (SOC) systems, such as online commerce, e-mail, Internet Protocol (IP) telephony, and grid computing, and the availability requirements accompanying such systems. In order to update SOC systems to fix existing software defects and vulnerabilities, system administrators need to strike a careful balance between bringing a system down for installing updates and keeping the system available for processing service requests.

[0005] Unfortunately, most SOC systems are long-running servers that amass considerable operational state data and, therefore, the option of launching another machine to test the effects of software patch(es) is limited.

[0006] System administrators can test software patches on a non-production machine and mirror the resulting traffic to the corresponding production system. Because the machines are disjoint, however, there are some drawbacks. First, comparison of two or more mirrored machines requires some level of cross-system synchronization, which, in turn, adds considerable overhead. Second, when dealing with encrypted traffic, a proxy would need to be employed, thereby adding complexity and causing performance degradation.

Summary

[0007] Systems, methods, and media for testing software patches are provided. In some embodiments, methods for testing software patches are provided. The methods include: injecting a software patch into a program; running multiple instances of a part of the program containing the software patch; obtaining outcomes of the multiple instances; and determining, using the outcomes, whether the software patch executes correctly.

[0008] In some embodiments, computer-readable media containing computer-executable instructions that, when executed by a processor, cause the processor to perform a method for testing software patches, are provided. This method includes: injecting a software patch into a program; running multiple instances of a part of the program containing the software patch; obtaining outcomes of the multiple instances; and determining whether, using the outcomes, the software patch executes correctly.

[0009] In some embodiments, systems for testing software patches are provided. The systems include: elements for injecting a software patch into a program; elements for running multiple instances of a part of the program containing the software patch; elements for obtaining outcomes of the multiple instances; and elements for determining, using the outcomes, whether the software patch executes correctly.

[0010] In some embodiments, systems for testing software patches are provided. The systems include: an execution environment capable of injecting a software patch into a

program and running multiple instances of a part of the program containing the software patch; and a supervision component capable of obtaining outcomes of the multiple instances and determining, using the outcomes, whether the software patch executes correctly.

Brief Description of the Drawings

[0011] FIG. 1 is a schematic diagram of a system for testing software patches in accordance with some embodiments of the disclosed subject matter.

[0012] FIG. 2 is a simple illustration of a method for testing software patches in accordance with some embodiments of the disclosed subject matter.

[0013] FIG. 3 is an illustrative diagram for injecting software patches into a program in accordance with some embodiments of the disclosed subject matter.

Detailed Description

[0014] Systems, methods, and media for testing software patches are provided. In some embodiments of the disclosed subject matter, systems, methods, and media are provided for testing software patches inserted into programs in a production machine, such as a production server, prior to system deployment. In some embodiments, software patches are injected while the programs are executing.

[0015] In some embodiments, a binary software patch can be injected into a program, such as a server application, using binary injection techniques so that when an instance of the program reaches targeted program segment(s), multiple instances of the program will be spawned to run the targeted program segment(s) either in parallel or in sequence. In some embodiments, multiple software patches are injected into a program. The output of the multiple instances can then be used to determine whether the patch(es) execute correctly. In some embodiments, software patches are injected while the program is executing. In some

embodiments, a validation mechanism of a software patch is also injected along with the software patch.

[0016] In some embodiments, an emulator, such as a virtual machine, is used to run multiple instances of the targeted segment(s) of the program. For example, the multiple instances can run in the emulator either in sequence or in parallel. The emulator, containing a set of logic for supervising running instances, can determine whether an exception indicating a failure is raised while each instance is executed. The emulator can also record the modifications made by each instance to the program memory and keep track of input and/or output transactions that each instance makes. In some embodiments, the emulator is injected into the program using an application programming interface (API) for runtime code patching.

[0017] In some embodiments, an operating system kernel is used to run multiple instances of the targeted segment(s) of the program. For example, the multiple instances can run in sequence or in parallel using a set of kernel functions, referred to as system calls, that can access and modify the memory space of the program. For instance, kernel functions can insert a software patch into the program, spawn and run multiple instances of the program to execute the targeted segment(s), detect and catch exceptions thrown by the multiple instances, and record program state changes.

[0018] Referring to FIG. 1, a system 100 includes a dynamic patch execution environment 102, a program 104 that needs to be updated, a patch 106, a detection component 108, and a supervision component 110.

[0019] Dynamic patch execution environment 102 can be a library of application program interfaces (API) for runtime code patching, such as Dyninst, or various other suitable runtime program instrumentation tools. The API can, for example, attach to program 104, create a new bit of code, and insert the new code into program 104.

[0020] Program 104 can continue execution while being modified and, therefore, program 104 does not need to be re-compiled, re-linked, and/or restarted. For instance, the next time the modified program executes the block of code that has been modified, the new code is executed in addition to, or in lieu of, the original code. The API can also change subroutine calls or remove them from program 104.

[0021] The API can make modifications to both the code and data of program 104. For example, the API can be used by a programmer, a user, or a system administrator. For instance, a programmer can insert instrumentation code during the development of a software application for subsequent patch testing.

[0022] Dynamic patch execution environment 102 can provide an environment in which patch 106 can be applied to multiple service instances of program 104. In some embodiments, patch 106 is inserted into program 104 and an instance of program 104 is run such that the part of program 104 containing patch 106 is executed multiple times within the same execution of program 104. In some embodiments, multiple instances of the part of program 104 containing patch 106 are run in parallel.

[0023] Program 104 can be a software application in a system, such as an SOC system. Program 104 also can be various other suitable programs that require high availability and/or frequent runtime updates or modifications.

[0024] Patch 106 can be a simple logic defect fix, a software update designed to close a vulnerability loophole, and/or any other suitable fix or update. In some embodiments, patch 106 is used to test implementation of new computing techniques. For example, an instance of the implementation of a controversial, unproven computing technique can be tested by placing the implementation into patch 106.

[0025] Detection component 108 can detect and examine software exceptions thrown by the multiple instances of the part of program 104 containing patch 106. Supervision

component 110 can obtain outcomes of the multiple instances. It can also determine whether patch 106 executes correctly using the outcomes of the multiple instances. Supervision component 110 can be invoked whenever all of the multiple instances reach predefined points in their execution or when software exceptions are detected by detection component 108. Both detection component 108 and supervision component 110 can be a shared library that can be injected at specific locations in program 104.

[0026] Referring to FIG. 2, a method 200 tests software patches to determine whether the software patches execute correctly.

[0027] As shown, at 202, a software patch, such as patch 106, is injected into a program, such as program 104. In some embodiments, a runtime instrumentation injection technique is employed. For example, Dyninst API can be used to inject instrumentation code into an active server application, such as a financial transaction server. For instance, Dyninst API can be used to insert instrumentation code, referred to as instrumentation trampolines, that can point to the different versions of the software patch under test. Dyninst API also can be used to insert instrumentation trampolines that can point to the instrumentation code that can examine and compare the output of the software patch.

[0028] At 204, multiple instances of a part of the program containing the software patch are run. In some embodiments, one or more execution threads are created and a function or a procedure is copied into the execution threads, in order for the execution threads to run the function or procedure in parallel. In some embodiments, the execution threads are created and run such that they do not interfere with one another (i.e., each thread is oblivious of the presence of other threads).

[0029] Suppose, for example, a software patch is inserted to modify the logic of a function in the program. When the program execution reaches the function, one or more

execution threads can be created. The function can be then copied onto each of the execution threads such that the execution threads can execute the function.

[0030] At 206, it is determined whether an exception is thrown by any of the multiple instances of the part of the program. In some embodiments, a separate module, such as detection component 108, is used to catch exceptions thrown by the multiple instances.

[0031] If an exception thrown by one of the multiple instances is detected at 206, the outcomes of the multiple instances are obtained at 208. For example, the outcomes of the multiple instances can be one or more characteristics indicative of the end-state of the execution of the multiple instances. In some embodiments, for instance, memory traces of each of the multiple instances are used as the outcomes of the multiple instances. For example, allocation and de-allocation (i.e., deletion) of memory blocks at the library/system call level can be tracked to fingerprint the execution of each multiple instances.

[0032] In some embodiments, input/output (I/O) transactions are used as the outcomes of the multiple instances. For example, a file system view can be duplicated to be attached to each instance of the part of the program. The file system view for each instance of the program can then be compared later to find any inconsistencies. In some embodiments, the program's external behavior(s) are used as the outcomes of the multiple instances. For example, information printed or displayed on a display screen can be used as the outcomes.

[0033] In some embodiments, the multiple instances are stopped before obtaining the outcomes when an exception is thrown. In some embodiments, only the program instance that has thrown an exception is stopped while the remaining instances of the program continue running. In some embodiments, the exception is handled to recover the excepted instance of the program from the failure causing the exception. In some embodiments,

detection component 108 handles the exception to recover the excepted instance of the program.

[0034] In some embodiments, the multiple instances are run one instance at a time in sequence. An instance of the part of the program, for example, can run first and its outcome, such as the resulting program state, can be saved before rolling back the program state to run the next instance. Once the program state is restored, the next instance of the program can run and the same process can repeat for subsequent instances. In some embodiments, each saved program state can be used to determine whether the part of the program containing the software patch executes correctly. In some embodiments, the process of running multiple instances is stopped once an instance of the program runs without raising an exception.

[0035] If, however, no exception is detected at 206, it is further determined at 212 whether all of the multiple instances of the program have reached a predefined synchronization point. For example, the predefined synchronization point can be the end of the software patch injected at 202. If it is determined at 212 that all of the multiple instances of the program have not reached the synchronization point yet, then 206 is repeated. If, however, it is determined at 212 that all of the multiple instances of the program have all reached the synchronization point, then the outcomes of the multiple instances are obtained at 208.

[0036] At 210, it is determined using the outcomes whether the software patch executes correctly. In some embodiments, determining the correctness of the software patch execution is based primarily on policy. For example, a correct base instance or observed normal operation can define the measure of correctness of the software patch execution in a program, such as program 104. Suppose, for instance, network routers are periodically retrofitted with vulnerability patches. If it is observed that the network routers operate

normally after the patches have been inserted, it can be concluded that the patches execute correctly.

[0037] In some embodiments, variations from well-defined invariants and specifications of the program state resulting from execution of the multiple instances can define the measure of correctness of the software patch execution in the program. For example, values returned from one or more functions invoked by the multiple instances can be checked against predefined correct return values from the functions.

[0038] Referring to FIG. 3, a diagram 300 illustrates a graphical example of injecting software patches into a program using Dyninst API.

[0039] To instrument a point in a program 302, one or more instructions in program 302 can be relocated and replaced with a jump to a base trampoline 304. Base trampoline 304 can be used to modify the original code of program 302 such that the original code can branch into the newly generated code. Base trampoline 304 contains a jump to a mini-trampoline 306 and contains space for the relocated instructions from original program 302.

[0040] Mini-trampoline 306 contains the instructions for performing the desired functionality stored in a patch 308 as well as for saving and restoring the state information of program 302, such as the register values and the program counter. For example, mini-trampoline 306 can call an existing function or run a snippet of code that has been created and inserted into program 302 using Dyninst API.

[0041] In some embodiments, more than one mini-trampoline 306 can be inserted at the same point in program 302. For example, multiple mini-trampolines 306 can be chained together such that the end of one mini-trampoline 306 can branch to the start of the next mini-trampoline 306, and the last mini-trampoline 306 can branch back to base trampoline 304, which, in turn, can branch back to program 302.

[0042] Although the invention has been described and illustrated in the foregoing illustrative embodiments, it is understood that the present disclosure has been made only by way of example, and that numerous changes in the details of implementation of the invention can be made without departing from the spirit and scope of the invention, which is only limited by the claims which follow. Features of the disclosed embodiments can be combined and rearranged in various ways.

What is claimed is:

1. A method for testing software patches, the method comprising:

injecting a software patch into a program;

running multiple instances of a part of the program containing the software patch;

obtaining outcomes of the multiple instances; and

determining, using the outcomes, whether the software patch is executed correctly.
2. The method of claim 1, wherein obtaining the outcomes of the multiple instances comprises one of:

obtaining memory traces of the multiple instances;

obtaining input and output transactions of the multiple instances; and

the combination thereof.
3. The method of claim 1, wherein determining, using the outcomes, whether the software patch is executed correctly comprises implementing a selection protocol by:

comparing the outcomes with normal outcomes of the program;

checking the outcomes against a set of predefined end-states of the program;

and

the combination thereof.
4. The method of claim 1, wherein running the multiple instances of the part of the program containing the software patch comprises generating at least one execution thread to run the part of the program containing the software patch in parallel.

5. The method of claim 1, wherein running the multiple instances of the part of the program containing the software patch comprises using a virtual machine to run the part of the program containing the software patch.

6. The method of claim 1, wherein running the multiple instances of the part of the program containing the software patch comprises using an operating system kernel to run the part of the program containing the software patch.

7. The method of claim 1, wherein instances are run in sequence and the first instance that runs without raising an exception is the last instance of the sequence to run.

8. The method of claim 1, wherein the outcomes are obtained upon all the multiple instances reaching a predefined synchronization point.

9. The method of claim 1, further comprising injecting a patch-validation mechanism into the program.

10. The method of claim 1, further comprising detecting at least one software exception thrown by at least one of the multiple instances.

11. The method of claim 10, wherein the outcomes are obtained when the at least one software exception is detected.

12. Computer-readable media containing computer-executable instructions that, when executed by a processor, cause the processor to perform a method for testing software patches, the method comprising:

injecting a software patch into a program;

running multiple instances of a part of the program containing the software patch;

obtaining outcomes of the multiple instances; and

determining, using the outcomes, whether the software patch is executed correctly.

13. The computer-readable media of claim 12, wherein obtaining the outcomes of the multiple instances comprises one of:

obtaining memory traces of the multiple instances;

obtaining input and output transactions of the multiple instances; and

the combination thereof.

14. The computer-readable media of claim 12, wherein determining, using the outcomes, whether the software patch is executed correctly comprises implementing a selection protocol by:

comparing the outcomes with normal outcomes of the program;

checking the outcomes against a set of predefined end-states of the program;

and

the combination thereof.

15. The computer-readable media of claim 12, wherein running the multiple instances of the part of the program containing the software patch comprises generating at least one execution thread to run the part of the program containing the software patch in parallel.

16. The computer-readable media of claim 12, wherein running the multiple instances of the part of the program containing the software patch comprises using a virtual machine to run the part of the program containing the software patch.

17. The computer-readable media of claim 12, wherein running the multiple instances of the part of the program containing the software patch comprises using an operating system kernel to run the part of the program containing the software patch.

18. The computer-readable media of claim 12, wherein instances are run in sequence and the first instance that runs without raising an exception is the last instance of the sequence to run.

19. The computer-readable media of claim 12, wherein the outcomes are obtained upon all the multiple instances reaching a predefined synchronization point.

20. The computer-readable media of claim 12, wherein the method further comprises injecting a patch-validation mechanism into the program.

21. The computer-readable media of claim 12, wherein the method further comprises detecting at least one software exception thrown by at least one of the multiple instances.

22. The computer-readable media of claim 21, wherein the outcomes are obtained when the at least one software exception is detected.

23. A system for testing software patches, the system comprising:

- means for injecting a software patch into a program;
- means for running multiple instances of a part of the program containing the software patch;
- means for obtaining outcomes of the multiple instances; and
- means for determining, using the outcomes, whether the software patch is executed correctly.

24. The system of claim 23, wherein means for obtaining the outcomes of the multiple instances comprises one of:

means for obtaining memory traces of the multiple instances;

means for obtaining input and output transactions of the multiple instances;

and

the combination thereof.

25. The system of claim 23, wherein means for determining, using the outcomes, whether the software patch is executed correctly comprises implementing a selection protocol using:

means for comparing the outcomes with normal outcomes of the program;

means for checking the outcomes against a set of predefined end-states of the program; and

the combination thereof.

26. The system of claim 23, wherein means for running the multiple instances of the part of the program containing the software patch comprises means for generating at least one execution thread to run the part of the program containing the software patch in parallel.

27. The system of claim 23, wherein means for running the multiple instances of the part of the program containing the software patch comprises means for using a virtual machine to run the part of the program containing the software patch.

28. The system of claim 23, wherein means for running the multiple instances of the part of the program containing the software patch comprises means for using an operating system kernel to run the part of the program containing the software patch.

29. The system of claim 23, wherein instances are run in sequence and the first instance that runs without raising an exception is the last instance of the sequence to run.

30. The system of claim 23, wherein the outcomes are obtained upon all the multiple instances reaching a predefined synchronization point.

31. The system of claim 23, further comprising means for injecting a patch-validation mechanism into the program.

32. The system of claim 23, further comprising means for detecting at least one software exception thrown by at least one of the multiple instances.

33. The system of claim 32, wherein the outcomes are obtained when the at least one software exception is detected.

34. A system for testing software patches, the system comprising:
an execution environment capable of:
injecting a software patch into a program; and
running multiple instances of a part of the program containing the software patch; and
a supervision component capable of:
obtaining outcomes of the multiple instances; and
determining, using the outcomes, whether the software patch is executed correctly.

35. The system of claim 34, wherein obtaining the outcomes of the multiple instances comprises one of:

obtaining memory traces of the multiple instances;

obtaining input and output transactions of the multiple instances; and

the combination thereof.

36. The system of claim 34, wherein determining, using the outcomes, whether the software patch is executed correctly comprises implementing a selection protocol by:

comparing the outcomes with normal outcomes of the program;

checking the outcomes against a set of predefined end-states of the program;

and

the combination thereof.

37. The system of claim 34, wherein running the multiple instances of the part of the program containing the software patch comprises generating at least one execution thread to run the part of the program containing the software patch in parallel.

38. The system of claim 34, wherein running the multiple instances of the part of the program containing the software patch comprises using a virtual machine to run the part of the program containing the software patch.

39. The system of claim 34, wherein running the multiple instances of the part of the program containing the software patch comprises using an operating system kernel to run the part of the program containing the software patch.

40. The system of claim 34, wherein instances are run in sequence and the first instance that runs without raising an exception is the last instance of the sequence to run.

41. The system of claim 34, wherein the outcomes are obtained upon all the multiple instances reaching a predefined synchronization point.

42. The system of claim 34, wherein the execution environment is further capable of injecting a patch-validation mechanism into the program.

43. The system of claim 34, further comprising a detection component capable of detecting at least one software exception thrown by at least one of the multiple instances.

44. The system of claim 43, wherein the outcomes are obtained when the at least one software exception is detected.

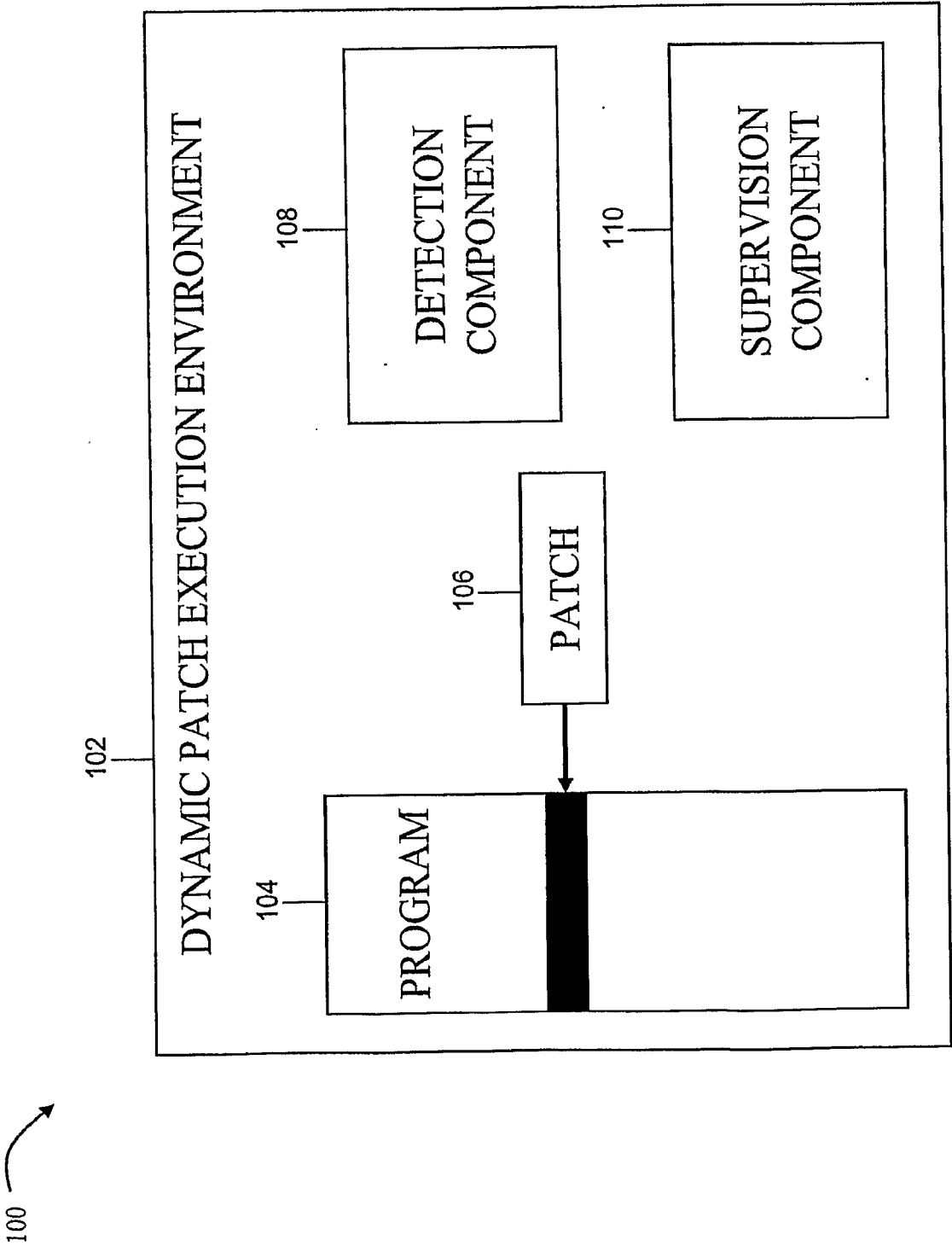
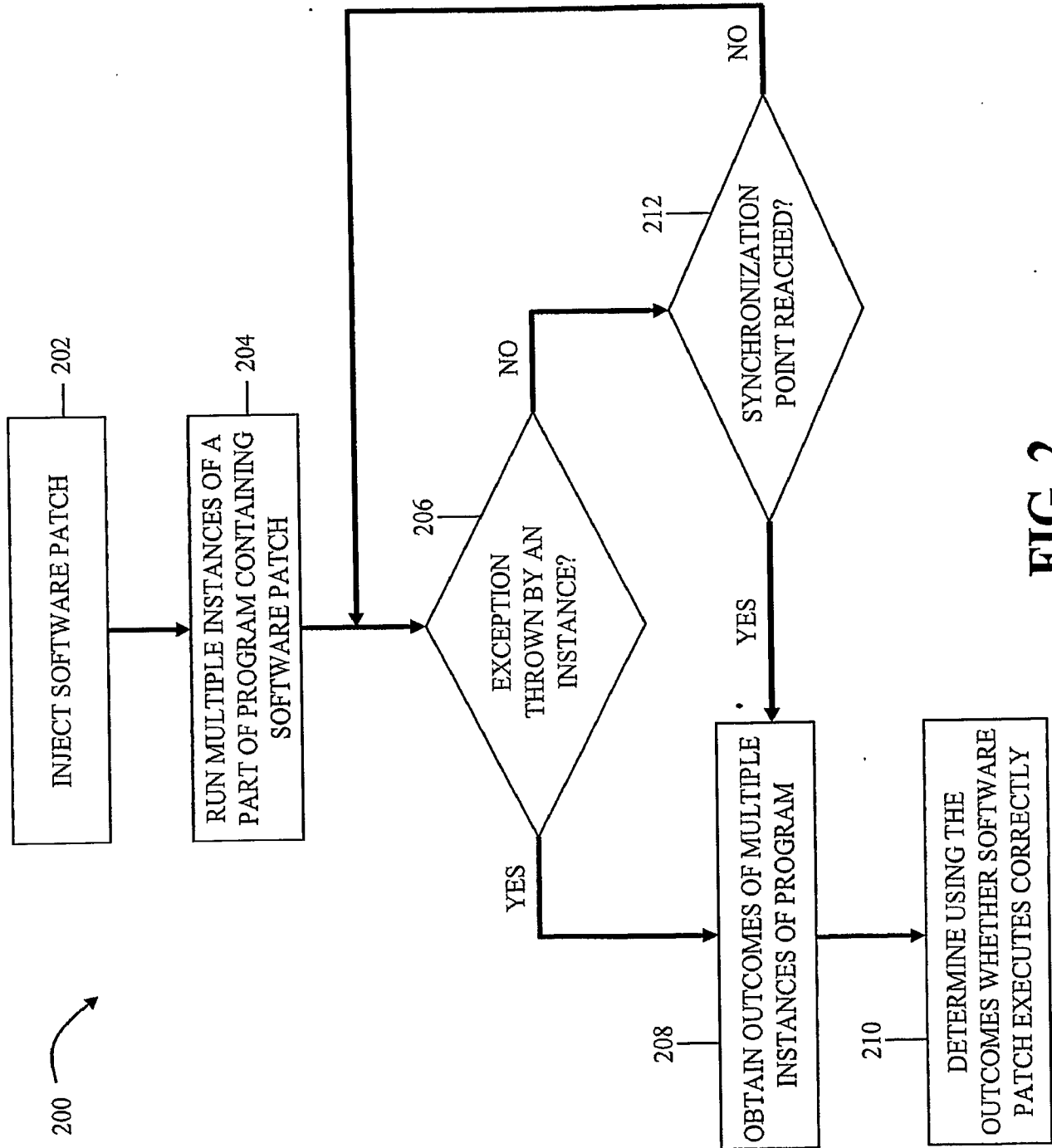


FIG. 1

**FIG. 2**

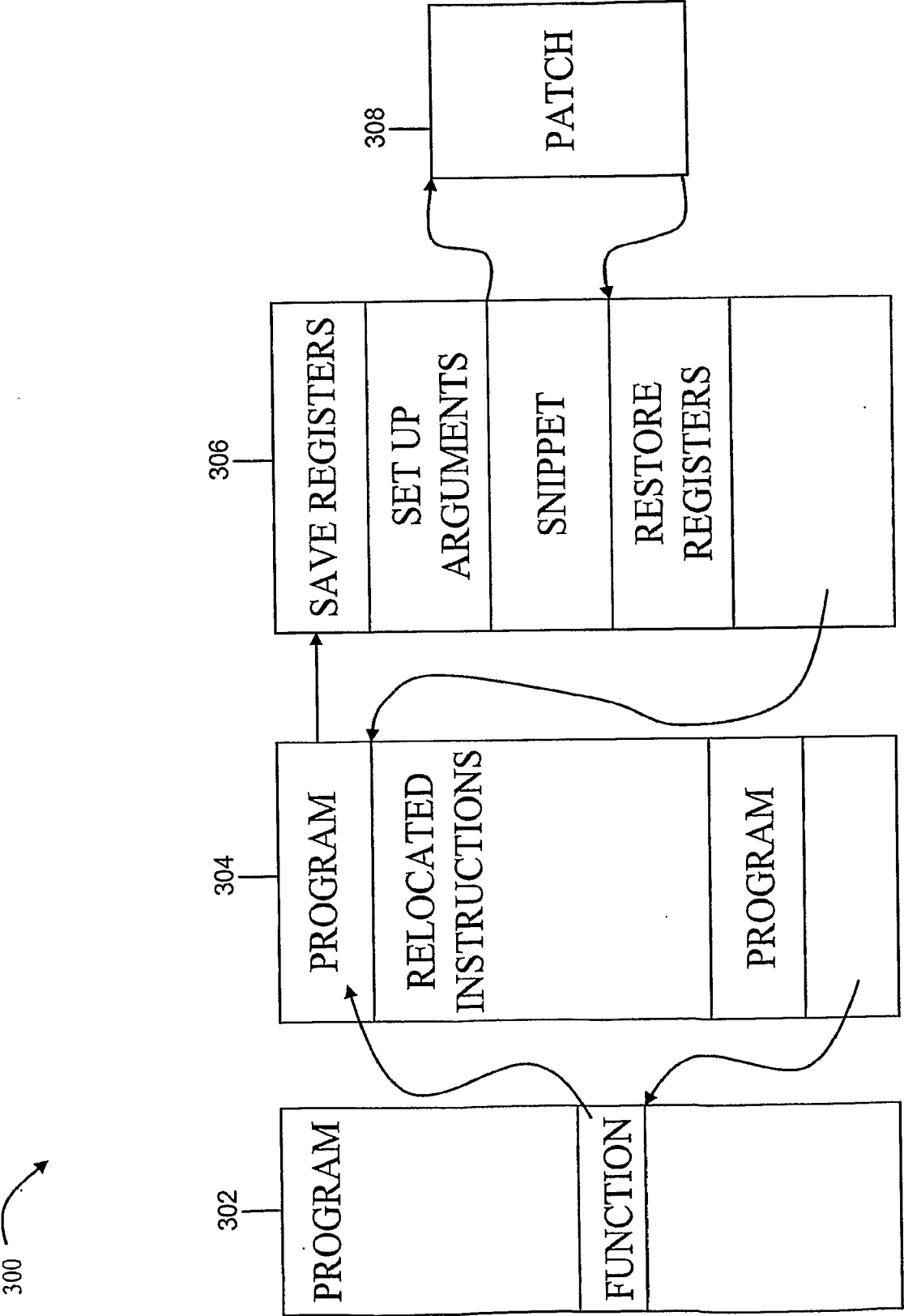


FIG. 3