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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
21 December 2017 (21.12.2017)

WIPO | PCT

(10) International Publication Number
WO 2017/218872 A1

(51) International Patent Classification:
G06F 21/51 (2013.01) *G06F 21/54* (2013.01)

(21) International Application Number:
PCT/US2017/037841

(22) International Filing Date:
16 June 2017 (16.06.2017)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
62/350,917 16 June 2016 (16.06.2016) US

(71) Applicant: **VIRSEC SYSTEMS, INC.** [US/US]; 226 Airport Parkway, Suite 350, San Jose, CA 95110 (US).

(72) Inventor: **GUPTA, Satya, Vrat**; 9699 Zac Court, Dublin, CA 94568 (US).

(74) Agent: **MEAGHER, Timothy, J.** et al.; HAMILTON, BROOK, SMITH & REYNOLDS, P.C., 530 Virginia Rd, P.O. Box 9133, Concord, MA 01742-9133 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, 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, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ,

(54) **Title:** SYSTEMS AND METHODS FOR REMEDIATING MEMORY CORRUPTION IN A COMPUTER APPLICATION

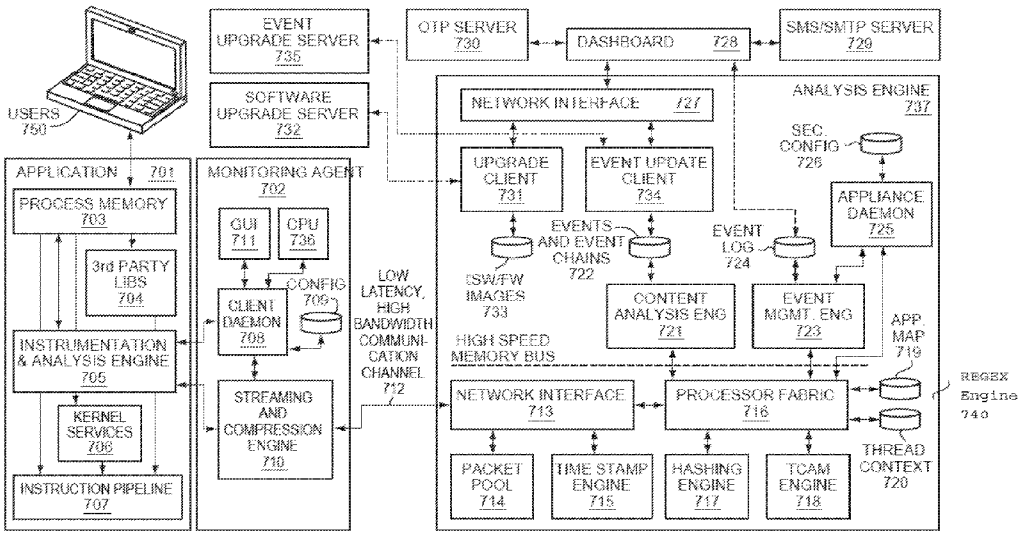

FIG. 7A

(57) **Abstract:** In example embodiments, systems and methods extract a model of a computer application during load time and store the model in memory. Embodiments may insert instructions into the computer application at run time to collect runtime state of the application, and analyze the collected data against the stored model to perform detection of security events. Embodiments may also instrument an exception handler to detect the security events based on unhandled memory access violations. Embodiments may, based upon the detection of the security events, dynamically respond, such as by modify a computer routine associated with an active process of the computer application. Modification may include installing or verifying an individual patch in memory associated with the computer application.

[Continued on next page]

WO 2017/218872 A1

TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *of inventorship (Rule 4.17(iv))*

Published:

— *with international search report (Art. 21(3))*

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments (Rule 48.2(h))*

SYSTEMS AND METHODS FOR REMEDIATING MEMORY CORRUPTION IN A COMPUTER APPLICATION

RELATED APPLICATION

[0001] This application claims the benefit of U.S. Provisional Application No. 62/350,917, filed on June 16, 2016. The entire teachings of the above application are incorporated herein by reference in their entirety.

BACKGROUND

[0002] Network accessible applications are often vulnerable to memory corruption attacks triggered remotely by malicious attackers. Malicious attackers have strived hard to exploit such vulnerability since it gives them unprecedented access to the remote user's computer network, often with elevated privileges. Once control has been seized, arbitrary code of the attacker's choosing can be executed by the attacker, as if the remote user owns the compromised machine. Usually the objective of the malicious attacker is to extract personal and/or confidential information from the user, but the objective could also include disrupting personal or business activity of the user for the purpose of inflicting loss of productivity.

[0003] Preparatory attacks may help to set the stage by placing strategic data in buffers on the stack, the heap segments, and other jump tables, including imports, exports, virtual pointers (VPTRs), and system call/system dispatch tables in the memory address space of the application. This allows subsequently launched attacks to manipulate the flow of execution, with the ultimate objective of causing code designed by a malicious hacker to execute instead of code that is natively part of the application. The most sophisticated attackers do not even need to insert their malicious code directly into the target application's memory space, instead, the attackers can re-purpose existing code by stitching together selectively chosen (i.e., cherry picked) chunks of code from the legitimately loaded application code and thereby execute their nefarious intent. There is an urgent need to protect the application at runtime from such advanced runtime memory corruption attacks.

SUMMARY

[0004] Embodiments of the present disclosure are directed to example systems and methods for protection against malicious attacks that are facilitated through memory corruption within one or more running processes. In some embodiments, the systems include one or more instrumentation engines and one or more analysis engines for performing operations to protect against malicious attacks. The one or more instrumentation engines may be located on the same or different hardware or computer system as the one or more analysis engines. In some embodiments, the systems and methods may extract a model of a computer application as the application code first loads into memory. The model may include, but is not limited to, building pairs of legal source and destination memory addresses, transitions, basic block boundary information, code segment bounds, import and export address table bounds, jump table bounds, or any other type of computer-routine-related information known to one skilled in the art. In some embodiments, the systems and methods may store the model of the computer application.

[0005] In some embodiments, the systems and methods may insert instructions into the computer application (optionally, at run time) prior to the computer application instructions being executed in memory in order to collect data at runtime and/or the execution state of the application. In some embodiments, the systems and methods may analyze the collected data at runtime against the stored model of the computer application to perform detection of one or more security events. In some embodiments, the systems and methods may, based upon the detection of the one or more security events, modify, in a manner that preserves continued execution of the computer application, at least one computer routine associated with at least one active process associated with the computer application (i.e., insert a patch).

[0006] According to some embodiments, the computer routine may be executed in association with the at least one process. In some embodiments, the one or more detected security events may be associated with a malicious movement to a different (unusual) code path within the computer application. Such a malicious movement may include, but is not limited to, a malicious jump routine, a trampoline to malicious code, an indirect jump vector, or any other malicious movement known to one skilled in the art.

[0007] In response to receipt of one or more aggregate patches by a user, some embodiments may perform at least one operation that modifies or removes the at least one

computer routine associated with the computer application, and modifying or removing one or more individual patches associated with the computer application. According to some embodiments, modifying may include verifying a patch or configuration associated with the computer application. According to some embodiments, the systems and methods may modify the stack associated with the at least one computer routine. In some embodiments, the systems and methods may modify one or more heaps associated with the at least one executing computer routine. In some other embodiments, the systems and methods may modify one or more jump tables.

[0008] Further, in some embodiments, the systems and methods may modify the at least one computer routine associated with the at least one process, while the at least one active process is executing the at least one computer routine. As such, some embodiments may employ hot patching (or live patching, or dynamic software patching/updating). According to some embodiments, a replacement function (i.e., different function) may be called as a result of a hot patch. In some embodiments, the systems and methods may, prior to modifying the at least one computer routine, pause execution of at least one active process (or computer application). In some embodiments, after modifying the at least one computer instruction, the systems and methods may resume execution of the at least one active process.

[0009] In some embodiments, the systems and methods may extract a model of a computer application during load time. According to some embodiments, the systems and methods may store the model of the computer application. In some embodiments, the systems and methods may insert instructions into the computer application (optionally, in memory) prior to the computer application being executed in memory in order to collect data at runtime. In some embodiments, the systems and methods may analyze the data collected at runtime against the stored model of the computer application to perform detection of one or more security events. In some embodiments, the systems and methods may, upon the detection of the one or more security events, temporarily remediate memory corruption associated with the computer application (i.e., restore one or more pointers) prior to executing one or more return instructions. In some embodiments, the systems and methods may report actionable information (i.e., report information to a vendor to create a patch) based upon the one or more detected security events. The actionable information may include, but is not limited to, information such as: where/how a security event occurs, where/how a trampoline

takes place, and where/how the memory in the stack or heap of a vulnerable function is corrupted.

[0010] In some embodiments, the systems and methods may modify, in a manner that preserves continued execution of the computer application, at least one computer routine associated with at least one process. In some embodiments, when a lite or full patch from the vendor is received by a user, an application runtime monitoring and analysis (ARMAS) application may be disabled, and new code may be loaded into computer memory associated with the process (i.e., a different location in memory) without shutting the process down. Some embodiments may deploy such lite patches released by a vendor as a shared static or dynamic library. Further, in some embodiments, the systems and methods may modify at least one computer instruction associated with at least one process, while the at least one process is executing. In some embodiments, the systems and methods may, prior to modifying the at least one computer instruction, pause execution of at least one process associated with the computer application. In some embodiments, after modifying the at least one computer instruction, the systems and methods may resume execution of the at least one process.

[0011] Some embodiments may remediate or prevent malicious attacks in real time until a patch is deployed. Some embodiments may provide an actionable remediation path to software vendor's developers. Some embodiments may detect heap based code and/or stack based code trampolines in real time. Some embodiments may hot-deploy lite patches (i.e., perform hot patching) without process termination. In other words, some embodiments may download and verify a lite patch, stop running all threads in the process temporarily, hot-patch a binary, and finally restart all the threads in the process (and/or application). Some embodiments may tie (i.e., associate) and track state of which patch is tied to which routine in which parent binary. Such state may include, but not be limited to, a checksum associated with the lite patch, and the address in the original binary itself. Some embodiments may untie (i.e., disassociate) a state that is presently associated with the process, either before or after deploying the patch. Some embodiments may protect a state relationship from hackers. By modifying one or more states (or associating or disassociating on or more states from a patch), some embodiments may protect a state relationship from hackers. Some embodiments include protection, including, but not limited to: (a) verification of a checksum associated with (or included in) a patch, (b) obtaining another copy of an original patch that is

deleted, and/or (c) encrypting contents of a patch, thereby preventing man-in-the-middle (MIM) attacks as well as deliberate/ accidental deletion.

[0012] In example embodiments, the systems and methods, for one or more code vulnerabilities of a computer application, map each of the code vulnerabilities to a respective system response in a table in memory (e.g., in a security policy database). In some example embodiments, at least one code vulnerability, and mapped system response, is provided from developers of the computer application. In some example embodiments, at least one of the code vulnerabilities and mapped system response is automatically determined by a code analyzer at load time or runtime of the computer application. The systems and methods next detect an event accessing a code vulnerability of the computer application.

[0013] The systems and methods, in response to the detection of the event, determine a system response mapped to the accessed code vulnerability in the table in memory. In some embodiments, an exception is triggered in response to an inappropriate operation by the application, such as inappropriately accessing the vulnerable code. The systems and methods instrument an exception handler to overwrite the kernel mode exception handler. The systems and methods intercept the triggered exception and associate the code vulnerability by an instrumented exception handler. In these embodiments, the systems and methods include querying, by the instrumented exception handler, the associated code vulnerability in the table. The querying returns the system response, configured as a system callback routine, mapped to the code vulnerability. The systems and methods execute the determined system response to prevent the event from exploiting the accessed code vulnerability. For example, the systems and methods execute the system callback routine to initiate instructions to prevent the event from exploiting the code vulnerability.

[0014] The system response (e.g., initiated instructions by the system callback routine) may include one or more of: logging the accessing of the code vulnerability as an error in a system log, dumping an image of an application process containing the accessed code vulnerability, restoring a copy of computer application prior to the accessing of the code vulnerability, dynamically loading one or more remedial patches from memory to modify at least one computer routine containing the code vulnerability, without restarting the computer application, continuing execution of the computer application until it terminates (e.g., crashes) based on the accessed code vulnerability, and terminating proactively the computer application.

BRIEF DESCRIPTION OF THE DRAWINGS

[0015] The foregoing will be apparent from the following more particular description of example embodiments of the invention, as illustrated in the accompanying drawings in which like reference characters refer to the same parts throughout the different views. The drawings are not necessarily to scale, emphasis instead being placed upon illustrating embodiments of the present invention.

[0016] FIG. 1 illustrates an example application infrastructure, according to some embodiments of the present disclosure.

[0017] FIG. 2 illustrates an example user-interaction with the application infrastructure of FIG. 1, according to some embodiments.

[0018] FIG. 3A shows an example flow chart of load time operations executed by a client.

[0019] FIG. 3B illustrates a flowchart of an example method (and system) for protection against malicious attacks that are facilitated through memory corruption, in embodiments of the present disclosure.

[0020] FIG. 3C illustrates a flowchart of an example method (and system) for protection against malicious attacks that exploit code vulnerabilities to cause memory corruption, in embodiments of the present disclosure.

[0021] FIG. 4A is a block diagram of a memory based attack, according to some embodiments.

[0022] FIG. 4B is a block diagram of a first stage of stack corruption detection functionality associated with a binary virtual patching application/probe (also known as "bvpProbe"), according to some embodiments.

[0023] FIG. 4C is a block diagram of a first stage of heap corruption detection functionality associated with a binary virtual patching application/probe (also known as "bvpProbe"), according to some embodiments.

[0024] FIG. 5A illustrates patching of an individual patch of a static or dynamic linked library (DLL) or shared object in real time, according to some embodiments.

[0025] FIG. 5B illustrates loading a patch at start up, according to some embodiments.

[0026] FIG. 5C illustrates a cycle associated with purging (or removing) one or more individual lite patches, according to some embodiments.

[0027] FIG. 6 is a block diagram of a patching timeline, according to some embodiments.

[0028] FIG. 7A illustrates an example block diagram of the application runtime monitoring and analysis solution (ARMAS) in embodiments of the present disclosure.

[0029] FIG. 7B illustrates an example protocol data unit (PDU) used to transmit data within the ARMAS infrastructure of FIG. 7A.

[0030] FIG. 8 illustrates a computer network or similar digital processing environment in which embodiments of the present disclosure may be implemented.

[0031] FIG. 9 illustrates a diagram of an example internal structure of a computer (e.g., client processor/device or server computers) in the computer system of FIG. 8.

DETAILED DESCRIPTION

[0032] A description of example embodiments of the invention follows.

In an application infrastructure of a data center for an enterprise, such as the one shown in FIG. 1, web servers receive incoming web requests (e.g., HTTP requests) from users (or from other machines through web services). However, the embodiments illustrated in FIG. 1 are not limited to a web application infrastructure or a data center, and may include, but are not limited to personal, enterprise, cloud-based and industrial control applications. In response to a web request, a web server authenticates the web service, and if the authentication is successful, establishes a session for the remote user to access information from the enterprise system (via web servers, portals, and application servers) in response to further web requests. The web service may access information from various applications, such as account services, trade finance, financial accounting, document management, and any other application without limitation, executing on computers of the enterprise system.

[0033] The web server internally maintains user and session (i.e., state) related data for the session, but does not pass on this data to the next server when forwarding an incoming request to one or more application servers for processing. That is, the user and session data is terminated at the web server of the web application infrastructure of FIG. 1 in the demilitarized zones. Then, when the application server sends the web server a response to the web request, the web server references the user and session data to determine which user (identified by their user name or IP address) to send the response. A web server may maintain the user and session data for simultaneously managing thousands of web sessions in this manner. Some of the thousands of web sessions (user and session data) may belong to

hackers that are attempting to exploit code vulnerabilities in the various applications executing on the computers of the enterprise system (and corrupt those applications).

Overview of Malware Attacks

[0034] The National Vulnerability Database (NVD) enumerated approximately 4100 application vulnerabilities in 2011 and approximately 5300 application vulnerabilities in 2012, which are divided into twenty-three attack categories. While a few of the attack categories result from negligence or misconfiguration, the largest number of attack categories involve a malicious actor purposely injecting, and later causing execution of, malicious content in an executing process of an organization. The process of injecting such malicious content involves identifying and exploiting some poorly designed code that performs inadequate input validation. For example, if the code lacks in user input size related validation, the code may allow buffer error style attacks that are included in the Buffer Errors attack category. In these attacks, the malicious actors are injecting malicious content in an attempt to infiltrate, determine content of value and then ex-filtrating such content. Malicious actors may also mutilate such content for profit. Content may include confidential information, such as credit card data, intellectual property, and social security numbers. The malicious actor may then use this confidential information to profit by selling this information to the highest bidder.

Example System Detecting Attacks

[0035] FIG. 2 illustrates example interactions within the web application infrastructure of FIG. 1. In a web application infrastructure (as shown in FIG. 2), a protected web server receives a web request (e.g., HTTP requests) from a user (via a web service client). Using information included in the web request (e.g., URL), the web server authenticates the web service user, and if the authentication is successful, establishes a connection (or session) for the web service user to access data within the web application infrastructure.

[0036] While connected to the application, an attacker may obtain access to code in memory through a trampoline. However, the application runtime monitoring and analysis (ARMAS) appliance of FIG. 2 communicates with the application to detect such memory corruption. Once such memory corruption is detected, the ARMAS appliance utilizes the shadow stack or Function Pointer Jump Table (which has stored the known pre-corruption

state) in order to temporarily restore the application to its known pre-corruption state. The unauthorized access may then be declared a security attack by a malicious adversary, which may be reported by the management server on the user interface for attack notification and corrective remedial action. Then, the ARMAS appliance, in communication with the management server, may obtain one or more patches and push the one or more patches out the affected servers, in order to remediate the corruption caused due to the hacker. In other embodiments, the ARMAS appliance may instead dynamically execute other responses at the web service, such as logging an error in relation to the corrupted memory (code instructions), dump an image of the application process containing the corrupted memory (e.g., a core dump), allow the corrupted application to continue executing until it terminates in a crash, or any other system response without limitation.

[0037] FIG. 3A shows the operations that an example client referred to herein as the Monitoring Agent (also known as "Resolve Client," according to some embodiments) may perform at load time to prepare for detecting malware activity, in accordance with principles of the present disclosure. Some embodiments may include one or more corresponding Application Maps and/or Application Map Databases as described in the "Automated Runtime Detection of Malware," PCT Application No. US2014/055469 (also PCT Publication No. WO2015/038944, filed September 12, 2014, incorporated herein by reference in its entirety), or other techniques known in the art. The Path Validation Engine is part of the Monitoring Agent that can reliably detect malware activity within microseconds from the point the malware starts to run. The Monitoring Agent first verifies the integrity and then analyzes each module of the application in order to extract a model of the application. The model of the application is stored in an Application Map Database that may contain the following tables: Code Table, Export Table, V Table, Other Table, Basic Block Table, Soft Spot Table, Memory Operand Table, Transition Table, Disassembly Table, and Critical OS Functions Table. In the embodiment in FIG. 3A, the Application Map Database is located on a remote system from the Monitoring Agent. In other embodiments, the Application Map Database can be saved on the same hardware where the application is executing or on hardware external to both the Monitoring Agent and Analysis Engine. The Monitoring Agent uses a Streaming Engine to package the extracted model of the application into Resolve Protocol Data Units (PDUs) to dispatch the data to be stored in the Application Map Database on the analysis system.

[0038] After the Monitoring Agent starts processing individual executable components of the application at load time at 302, the same operations are performed in a loop for each module of the computer application at 304 and 306. As each module of the application loads in memory, the Monitoring Agent examines all the instructions of the given module. The modules of the application file are in a standard file format, such as a Portable Executable (PE), Executable and Linkable Format (ELF) or Common Object File Format (COFF). In this format, the modules of the application are organized into sections that include a code section, exported data section, v-table section, and many other additional sections. As each module of the application loads in memory, the Monitoring Agent extracts relevant information as part of the model of the application. The bounds and access attributes of the code section of the module are dispatched and saved to the Application Map Database in the Code Table at 314. Each record in this table is of the form {Start Address, End Address}. The bounds and number of instructions of each basic block in the code section of the module are dispatched and saved in the Application Map database in the Basic Block Table at 330. Each record in this table is of the form {Start Address, End Address, and Number of instructions}. The bounds and access attributes of the exported data section of the module are saved in the Application Map database in the Export Table at 318. Each record in this table is of the form {Start Address, End Address}. The bounds and access attributes of a v-table section (if any) of the module are dispatched and saved in the Application Map database in the V Table at 322. Each record in this table is of the form {Start Address, End Address}. The bounds and access attributes of all other sections of the module are dispatched and saved in the Application Map database in the Other Table at 326. Each record in this table is of the form {Start Address, End Address, and Protection Attributes}.

[0039] As each module loads into memory, the Monitoring Agent also extracts other memory mapping data 336 and soft spot data 334 from the modules of the application. Memory mapping data includes instructions for memory allocation, memory de-allocation, and memory writes to critical segments of memory. Soft spot data includes instructions for manipulating large memory buffers (spot spots) including instructions that execute loops (such as instructions with REP style opcodes). The address of soft spot instructions and the size of each memory write are dispatched and saved in the Application Map database in the Soft Spot Table at 334. Each record in this table is of the form {Address, Write size}. The address and the write size will be stored for memory write instructions where the destination

is a memory operand. This data is stored in the Application Map Database in the Memory Operand Write Table at 340. Each record in this table is of the form {Source Address, Memory Write Size}.

[0040] As each module of the application loads into memory, the Monitoring Agent also extracts transition mapping data (branch transfer or transition data) from the module. The transition mapping data can be for a direct transition mapping where transition instructions for the target address can be presently determined or for an indirect memory mapping where transition instructions for the target address have run time dependency preventing these instructions from being fully determined until runtime. The full disassembly of instructions where indirect transitions occur are dispatched and saved in the Application Map Database in the Disassembly Table at 324. All the extracted transition mappings are also dispatched and saved in the Application Map Database in the Transition Table at 324 and 332. Each record in this table is of the form {Source Address, Destination Address}. In addition, an operator can manually add Transition Mapping Data into the Map Transition Table prior to runtime at 320. In order to add records manually into the Map Transition Table, an operator may be required to authenticate themselves using a 2-factor authentication process to eliminate possible tampering of the Transition Table by malware.

[0041] As each module of the application loads into memory, the Monitoring Agent also checks the application for integrity at 308. In one embodiment, this is accomplished by computing a checksum such as the MD5 hash of the code as it is loading and comparing it against its corresponding known good checksum saved in a Checksum database. Alternatively, a trusted checksum verification service can also be leveraged. This ensures that the code of the currently loading module is not already corrupted with malware. The Monitoring Agent may be configured to throw an alarm if the integrity check fails at 310.

[0042] At load time, particular OS functions and system calls that affect access permissions and privileges are also identified and their addresses are dispatched and saved in the Critical OS Functions Table at 312 and 316. The particular OS functions and system calls dispatched by the Monitoring Agent have long reaching effects on the execution path of the executable. These administrative and critical OS functions and system calls change access permissions of memory segments, bump up access privileges, change the no-execute policy, change the Structured Exception Handler protection, shut down the Address Space Layout

Randomization policy, allocate and de-allocate memory, create a new process, create a new thread, or are involved in encrypting and decrypting data.

[0043] As each module of the application loads into memory, the Monitoring Agent additionally instruments instructions that are inserted into the module of the application to collect data at runtime. The instrumented code is inserted into the modules of the application using a dynamic binary analysis engine and/or a byte code instrumentation engine. Soft spot instructions are instrumented in areas within the modules that malware tend to attack, such as instructions that execute loops, to collect data to track activities in these areas at runtime at 338. Direct and indirect transition mapping instructions are instrumented in the modules to collect data to track activities involving transition mappings at runtime at 328. Memory Operand Write instructions are instrumented in the modules to collect data on memory write activities at runtime at 336. In the presence of self-modifying code, the basic blocks may change at run time. Additionally, instructions are instrumented in the application to collect data for activities involving OS functions and systems calls stored in the Critical OS Function Table at 312 and 316.

[0044] As a result of the instrumentation inserted at load time, critical information is generated at run time and collected for analysis. As the transition mapping data related instrumentation is accessed, the Resolve Client collects the thread ID, current instruction address, destination instruction address and optionally data contained in each general purpose register. As the Soft Spot instrumentation is accessed before the instruction is executed, the Monitoring Agent captures the thread ID and the bounds of the stack through appropriate registers. As the soft spot instrumentation is completed, the Monitoring Agent captures the thread ID and a few general purpose registers that allow it to estimate the region of memory updated by this write operation. As the critical API or OS call instrumentation is accessed before the call is executed, the Monitoring Agent captures the thread ID, API name or System Call number and input parameters. As the critical API or OS call instrumentation is accessed after the call is executed, the Monitoring Agent captures the thread ID, API name or System Call number and return value. Instrumentation in the OS functions or system calls that allocate or de-allocate memory helps to track the regions of memory that are currently involved in the various heaps the application may have created. This memory envelope is leveraged to track the target of indirect memory writes run time in order to find if the malware wants to overrun control structures in the heap. In addition, by tracking the bounds

of basic blocks using a cache, the Analysis Engine can determine if the basic block has changed. When the determination is positive, the Basic Block Table in the model database can be updated.

Methods of Preventing Attack

[0045] FIG. 3B illustrates a flowchart of an example method (and system) 300 for protection against malicious attacks that are facilitated through memory corruption, in embodiments of the present disclosure. In some embodiments, the systems and methods may extract 382 a model of a computer application during load time. The model may include, but is not limited to, source information (e.g., source memory addresses), destination information (e.g., destination memory addresses), transitions, branch boundary information, basic block boundary information, code segment bounds, import and export table bounds, jump table bounds, or any other type of computer-routine-related information known to one skilled in the art. The systems and methods may store 384 the model of the computer application. The systems and methods may insert 386 instructions into the computer application to collect data at runtime. The systems and methods may analyze 388 the data collected at runtime against the stored model of the computer application to perform detection of one or more security events. The systems and methods may, based upon the detection of the one or more security events, modify 389 at least one computer routine associated with at least one active process associated with the computer application (i.e., insert a patch).

[0046] Embodiments of the method (and system) 300 remedy/protect against various types of attacks. Those attacks that target process memory at runtime present many serious technical and operational challenges (which are remedied by some embodiments). For example, most cyber security solutions do not have the ability to observe operations in process memory at the granularity that is required to deterministically declare if an attack is in progress or not. As a result, sophisticated memory based attacks like APTs (advanced persistent threats) can go undetected for years. Ordinarily, until a process running vulnerable code is restarted, such code cannot be swapped out for non-vulnerable code. As a result, the enterprise is forced into one of two unpalatable choices – keep running and become a target for even the most unsophisticated attacker, or reboot and suffer from discontinuity of operations and revenue. An attack that leverages sophisticated memory corruption may begin with the application's control flow being maliciously altered, such that instead of the

application's own code executing thereafter, adversary driven code begins to execute. Another outcome of application code ceding control may be that in other attack scenarios, the application may take an unhandled exception and crash. This form of attack is effectively a denial of service attack.

[0047] FIG. 3C is a flowchart of an example method (and system) 390 for protection against malicious attacks that exploit code vulnerabilities to cause memory corruption, in embodiments of the present disclosure. The method 390 creates 391 a security policy database in memory of the web service infrastructure of FIG. 2 (or other system infrastructure). The security policy database includes tables of policies, which map code vulnerabilities of a computer application to corresponding system responses. The code vulnerabilities and mapped system responses may be provided by a development team (or other individuals) associated with the computer application, or automatically or statically determined by a code analyzer or other such system tool (e.g., of the ARMAS appliance) analyzing the code of the computer application at load time or runtime. The method 390 detects 392 an event accessing vulnerable code (a code vulnerability) of the computer application. For example, the accessing of the vulnerable code may cause a memory access violation or other memory or system violation, which triggers an unhandled hardware or software exception which would normally cause the thread or process of the application executing the vulnerable code to terminate (i.e., crash). In other embodiments, the method 390 may compare a saved copy of the computer application to the presently loaded or executed computer application to detect the memory corruption due to an accessed code vulnerability. In these embodiments, the method 390 may proactively trigger a hardware or software exception. The method 390 then detects the accessing of the code vulnerability by instrumenting an exception handler that intercepts (catches) the triggered hardware or software exception and associates the code vulnerability prior to the exception causing the termination of the computer application. To do so, the method 390 overwrites the kernel mode exception handler with the instrumented exception handler so that the method 390 may take control of the computer application back from the kernel and initiate a system response to the triggered exception. Otherwise, the kernel would crash the computer application in response to the triggered exception.

[0048] The method 390 provides the instrumented exception handler to include one or more instructions that automatically query the accessed code vulnerability associated with the

exception in the security policy database in response to intercepting the exception. By the automatic query of the instrumented exception handler, the method 390 determines 393 the system response mapped to the accessed code vulnerability in a policy contained in the security policy database. The determined system response is retrieved from the policy in the form of a system callback routine. The instrumented exception handler automatically executes 394 the sysetm response (i.e., system callback routine) to prevent the exploitation of the accessed code vulnerability. The system callback routine may include, without limitation, logging the accessing of the code vulnerability as an error in a system log, dumping an image of the application thread or process containing the accessed code vulnerability, restoring a saved copy of the computer application prior to accessing the code vulnerability, dynamically loading one or more remedial patches from memory in the web service infrastructure to modify the computer routine containing the code vulnerability, continue execution of the computer application until termination results (i.e., the computer application crashes) due to the memory violation, immediately terminating the computer application, or any other system response in relation to the code vulnerability.

ARMAS Probe

[0049] An example of a malicious adversary seizing control from the application is depicted in FIG. 4A. The code (blocks of instructions) 405 of a function Foo() may include vulnerabilities that may be exploited by an attacker. As illustrated in FIG. 4A, an adversary may flood the stack memory 415 of the vulnerable function Foo(). As shown, the stack memory 415 of Foo() includes local variables, a saved base pointer, a saved return pointer 418, and parameters. When the memory location of saved return pointer of the stack memory 415 of Foo() is determined and overrun by an attacker, then when the Foo() exits (returns from the stack), the address of the next instruction may be retrieved from the saved return pointer 418 by the attacker, and the application may come under attack. Attackers may use such a technique to make a malicious movement to a different code path, such as trampoline (jump) from the application code into their own malicious code (adversary trampoline code 410).

[0050] In FIG. 4A, the target of an attack may be the Instruction Pointer Register. The attacker may also target any other register as well. Many function pointers may be declared in the application code and may be are loaded using indirect memory addresses whose value

is read from various registers. By inserting malicious data in a targeted register, an attacker may alter the program execution flow. For example, the attacker may target register EAX (accumulator register) by looking for a ROP (return oriented programming) gadget that targets an instruction in code 405 of Foo() like:

```
move rax, rsp
ret
```

[0051] This may have the effect of loading an adversary controlled value from the stack 415 into other registers such as RAX. Next, the adversary may trigger a trampoline from application code 405 of Foo() to their code 410 by executing an instruction such as "call [RAX]." Regardless of which register is targeted, the effect may be the same – adversary code may start to execute.

[0052] In response, embodiments include an application include a runtime monitoring and analysis (ARMAS) Probe/Application. This ARMAS Probe executes instrumented instructions in line in every thread of the process and collects source and destination information for branch transfer instructions, such as the function calls (e.g., move rax, rsp shown above) and return function calls (e.g., rets shown above) that the Probe encounters at runtime. The instrumented instructions then relay the highly granular information of the threads to the ARMAS Probe, which may make the determination whether a branch transfer is really a code trampoline into an adversary's code or not. Embodiments include a smarter ARMAS Probe, called the binary virtual patching application/probe (or "bvpProbe"), which overcomes the limitation of constantly reaching out to the ARMAS Appliance for confirmation.

[0053] According to some embodiments, the operation of the bvpProbe can be broken into three stages (stages 1 through 3, as illustrated to follow):

Stage 1 Functionality of the bvpProbe

[0054] The bvpProbe may protect against stack based and/or heap based attack trampolines. FIG. 4B is a block diagram of a first stage of stack corruption detection functionality associated with bvpProbe, according to some embodiments. FIG. 4C is a block diagram of a first stage of heap corruption detection functionality associated with a bvpProbe, according to some embodiments.

[0055] As illustrated in FIG. 4B, for a stack based trampoline, as a first step (step 1), some embodiments may save the intended target address 420 of the computer routine. As illustrated in FIG. 4B, in the next step (step 2), memory corruption 422 may occur by one or more attackers (e.g., as described in references to FIG. 4A). As illustrated in step 3 of FIG. 4B, some embodiments may check 424 for (detect) the memory corruption. As illustrated in step 4 if FIG. 4B, some embodiments may bypass the memory corruption by applying a copy of the saved target 426 to the target address and executing 428 the saved target. Some embodiments may handle memory corruption using one or more techniques for memory corruption handling known to those skilled in the art, including but not limited to techniques such as the "System and Methods for Run Time Detection and Correction of Memory Corruption," U.S. Patent No. 8,966,312, or other techniques known in the art. As such, some embodiments may detect a memory corruption 424 of at least one portion of memory during run-time and correct the memory corruption of the at least one portion of memory by replacing the at least one portion of memory with a backup copy of the at least one portion of memory (i.e., the copy of the saved target 426). In this way, memory corruption may be corrected in a timely fashion while minimizing security risks.

[0056] As also illustrated in FIG. 4B, for stack based trampoline, instead of simply dispatching runtime transition data to the ARMAS Appliance, the bvpProbe may save critical state at the point of entry into a routine. If the bvpProbe determines 424 that the branch transfer operation on a given thread has been maliciously altered as shown in step 3 of FIG. 4B, it can undo the memory corruption and restores 426-428 the contextually appropriate destination by restoring the saved intended target 420 back onto the stack from where the address for the next instruction to be executed is retrieved.

[0057] FIG. 4C illustrates corruption detection and handling for heap based trampolines. As illustrated in FIG. 4C, as the application runs and executes in step 1 of FIG. 4C, adversary controlled input is written to an instance (Instance 1) of ObjA in step 2 of FIG. 4C. Instance 1 of ObjA, which contains vulnerable code, was previously declared in the heap section of application memory. Ordinarily each object (e.g., ObjA and ObjM) of the application has one or more virtual function pointers (vptrs), such as Function Pointers 1,2…N,…,X, that points to the common virtual pointer table (vtable) in a read only data section. According to some embodiments, function pointers may be declared in the read only data section of the application memory. When the adversary data overwrites a vptr (which correspond to

vulnerable code of ObjA), the pointer from vptr to vtable may be prevented in step 2 of FIG. 4C. At some later point in time, when known good code accesses ObjA in step 3 of FIG. 4C, instead of executing a known good function pointer, the adversary chosen malicious code begins to execute as in step 4 of FIG. 4C. The bvpProbe detects the trampoline at step 4 of FIG. 4C by comparing the destinations (malicious code) with known good destinations.

Stage 2 Functionality of the bvpProbe

[0058] The bvpProbe may collect deep insight into the vulnerability and provide highly actionable information to the development team. Once the development team has developed and tested a non-vulnerable version of the vulnerable function, then the new code for that function may be packaged into a library and released as an individual patch ("lite" patch). Once a verifiable and signed version of the "lite" patch is made available at the ARMAS Dashboard, it can be pushed to one or more affected servers where the bvpProbe may be provisioned to run. The "lite" patch may be stored in memory at the affected servers, such that the "lite" patch may be injected, as necessary, from memory into a computer application (rather than repeatedly accessing from a file on the ARMAS Dashboard or over the network from a developer site).

[0059] On receiving the individual patch ("lite" patch), the ARMAS probe may save the configuration data for the corresponding individual patch ("lite" patch). Then, either on user command (or automatically), or in response to an error, process threads may be suspended. As illustrated in FIG. 5A, process threads may be suspended in response to a buffer error (BE) vulnerability hit.

[0060] As illustrated in FIG. 5A, in suspending the process threads, the ARMAS probe may halt one or more threads in the application using an approach including but not limited to one or more of the following mechanisms, based on the complexity of the application. The mechanism may use intrinsics provided by the hardware running the application. A first example mechanism (method) uses the SuspendThread() API. In this method, a process is running 505 and branch execution vulnerability is hit 510 by an attacker. The method suspends 515 every thread in the process recursively using the SuspendThread() API. The method then injects 520 a lite patch DLL (as described above) into the suspended thread of the process, which patches 525 the vulnerable function and updates 530 the corresponding

configuration file. Once the code has been patched, the probe can call the ResumeThread() API to resume 535 the process.

[0061] This approach may land a multithreaded application into operational issues due to timeouts getting truncated, race conditions due to processes waking up out of order and/or deadlock due to some lower priority thread acquiring a semaphore on being woken up. A second example mechanism is the use of one or more system calls (such as NtSuspendProcess()) implemented in the kernel that can suspend a process. A third example mechanism is the use of the Debugger Interface (DebugActiveProcess() to halt and DebugActiveProcessStop() to resume, in order to stop the process temporarily.

[0062] As illustrated in FIG. 5A, once the process is "frozen" (i.e., suspended 515) it is safe to inject 520 new code (i.e., inject a "lite" patch DLL and perform a "hot" patch for the vulnerable function) and create trampolines from the old vulnerable code to the new non-vulnerable code. These non-malicious trampolines may be implemented using, but not limited to using, one of the following mechanisms. A first example mechanism is to directly edit the functionality in the code segment to jump to where the new "lite" functionality is loaded in memory. The second example mechanism is useful in the case that the affected functionality is an exported functionality. The second example mechanism changes the address in the import tables so that instead of the old vulnerable code being called, the new non-vulnerable code is invoked. A third example mechanism inserts code that creates an event and tie new code to the event handler.

[0063] As illustrated in FIG. 5A, the bvpProbe may wake up the suspended process (i.e., resume 535 the process). However, as also illustrated in FIG. 5A, before waking up the suspended process, configuration information may be updated 530 in the configuration file.

[0064] The configuration information may include information for multiple patches (i.e., multiple "lite" patches). It is important to save configuration related information for the patch. This configuration information may allow the bvpProbe to tie (associate) one or more individual ("lite") patches to a specific release of an executable module. Then, until the configuration is changed, the bvpProbe may reload the one or more individual ("lite") patches on start up. The configuration file itself may be a target for malicious adversaries. Therefore, configuration file contents may be encrypted with an endpoint specific "canary" and its file integrity checksum information may be published, so that the configuration file may be verified before the bvpProbe starts to utilize it for reloading patches.

[0065] It is also important that the above-mentioned operation of virtual patching (e.g., using a "lite" patch, as illustrated in FIG. 5A) may occur without the process being terminated. This is hugely important in scenarios where the application delivers mission critical operations to the enterprise, or if the cost to patch the application is large.

[0066] However, as illustrated in FIG. 5B, if the process is restarted before the full patch (i.e., aggregate patch, which may include one or more individual or "lite" patches) is delivered, the "lite" patch insertion preferably occurs at start up. As illustrated in FIG. 5B, when a power cycle occurs, the bvpProbe application initiates the process 540, and reads 542 the configuration file to retrieve saved configuration information (including but not limited to patch information). Next, as illustrated in FIG. 5B, the bvpProbe application loads 543 one or more individual ("lite") patches onto each module of the application, in order to patch 545 potentially vulnerable functions for each module. After loading the desired "lite" patches, the bvpProbe may start running the application process.

Stage 3 of the bvpProbe

[0067] At some subsequent time, one or more users (such as a software development team) may release a full patch (i.e., aggregate patch, which may include one or more individual or "lite" patches) comprising of fixes to multiple vulnerabilities. A full patch may include one or more lite patches. At that time, one or more in-memory, "lite" patches may be explicitly removed, as illustrated in FIG. 5C. FIG. 5C illustrates a cycle associated with purging (or removing) one or more individual patches, according to some embodiments.

[0068] As illustrated in FIG. 5C, one or more communication messages (purge patch messages) may be dispatched 570 to/from the ARMAS dashboard from/to the ARMAS bvpProbe, so that the lite patch configuration information may be updated in the configuration file. The bvpProbe registers 571 the purge patch messages and update 572 the configuration file, which may include information for multiple lite patches, to reflect the most recent version(s) of lite patches that are applicable to one or more modules. In this way, the one or more versions of lite patches that are inapplicable to the one or more modules may not be loaded onto the module (i.e., "purged," or not loaded). Then, as illustrated in FIG. 5C, the bvpProbe starts 573 the process and loads 574 each module. Once the modules are loaded, the bypProbe loads 575 onto each module the applicable lite patches (but not the inapplicable lite patches), which patches 576 the corresponding potentially vulnerable functions prior to resuming the process. The bvpProbe then resumes 577 the running of the process. In this

way, some embodiments may the bvpProbe purge the lite patches for which a full patch has been released.

Virtual Patching Timeline

[0069] FIG. 6 is a block diagram of a patching timeline, as well as a summary of bvpProbe functionality (as described herein), according to some embodiments. As illustrated in the timeline of FIG. 6, after an application is deployed, a memory based attack (i.e., zero-day attack) may occur in the stack of a function, such as stack 415 of function Foo 405, as described by the memory based attack (or malicious trampoline) of FIG. 4A. As illustrated in FIG. 6, according to some embodiments, the bvpProbe (in stage 1) may inhibit the malicious trampoline (adversary trampoline code 410) that otherwise would have executed the zero-day attack. As illustrated in FIG. 6, according to some embodiments, in Stage 2, the bvpProbe may load a "lite" patch 580, and in Stage 3 the bvpProbe may revert (or remove or purge) the loaded "lite" patch 580 in lieu of a full released upgrade 582, which may require a reboot.

Virtual Patching By Exception Handling

[0070] In other embodiments, the ARMAS appliance executes a virtual patching application that can detect accessing of code vulnerabilities prior to memory corruption. In these embodiments, as development teams (e.g., application vendors) determine code vulnerabilities of an application via the use of the ARMAS application or by other means of dynamic or static code analysis (e.g., code analyzers), the code vulnerabilities of the application are configured as policies. Each configured policy includes a determined code vulnerability and a corresponding system response configured as a system callback routine. When a development team configures a policy for a determined code vulnerability, the team also programs the corresponding system callback routine to execute a recommended system response to the determined code vulnerability. The recommended system responses (programmed as callback routines) include, without limitation, logging the accessing of the code vulnerability as an error in a system log, dumping an image of the computer application thread or process containing the accessed code vulnerability, restoring a saved copy of the stack of the computer application, dynamically loading one or more remedial patches from memory in the web service infrastructure to modify the computer routine containing the code vulnerability, continue execution of the computter application until it crashes due to the

memory violation, immediately terminating the computer application, or any other sysetm response in relation to the code vulnerability. The team may also configure a default policy for the computer application with a callback routine programmed to execute one of the above recommended system responses. The configured policies are stored in tables of a security policy database that is stored at a network location accessible to the ARMAS appliance.

[0071] The ARMAS appliance instruments a hardware or software exception handler at the application servers of the network. When an unhandled memory access violation, or other unhandled memory or system violation, occurs because of an event (e.g., web service request) accesses vulnerable application code, the hardware or software (e.g., operating system) executing the application triggers an exception. The hardware or software exception handler instrumented by the ARMAS appliance intercepts the triggered exception, and associates the code vulnerability, and queries the security policy database for a policy containing the associated code vulnerability. To do so, the ARMAS appliance overwrites the kernel mode exception handler with the instrumented exception handle to take back control of the application from the kernel in order to initiate a system response to the triggered exception. Otherwise, the kernel would crash the application in response to the triggered exception. If a corresponding policy is located in the security policy database, the instrumented exception handler executes the callback routine of the policy in response to the accessing of the associated code vulnerability. If no corresponding policy is located, the instrumented exception handler executes the callback routine from the default policy for the computer application in the security policy database.

[0072] The callback routine functions as a virtual patch for responding to the vulnerability in the accessed code until a full patch, provided by the development team of the computer application, is available for download to the application server. Below is an example of the exception handle code. Option 5, shown in the example, may comprise injecting a “lite” patch saved in memory as describe above in reference to FIGs. 5A-5C.

```
catch (Exception e)
{
    virtual patch = Retrieve programmable callback from Security Policy
                    database routine for code associated with unhandled
                    memory violation;
```

```
Execute virtual patch;
        // Executing virtual patch may comprises one of the following
    examples:
            // Option 1: Console.WriteLine("An error occurred: '{0}'", e);
            // Option 2: Save process image to disk (core dump);
            // Option 3: Restore Stack and move on;
            // Option 4: Let process continue and crash;
            // Option 5: Patch process without restarting process
            // .
            // .
            // .
            // Option x (without limitation)

        }
```

Application Runtime Monitoring and Analysis (ARMAS) Infrastructure

[0073] FIG. 7A depicts a high level block diagram of an example application runtime monitoring and analysis (ARMAS) infrastructure. This infrastructure may be configured on a various hardware including computing devices ranging from smartphones, tablets, laptops, desktops to high end servers. As shown in this figure, data collection performed by the Monitoring Agent 702 may be segregated from analysis performed by the Analysis Engine 737 to improve application performance. The infrastructure provides high availability to prevent hackers from subverting its protection against malware attacks. The Monitoring Agent 702 interacts with an application to gather load time and runtime data. The infrastructure of the application 701 includes process memory 703, third-party libraries 704, kernel services 706, and an instruction pipeline 707. The infrastructure of the Monitoring Agent 702 includes the Instrumentation & Analysis Engine (instrumentation engine) 705, graphical user interface (GUI) 711, Client Daemon 708, Configuration database 709, and Streaming and Compression Engine 710, and central processing unit (CPU) 736. Local or remote users 750 of the application 701 interact with the application either through devices like keyboards, mice or similar I/O devices or over a network through a communication channel that may be established by means of pipes, shared memory or sockets. In response

the application process 703 dispatches appropriate sets of instructions into the instruction pipeline 707 for execution. The application may also leverage its own or third party libraries 704 such as libc.so (Linux) or msvcrtxx.dll (Windows). As functionality from these libraries is invoked, appropriate instructions from these libraries are also inserted into the instruction pipeline for execution 707. In addition the application may leverage system resources such as memory, file I/O etc. from the kernel 706. These sequences of instructions from the application, libraries and the kernel put together in a time ordered sequence deliver the application functionality desired by a given user.

[0074] As the application's code begins to load into memory, the instrumentation engine 705 performs several different load time actions. Once all the modules have loaded up, the instrumented instructions of the application generate runtime data. The Client Daemon 708 initializes the Instrumentation and Analysis Engine 705, the Streaming Engine 710 and the GUI 711 processes in the CPU at 736 by reading one or more configuration files from the Configuration database 709. It also initializes intercommunication pipes between the instrumentation engine, Streaming Engine, GUI, Analysis Engine 737 and itself. The Client Daemon also ensures that if any Monitoring Agent 702 process, including itself, becomes unresponsive or dies, it will be regenerated. This ensures that the Monitoring Agent 702 is a high availability enterprise grade product.

[0075] The Instrumentation and Analysis Engine 737 pushes load and runtime data collected from the application into the Streaming Engine. The Streaming Engine packages the raw data from the Monitoring Agent 702 into the PDU. Then it pushes the PDU over a high bandwidth, low latency communication channel 712 to the Analysis Engine 737. If the Monitoring Agent 702 and the Analysis Engine 737 are located on the same machine this channel can be a memory bus. If these entities are located on different hardware but in the same physical vicinity, the channel can be an Ethernet or Fiber based transport, which allows remote connections to be established between the entities to transport the load and runtime data across the Internet.

[0076] The infrastructure of the Analysis Engine 737 includes the Network Interface Card (NIC) 713, the Packet Pool 714, the Time Stamp Engine 715, the Processor Fabric 716, the Hashing Engine 717, the TCAM Engine 718, the Application Map database 719, and the Thread Context database 720, which makes up the REGEX Engine 740. The infrastructure of the Analysis Engine 737 further includes the Content Analysis Engine 721, the Events and

Event Chains 722, the Event Management Engine 723, the Event Log 724, the Application Daemon 725, the Analysis Engine Configuration database 726, the Network Interface 727, the Dashboard or CMS 728, the SMS/SMTP Server 729, the OTP Server 730, the Upgrade Client 731, the Software Upgrade Server 732, Software Images 733, the Event Update Client 734, and the Event Upgrade Server 735.

[0077] The PDU together with the protocol headers is intercepted at the Network Interface Card 713 from where the PDU is pulled and put into the Packet Pool 714. The timestamp fields in the PDU are filled up by the Time Stamp Engine 715. This helps to make sure that no packet is stuck in the packet Pool buffer for an inordinately long time.

[0078] The Processor Fabric 716 pulls packets from the packet buffer and the address fields are hashed and replaced in the appropriate location in the packet. This operation is performed by the Hashing Engine 717. Then the Processor Fabric starts removing packets from the packet buffer in the order they arrived. Packets with information from the load time phase are processed such that the relevant data is extracted and stored in the Application Map database 719. Packets with information from the runtime phase are processed in accordance with Figure 5. The efficiency of the Analysis Engine 737 can be increased or decreased based on the number of processors in the Processor Fabric.

[0079] The transition target data is saved in the Thread Context database 720 which has a table for each thread. The Processor fabric also leverages the TCAM Engine 718 to perform transition and memory region searches. Since the processor fabric performing lookups using hashes, the actual time used is predictable and very short. By choosing the number of processors in the fabric carefully, per packet throughput can be suitable altered.

[0080] When the Analysis Engine 737 performs searches, it may, from time to time find an invalid transition, invalid operation of critical/admin functions or system calls, or find a memory write on undesirable locations. In each of these cases, the Analysis Engine 737 dispatches an event of the programmed severity as described by the policy stored in the Event and Event Chain database 722 to the Event Management Engine 723. The raw event log is stored in the Event Log Database 724. The Dashboard can also access the Event Log and display application status.

[0081] A remedial action is also associated with every event in the Event and Event Chain database 722. A user can set the remedial action from a range of actions from ignoring the event in one extreme to terminating the thread in the other extreme. A recommended

remedial action can be recommended to the analyst using the Event Update Client 734 and Event Upgrade Server 735. In order to change the aforementioned recommended action, an analyst can use the Dashboard 728 accordingly. The Dashboard provides a GUI interface that displays the state of each monitored application and allows a security analyst to have certain control over the application, such as starting and stopping the application. When an event is generated, the Event Chain advances from the normal state to a subsequent state. The remedial action associated with the new state can be taken. If the remedial action involves a non-ignore action, a notification is sent to the Security Analyst using and SMS or SMTP Server 729. The SMS/ SMTP address of the security analyst can be determined using an LDAP or other directory protocol. The process of starting or stopping an application from the Dashboard requires elevated privileges so the security analyst must authenticate using an OTP Server 730.

[0082] New events can also be created and linked into the Event and Event Chain database 722 with a severity and remedial action recommended to the analyst. This allows unique events and event chains for a new attack at one installation to be dispatched to other installations. For this purpose, all new events and event chains are loaded into the Event Upgrade Server 735. The Event Update Client 734 periodically connects and authenticates to the Event Upgrade Server 735 to retrieve new events and event chains. The Event Update Client then loads these new events and event chains into the Events and Events Chain database 722. The Content Analysis Engine 721 can start tracking the application for the new attacks encapsulated into the new event chains.

[0083] Just as with the Client Daemon, the Appliance Daemon 725 is responsible for starting the various processes that run on the Analysis Engine 737. For this purpose, it must read configuration information from the Analysis Engine Configuration database 726. The daemon is also responsible for running a heartbeat poll for all processes in the Analysis Engine 737. This ensures that all the devices in the Analysis Engine 373 ecosystem are in top working condition at all times. Loss of three consecutive heartbeats suggests that the targeted process is not responding. If any process has exited prematurely, the daemon will revive that process including itself.

[0084] From time to time, the software may be upgraded in the Appliance host, or of the Analysis Engine 737 or of the Client for purposes such as fixing errors in the software. For this purpose, the Upgrade Client 731 constantly checks with the Software Upgrade Server

732 where the latest software is available. If the client finds that the entities in the Analysis Engine 737 or the Client are running an older image, it will allow the analysts to upgrade the old image with a new image from the Software Upgrade Server 732. New images are bundled together as a system image 733. This makes it possible to provision the appliance or the host with tested compatible images. If one of the images of a subsystem in the Analysis Engine 737 or the Monitoring Agent 702 does not match the image for the same component in the System image, then all images will be rolled to a previous known good system image.

PDU for ARMAS Communications

[0085] FIG. 7B illustrates an example protocol data unit (PDU) used to transmit data between the Monitoring Agent 702 and Analysis Engine 737 of FIG. 7A. In order for the Monitoring Agent 702 and the Analysis Engine 737 to work effectively with each other, they communicate with each other using the PDU. The PDU can specifically be used by the Monitoring Agent 702 to package the extracted model of the application and/or collected runtime data for transmission to the Analysis Engine 737. The PDU contains fields for each type of information to be transmitted between the Monitoring Agent 702 and the Analysis Engine 737. The PDU is divided into the Application Provided Data Section, the HW/CVE Generated, and Content Analysis Engine or Raw Data sections.

[0086] The Application Provided Data Section contains data from various registers as well as source and target addresses that are placed in the various fields of this section. The Protocol Version contains the version number of the PDU 752. As the protocol version changes over time, the source and destination must be capable of continuing to communicate with each other. This 8 bit field describes the version number of the packet as generated by the source entity. A presently unused reserved field 756 follows the Protocol Version field.

[0087] The next field of the Application Provided Data Section is the Message Source/Destination Identifiers 757, 753, and 754 are used to exchange traffic within the Analysis Engine infrastructure as shown in FIG. 7A. From time to time, the various entities shown in FIG. 7, exchange traffic between themselves. Not all these devices have or need IP addresses and therefore, the two (hardware and host) Query Router Engines uses the Message Source and Destination fields to route traffic internally. Some messages need to go across the network to entities in the Analysis Engine 737. For this purpose, the entities are assigned the following IDs. A given Analysis Engine appliance may have more than one accelerator card.

Each card will have a unique IP address; therefore, the various entities will have a unique ID. The aforementioned infrastructure may also be running more than one application. Since each application server will have a unique IP address, the corresponding Monitoring Agent side entity will also have a unique ID.

Monitoring Agent Side Entities

1. GUI
2. Instrumentation and Analysis Engine
3. Client Message Router
4. Streaming Engine
5. Client Side Daemon
6. CLI Engine
7. Client Watchdog
8. Client Compression Block
9. Client iWarp Ethernet Driver (100 Mb/1Gb/10Gb)

Per PCI Card Entities (starting address = 20 + n*20)

20. Securalyzer TOE block
21. Securalyzer PCI Bridge
22. Decompression Block
23. Message Verification Block
24. Packet Hashing Block
25. Time-Stamping Block
26. Message Timeout Timer Block
27. Statistics Counter Block
28. Securalyzer Query Router Engine
29. Securalyzer Assist

Securalyzer Host Entities

200. Securalyzer PCIe Driver
201. Host Routing Engine
202. Content Analysis Engine
203. Log Manager

204. Daemon
205. Web Engine
206. Watchdog
207. IPC Messaging Bus
208. Configuration Database
209. Log Database

SIEM Connectors

220. SIEM Connector 1 – Virsec Dashboard
221. SIEM Connector 2 – HP ArcSight
222. SIEM Connector 3 – IBM QRadar
223. SIEM Connector 4 – Alien Vault USM

Securalyzer Infrastructure Entities

230. Virsec dashboard
231. SMTP Server
232. LDAP Server
233. SMS Server
234. Entitlement Server
235. Database Backup Server
236. OTP Client
237. OTP Server
238. Checksum Server
239. Ticketing Server
240. Virsec Rules Server
241. Virsec Update Server

All user applications

255. User Applications – Application PID is used to identify the application issuing a query

[0088] Another field of the Application Provided Data section is the Message Type field which indicates the type of data being transmitted 755. At the highest level, there are three

distinct types of messages that flow between the various local Monitoring Agent side entities, between the Analysis Engine appliance side entities and between Client side and appliance side entities. Furthermore, messages that need to travel over a network must conform to the OSI model and other protocols.

[0089] The following field of the Application Provided Data section is the Packet Sequence Number field containing the sequence identifier for the packet 779. The Streaming Engine will perform error recovery on lost packets. For this purpose it needs to identify the packet uniquely. An incrementing signed 64 bit packet sequence number is inserted by the Streaming Engine and simply passes through the remaining Analysis Engine infrastructure. If the sequence number wraps at the 64 bit boundary, it may restart at 0. In the case of non-application packets such as heartbeat or log message etc., the packet sequence number may be -1.

[0090] The Application Provided Data section also contains the Canary Message field contains a canary used for encryption purposes 761. The Monitoring Agent 702 and the Analysis Engine 737 know how to compute the Canary from some common information but of a fresh nature such as the Application Launch time, PID, the license string, and an authorized user name.

[0091] The Application Provided Data section additionally contains generic fields that are used in all messages. The Application Source Instruction Address 780, Application Destination Instruction Address 758, Memory Start Address Pointer 759, Memory End Address Pointer 760, Application PID 762, Thread ID 763, Analysis Engine Arrival Timestamp 764, and Analysis Engine Departure Timestamp 765 fields which hold general application data.

[0092] The PDU also contains the HW/CAE Generated section. In order to facilitate analysis and to maintain a fixed time budget, the Analysis Engine 737 hashes the source and destination address fields and updates the PDU prior to processing. The HW/ CAE Generated section of the PDU is where the hashed data is placed for later use. This section includes the Hashed Application Source Instruction Address 766, Hash Application Destination Instruction Address 767, Hashed Memory Start Address 768, and Hashed Memory End Address 769 fields. The HW/CAW Generated section additionally contains other fields related to the Canary 771 including the Hardcoded Content Start Magic header, API Name

Magic Header, Call Context Magic Header and Call Raw Data Magic Header are present in all PDU packets.

[0093] The HW/CAW Generated section also includes a field 770 to identify other configuration and error data which includes Result, Configuration Bits, Operating Mode, Error Code, and Operating Modes data. The Result part of the field is segmented to return Boolean results for the different Analysis Engine queries – the transition playbook, the code layout, the Memory (Stack or Heap) Overrun, and the Deep Inspection queries. The Configuration Bits part of the field indicates when a Compression Flag, Demo Flag, or Co-located Flag is set. The presence of the flag in this field indicates to the Analysis Engine 737 whether the packet should be returned in compression mode. The Demo Flag indicates that system is in demo mode because there is no valid license for the system. In this mode, logs and events will not be available in their entirety. The Co-located Flag indicates that the application is being run in the Analysis Engine 737 so that Host Query Router Engine can determine where to send packets that need to return to the Application. If this flag is set, the packets are sent via the PCI Bridge, otherwise they are sent over the Ethernet interface on the PCI card. The Operating Mode part of the field indicates whether the system is in Paranoid, Monitor, or Learn mode. These modes will be discussed in more details later in this section. Lastly, the Error Code part of the field indicates an error in the system. The first eight bits of the error code will correspond to the message source. The remaining 12 bits will correspond to the actual error reported by each subsystem.

[0094] The PDU also contains the Content Analysis Engine or Raw Data. All variable data such as arguments and return value of the OS library calls and System Calls is placed in this section of the PDU. The data in this section contains the content of the data collected from the application and is primarily targeted at the Content Analysis Engine 721. This section contains the Variable Sized API Name or Number 772, the Call Content Magic Header 777, the Variable Sized Call Content 774, the Call Raw Data Magic Header 778, Variable Sized Raw Data Contents 776, and two reserved 773 and 775 fields. Furthermore, these fields can be overloaded for management messages.

Digital Processing Infrastructure

[0095] FIG. 8 illustrates a computer network or similar digital processing environment in which embodiments of the present disclosure may be implemented.

[0096] Client computer(s)/devices 50 and server computer(s) 60 provide processing, storage, and input/output devices executing application programs and the like. The client computer(s)/devices 50 can also be linked through communications network 70 to other computing devices, including other client devices/processes 50 and server computer(s) 60. The communications network 70 can be part of a remote access network, a global network (e.g., the Internet), a worldwide collection of computers, local area or wide area networks, and gateways that currently use respective protocols (TCP/IP, Bluetooth®, etc.) to communicate with one another. Other electronic device/computer network architectures are suitable.

[0097] Client computers/devices 50 may be configured as the security monitoring agent. Server computers 60 may be configured as the analysis engine which communicates with client devices (i.e., security monitoring agent) 50 for detecting database injection attacks. The server computers 60 may not be separate server computers but part of cloud network 70. In some embodiments, the server computer (e.g., analysis engine) may analyze a set of computer routines, identify one or more patches to be applied, and apply one or more patches to the computer routines. The client (security monitoring agent) 50 may communicate patches and patch requests, to/from the server (analysis engine) 60. In some embodiments, the client 50 may include client applications or components (e.g., instrumentation engine) executing on the client (i.e., security monitoring agent) 50 for capturing requests and queries, and detecting corrupted memory for which patches are required, as well as providing patches, and the client 50 may communicate this information to the server (e.g., analysis engine) 60.

[0098] FIG. 9 is a diagram of an example internal structure of a computer (e.g., client processor/device 50 or server computers 60) in the computer system of FIG. 8. Each computer 50, 60 contains a system bus 79, where a bus is a set of hardware lines used for data transfer among the components of a computer or processing system. The system bus 79 is essentially a shared conduit that connects different elements of a computer system (e.g., processor, disk storage, memory, input/output ports, network ports, etc.) that enables the transfer of information between the elements. Attached to the system bus 79 is an I/O device interface 82 for connecting various input and output devices (e.g., keyboard, mouse, displays, printers, speakers, etc.) to the computer 50, 60. A network interface 86 allows the computer to connect to various other devices attached to a network (e.g., network 70 of FIG. 8). Memory 90 provides volatile storage for computer software instructions 92 and data 94 used

to implement an embodiment of the present disclosure (e.g., security monitoring agent, instrumentation engine, and analysis engine elements described herein). Disk storage 95 provides non-volatile storage for computer software instructions 92 and data 94 used to implement an embodiment of the present disclosure. A central processor unit 84 is also attached to the system bus 79 and provides for the execution of computer instructions.

[0099] Embodiments or aspects thereof may be implemented in the form of hardware including but not limited to hardware circuitry, firmware, or software. If implemented in software, the software may be stored on any non-transient computer readable medium that is configured to enable a processor to load the software or subsets of instructions thereof. The processor then executes the instructions and is configured to operate or cause an apparatus to operate in a manner as described herein.

[00100] Some embodiments may transform the behavior and/or data of a set of computer routines by asynchronously and dynamically manipulating at least one of the computer routines through patch updates. The patch may include (but is not limited to) modification of a value, input parameter, return value, or code body associated with one or more of the computer routines, thereby transforming the behavior (and/or data) of the computer routine.

[00101] Some embodiments may provide functional improvements to the quality of computer applications, computer program functionality, and/or computer code by detecting malicious handling of computer routines and/or vulnerabilities in the computer applications and/or computer code. Some embodiments may deploy one or more patches to correct and/or replace improperly executing computer routines to avoid the unexpected and/or incorrect behavior. As such, some embodiments may detect and correct computer code functionality, thereby providing a substantial functional improvement.

[00102] Some embodiments solve a technical problem (thereby providing a technical effect) by improving the robustness of functionality of software and its error handling functionality. Some embodiments also solve a technical problem of detecting and remediating code corruption that may be hard to do using existing approaches (thereby providing a technical effect).

[00103] Further, hardware, firmware, software, routines, or instructions may be described herein as performing certain actions and/or functions of the data processors. However, it should be appreciated that such descriptions contained herein are merely for convenience and

that such actions in fact result from computing devices, processors, controllers, or other devices executing the firmware, software, routines, instructions, etc.

[00104] It should be understood that the flow diagrams, block diagrams, and network diagrams may include more or fewer elements, be arranged differently, or be represented differently. But it further should be understood that certain implementations may dictate the block and network diagrams and the number of block and network diagrams illustrating the execution of the embodiments be implemented in a particular way.

[00105] Accordingly, further embodiments may also be implemented in a variety of computer architectures, physical, virtual, cloud computers, and/or some combination thereof, and, thus, the data processors described herein are intended for purposes of illustration only and not as a limitation of the embodiments.

[00106] While this disclosure has been particularly shown and described with references to example embodiments thereof, it will be understood by those skilled in the art that various changes in form and details may be made therein without departing from the scope of the disclosure encompassed by the appended claims.

CLAIMS

What is claimed is:

1. A computer-implemented method comprising:

extracting a model of a computer application during load time;

storing the model of the computer application;

inserting instructions into the computer application to collect data at runtime;

analyzing the data collected at runtime against the stored model of the computer application to perform detection of one or more security events; and

based upon the detection of the one or more security events, modifying, in a manner that preserves continued execution of the computer application, at least one computer routine associated with at least one active process associated with the computer application.

2. The method of Claim 1 wherein the computer routine is executed in association with the at least one process.

3. The method of Claim 1 wherein the one or more detected security events is associated with a malicious movement to a different code path within the computer application.

4. The method of Claim 1 wherein modifying includes verifying a patch or configuration associated with the computer application.

5. The method of Claim 1 further comprising:

in response to receipt of one or more aggregate patches by a user, performing at least one of:

modifying or removing the at least one computer routine associated with the computer application; and

modifying or removing one or more individual patches associated with the computer application.

6. The method of Claim 1 further comprising:

modifying one or more stacks associated with the at least one computer routine.

7. The method of Claim 1 further comprising:

modifying one or more heaps associated with the at least one executing computer routine.

8. The method of Claim 1 further comprising:

modifying the at least one computer routine associated with the at least one process, while the at least one active process is executing the at least one computer routine.

9. The method of Claim 1 further comprising:

prior to modifying the at least one computer routine, pausing execution of at least one active process; and

after modifying the at least one computer routine, resuming execution of the at least one active process.

10. A computer system comprising:

an instrumentation engine configured to:

extract a model of a computer application during load time;

store the model of the computer application;

insert instructions into the computer application to collect data at runtime; and

an analysis engine configured to:

analyze the data collected at runtime against the stored model of the computer application to perform detection of one or more security events; and

based upon the detection of the one or more security events, modifying, in a manner that preserves continued execution of the computer application, at least one computer routine associated with at least one active process associated with the computer application.

11. The system of Claim 10 wherein the computer routine is executed in association with the at least one process.

12. The system of Claim 10 wherein the one or more detected security events is associated with a malicious movement to a different code path within the computer application.

13. The system of Claim 10 wherein the analysis engine is further configured to verify a patch or configuration associated with the computer application.

14. The system of Claim 10, wherein the analysis engine is further configured to:

in response to receipt of one or more aggregate patches by a user, perform at least one of:

modifying or removing the at least one computer routine associated with the computer application; and

modifying or removing one or more individual patches associated with the computer application.

15. The system of Claim 10, wherein the analysis engine is further configured to:

modify one or more stacks associated with the at least one computer routine.

16. The system of Claim 10, wherein the analysis engine is further configured to:

modify one or more heaps associated with the at least one executing computer routine.

17. The system of Claim 10, wherein the analysis engine is further configured to:

modify the at least one computer routine associated with the at least one process, while the at least one active process is executing the at least one computer routine.

18. The system of Claim 10, wherein the analysis engine is further configured to:

prior to modifying the at least one computer routine, pause execution of at least one active process; and

after modify the at least one computer routine, resuming execution of the at least one active process.

19. A computer-implemented method comprising:

extracting a model of a computer application during load time;

storing the model of the computer application;

inserting instructions into the computer application to collect data at runtime;

analyzing the data collected at runtime against the stored model of the computer application to perform detection of one or more security events;

upon the detection of the one or more security events, temporarily remediating memory corruption associated with the computer application prior to executing one or more return instructions;

reporting actionable information based upon the one or more detected security events; and

based upon the detection of the one or more security events, modifying, in a manner that preserves continued execution of the computer application, at least one computer routine associated with at least one active process associated with the computer application.

20. The method of Claim 19 further comprising:

modifying at least one computer instruction associated with at least one process, while the at least one process is executing.

21. The method of Claim 19 further comprising:

prior to modifying the at least one computer routine, pausing execution of at least one process associated with the computer application; and

after modifying the at least one computer routine, resuming execution of the at least one process.

22. A computer system comprising:

an instrumentation engine configured to:

extract a model of a computer application during load time;

store the model of the computer application;

insert instructions into the computer application to collect data at runtime; and

an analysis engine configured to:

analyze the data collected at runtime against the stored model of the computer application to perform detection of one or more security events;

upon the detection of the one or more security events, temporarily remediate memory corruption associated with the computer application prior to executing one or more return instructions;

report actionable information based upon the one or more detected security events; and

based upon the detection of the one or more security events, modify, in a manner that preserves continued execution of the computer application, at least one computer routine associated with at least one active process associated with the computer application.

23. The system of Claim 22, wherein the analysis engine is further configured to:

modify at least one computer instruction associated with at least one process, while the at least one process is executing.

24. The system of Claim 22, wherein the analysis engine is further configured to:

prior to modifying the at least one computer routine, pause execution of at least one process associated with the computer application; and

after modifying the at least one computer routine, resume execution of the at least one process.

25. A computer-implemented method comprising:

for one or more code vulnerabilities of a computer application, mapping each code vulnerability to a respective system response in a table in memory;

detecting an event accessing a code vulnerability of the computer application;

in response to the detection of the event, dynamically determining a system response mapped to the accessed code vulnerability in the table in memory; and

executing the determined system response, the executing prevents the event from exploiting the accessed code vulnerability.

26. The method of Claim 25, wherein at least one code vulnerability and mapped system response is provided from developers of the computer application.

27. The method of Claim 25, wherein at least one code vulnerability and mapped system response is automatically determined by a code analyzer at load time or runtime of the computer application.

28. The method of Claim 25, wherein the system response comprises a system callback routine programmable by a system or user.

29. The method of Claim 28, further comprising:

instrumenting an exception handler to overwrite a kernel mode exception handler;

triggering an exception in response to the accessing of the code vulnerability;

intercepting the triggered exception and associating the code vulnerability at the instrumented exception handler;

querying, by the instrumented exception handler, the associated code vulnerability in the table in memory, the querying returning the system callback routine mapped to the code vulnerability; and

executing, by the instrumented exception handler, the system callback routine to initiate instructions to prevent the event from exploiting the code vulnerability.

30. The method of Claim 25, wherein the system response includes one or more of:

logging the accessing of the code vulnerability as an error in a system log;

dumping an image of an application process containing the accessed code vulnerability;

restoring a copy of the computer application prior to the accessing of the code vulnerability;

dynamically loading one or more remedial patches from memory, the loading using the remedial patch to modify at least one computer routine containing the code vulnerability, the loading modifying the at least one computer routine without restarting the computer application;

continuing execution of the computer application until termination results based on the accessed code vulnerability; and

terminating proactively the computer application.

31. The method of Claim 30, wherein dynamically loading includes injecting the remedial patches directly from memory of a server executing the computer application.

32. A computer system comprising:

an instrumentation engine configured to:

for one or more code vulnerabilities of a computer application, map each code vulnerability to a respective system response in a table in memory; and

an analysis engine configured to:

detect an event accessing a code vulnerability of the computer application;

in response to the detection of the event, dynamically determine a system response mapped to the accessed code vulnerability in the table in memory; and

execute the determined system response, the executing prevents the event from exploiting the accessed code vulnerability.

33. The system of Claim 32, wherein at least one code vulnerability and mapped system response is provided from developers of the computer application.

34. The system of Claim 32, wherein at least one code vulnerability and mapped system response is automatically determined by a code analyzer at load time or runtime of the computer application.

35. The system of Claim 32, wherein the system response comprises a system callback routine programmable by a system or user.

36. The system of Claim 35, wherein the analysis engine is further configured to:

instrument an exception handler to overwrite a kernel mode exception handler;

trigger an exception in response to the accessing of the code vulnerability;

intercept the triggered exception and associate the code vulnerability at the instrumented exception handler;

query, by the instrumented exception handler, the associated code vulnerability in the table, the querying returning the system callback routine mapped to the code vulnerability; and

execute, by the instrumented exception handler, the system callback routine to initiate instructions to prevent the event from exploiting the code vulnerability.

37. The system of Claim 32, wherein the system response includes one or more of:

logging the accessing of the code vulnerability as an error in a system log;

dumping an image of an application process containing the accessed code vulnerability;

restoring a copy of computer application prior to the accessing of the code vulnerability;

dynamically loading one or more remedial patches from memory, the loading using the remedial patch to modify at least one computer routine containing the code vulnerability, the loading modifying the at least one computer routine without restarting the computer application;

continuing execution of the computer application until termination results based on the accessed code vulnerability; and

terminating proactively the computer application.

38. The system of Claim 37, wherein the dynamically loading includes injecting the remedial patches directly from memory of a server executing the computer application.

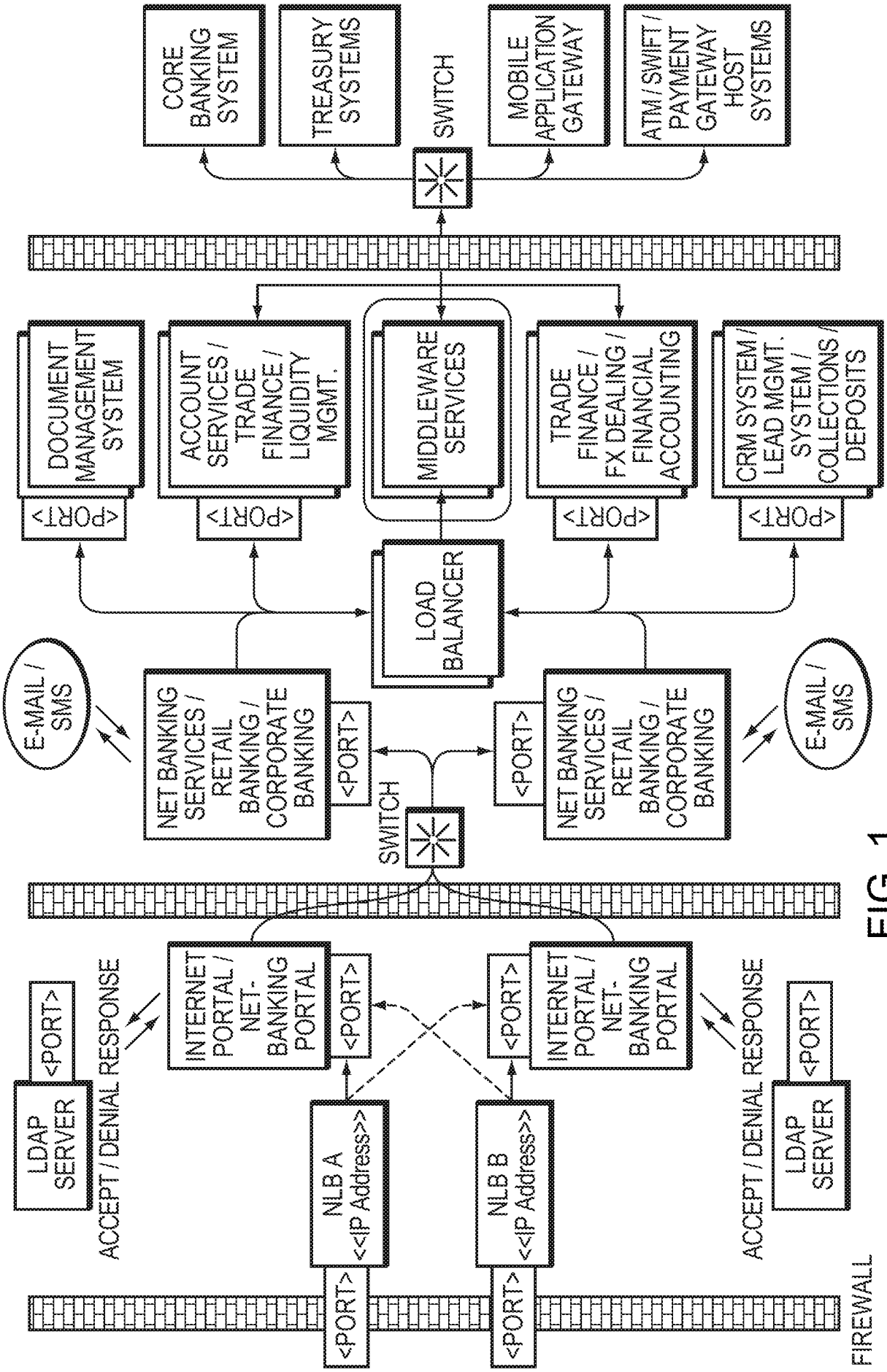
CORE BANKING SYSTEM
TREASURY SYSTEMS
SWITCH
MOBILE APPLICATION GATEWAY
ATM / SWIFT / PAYMENT GATEWAY HOST SYSTEMS
DOCUMENT MANAGEMENT SYSTEM
ACCOUNT SERVICES / TRADE FINANCE / LIQUIDITY MGMT.
MIDDLEWARE SERVICES
TRADE FINANCE / FX DEALING / FINANCIAL ACCOUNTING
CRM SYSTEM / LEAD MGMT. SYSTEM / COLLECTIONS / DEPOSITS
<PORT>
LOAD BALANCER
E-MAIL / SMS
NET BANKING SERVICES / RETAIL BANKING / CORPORATE BANKING
SWITCH
INTERNET PORTAL / NET-BANKING PORTAL
LDAP SERVER
ACCEPT / DENIAL RESPONSE
NLB A <<IP Address>>
NLB B <<IP Address>>
FIREWALL

FIG. 1

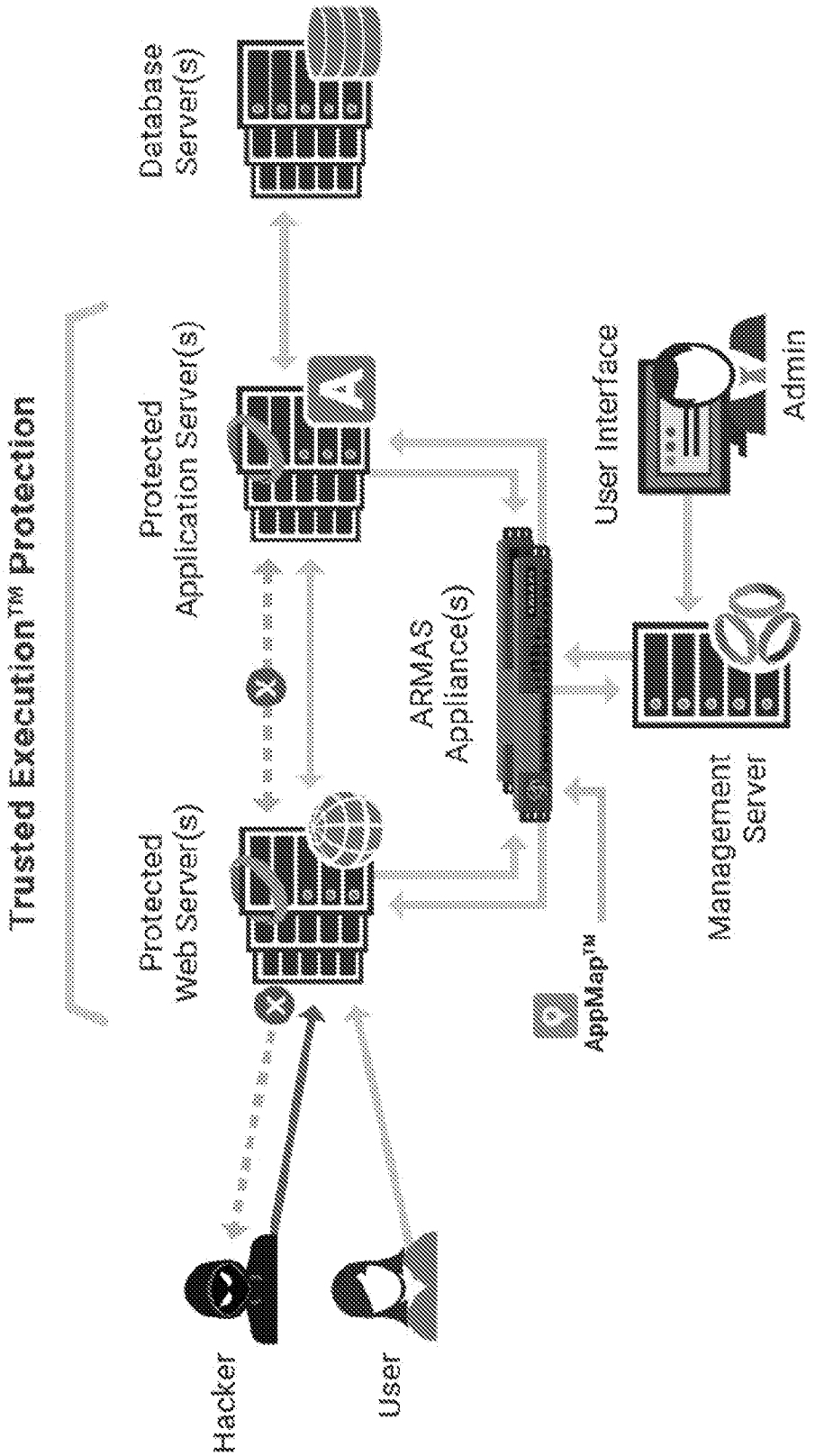
Trusted Execution™ Protection
Database Server(s)
Protected Application Server(s)
Protected Web Server(s)
ARMAS Appliance(s)
User Interface
Admin
Management Server
AppMap™
Hacker
User

FIG. 2

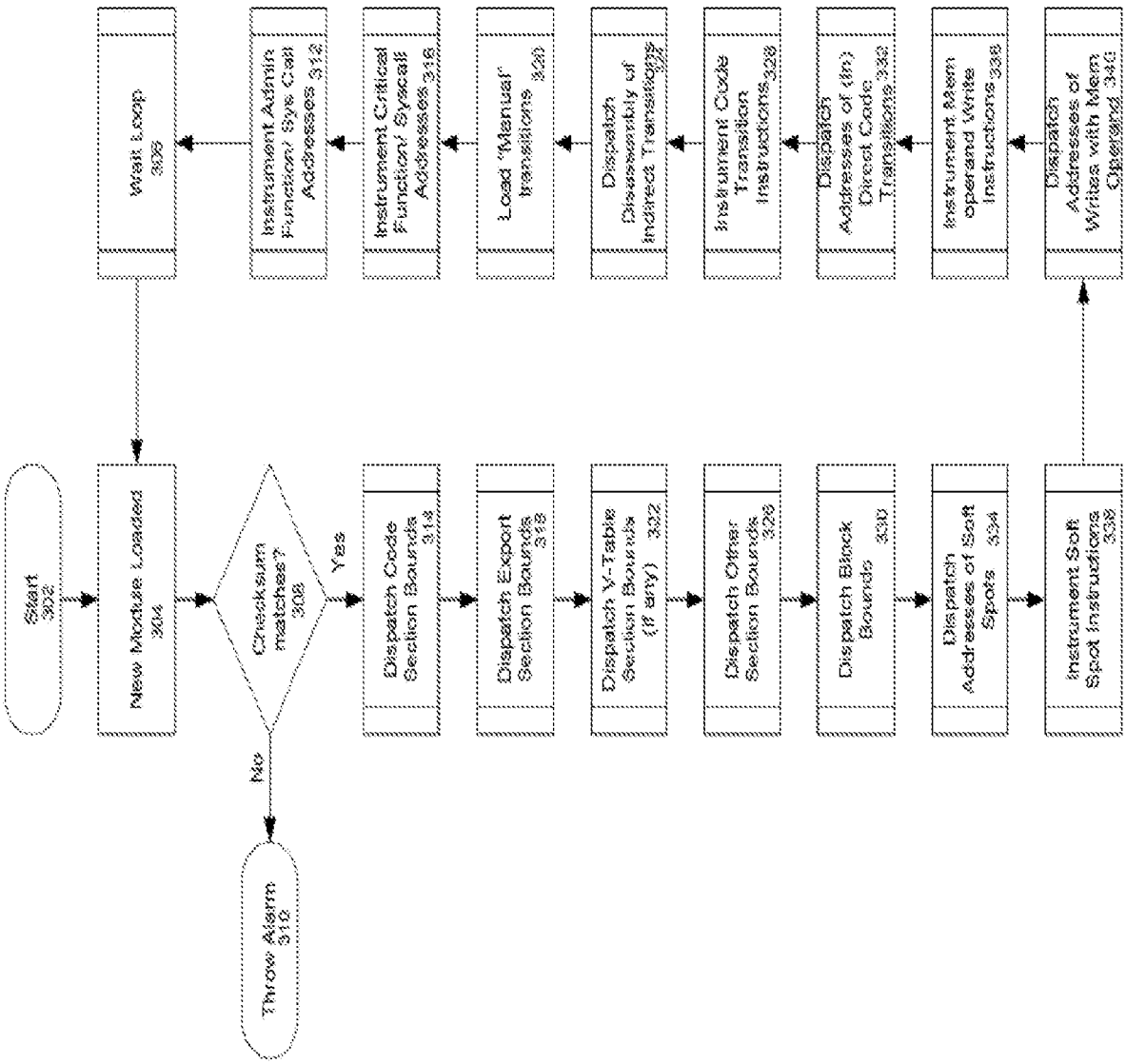
Start 302
New Module Loaded 304
Checksum matches? 306
Yes
No
Throw Alarm 310
Dispatch Code Section Bounds 314
Dispatch Export Section Bounds 316
Dispatch V-Table Section Bounds (if any) 322
Dispatch Other Section Bounds 326
Dispatch Block Bounds 330
Dispatch Addresses of Soft Spots 334
Instrument Soft Spot Instructions 338
Dispatch Addresses of Writes with Mem Operand 340
Instrument Mem operand Write Instructions 336
Dispatch Addresses of (In) Direct Code Transitions 332
Instrument Code Transition Instructions 328
Dispatch Disassembly of Indirect Transitions 324
Load "Manual" transitions 320
Instrument Critical Function/ Syscall Addresses 318
Instrument Admin Function/ Sys Call Addresses 312
Wait Loop 309

FIG. 3A

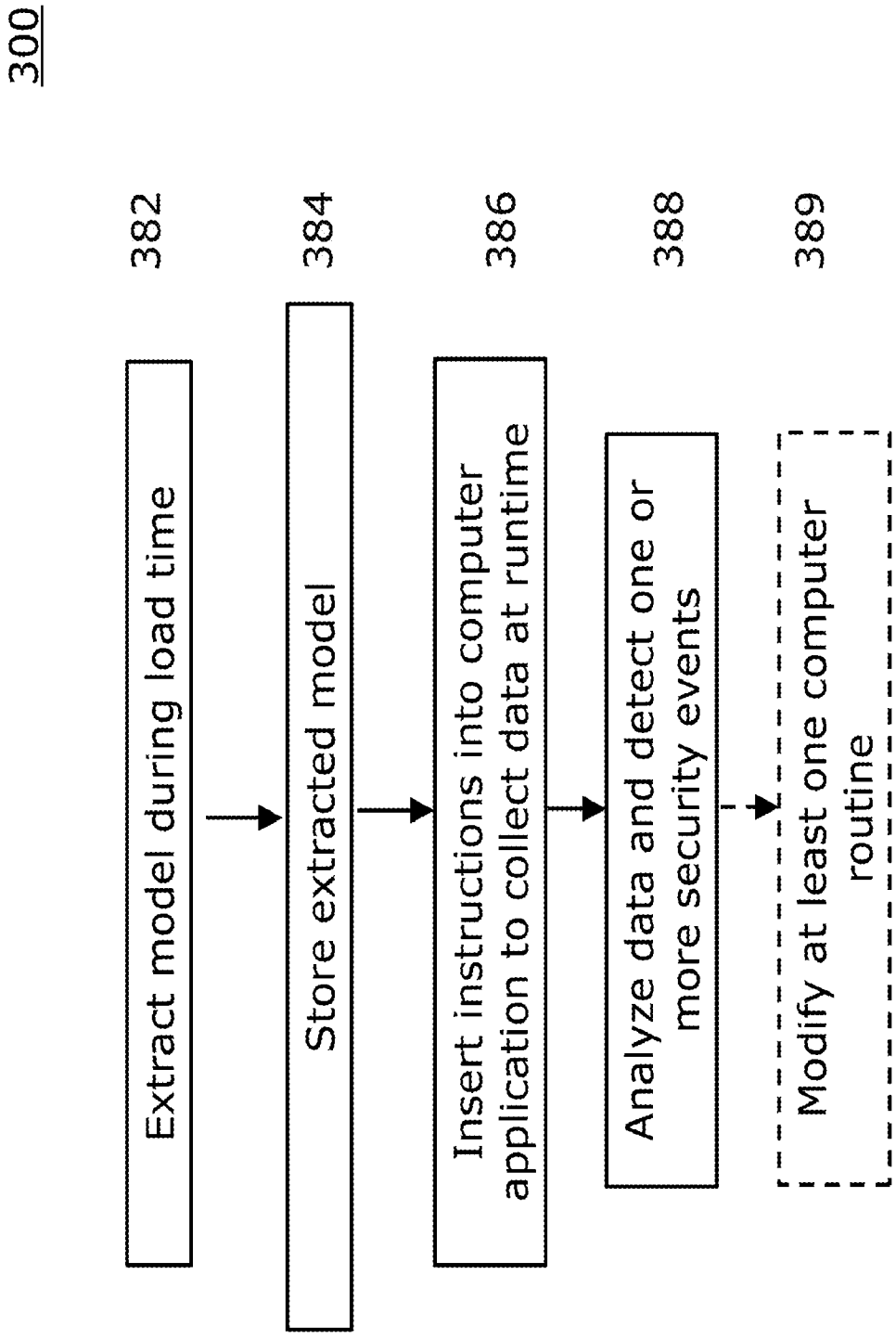
300
382
Extract model during load time
384
Store extracted model
386
Insert instructions into computer application to collect data at runtime
388
Analyze data and detect one or more security events
389
Modify at least one computer routine

FIG. 3B

390

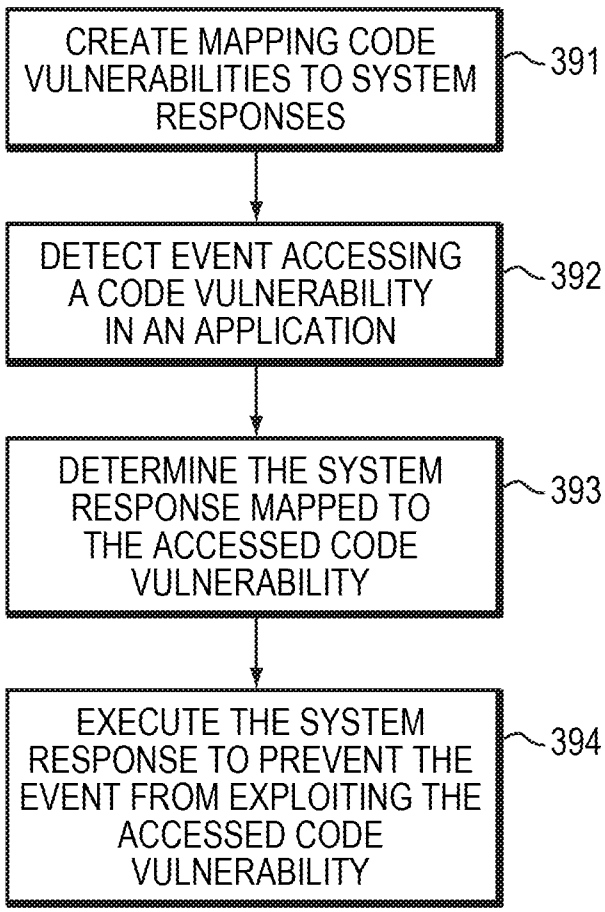
CREATE MAPPING CODE VULNERABILITIES TO SYSTEM RESPONSES
391
DETECT EVENT ACCESSING A CODE VULNERABILITY IN AN APPLICATION
392
DETERMINE THE SYSTEM RESPONSE MAPPED TO THE ACCESSED CODE VULNERABILITY
393
EXECUTE THE SYSTEM RESPONSE TO PREVENT THE EVENT FROM EXPLOITING THE ACCESSED CODE VULNERABILITY
394

FIG. 3C

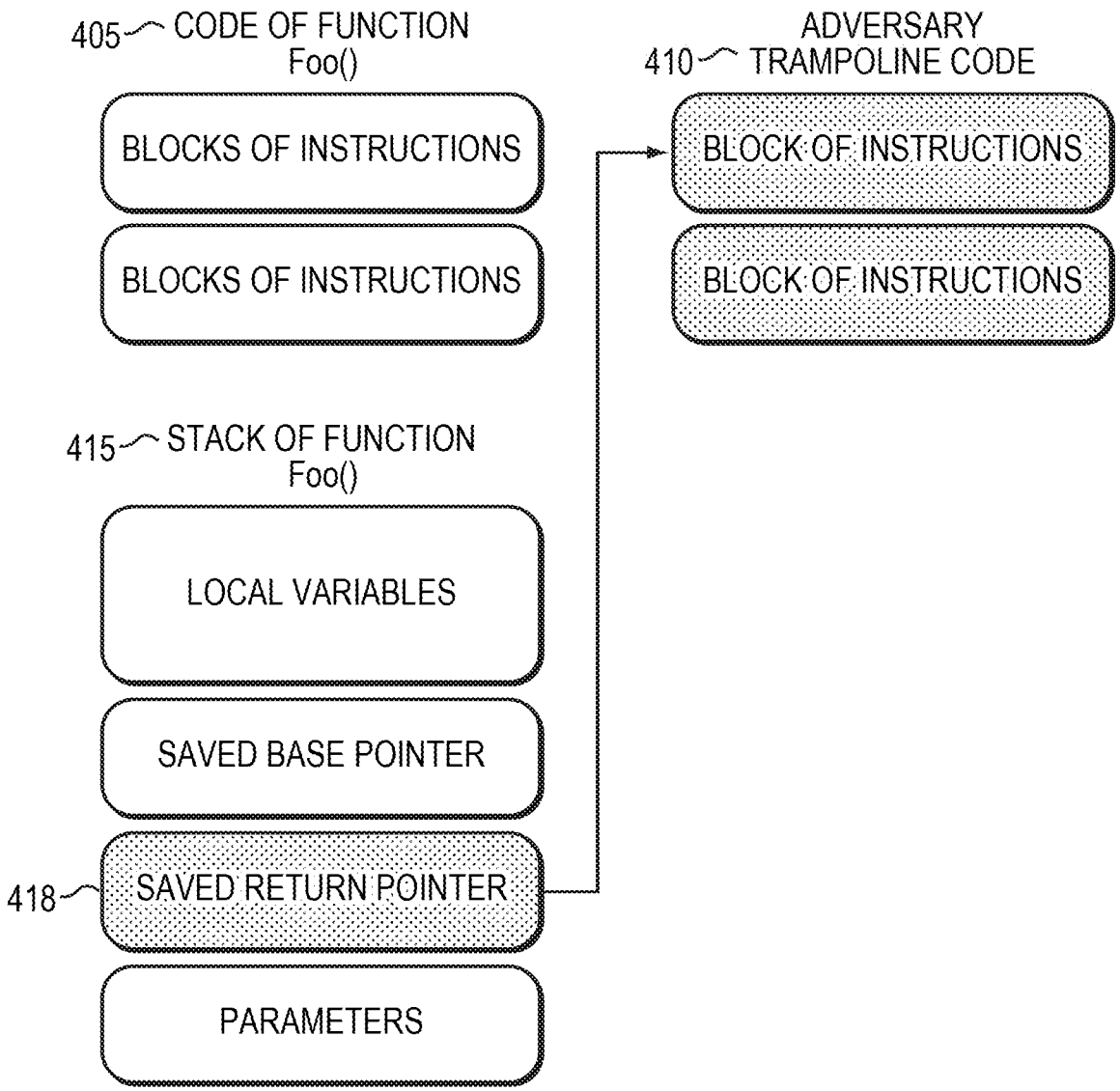
405
CODE OF FUNCTION Foo()
BLOCKS OF INSTRUCTIONS
BLOCKS OF INSTRUCTIONS
410
ADVERSARY TRAMPOLINE CODE
BLOCK OF INSTRUCTIONS
BLOCK OF INSTRUCTIONS
415
STACK OF FUNCTION Foo()
LOCAL VARIABLES
SAVED BASE POINTER
418
SAVED RETURN POINTER
PARAMETERS

FIG. 4A

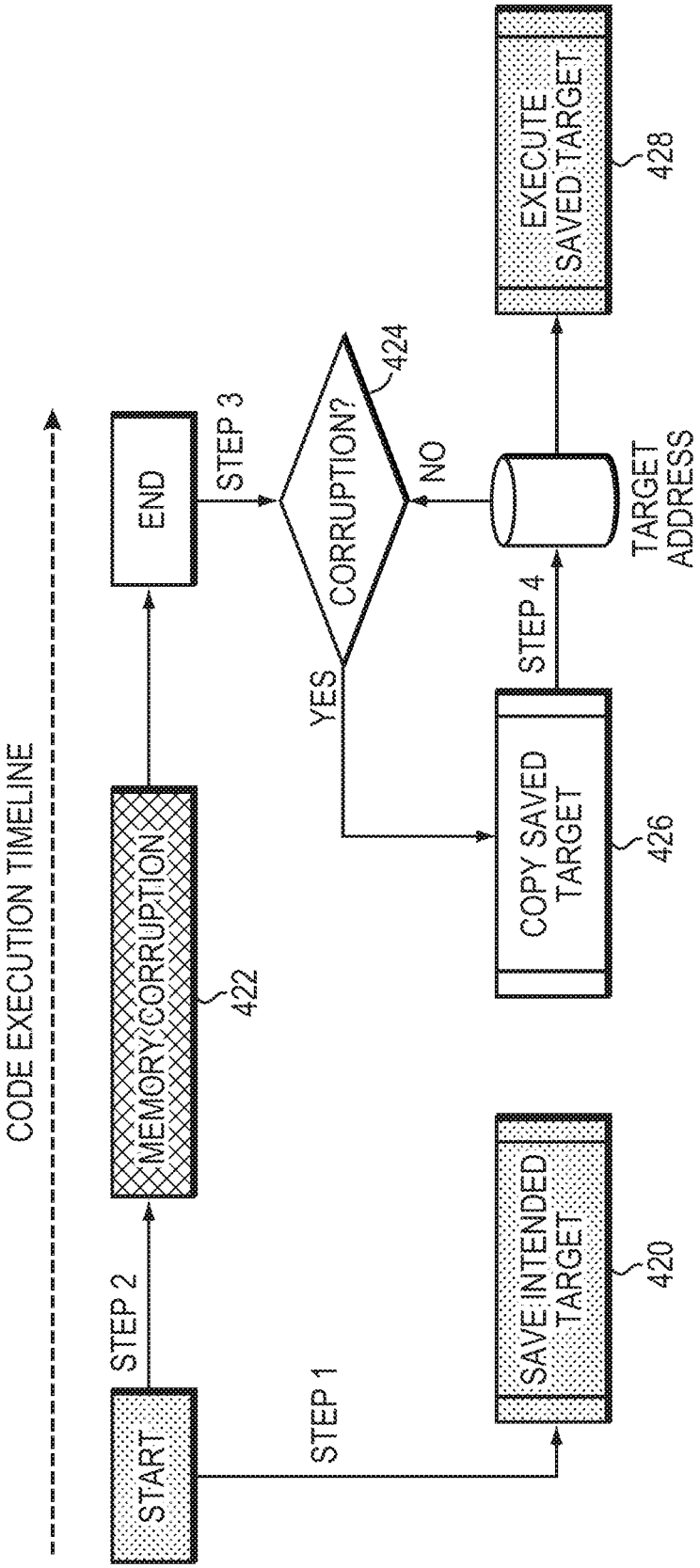
CODE EXECUTION TIMELINE
START
STEP 1
SAVE INTENDED TARGET
420
STEP 2
MEMORY CORRUPTION
422
END
STEP 3
CORRUPTION?
424
YES
NO
COPY SAVED TARGET
426
STEP 4
TARGET ADDRESS
EXECUTE SAVED TARGET
428

FIG. 4B

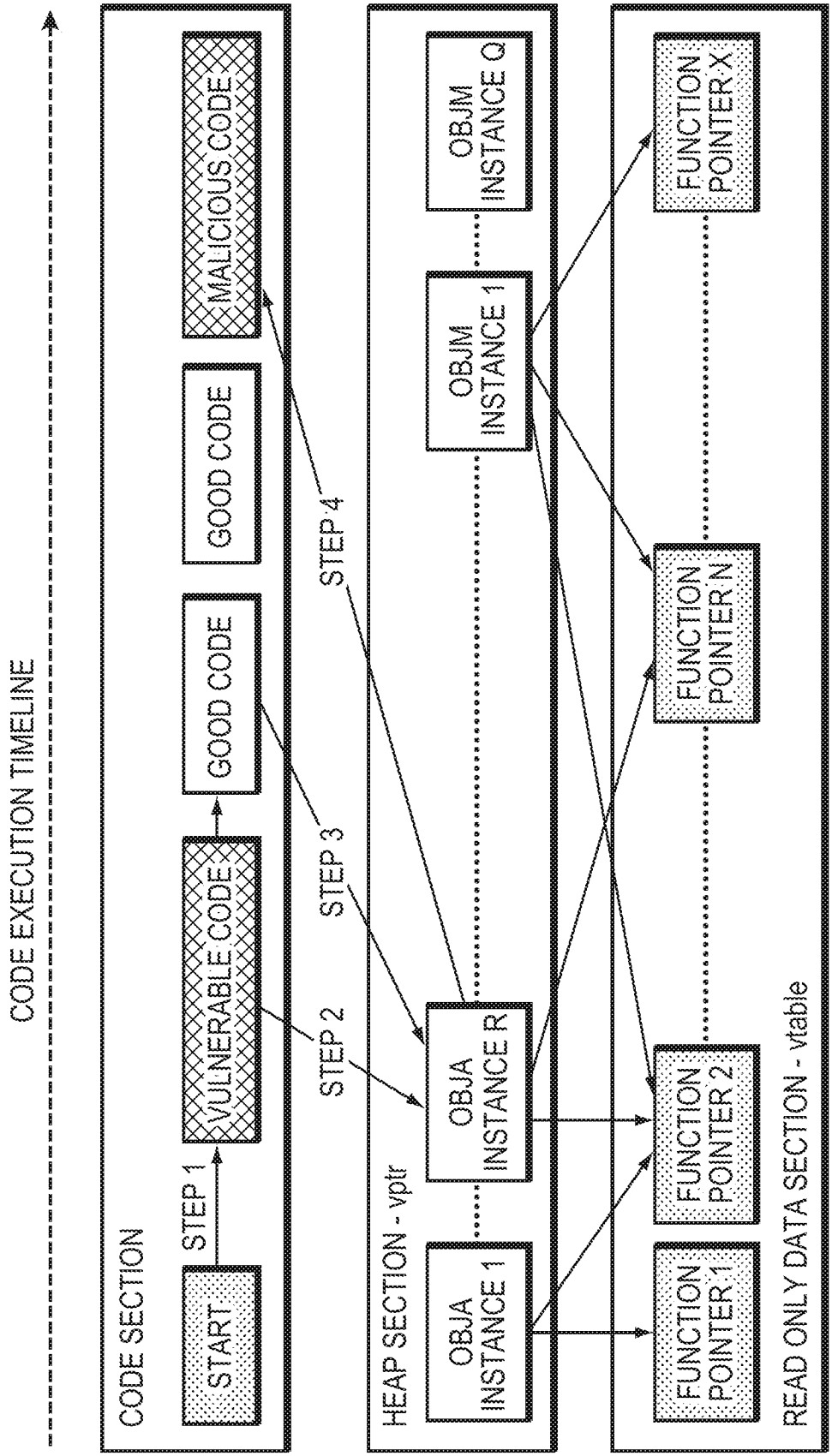
CODE EXECUTION TIMELINE
CODE SECTION
START
STEP 1
VULNERABLE CODE
GOOD CODE
GOOD CODE
MALICIOUS CODE
STEP 2
STEP 3
STEP 4
HEAP SECTION - vptr
OBJA INSTANCE 1
OBJA INSTANCE R
OBJM INSTANCE 1
OBJM INSTANCE Q
READ ONLY DATA SECTION - vtable
FUNCTION POINTER 1
FUNCTION POINTER 2
FUNCTION POINTER N
FUNCTION POINTER X

FIG. 4C

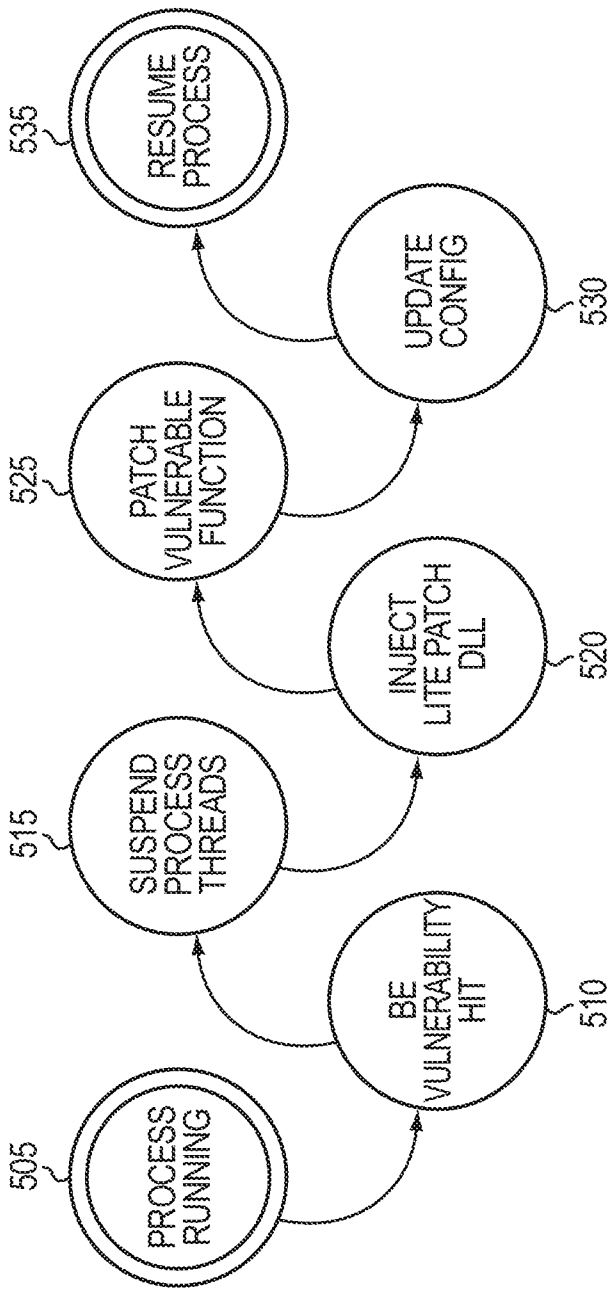
505
PROCESS RUNNING
510
BE VULNERABILITY HIT
515
SUSPEND PROCESS THREADS
520
INJECT LITE PATCH DLL
525
PATCH VULNERABLE FUNCTION
530
UPDATE CONFIG
535
RESUME PROCESS

FIG. 5A

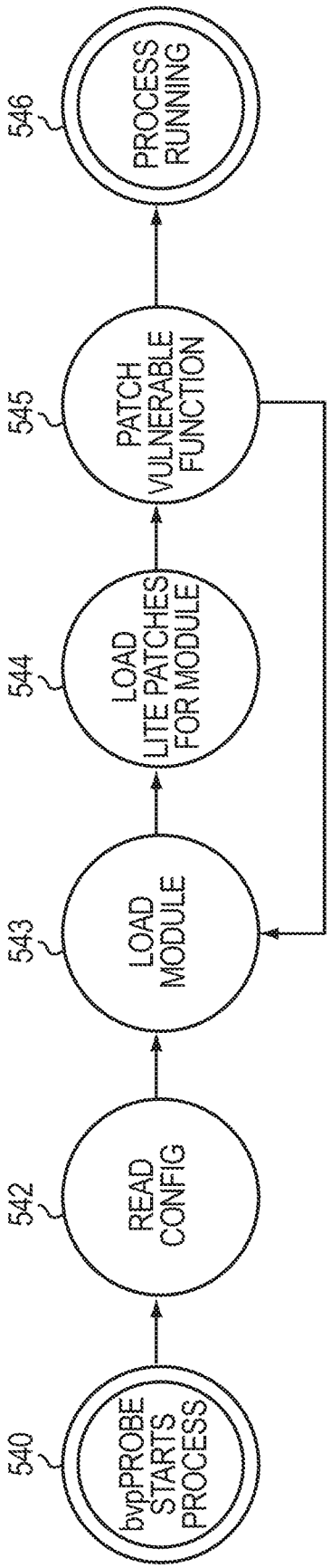
540
bvpPROBE STARTS PROCESS
542
READ CONFIG
543
LOAD MODULE
544
LOAD LITE PATCHES FOR MODULE
545
PATCH VULNERABLE FUNCTION
546
PROCESS RUNNING

FIG. 5B

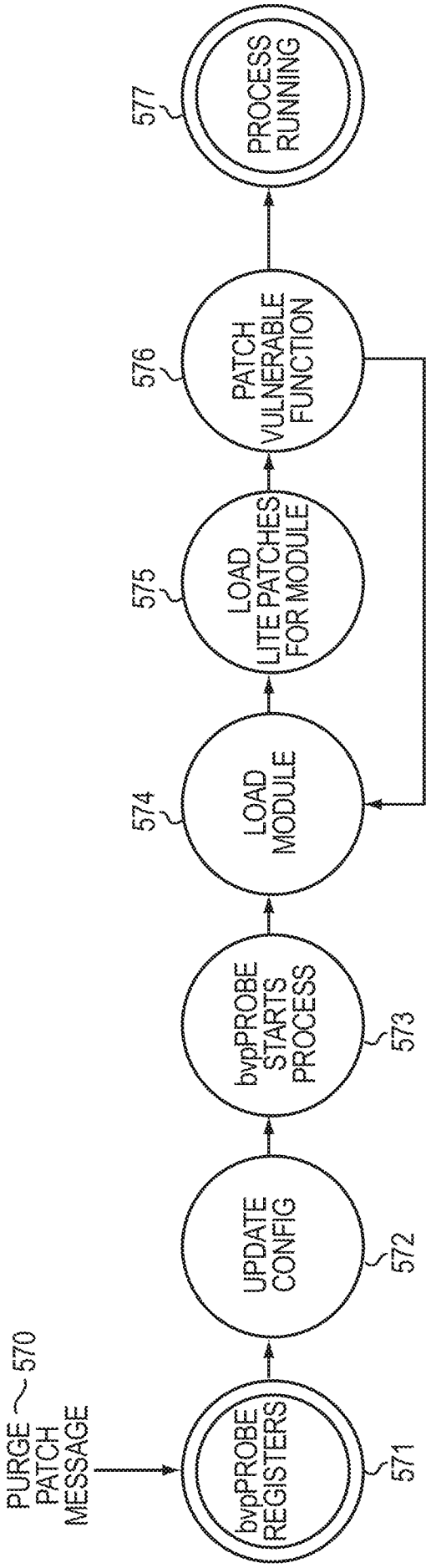
PURGE PATCH MESSAGE
570
bvpPROBE REGISTERS
571
UPDATE CONFIG
572
bvpPROBE STARTS PROCESS
573
LOAD MODULE
574
LOAD LITE PATCHES FOR MODULE
575
PATCH VULNERABLE FUNCTION
576
PROCESS RUNNING
577

FIG. 5C

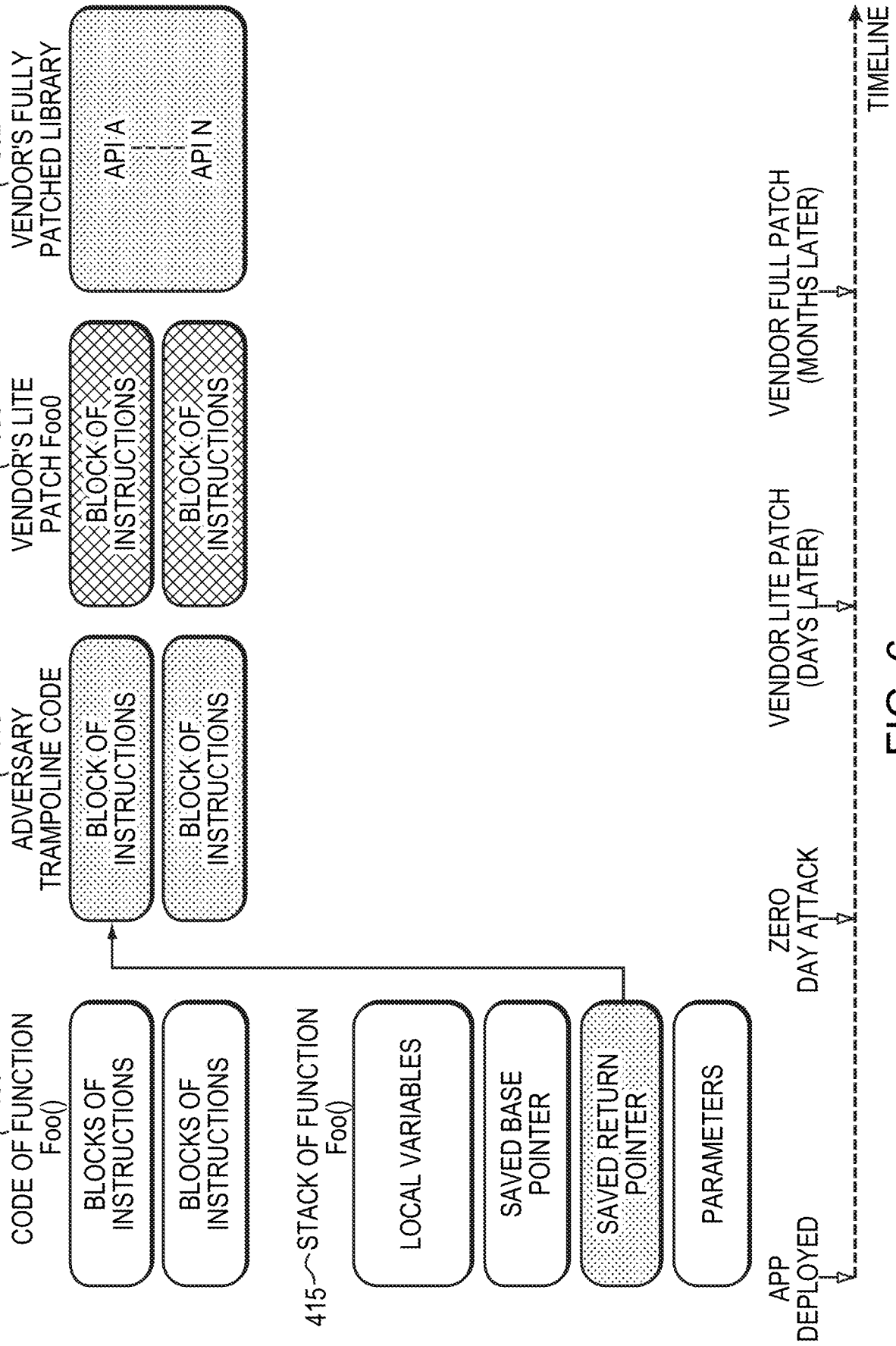
405
CODE OF FUNCTION Foo()
BLOCKS OF INSTRUCTIONS
BLOCKS OF INSTRUCTIONS
415
STACK OF FUNCTION Foo()
LOCAL VARIABLES
SAVED BASE POINTER
SAVED RETURN POINTER
PARAMETERS
410
ADVERSARY TRAMPOLINE CODE
BLOCK OF INSTRUCTIONS
BLOCK OF INSTRUCTIONS
580
VENDOR'S LITE PATCH Foo0
BLOCK OF INSTRUCTIONS
BLOCK OF INSTRUCTIONS
582
VENDOR'S FULLY PATCHED LIBRARY
API A
API N
APP DEPLOYED
ZERO DAY ATTACK
VENDOR LITE PATCH (DAYS LATER)
VENDOR FULL PATCH (MONTHS LATER)
TIMELINE

FIG. 6

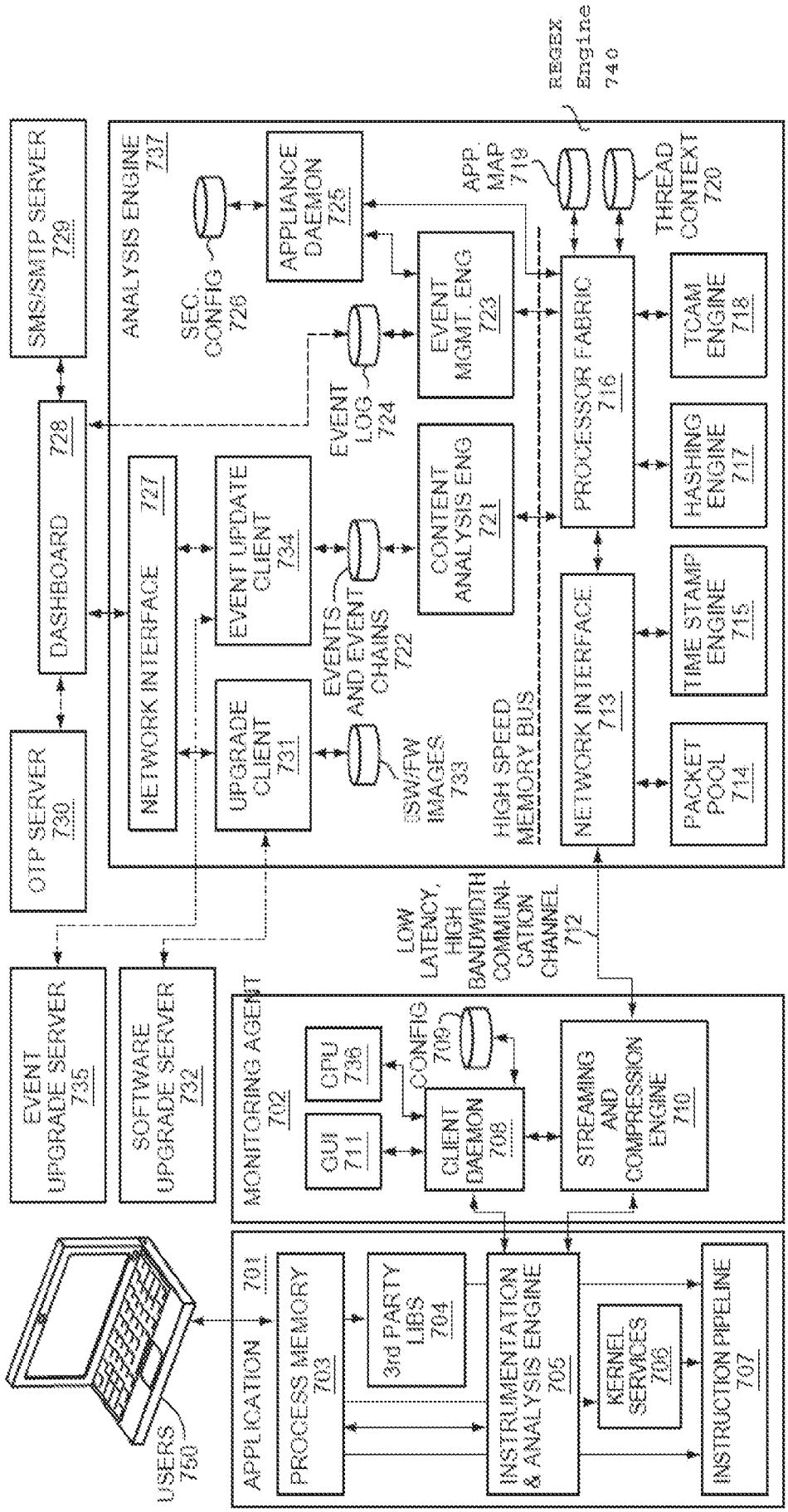
USERS 701
APPLICATION 701
PROCESS MEMORY 703
3rd PARTY LIBS 704
INSTRUMENTATION & ANALYSIS ENGINE 705
KERNEL SERVICES 706
INSTRUCTION PIPELINE 707
MONITORING AGENT 702
GUI 711
CPU 736
CLIENT DAEMON 708
CONFIG 709
STREAMING AND COMPRESSION ENGINE 710
LOW LATENCY, HIGH BANDWIDTH COMMUNI-CATION CHANNEL 712
EVENT UPGRADE SERVER 735
SOFTWARE UPGRADE SERVER 732
OTP SERVER 730
SMS/SMTP SERVER 729
DASHBOARD 728
ANALYSIS ENGINE 737
NETWORK INTERFACE 727
UPGRADE CLIENT 731
EVENT UPDATE CLIENT 734
SEC. CONFIG 726
APPLIANCE DAEMON 726
EVENT LOG 724
EVENTS AND EVENT CHAINS 722
SW/FW IMAGES 733
CONTENT ANALYSIS ENG 721
EVENT MGMT. ENG 723
HIGH SPEED MEMORY BUS
NETWORK INTERFACE 713
PACKET POOL 714
TIME STAMP ENGINE 715
PROCESSOR FABRIC 716
HASHING ENGINE 717
TCAM ENGINE 718
REGEX Engine 740
APP. MAP 719
THREAD CONTEXT 720

FIG. 7A

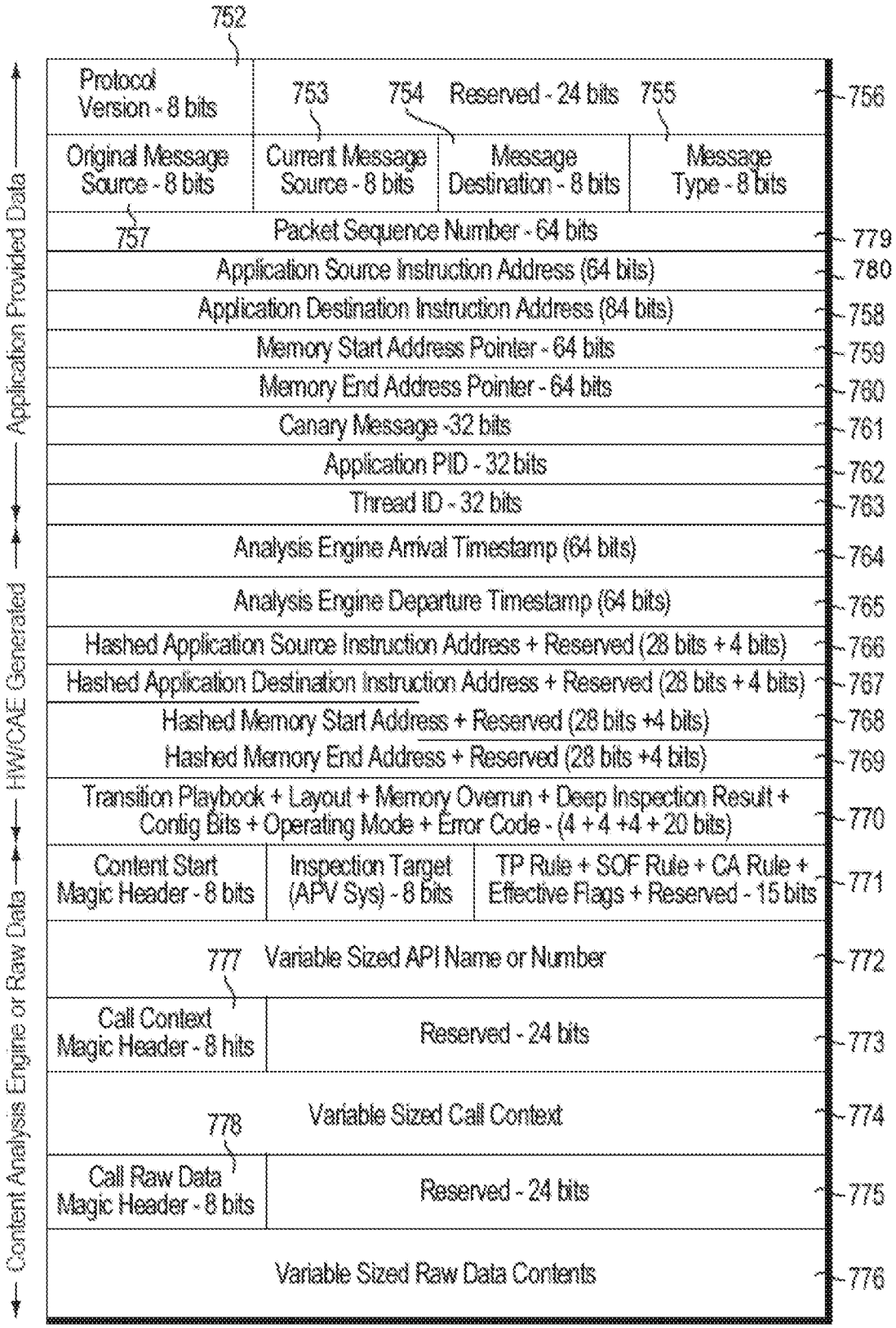
752
Protocol Version - 8 bits
753
754
Reserved - 24 bits
755
756
Original Message Source - 8 bits
Current Message Source - 8 bits
Message Destination - 8 bits
Message Type - 8 bits
757
Packet Sequence Number - 64 bits
779
Application Source Instruction Address (64 bits)
780
Application Destination Instruction Address (84 bits)
758
Memory Start Address Pointer - 64 bits
759
Memory End Address Pointer - 64 bits
760
Canary Message -32 bits
761
Application PID - 32 bits
762
Thread ID - 32 bits
763
Analysis Engine Arrival Timestamp (64 bits)
764
Analysis Engine Departure Timestamp (64 bits)
765
Hashed Application Source Instruction Address + Reserved (28 bits + 4 bits)
766
Hashed Application Destination Instruction Address + Reserved (28 bits + 4 bits)
767
Hashed Memory Start Address + Reserved (28 bits +4 bits)
768
Hashed Memory End Address + Reserved (28 bits +4 bits)
769
Transition Playbook + Layout + Memory Overrun + Deep Inspection Result + Contig Bits + Operating Mode + Error Code - (4 + 4 +4 + 20 bits)
770
Content Start Magic Header - 8 bits
Inspection Target (APV Sys) - 8 bits
TP Rule + SOF Rule + CA Rule + Effective Flags + Reserved - 15 bits
771
777
Variable Sized API Name or Number
772
Call Context Magic Header - 8 bits
Reserved - 24 bits
773
778
Variable Sized Call Context
774
Call Raw Data Magic Header - 8 bits
Reserved - 24 bits
775
Variable Sized Raw Data Contents
776
Content Analysis Engine or Raw Data
HW/CAE Generated
Application Provided Data

FIG. 7B

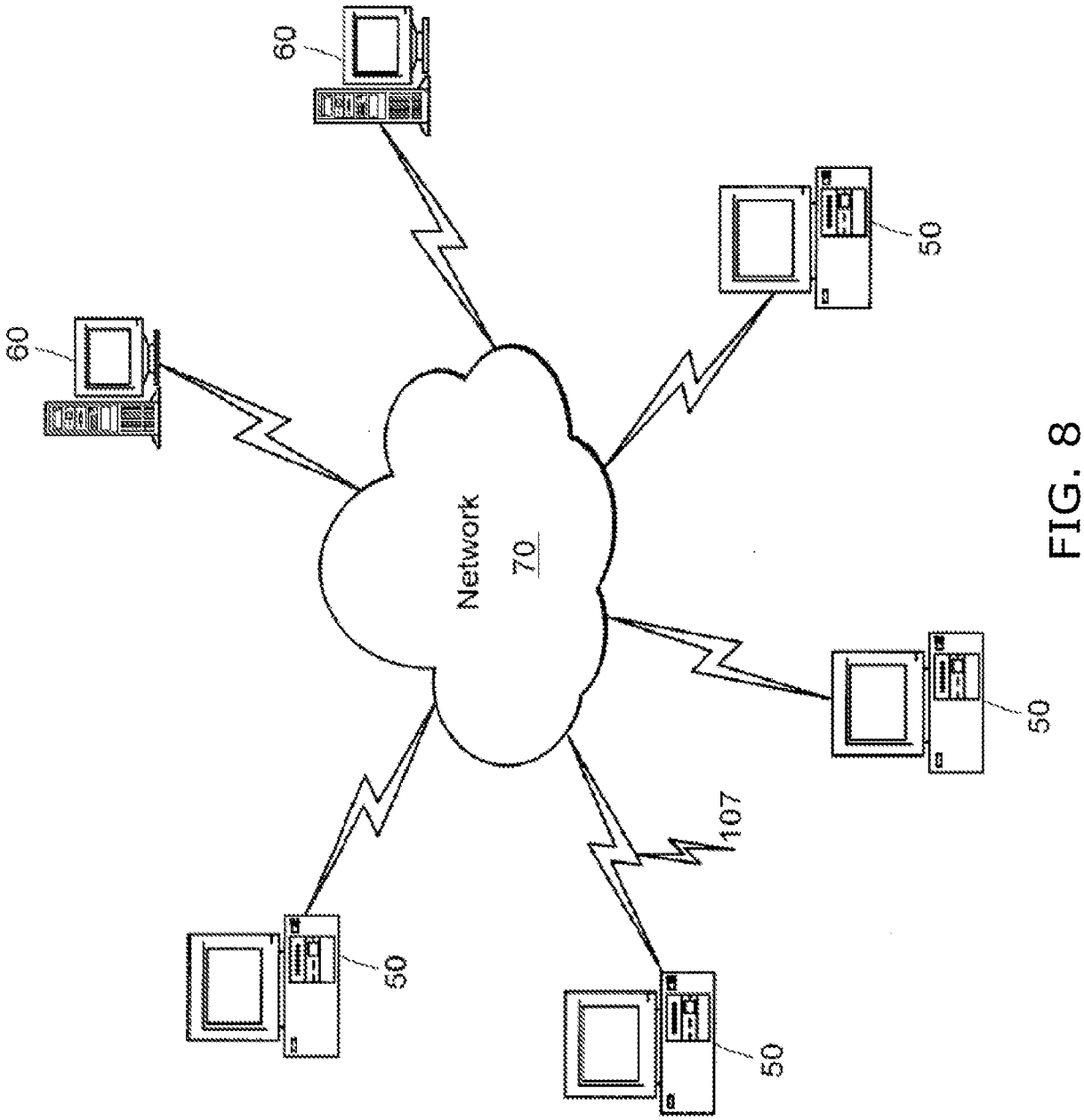

FIG. 8

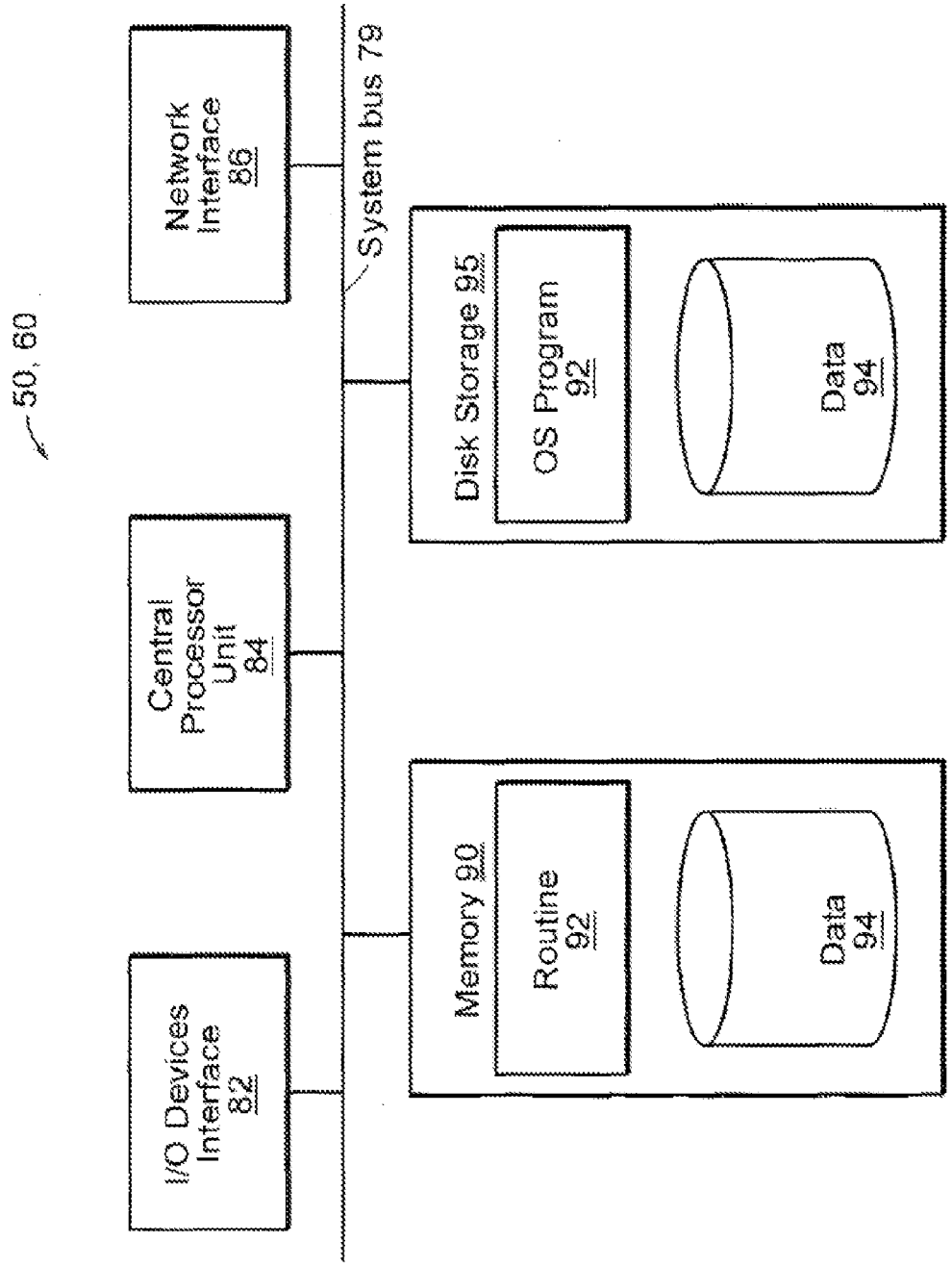
50, 60
Network Interface 86
System bus 79
Disk Storage 95
OS Program 92
Data 94
Central Processor Unit 84
Memory 90
Routine 92
Data 94
I/O Devices Interface 82

FIG. 9

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2017/037841

Box No. II Observations where certain claims were found unsearchable (Continuation of item 2 of first sheet)

This international search report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. ☐ Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely:

2. ☐ Claims Nos.:
because they relate to parts of the international application that do not comply with the prescribed requirements to such an extent that no meaningful international search can be carried out, specifically:

3. ☐ Claims Nos.:
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

Box No. III Observations where unity of invention is lacking (Continuation of item 3 of first sheet)

This International Searching Authority found multiple inventions in this international application, as follows:

see additional sheet

1. ☐ As all required additional search fees were timely paid by the applicant, this international search report covers all searchable claims.

2. ☐ As all searchable claims could be searched without effort justifying an additional fees, this Authority did not invite payment of additional fees.

3. ☐ As only some of the required additional search fees were timely paid by the applicant, this international search report covers only those claims for which fees were paid, specifically claims Nos.:

4. ☒ No required additional search fees were timely paid by the applicant. Consequently, this international search report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:

1-24

Remark on Protest

☐ The additional search fees were accompanied by the applicant's protest and, where applicable, the payment of a protest fee.

☐ The additional search fees were accompanied by the applicant's protest but the applicable protest fee was not paid within the time limit specified in the invitation.

☐ No protest accompanied the payment of additional search fees.

INTERNATIONAL SEARCH REPORT

International application No
PCT/US2017/037841

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06F21/51 G06F21/54
ADD.

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)
G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
EPO-Internal, WPI Data

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	WO 2015/038944 A1 (VIRSEC SYSTEMS INC [US]) 19 March 2015 (2015-03-19) cited in the application claim 1 paragraph [0063] - paragraph [0064] -----	1-24
Y	BUCK B ET AL: "AN API FOR RUNTIME CODE PATCHING", 20000101, vol. 14, no. 4, 1 January 2000 (2000-01-01), pages 317-329, XP008079534, page 317 ----- -/--	1-24

[X] Further documents are listed in the continuation of Box C.

[X] 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 another 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 September 2017	13/11/2017

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016	Mäenpää, Jari

2

Form PCT/ISA/210 (second sheet) (April 2005)

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	"Software Instrumentation ED - Wah B", 1 January 2008 (2008-01-01), WILEY ENCYCLOPEDIA OF COMPUTER SCIENCE AND ENGINEE, WILEY, PAGE(S) 1 - 11, XP007912827, the whole document -----	1-24

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No
PCT/US2017/037841

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 2015038944 A1	19-03-2015	AU 2014318585 A1	17-03-2016
		CA 2923231 A1	19-03-2015
		EP 3044719 A1	20-07-2016
		JP 2016534479 A	04-11-2016
		KR 20160114037 A	04-10-2016
		US 2016212159 A1	21-07-2016
		WO 2015038944 A1	19-03-2015

FURTHER INFORMATION CONTINUED FROM PCT/ISA/ 210

This International Searching Authority found multiple (groups of) inventions in this international application, as follows:

1. claims: 1-24

   Computer implemented method and computer system wherein based upon the detection of security events, detected through load time model extraction and instrumentation based monitoring and analysis mechanism, a routine associated with an application is modified in a manner that preservers continued execution of the computer application.

   ---

2. claims: 25-38

   Computer implemented method and computer system wherein an event accessing a code vulnerability is detected and respective response is determined and executed to prevent the event from exploiting the the accessed code vulnerability.
