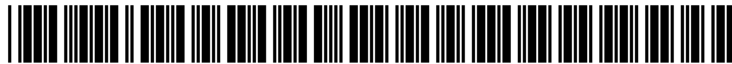


US 20260127288A1

(19) **United States**

(12) **Patent Application Publication**

(10) **Pub. No.: US 2026/0127288 A1**

(43) **Pub. Date: May 7, 2026**

Moskalchuk et al.

(54) **DYNAMIC KERNEL SECURITY MODULE WITH KERNEL-LEVEL SIGNED BINARY VALIDATION**

(71) Applicant: **CyberArk Software Ltd.**, Petach-Tikva (IL)

(72) Inventors: **Dmitry Moskalchuk**, Petach-Tikva (IL); **Ilya Abramovich**, Petach-Tikva (IL)

(73) Assignee: **CyberArk Software Ltd.**, Petach-Tikva (IL)

(21) Appl. No.: **19/435,500**

(22) Filed: **Dec. 29, 2025**

Related U.S. Application Data

(63) Continuation-in-part of application No. 19/354,412, filed on Oct. 9, 2025, which is a continuation of application No. 18/883,251, filed on Sep. 12, 2024, now Pat. No. 12,462,035.

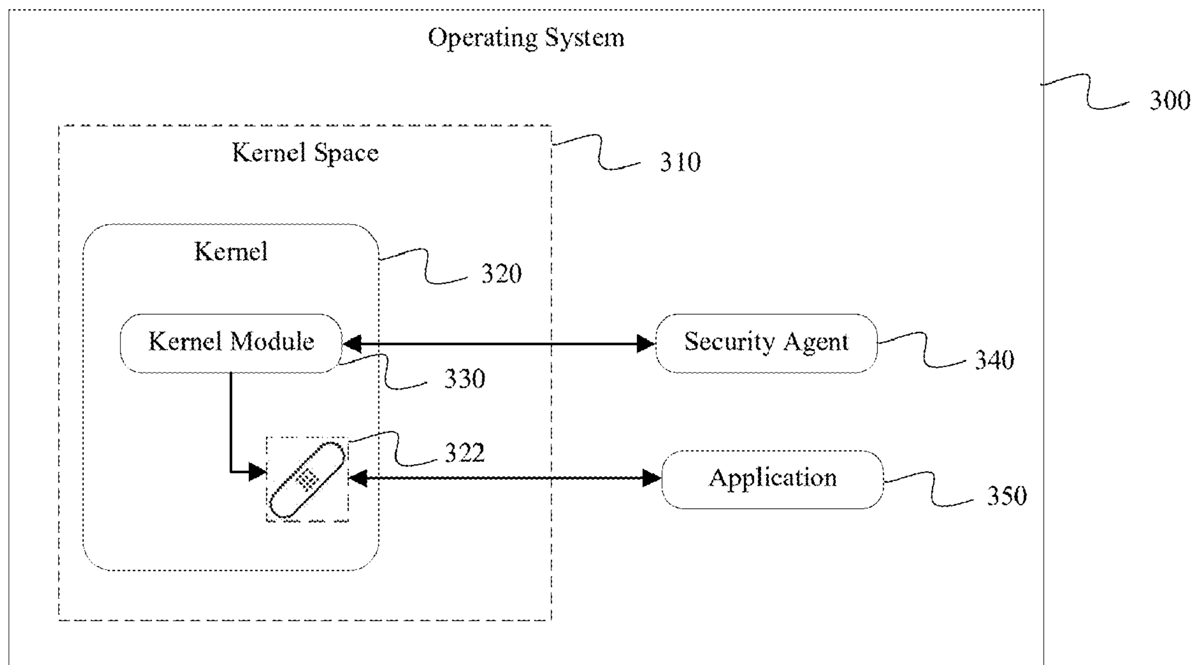
Publication Classification

(51) **Int. Cl.**
G06F 21/57 (2013.01)
G06F 21/54 (2013.01)

(52) **U.S. Cl.**
CPC *G06F 21/572* (2013.01); *G06F 21/54* (2013.01)

(57) **ABSTRACT**

Disclosed embodiments relate to systems and methods for securing kernel-level system functions and validating kernel-level verification of signed binaries. Techniques include identifying, by a system patched by a kernel module, a file signed by a user; receiving, by the kernel module, a second key from the user; and validating the signature embedded in the file, by identifying the file; determining whether the signature is embedded in the file; and validating that the signature corresponds to the second key.



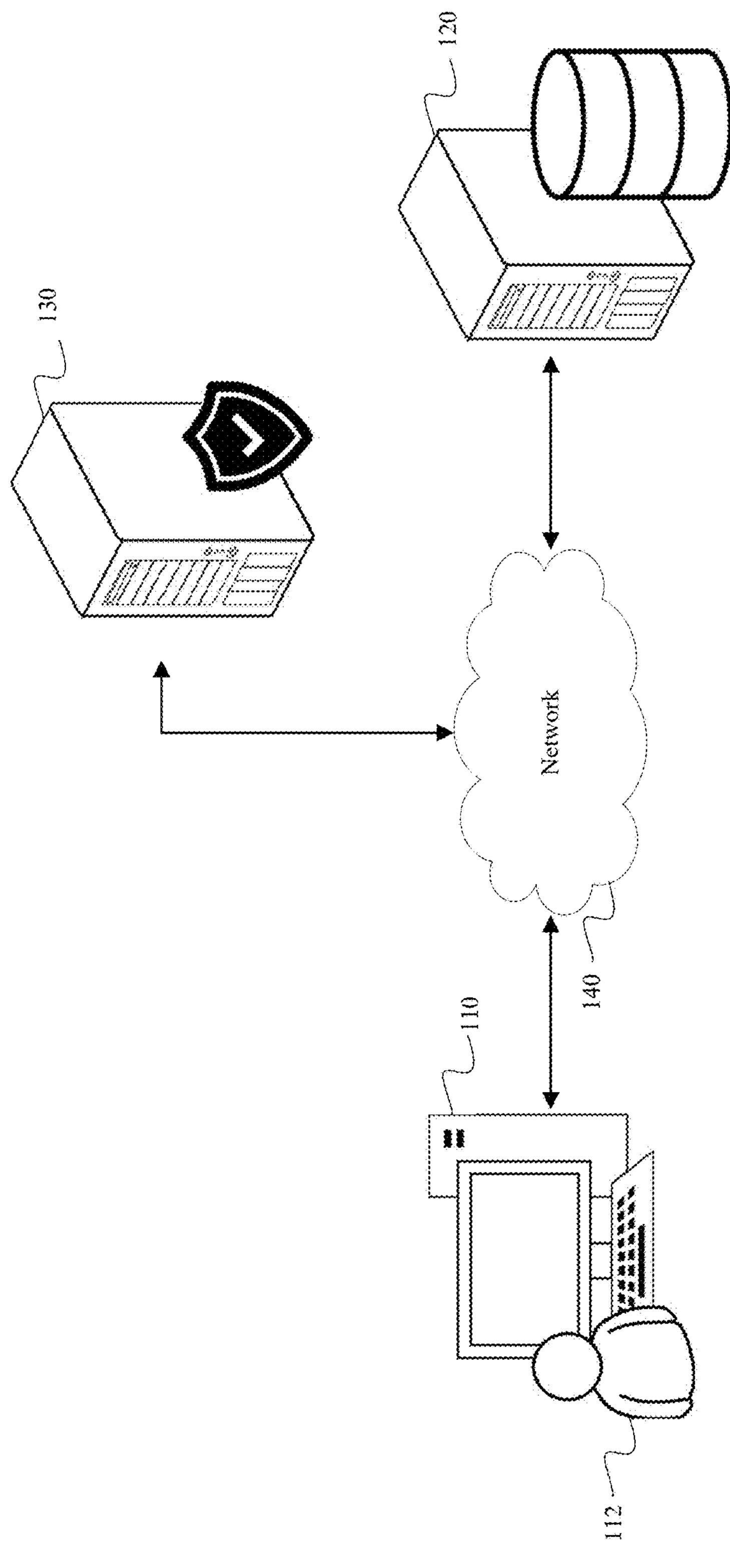


FIG. 1

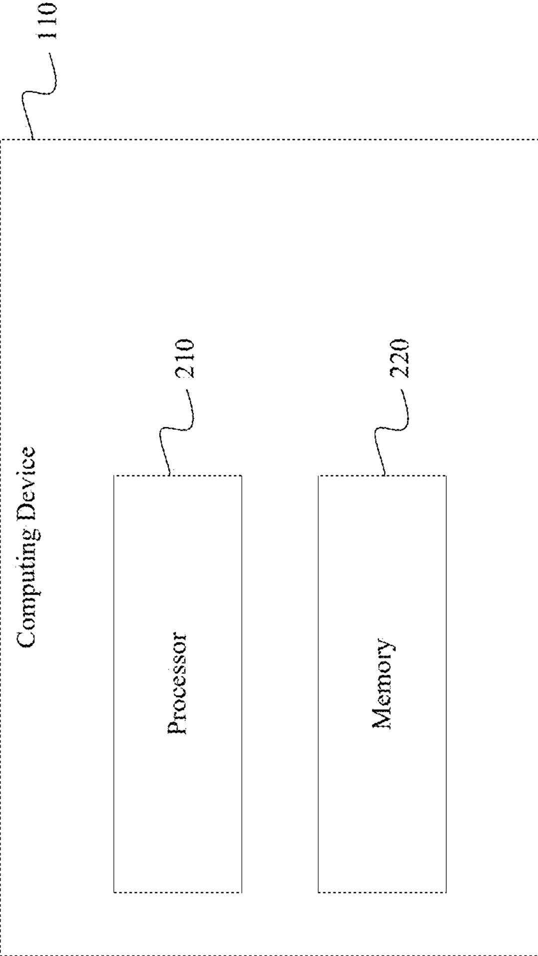


FIG. 2

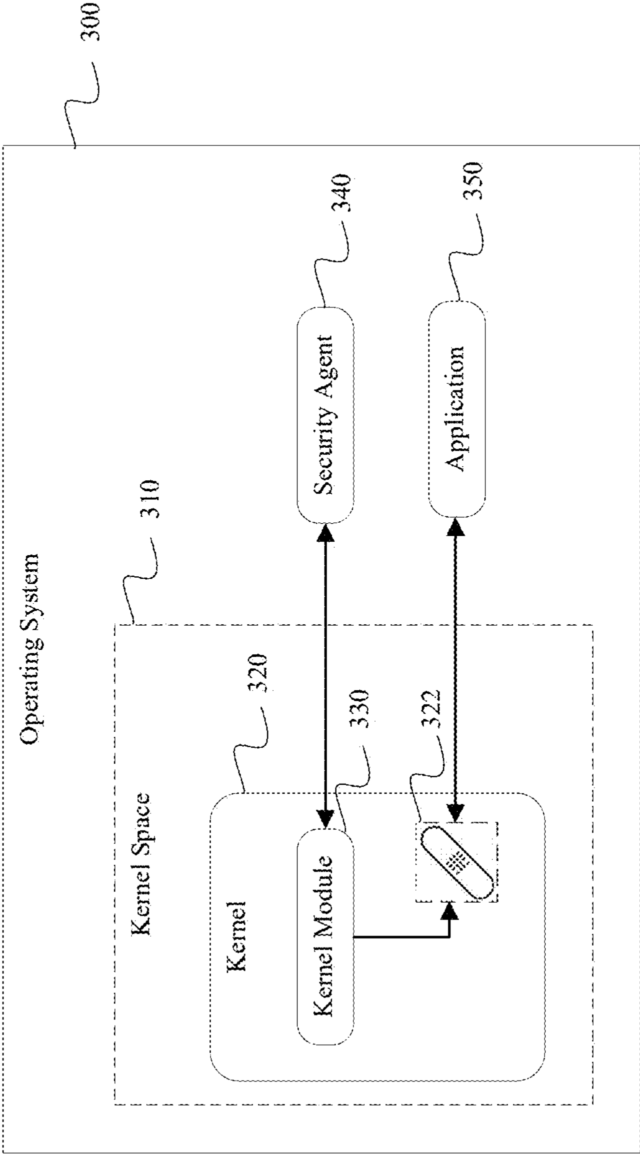


FIG. 3

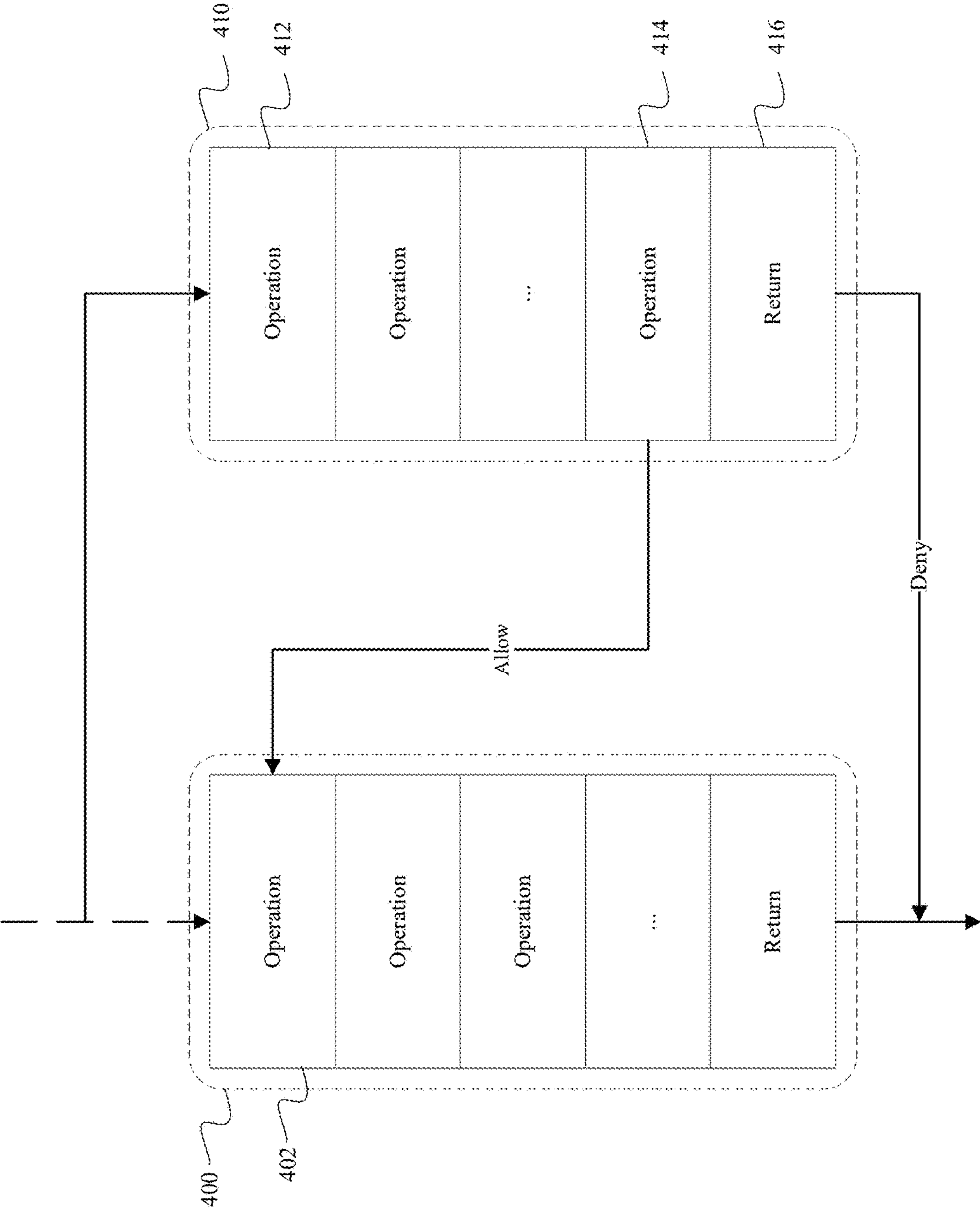


FIG. 4

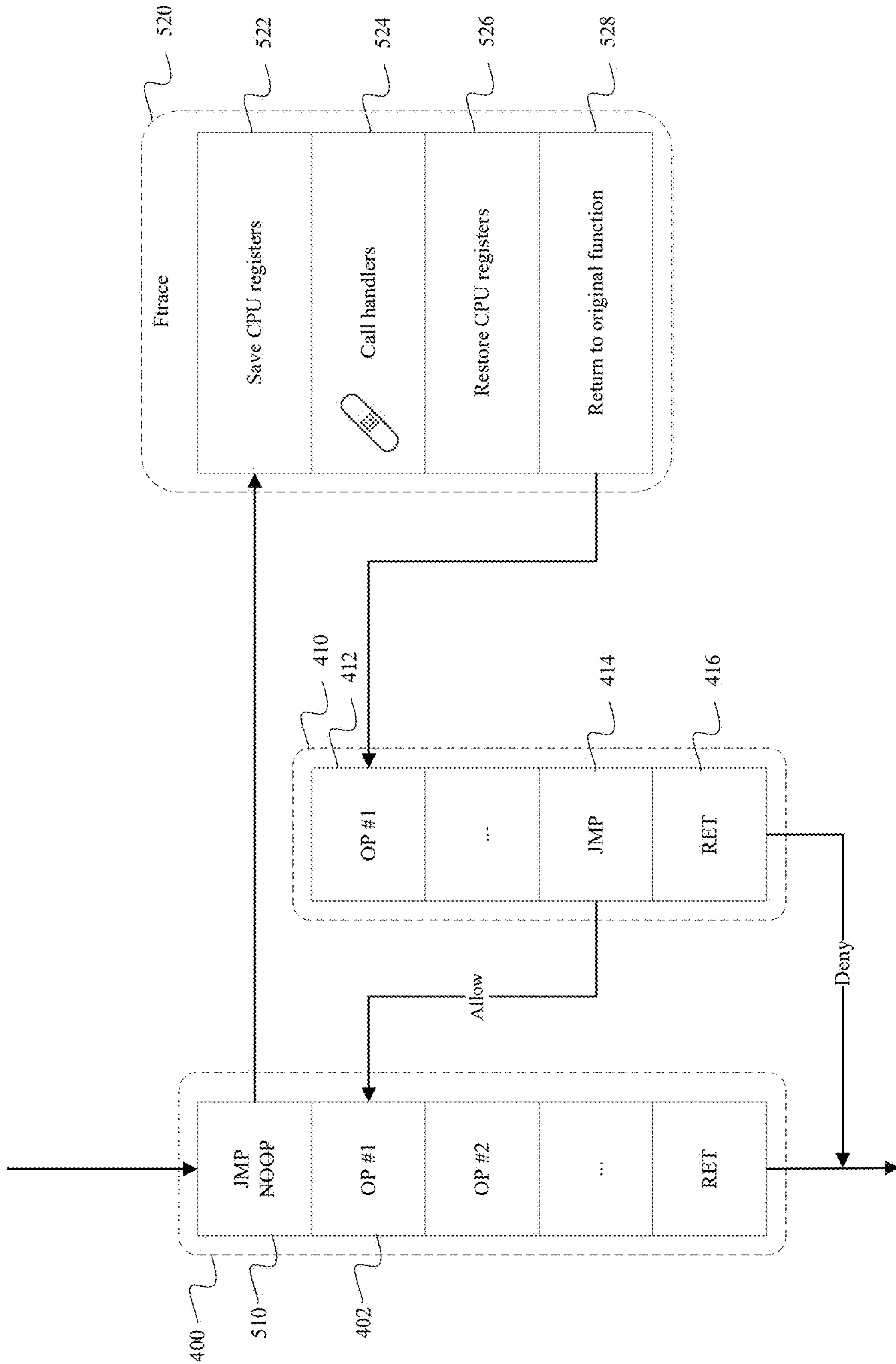


FIG. 5A

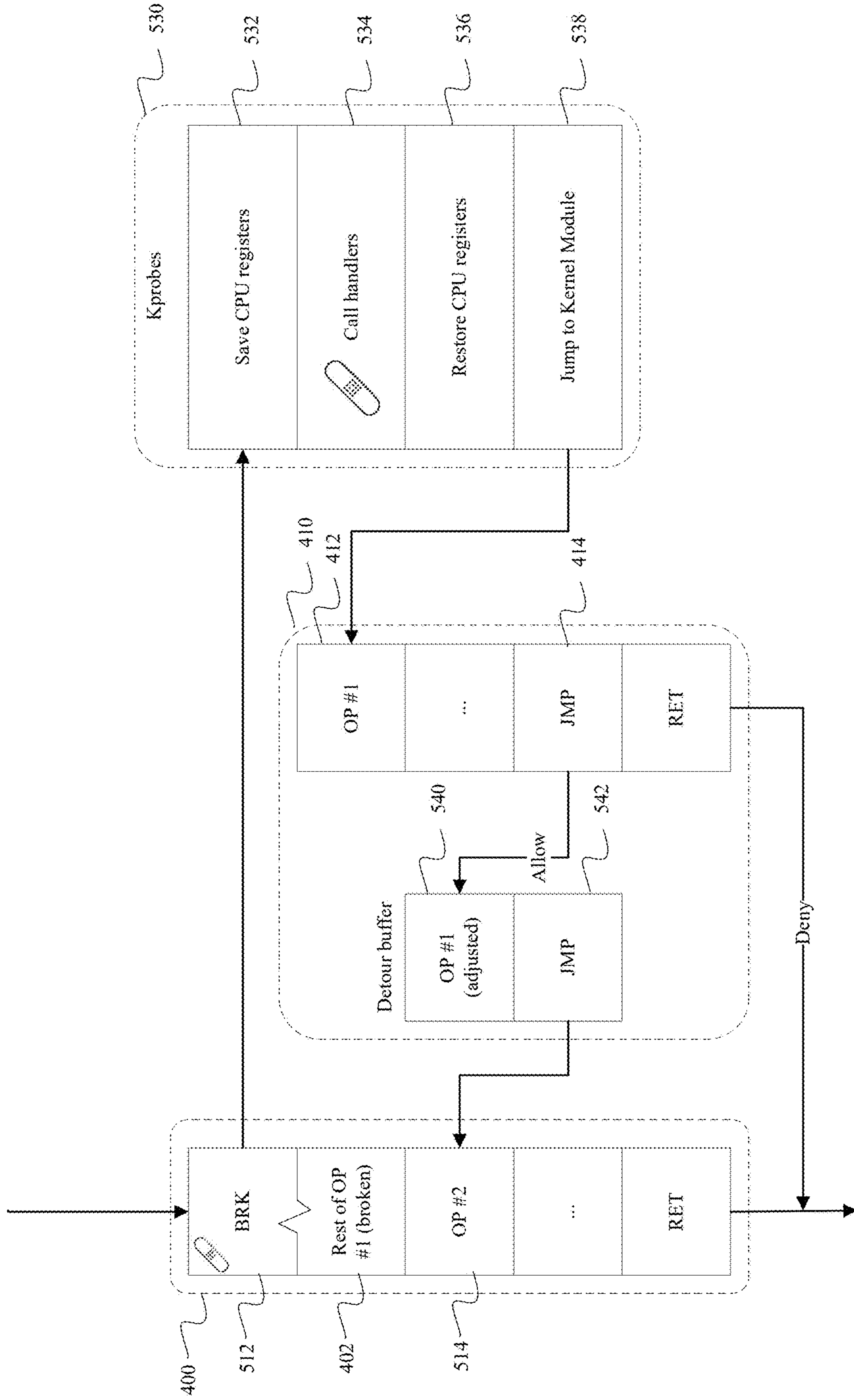


FIG. 5B

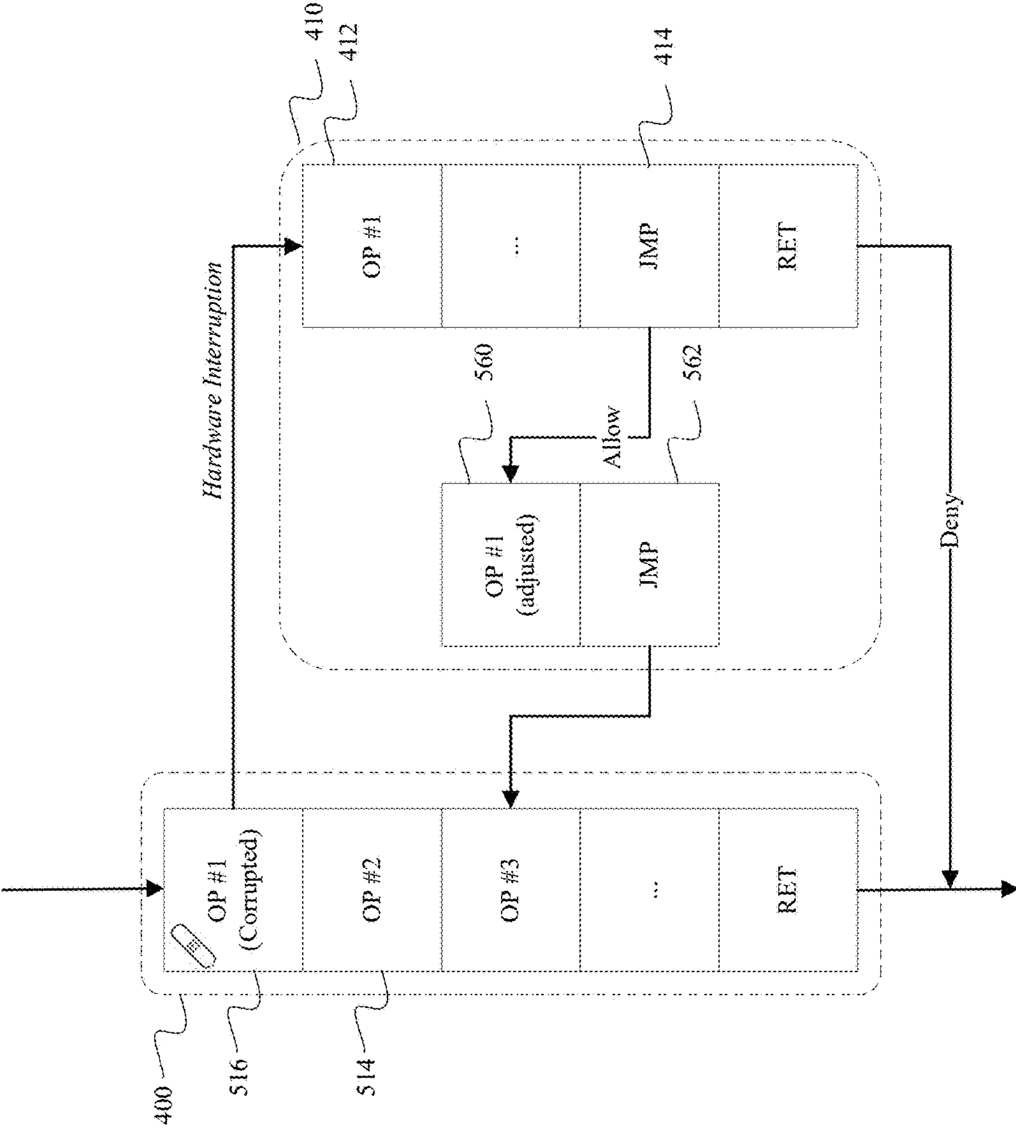


FIG. 5C

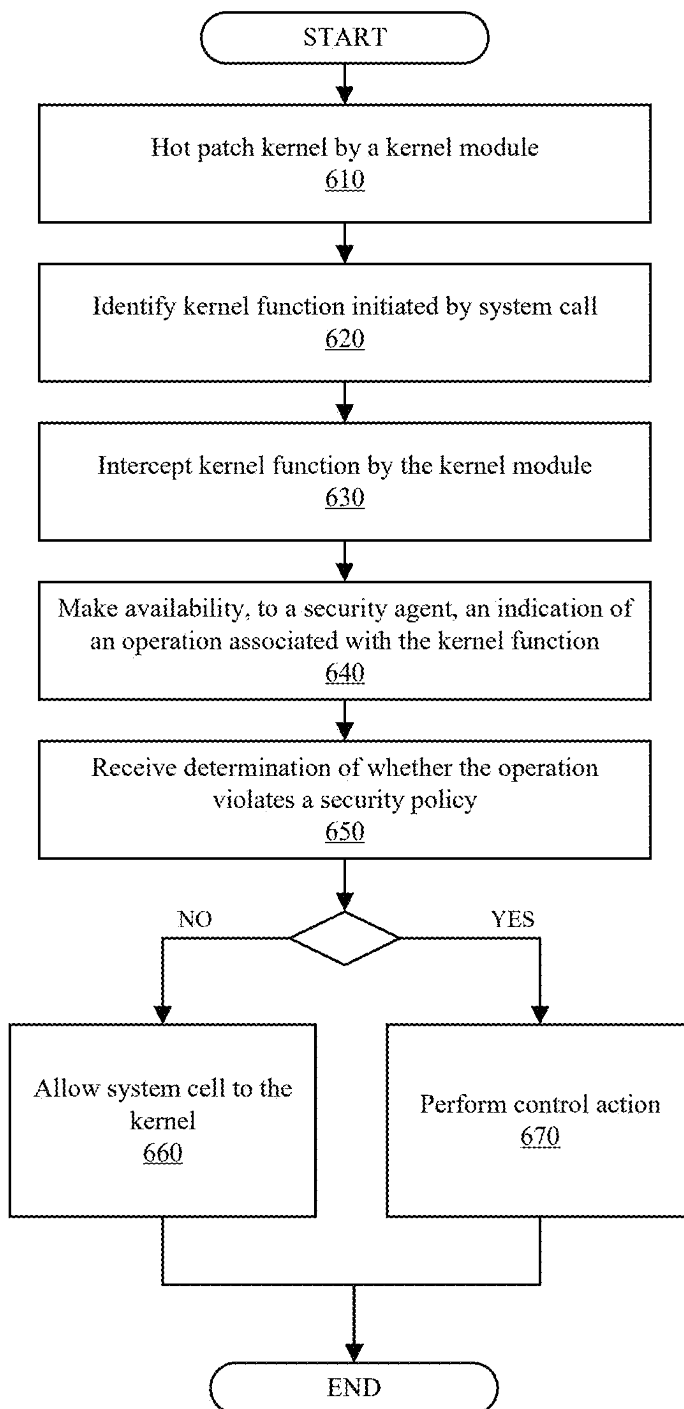


FIG. 6

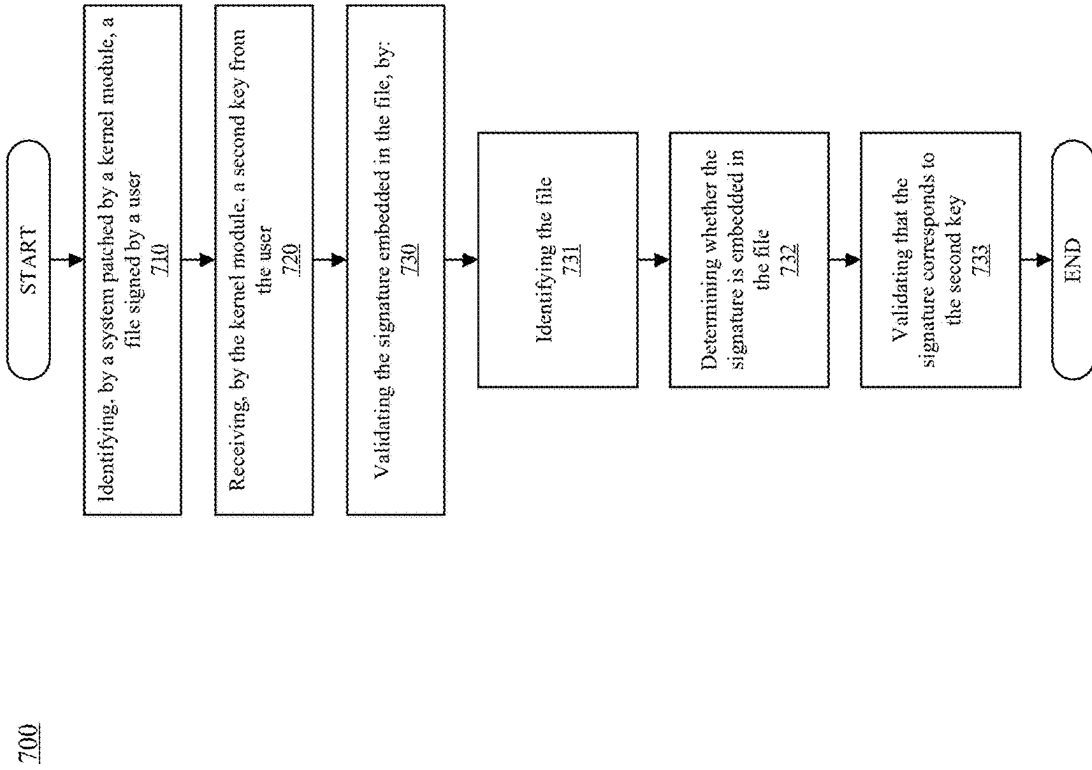


FIG. 7

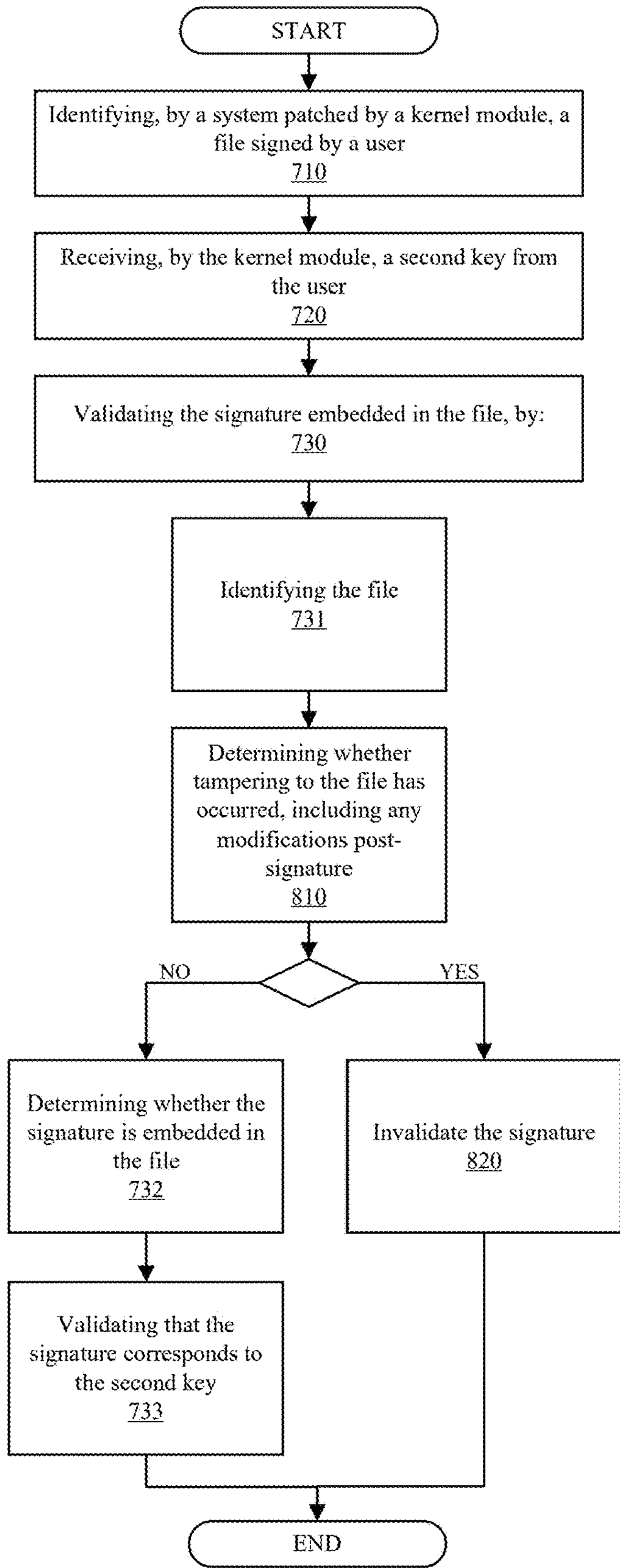


FIG. 8

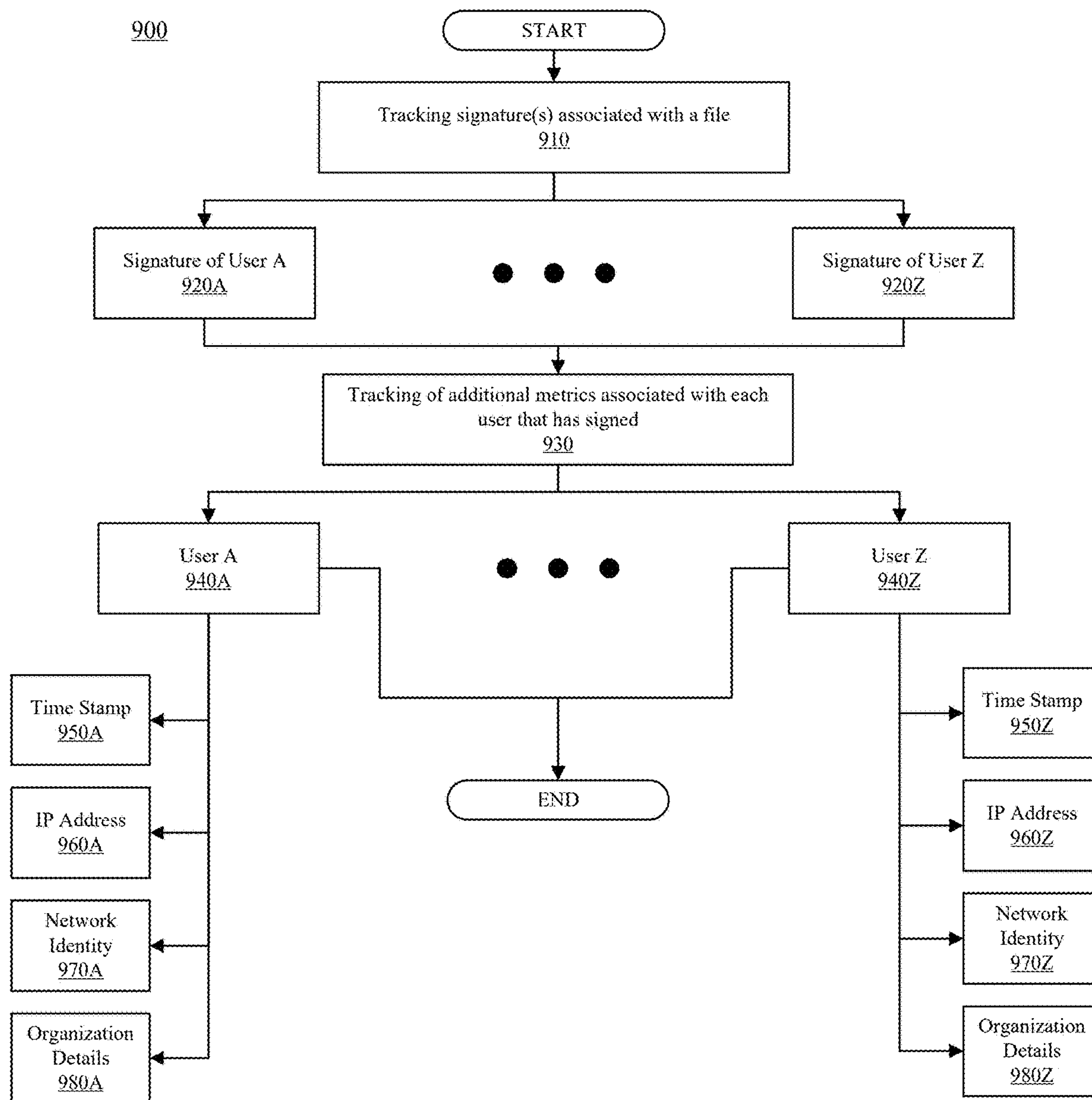


FIG. 9

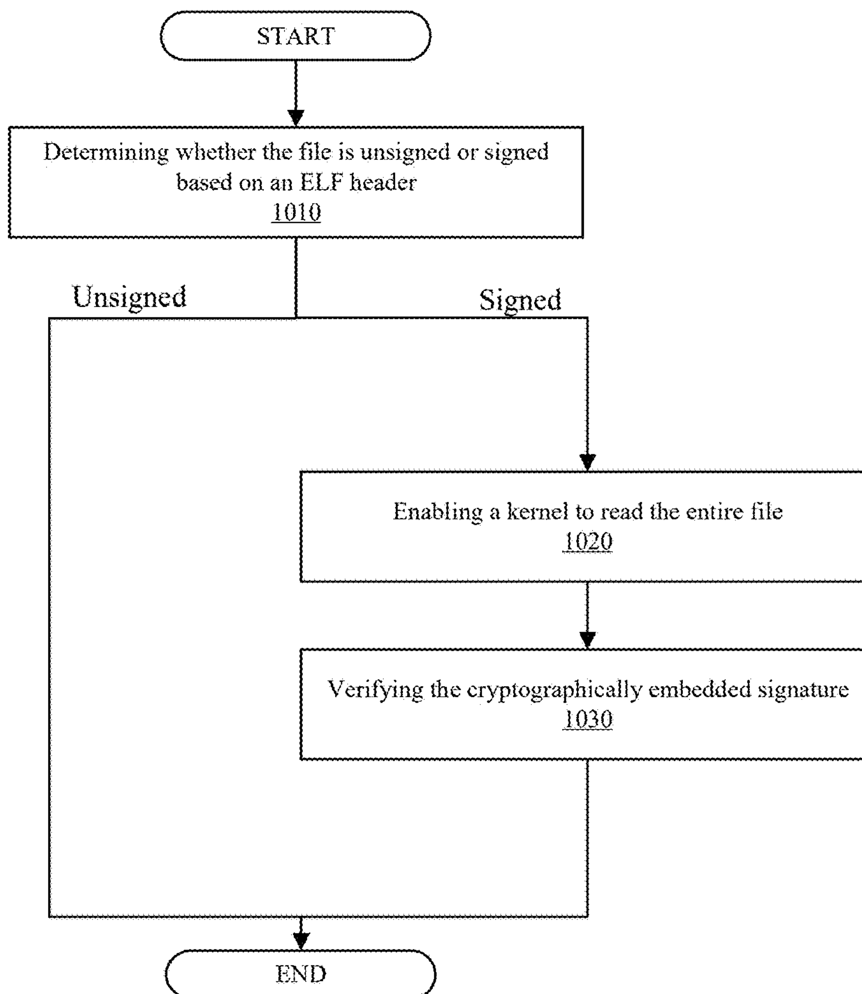


FIG. 10

DYNAMIC KERNEL SECURITY MODULE WITH KERNEL-LEVEL SIGNED BINARY VALIDATION

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application is a continuation-in-part of and claims the benefits of priority to U.S. application No. Ser. No. 19/354,412, filed Oct. 9, 2025, which is a continuation of U.S. application No. Ser. No. 18/883,251, filed on Sep. 12, 2024, each of which is incorporated by reference herein in its entirety.

TECHNICAL FIELD

[0002] The present disclosure relates generally to cybersecurity and, more specifically, to techniques for securing kernel-level system functions through hot patching a kernel, and validating kernel-level verification of signed binaries.

BACKGROUND INFORMATION

[0003] In modern network-based environments, it is increasingly important for organizations and individuals alike to securely control which users and processes are authorized to perform sensitive operations. Many computer systems enforce privileges to perform security-relevant functions through the use of one or more security policies. However, as various techniques are developed for enforcing these security policies, attackers continuously find ways to circumvent these measures.

[0004] One solution to thwart would-be attackers is to implement security policies at the kernel level. For example, the Linux™ operating system supports the use of Linux Security Modules (LSM), which enable implementing a mandatory access control (MAC) module to protect such systems from attacks. These techniques allow individual applications to be isolated, which may limit access to attackers who have compromised part of a system. Example LSMs include AppArmor™ and SELinux™, which are included in many Linux™ kernel distributions.

[0005] Despite the term "module," however, these LSMs are a static part of the Linux™ kernel. These LSMs can be either enabled or disabled for a specific Linux kernel at build time and this choice cannot be changed thereafter. The LSMs become part of the module chain and are called by the LSM subsystem every time the security check is needed inside the Linux™ kernel. These techniques thus provide very minimal flexibility and cannot be altered in any way during runtime.

[0006] Another example of security policies at the kernel level, whose implementation would allow for increased protection from would-be attackers, includes kernel-level verification of signed binaries. When executable files are introduced into a system or shared across systems, verifying their authenticity and integrity is important to ensure that executable code originates from a trusted source and has not been altered. Kernel-level validation based on signed binaries, for example using cryptographic signatures generated using a private key and verified using a corresponding public key, allows this verification to occur inside the Linux™ kernel prior to, or in connection with, execution of the executable file.

[0007] Accordingly, in view of these and other deficiencies in existing techniques, technological solutions are needed for improving security at the kernel level while maintaining flexibility for users. In particular, solutions should advantageously provide the ability to enable or disable the security module at runtime, without requiring the kernel to be rebuilt, and to validate at the kernel level signed binaries to verify security or privacy of a file.

SUMMARY

[0008] The disclosed embodiments describe non-transitory computer readable media, systems, and methods for dynamically securing kernel-level system functions. For example, in an embodiment, a non-transitory computer readable medium may include instructions that, when executed by at least one processor, cause the at least one processor to perform operations for dynamically securing kernel-level system functions. The operations may comprise hot patching of a kernel by a kernel module loaded into the kernel; identifying a kernel function initiated by a system call associated with a user-level application; intercepting the kernel function by the kernel module; making available, to a security agent, an indication of at least one operation associated with the kernel function; receiving, from the security agent, a determination of whether the at least one operation associated with the kernel function violates at least one security policy; and based on the determination indicating the at least one operation does not violate the at least one security policy, allowing the system call to the kernel; or based on the determination indicating the at least one operation violates at least one security policy, performing at least one control action.

[0009] According to a disclosed embodiment, prior to the hot patching, the kernel may be unaltered relative to a build time of the kernel.

[0010] According to a disclosed embodiment, the kernel may be associated with a kernel distribution.

[0011] According to a disclosed embodiment, the hot patching of the kernel may be performed based on a request from an additional application.

[0012] According to a disclosed embodiment, the hot patching of the kernel may include replacing an instruction of a kernel function with a replacement instruction.

[0013] According to a disclosed embodiment, the instruction may be a no-op instruction and the replacement instruction may be a JMP instruction directed to an ftrace function.

[0014] According to a disclosed embodiment, the ftrace function may be configured to replace a value of an instruction pointer register with an address of a replacement function.

[0015] According to a disclosed embodiment, the hot patching of the kernel may include corrupting an original instruction of a kernel function to generate a corrupted instruction.

[0016] According to a disclosed embodiment, the corrupted instruction may trigger a kprobes mechanism.

[0017] According to a disclosed embodiment, the kprobes mechanism may be configured to replace a value of an instruction pointer register with an address of a replacement function.

[0018] According to a disclosed embodiment, allowing the system call to the kernel may include invoking a detour buffer configured to perform the original instruction of the kernel function.

[0019] According to a disclosed embodiment, the original instruction of the kernel function performed by the detour buffer may be adjusted.

[0020] According to a disclosed embodiment, performing the hot patching of the kernel may include corrupting a CPU opcode associated with the kernel function to generate a corrupted CPU opcode.

[0021] According to a disclosed embodiment, the corrupted CPU opcode may be configured to trigger a hardware interruption to invoke the kernel module.

[0022] According to a disclosed embodiment, the at least one CPU opcode may be corrupted based on a determination that at least one of an ftrace mechanism or a kprobes mechanism has been disabled.

[0023] According to another disclosed embodiment, there may be a computer-implemented method for dynamically securing kernel-level system functions. The method may comprise hot patching of a kernel by a kernel module loaded into the kernel; identifying a kernel function initiated by a system call associated with a user-level application; intercepting the kernel function by the kernel module; making available, to a security agent, an indication of at least one operation associated with the kernel function; receiving, from the security agent, a determination of whether the at least one operation associated with the kernel function violates at least one security policy; and based on the determination indicating the at least one operation does not violate the at least one security policy, allowing the system call to the kernel; or based on the determination indicating the at least one operation violates at least one security policy, performing at least one control action.

[0024] According to a disclosed embodiment, the at least one control action may include preventing the at least one operation.

[0025] According to a disclosed embodiment, making available the indication of the at least one operation may include querying the security agent.

[0026] According to a disclosed embodiment, the security agent may be an application executing in a user space.

[0027] According to a disclosed embodiment, the method may further comprise authenticating the security agent.

[0028] According to a disclosed embodiment, the security agent may include verifying a signature of the security agent using a cryptographic key.

[0029] According to a disclosed embodiment, the at least one operation may include at least one of: execution of an executable file, writing to at least one file, or removing at least one file.

[0030] The disclosed embodiments also describe non-transitory computer readable media, systems, and methods for dynamically validating kernel-level verification of signed binaries. For example, in an embodiment, a non-transitory computer readable medium may include instructions that, when executed by at least one processor, cause the at least one processor to perform operations for dynamically validating kernel-level verification of signed binaries. The operations may comprise identifying, by a system patched by a kernel module, an executable file requested to be executed, wherein the executable file includes a cryptographic signature generated using a first key associated with a user, and wherein the cryptographic signature is embedded in the executable file, determining, by the kernel module, whether the executable file is signed based on at least one of file metadata, a file format structure, or a designated marker associated with the executable file, based on determining that the executable file is signed, identifying, by the kernel module, a second key associated with the user, wherein the second key is at least one of: stored in association with the kernel module or is provided to the kernel module by a user-space security agent, and validating the cryptographic signature embedded in the executable file, the validating comprising verifying the cryptographic signature using the second key; and based on a result of the verifying, allowing the request to be executed or denying the request to be executed.

[0031] According to a disclosed embodiment, the first key and the second key may be a part of a public-private key pair.

[0032] According to a disclosed embodiment, the first key may be stored locally, and may not leave the computing environment in which the signing occurs.

[0033] According to a disclosed embodiment, the second key may be stored in association with the kernel module and may be used for signature verification.

[0034] According to a disclosed embodiment, the operations may further comprise detecting tampering to the file, wherein any modifications to the file may invalidate the signature.

[0035] According to a disclosed embodiment, the operations may further comprise tracking the signature, wherein the file may be signed multiple times by multiple users using multiple different keys, and tracking the signature may comprise maintaining a log of all keys used to sign the file together with additional metrics, which may comprise at least one of, a time stamp, an IP address, a network identity associated with the user, or the details of the organization.

[0036] According to a disclosed embodiment, the operations may be compatible with ELF files.

[0037] According to a disclosed embodiment, the checking whether the signature has been embedded in the file may further comprise determining whether the file is unsigned or signed based on an ELF header.

[0038] According to a disclosed embodiment, when the file is determined to be unsigned, the checking whether the signature has been embedded in the file may further comprise skipping subsequent steps to reduce computing overhead.

[0039] According to a disclosed embodiment, when the file is determined to be signed, the checking whether the signature has been embedded in the file may further comprise enabling a kernel to read the entire file, or verifying the embedded signature.

[0040] According to a disclosed embodiment, the system may include a user-selectable control configured to enable selective signing of critical files.

[0041] According to another disclosed embodiment, there may be a computer-implemented method for dynamically validating kernel-level verification of signed binaries. The method may comprise identifying, by a system patched by a kernel module, an executable file requested to be executed, wherein the executable file includes a cryptographic signature generated using a first key associated with a user, and wherein the cryptographic signature is embedded in the executable file, determining, by the kernel module, whether the executable file is signed based on at least one of file metadata, a file format structure, or a designated marker associated with the executable file, based on determining that the executable file is signed, identifying, by the kernel

module, a second key associated with the user, wherein the second key is at least one of: stored in association with the kernel module or is provided to the kernel module by a user-space security agent, and validating the cryptographic signature embedded in the executable file, the validating comprising verifying the cryptographic signature using the second key; and based on a result of the verifying, allowing the request to be executed or denying the request to be executed.

[0042] Aspects of the disclosed embodiments may include tangible computer-readable media that store software instructions that, when executed by one or more processors, are configured for and capable of performing and executing one or more of the methods, operations, and the like consistent with the disclosed embodiments. Also, aspects of the disclosed embodiments may be performed by one or more processors that are configured as special-purpose processor (s) based on software instructions that are programmed with logic and instructions that perform, when executed, one or more operations consistent with the disclosed embodiments.

[0043] It is to be understood that both the foregoing general description and the following detailed description are exemplary and explanatory only, and are not restrictive of the disclosed embodiments, as claimed.

BRIEF DESCRIPTION OF THE DRAWINGS

[0044] The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate disclosed embodiments and, together with the description, serve to explain the disclosed embodiments. In the drawings:

[0045] FIG. **1** illustrates an example system environment for dynamically securing kernel-level system functions, consistent with the disclosed embodiments.

[0046] FIG. **2** is a block diagram showing an example computing device, consistent with the disclosed embodiments.

[0047] FIG. **3** is a block diagram showing an example operating system of a computing device, consistent with the disclosed embodiments.

[0048] FIG. **4** is a block diagram showing an example execution path for a kernel implementing a kernel module, consistent with the disclosed embodiments.

[0049] FIG. **5**A illustrates an example execution flow employing an ftrace technique for hot patching a kernel, consistent with the disclosed embodiments.

[0050] FIG. **5**B illustrates an example execution flow employing a kprobes technique for hot patching a kernel, consistent with the disclosed embodiments.

[0051] FIG. **5**C illustrates an example execution flow employing a hardware interruption technique for hot patching a kernel, consistent with the disclosed embodiments.

[0052] FIG. **6** is a flowchart showing an example process for dynamically securing kernel-level system functions, consistent with the disclosed embodiments.

[0053] FIG. **7** is a flowchart showing an example process for dynamically validating kernel-level verification of signed binaries, consistent with the disclosed embodiments.

[0054] FIG. **8** is a flowchart showing an example process of invalidating a signature based on modifications to a file, consistent with the disclosed embodiments.

[0055] FIG. **9** is a flowchart showing an example process for dynamically validating multiple signatures and tracking additional metrics per file, consistent with the disclosed embodiments.

[0056] FIG. **10** is a flowchart showing an example process for checking whether a signature has been embedded into the file, consistent with the disclosed embodiments.

DETAILED DESCRIPTION

[0057] In the following detailed description, numerous specific details are set forth in order to provide a thorough understanding of the disclosed example embodiments. However, it will be understood by those skilled in the art that the principles of the example embodiments may be practiced without every specific detail. Well-known methods, procedures, and components have not been described in detail so as not to obscure the principles of the example embodiments. Unless explicitly stated, the example methods and processes described herein are not constrained to a particular order or sequence, or constrained to a particular system configuration.

[0058] Additionally, some of the described embodiments or elements thereof can occur or be performed simultaneously, at the same point in time, or concurrently.

[0059] The techniques for dynamically securing kernel-level system functions described herein overcome several technological problems relating to security, efficiency, and flexibility in the fields of cybersecurity and network security. In particular, the disclosed embodiments provide a dynamic security module that may be enabled or disabled at any time. As discussed above, LSMs provide kernel-level enforcement of security policies but provide no flexibility for dynamically enabling or disabling the security modules. To address this, the disclosed systems may provide similar security features through a loadable kernel module (LKM). Despite the terms "loadable kernel module" and "Linux™ Security Module" both including the word "module," these concepts are implemented in very different ways. An LKM is an object file that contains code to extend the running kernel, or so-called base kernel, of an operating system, and is typically used to add support for new hardware (as device drivers) and/or filesystems, or for adding system calls. When the functionality provided by an LKM is no longer required, it can be unloaded to free memory and other resources.

[0060] The disclosed embodiments may enforce a security policy through an LKM. When the LSM is loaded into the running kernel, it may automatically locate various places in the running kernel memory and hot patch them accordingly. The hot patching may be implemented safely and seamlessly, such that for every specific execution thread, the execution flow either follows the original execution flow or is diverted into the LKM code. When the LKM is unloaded, it reverts all the changes done at the loading stage and returns the running Linux™ kernel into its initial state. Accordingly, the disclosed techniques enable a security policy to be enforced at the kernel level without disrupting existing kernel functionality. Moreover, the security policy enforcement can be enabled or disabled dynamically, even when the kernel is running.

[0061] Reference will now be made in detail to the disclosed embodiments, examples of which are illustrated in the accompanying drawings.

[0062] FIG. **1** illustrates an example system environment **100** for dynamically securing kernel-level system functions,

consistent with the disclosed embodiments. System environment **100** may include one or more computing devices **110**, one or more target resources **120**, and one or more security servers **130**, as shown in FIG. **1**. System environment **100** may represent a system or network environment in which various computing operations may be performed. For example, computing device **110** (or an entity associated with computing device **110**, such as identity **112**) may request to perform a computing operation within system environment **100**. In some embodiments, this may include a network-based computing operation. For example, this may include an operation involving a file or other data on target resource **120**. Alternatively or additionally, this may include a local computing operation. For example, the local computing operation may be an operation involving a file stored in computing device **110**. Accordingly, while system environment **100** is shown in FIG. **1** to include target resource **120** and security server **130** separately from computing device **110** by way of example, in some embodiments, one or both of target resource **120** and security server **130** may be integrated with computing device **110**. For example, target resource **120** may be a local resource of computing device **110** and security server **130** may be an agent or other process running on computing device **110**. Accordingly, system **100** may not necessarily be a network-based system environment and may be a local environment of computing device **110**.

[0063] The various components of system environment **100** may communicate over a network **140**. Such communications may take place across various types of networks, such as the Internet, a wired Wide Area Network (WAN), a wired Local Area Network (LAN), a wireless WAN (e.g., WiMAX), a wireless LAN (e.g., IEEE 802.11, etc.), a mesh network, a mobile/cellular network, an enterprise or private data network, a storage area network, a virtual private network using a public network, a near-field communications technique (e.g., Bluetooth, infrared, etc.), or various other types of network communications. In some embodiments, the communications may take place across two or more of these forms of networks and protocols. While system environment **100** is shown as a network-based environment, it is understood that in some embodiments, one or more aspects of the disclosed systems and methods may also be used in a localized system, with one or more of the components communicating directly with each other.

[0064] As noted above, system environment **100** may include one or more computing devices **110**. Computing device **110** may include any device that may be used for performing various computing operations as described herein. Accordingly, computing device **110** may include various forms of computer-based devices, such as a workstation or personal computer (e.g., a desktop or laptop computer), a mobile device (e.g., a mobile phone or tablet), a wearable device (e.g., a smart watch, smart jewelry, implantable device, fitness tracker, smart clothing, head-mounted display, etc.), an IoT device (e.g., smart home devices, industrial devices, etc.), or any other device that may be capable of performing a computing operation. In some embodiments, computing device **110** may be a virtual machine (e.g., based on AWS™, Azure™, IBM Cloud™, etc.), container instance (e.g., Docker™ container, Java™ container, Windows Server™ container, etc.), or other virtualized instance.

[0065] In some embodiments, computing device **110** may be associated with an identity **112**. Identity **112** may be any entity that may be associated with one or more privileges required to perform a computing operation. For example, identity **112** may be a user, an account, an application, a process, a service, an electronic signature, or any other entity or attribute associated with one or more components of system environment **100**. In some embodiments, identity **112** may be a user requesting to perform a computing operation through computing device **110**. As noted above, this may be a computing operation associated with data on computing device **110**, target resource **120**, and/or security server **130**.

[0066] Target resource **120** may include any form of remote computing device that may be the target of a computing operation or computing operation request. Examples of network resource **120** may include SQL servers, databases or data structures holding confidential information, restricted-use applications, operating system directory services, access-restricted cloud-computing resources (e.g., an AWS™ or Azure™ server), sensitive IoT equipment (e.g., physical access control devices, video surveillance equipment, etc.), and/or any other computer-based equipment or software that may be accessible over a network. Target resource **120** may include various other forms of computing devices, such as a mobile device (e.g., a mobile phone or tablet), a wearable device (a smart watch, smart jewelry, implantable device, fitness tracker, smart clothing, or head-mounted display, etc.), an IoT device (e.g., a network-connected appliance, vehicle, lighting, thermostat, room access controller, building entry controller, parking garage controller, sensor device, etc.), a gateway, switch, router, portable device, virtual machine, or any other device that may be subject to computing operations. In some embodiments, target resource **120** may be a privileged resource, such that access to the target resource **120** may be limited or restricted. For example, access to the target resource **120** may call for a privileged credential (e.g., a password, a username, an SSH key, an asymmetric key, a security or access token, etc.). In some embodiments, target resource **120** may not necessarily be a separate device from computing device **110** and may be a local resource. Accordingly, target resource **120** may be a local hard drive, database, data structure, or other resource integrated with computing device **110**.

[0067] Security server **130** may be configured to monitor and/or manage one or more security policies within system environment **100**. For example, security server **130** may manage one or more privileges associated with identity **112** (or computing device **110**) required to perform computing operations within system environment **100**. In some embodiments, security server **130** may represent a privileged access management (PAM) system or other access management system implemented within system environment **100**. Alternatively or additionally, security server **130** may be a security information and event management (SIEM) resource implemented within system environment **100**. Security server **130** may be configured to grant, track, monitor, store, revoke, validate, or otherwise manage privileges of various identities within system environment **100**. While illustrated as a separate component of system environment **100**, it is to be understood that security server **130** may be integrated with one or more other components of system environment **100**. For example, in some embodiments, security server **130** may be implemented as part of target network resource **120**, computing device **110**, or

another device of system environment **100**. In some embodiments, a separate security server may not be used and a security policy may be enforced through a security agent running on a computing device, such as security agent **340** as described below. Alternatively or additionally, the security agent may communicate with security server **130** to enforce security policies.

[0068] FIG. **2** is a block diagram showing an example computing device **110**, consistent with the disclosed embodiments. As described above, computing device **110** may be a device configured to perform (or request to perform) one or more computing operations and may include one or more dedicated processors and/or memories. For example, computing device **110** may include a processor (or multiple processors) **210**, and a memory (or multiple memories) **220**, as shown in FIG. **2**.

[0069] Processor **210** may take the form of, but is not limited to, a microprocessor, embedded processor, or the like, or may be integrated in a system on a chip (SoC). Furthermore, according to some embodiments, processor **210** may be from the family of processors manufactured by Intel®, AMD®, Qualcomm®, Apple®, NVIDIA®, or the like. Processor **210** may also be a processor based on the ARM architecture, a processor based on the RISC-V architecture, a mobile processor, a graphics processing unit, or any other form of processor. The disclosed embodiments are not limited to any particular type of processor configured in computing device **110**.

[0070] Memory **220** may include one or more storage devices configured to store instructions used by the processor **210** to perform functions related to computing device **110** described herein. The disclosed embodiments are not limited to particular software programs or devices configured to perform dedicated tasks. For example, the memory **220** may store a single program, such as a user-level application, that performs the functions associated with the disclosed embodiments, or may comprise multiple software programs. Additionally, the processor **210** may, in some embodiments, execute one or more programs (or portions thereof) remotely located from computing device **110**. Furthermore, memory **220** may include one or more storage devices configured to store data for use by the programs. Memory **220** may include, but is not limited to a hard drive, a solid state drive, a CD-ROM drive, a transient or temporary storage device (e.g., a random-access memory ("RAM")), a peripheral storage device (e.g., an external hard drive, a USB drive, etc.), a network drive, a cloud storage device, or any other storage device.

[0071] Computing device **110** may include various security components configured to evaluate kernel-level system functions performed on computing device **110**. For example, when a user-level application performs a system call involving a kernel function, the function may be intercepted by a kernel module and evaluated against a security policy. As described herein, the kernel module may be implemented through a hot patching technique such that the security policy may be enforced dynamically. For example, the kernel module may be enabled or disabled as needed during runtime of the kernel instead of at build time. Moreover, the security policy itself can be updated dynamically to address evolving security needs.

[0072] FIG. **3** is a block diagram showing an example operating system **300**, consistent with the disclosed embodiments. Operating system **300** may represent an operating system of a computing device through which a computing operation is performed (or requested to be performed). For example, operating system **300** may be an operating system of computing device **110** and thus may be executing using processor **210** and/or memory **220**, as described above. Operating system **300** may be a Linux™ operating system or distributions thereof, such as Debian-, Pacman-, RPM-, Gentoo-, Slackware-, or Android-based distributions, or similar distributions. One of ordinary skill in the art would further recognize that various aspects of the disclosed embodiments may equally apply in other types of operating platforms, such as Microsoft Windows™, Apple macOS™, Apple iOS™, Google Android™, or the like. Operating system **300** may include a kernel space **310**, as shown in FIG. **3**. Kernel space **310** may represent a protected space of operating system **300**, which may be reserved for running a kernel **320**. For example, operating system **300** may be a Linux™ operating system and kernel **320** may be a Linux™ kernel executing within kernel space **310**. Operating system **300** may further include a user space dedicated for running user applications outside of the system's kernel.

[0073] Operating system **300** may include an application **350**, which may be a user-level application. For example, application **350** may operate in a user space (which may be distinct from the kernel space), as shown in FIG. **3**. Application **350** may be any process or application executing on operating system **300** that may request to perform a computing operation associated with a kernel function. This may include a request to access a file, modify a file, delete a file (or a portion of a file), obtain a new credential, add a new user or user group, mount an image file, execute a kernel space code, perform network-related tasks, or any other action that may require privileges. For example, identity **112** may operate computing device **110** to perform a computing operation through application **350** and, as a result, application **350** may request to perform the computing operation. Accordingly, application **350** may be a product of an executable file that was executed by a privileged user or process, such as identity **112**. Alternatively or additionally, application **350** itself may be the product of running a file with privileges. The target of the computing operation may be local to computing device **110** or may be remote (e.g., at target resource **120**, etc.), as described above.

[0074] Consistent with the disclosed embodiments, a kernel module **330** may be loaded into kernel **320**. Kernel module **330** may be configured to enable the various techniques for dynamically securing kernel-level system functions described herein. Notably, kernel module **330** may be a type of module that may be implemented dynamically during a runtime of kernel **320**. For example, if kernel **320** is a Linux™ kernel, kernel module **330** may be implemented as a loadable kernel module (LKM) to extend the running kernel of operating system **300**. Accordingly, kernel module **330** may be loaded dynamically, similar to LKMs used for adding new device drivers, filesystems, or system calls. Kernel module **330** may thus be distinct from a Linux™ Security Module (LSM) which must be implemented during build time of kernel **320**. Accordingly, kernel module **330** may provide improved flexibility over a LSM, allowing a user to enable or disable kernel module **330** as needed.

[0075] Once loaded into kernel **320**, kernel module **330** may be configured to perform a hot patching technique to enable the various security functions disclosed herein. For example, this may include implementing hot patch **322**, as

shown in FIG. **3**. As used herein "hot patching" (also referred to as "hotpatching") may refer to any form of patching or modification to a kernel function. In some embodiments, hot patching may include exploiting various features of the kernel to redirect the execution flow of one or more kernel functions from the normal kernel to kernel module **330**. For example, hot patch **322** may enable kernel module **330** to intercept various kernel functions invoked by system calls of application **350**. Kernel module **330** may then analyze these kernel functions to determine whether they violate one or more security policies and may take necessary steps to enforce the security policies. This hot patching technique may thus enable kernel module **330** to provide kernel-level security functions, similar to a Linux™ Security Module, while maintaining the flexibility of a loadable kernel module. Various example hot patching techniques are described in greater detail below.

[0076] While a Linux™ operating system is used by way of example, one of ordinary skill in the art would recognize that similar forms of modules may be implemented in various other systems. For example, kernel module **330** may be implemented as a kernel loadable module (kld) in a FreeBSD™ operating system, a kernel extension (kext) in a macOS™ operating system, a kernel extension module in an AIX™ operating system, a dynamically loadable kernel module in an HP-UX™ operating system, a kernel-mode driver in a Windows NT operating system, a downloadable kernel module (DKM) in a VxWorks™ operating system, or the like.

[0077] Moreover, even within a specific OS family, kernel module **330** may be applied to many different distributions of the kernel. For example, within the Linux™ family, kernel module **330** may be used in many different distributions, such as Debian™, Fedora™, Arch™, Ubuntu™, or other distributions. Prior to hot patching by kernel module **330**, kernel **320** may be unaltered relative to a build time of kernel **320**. In other words, the hot patching may not require any form of treatment to the kernel prior to the hot patching. The security advantages described herein may thus be achieved using a wide variety of kernel types without any required special customizations. It is to be understood that a kernel unaltered relative to a build time of the kernel may already include the various customizations based on different distributions described above. However, it is the hot patching techniques described herein that alter the kernel's behavior to provide the disclosed security improvements.

[0078] According to some embodiments, kernel module **330** may interface with a security agent, such as security agent **340**, to analyze one or more kernel functions. In some embodiments, this may include providing an indication of an operation associated with a kernel function to security agent **340**, which may then determine whether the operation violates a security policy. Accordingly, security agent **340** may store or have access to one or more security policies associated with system environment **100**. In some embodiments, security agent **340** may be an application executing on computing device **110**. For example, as shown in FIG. **3**, security agent **340** may be an application executing in a user space of operating system **300**. As one example, security agent **340** may be a privilege management application, such as the CyberArk™ Endpoint Privilege Manager (EPM), which may enforce various security policies within an operating system. In some embodiments, security agent **340** (or another application) may provide a request or other form of instruction to kernel module **330** to perform hot patch **322**.

[0079] In some embodiments, security agent **340** may communicate with one or more external sources for enforcing a security policy. For example, security agent **340** may be configured to provide an indication of an operation associated with a kernel function to security server **130**, which may analyze the operation and provide an indication of whether the operation violates a security policy. Alternatively or additionally, security server **130** may store one or more security policies and security agent **340** may access the security policies to evaluate the operation. The various functions of security agent **340** may be shared with security server **130** in various ways, which may depend on the particular application.

[0080] As indicated above, hot patch **322** may redirect the normal execution flow of one or more kernel functions of kernel **320** through kernel module **330**. FIG. **4** is a block diagram showing an example execution path for a kernel implementing kernel module **330**, consistent with the disclosed embodiments. Kernel **320** may define a kernel function **400**, which may include a series of operations. Normally, when kernel function **400** is initiated (e.g., by a system call of application **350**) an operation **402** may be invoked, as shown in FIG. **4**. However, based on hot patch **322**, an alternate function **410** defined by kernel module **330** may be called instead. For example, the system call may cause initiation of operation **412**, as shown in FIG. **4**. Operation **410** may enable kernel module **330** to evaluate operation **402** (or various other operations associated with kernel **320**) against a security policy, as indicated above. The operation can be any operation that may be important from a security point of view. For example, operation **402** may include execution of a specific executable file, opening a specific file for writing, removal of a file, or various other operations. Operation **414** may determine that operation **402** does not violate a security policy and thus may cause the flow to return to operation **402**. Alternatively, based on a determination that operation **402** does violate the security policy, operation **416** may direct the flow to skip operation **402** (and/or various other operations), effectively denying the kernel function invoked by the system call.

[0081] As indicated in FIG. **4**, the hot patching techniques performed by kernel module **330** may be implemented in an atomic and safe way. For any given execution thread, the execution flow will either follow the original flow, or will be diverted to kernel module **330** code. Notably, kernel module **330** may not necessarily replace any original kernel functions. For example, if allowed, the execution flow is still directed to original kernel function **400**, effectively calling them in a build-time-defined order. Accordingly, kernel module **330** may be implemented with no (or at least minimal) side effects relative to the original kernel function.

[0082] The specific techniques utilized for hot patch **322** may vary depending on the system and the particular build of kernel **320**. Accordingly, there may be no guarantee that specific functions are located at specific addresses. For example, function addresses may be highly volatile and may be managed by the compiler and the linker at build time, and by the boot loader (GRUB, UEFI, etc.) at the boot time. Kernel module **330** may thus be configured to locate a particular address of kernel function **400** and various other functions to be patched within kernel **320**. In some embodi-

ments, this search may be performed dynamically to accommodate a variety of different builds of kernel **320**. In the example of a Linux™ kernel, this dynamic searching may include using a "kallsyms_lookup_name" technique to find an address of kernel function **400**. In some embodiments, (e.g., for Linux™ kernel versions 5.7.0 or later), this may further include the use of a kernel probe (also referred to as "kprobes") mechanism to find the address of "kallsyms_lookup_name." In some cases (e.g., where "kallsyms_lookup_name" fails), the kprobes technique may be used directly to find the addresses of specific functions. Alternatively or additionally (e.g., if kprobes fails), a printk-based heuristic technique may be used to find an address of kernel function **400**. Once kernel function **400** has been identified, various techniques for implementing hot patch **322** may be used, which may also depend on a particular build of kernel **320**. Various example techniques are shown in FIGS. **5**A-**5**C and described below.

[0083] In some embodiments, hot patch **322** may be implemented using a function tracer (or "ftrace") mechanism associated with Linux kernels. Normally, ftrace may be exploited for collecting a status of a running kernel or for kernel debugging. However, ftrace may also allow the execution of custom code to be injected at the beginning of a Linux kernel function. Relevant to the present disclosure, this ftrace mechanism may be used to implement kernel module function **410**, as described above. FIG. **5**A illustrates an example execution flow **500**A employing an "ftrace" technique for hot patching kernel **320**, consistent with the disclosed embodiments.

[0084] For the disclosed ftrace hot patching techniques to work, support from the compiler and the linker may be utilized. In other words, kernel **320** may be built with "ftrace" functionality enabled. If enabled, the build system of kernel **320** may automatically enable all the needed compiler and linker options, which may instruct the compiler and the linker to generate a so-called "prologue" at the beginning of every kernel function before its actual code. For example, if ftrace is enabled, kernel function **400** may include prologue operation **510**, as shown in FIG. **5**A. Consistent with the present disclosure, prologue operation **510** may be in the form of a "no-op" CPU instruction and thus may effectively do nothing when executed by the CPU. However, the presence of prologue operation **510** provides the ability for prologue operation **510** to be replaced by another one—for example, by a jump to another address (or "JMP") operation. By registering a tracer for the kernel function **400**, the ftrace mechanism replaces the first no-op CPU instruction at the beginning of the corresponding function by the JMP instruction, as shown in FIG. **5**A. This may force the execution flow to continue from the internal ftrace handler to perform ftrace function **520**.

[0085] Ftrace function **520** may include an operation **522** of saving the CPU registers values in its internal memory. In operation **524**, ftrace function **520** looks up the return address and finds the corresponding tracer to be called. It then calls the user-registered handler. In operation **526**, ftrace function **520** may then restore the CPU registers values from the saved location. This may be advantageous because execution of the registered handlers in operation **524** may change the CPU registers. Accordingly, to make the original execution flow unaffected by these changes, restoring of previous values of CPU registers may be performed. After completion of the handlers, ftrace function **520** returns back to the original function in operation **528**.

[0086] Typically, after returning to the original function, the CPU continues execution of the code starting from the second CPU instruction (i.e., operation **402**) which, thanks to the compiler help, is the first actual instruction of the kernel function. To implement hot patch **322** using the ftrace mechanism, kernel module **330** may register an ftrace handler that replaces the value of an instruction pointer (IP) register with the address of kernel module function **410**. Accordingly, when ftrace function **520** restores the saved CPU registers in operation **526** before continuing execution of the original kernel function, it writes the address of kernel module function **410** into the CPU IP register. Therefore, after returning from ftrace function **520**, execution continues at operation **412** instead of operation **402** of the original kernel function **400**, as shown in FIG. **5**A. Kernel module function **410** may then proceed to enforce a security policy, as described above with respect to FIG. **4**. If the requested operation does not violate a security policy, operation **414** may include a JMP operation to jump to operation **402**. Alternatively, if the requested operation does violate a security policy, operation **416** may include a return operation to return from kernel module function **410**, which may be interpreted by the initial caller as a return from the original kernel function **400**.

[0087] In some embodiments, hot patch **322** may be implemented using a kernel dynamic probe (or "kprobes") mechanism associated with Linux kernels. Kprobes provides a lightweight interface for kernel modules to implant probes and register corresponding probe handlers, similar to ftrace handlers discussed above. Specifically, the kprobes mechanism allows a break-point instruction to be inserted into almost any place in the kernel memory. The resulting interruption will be handled by the assigned interrupt handler. Relevant to the present disclosure, this kprobes mechanism may be used to implement kernel module function **410**, as described above. FIG. **5**B illustrates an example execution flow **500**B employing a "kprobes" technique for hot patching kernel **320**, consistent with the disclosed embodiments.

[0088] Similar to the ftrace mechanism described above, the kprobes mechanism must be enabled at the kernel build time. The kprobes mechanism operates by allowing a breakpoint CPU instruction to be inserted at a specific place in a kernel function. For example, a breakpoint **512** may be inserted into kernel function **400**, as shown. This effectively corrupts the CPU instruction at this breakpoint. As a result of the corrupted instruction, the execution flow will be directed to a kprobes interrupt handler **530**. To deal with the CPU instruction corruption, the kprobes mechanism first saves the affected instruction (in this case, operation **402**) and then spoils it. Similar to ftrace function **510**, kprobes interruption handler **530** saves the CPU registers in operation **532**, calls registered kprobes handlers in operation **534**, and restores the CPU registers upon returning from the handlers in operation **536**. Typically, when all the registered kprobes handlers are completed, kprobes interruption handler **530** will single-step the original saved instruction of the now-corrupted operation **402** and then will jump to the next non-corrupted CPU instruction of the original code, in this case, operation **514**.

[0089] To implement hot patch **322** using the kprobes mechanism, kernel module **330** may register a kprobes

handler that replaces the value of an instruction pointer (IP) register with the address of kernel module function **410**, much like the ftrace handler described above. Accordingly, when kprobes interruption handler **530** restores the saved CPU registers in operation **536**, it writes the address of kernel module function **410** into the CPU IP register. The kprobes handler registered by kernel module **330** also clears the portion of code instructing kprobes interruption handler **530** to single-step the saved CPU instruction before returning.

[0090] As a result, after restoring the CPU registers, kprobes interruption handler **530** does not perform the saved version of operation **402** and jumps to operation **412** instead of operation **514**, as shown in FIG. **5**B. Kernel module function **410** may then proceed to enforce a security policy, as described above with respect to FIG. **4**. If the requested operation does violate a security policy, operation **416** may include a return operation to return from kernel module function **410**, which may be interpreted by the initial caller as a return from the original kernel function **400**. If the requested operation does not violate a security policy, operation **414** may include a JMP operation to return to kernel function **400**. However, due to the kprobes handler clearing the portion of code instructing kprobes interruption handler **530** to single-step the saved CPU instruction, operation **402** may not have been completed. Accordingly, kernel module function **410** may prepare a detour buffer saving its own copy of the original operation **402**. Instead of jumping to the next uncorrupted operation **514**, operation **414** may jump to the saved operation **540**, and then, in operation **542**, may jump to operation **514**. In some embodiments, operation **540** may be adjusted relative to original operation **402**, for example, to account for breakpoint **512**.

[0091] Both the ftrace and kprobes hot patching techniques described above employ built-in features of the Linux™ kernel. A person of ordinary skill in the art would recognize that similar techniques may be employed for other built-in mechanisms, including mechanisms of kernels in other forms of operating systems. As indicated above, both the ftrace and kprobes hot patching techniques require the mechanisms to be enabled during build time. Accordingly, kernel module **330** may attempt various techniques disclosed herein in a predetermined order of priority. For example, kernel module **330** may first attempt the ftrace technique and, if ftrace is not enabled, attempt the kprobes technique (or vice versa). In some embodiments, hot patch **322** may be implemented using a "raw" approach, which may rely on a hardware interruption technique rather than any built-in mechanisms being enabled. Accordingly, this technique may be applied as an alternative to the ftrace and kprobes techniques described above. For example, the hardware interruption technique may be attempted if neither the ftrace or kprobes mechanisms are enabled. While an example order for attempting the various techniques is provided herein, the present disclosure is not limited to any particular order.

[0092] FIG. **5**C illustrates an example execution flow **500**C employing a hardware interruption technique for hot patching kernel **320**, consistent with the disclosed embodiments. The hardware interruption technique differs from flows **500**A and **500**B described above in that it implements the hot patching by editing the CPU opcodes directly. Execution flow **500**C is similar to kprobes-based approach illustrated in execution flow **500**B in that it corrupts the needed CPU instructions but, in contrast with kprobes, the hardware interruption technique includes placing an illegal CPU opcode instead of a break-point one. For example, as shown in FIG. **5**C, kernel module **330** may corrupt operation **402** to include an illegal operation **516**. When the execution thread hits such illegal operation **516**, a hardware interruption is triggered, which may be handled by kernel module function **410**. In some embodiments, similar to with the kprobes technique, kernel module function **410** may store a copy of original operation **402** as operation **550**, which may be triggered in operation **414**. In operation **552**, the execution flow may jump to the first non-corrupted operation **514**.

[0093] As described above, kernel module **330** may not necessarily evaluate an operation against a security policy itself. For example, kernel module **330** may communicate with a user-level application, such as security agent **340**, which may determine whether a specific operation violates a security policy and should be denied. Accordingly, kernel module **330** may implement a communicator function, which may enable efficient communication between kernel and user space components without significant overhead. In some embodiments, kernel module **330** may employ additional security features to ensure these communications are with security agent **340** and not a hostile actor. For example, to identify security agent **340**, kernel module **330** may use a reliable public key signature verification technique, in which security agent **340** binaries are signed with a private key at the moment of build. Kernel module **330** may store or have access to the corresponding public key (e.g., in its source code), which it may use to verify signature of security agent **340** binaries. This public key may be configured such that it may only be used for signature verification, not for the signature substitution. This approach may allow kernel module **330** to reliably identify security agent **340** binaries as well as to make sure that the binaries were not altered in any way after they were signed.

[0094] FIG. **6** is a flowchart showing an example process **600** for dynamically securing kernel-level system functions, consistent with the disclosed embodiments. Process **600** may be performed by at least one processor of a computing device, such as processor **210** described above. It is to be understood that throughout the present disclosure, the term "processor" is used as a shorthand for "at least one processor." In other words, a processor may include one or more structures that perform logic operations whether such structures are collocated, connected, or dispersed. In some embodiments, a non-transitory computer readable medium may contain instructions that when executed by a processor cause the processor to perform process **600**. Further, process **600** is not necessarily limited to the steps shown in FIG. **6**, and any steps or processes of the various embodiments described throughout the present disclosure may also be included in process **600**, including those described above with respect to, for example, FIGS. **3**, **4**, and **5**A-C.

[0095] In step **610**, process **600** may include hot patching of a kernel by a kernel module loaded into the kernel. For example, this may include performing hot patch **322** by kernel module **330**, as described above. Prior to the hot patching, the kernel may be unaltered relative to a build time of the kernel. In other words, other than customizations associated with a particular distribution of the kernel (or any other variations introduced during build time), the kernel may be untreated prior to the hot patching. Accordingly, hot patching of the unaltered kernel may result in changing the

kernel's behavior, as described above. In some embodiments, the hot patching of the kernel may be performed based on a request from an additional application. For example, the additional application may be security agent **340** described above.

[0096] Hot patching of the kernel may be implemented in a variety of ways, as described above. For example, consistent with the ftrace technique described above, hot patching of the kernel may include replacing an instruction of a kernel function with a replacement instruction. In some embodiments, the instruction may be a no-op instruction (e.g., operation **510**) and the replacement instruction may be a JMP instruction directed to an ftrace function, as described above. The ftrace function may be configured to replace a value of an instruction pointer register with an address of a replacement function. For example, as described above, kernel module **330** may register an ftrace handler that replaces the value of an instruction pointer (IP) register with the address of kernel module function **410**. Accordingly, when the ftrace function restores saved CPU registers it may write the address of a kernel module function into the CPU IP register, as described above.

[0097] As another example, consistent with the kprobes technique described above, hot patching of the kernel may include corrupting an original instruction of a kernel function to generate a corrupted instruction. For example, this may include corrupting operation **402** to include breakpoint **512**, as described above. The corrupted instruction may trigger a kprobe mechanism (i.e., triggering kprobes function **530**, as described above). The kprobe mechanism may be configured to replace a value of an instruction pointer register with an address of a replacement function. For example, as described above, kernel module **330** may register a kprobes handler that replaces the value of an instruction pointer register with the address of kernel module function **410**.

[0098] As another example, consistent with the hardware interruption technique described above, hot patching of the kernel may include corrupting a CPU opcode associated with the kernel function. The corrupted CPU opcode may be configured to trigger a hardware interruption to invoke the kernel module. For example, the hot patching may include corrupting operation **516**, as described above. In some embodiments, an ftrace or kprobes technique may be prioritized over the hardware interruption technique. For example, the at least one CPU opcode may be corrupted based on a determination that at least one of an ftrace mechanism or a kprobes mechanism has been disabled.

[0099] In step **620**, process **600** may include identifying a kernel function initiated by a system call associated with a user-level application. For example, step **620** may include identifying kernel function which may be initiated by a system call associated with application **350**, as described above.

[0100] In step **630**, process **600** may include intercepting the kernel function by the kernel module. For example, step **630** may include intercepting kernel function **400** by kernel module **330**. The kernel function may be intercepted as a result of the various hot patching techniques described above. For example, the kernel function may be intercepted through registering a handler invoked by an ftrace or kprobes mechanism, where the handler replaces a value of an instruction pointer register, as described above. Alternatively or additionally, the kernel function may be intercepted through a hardware interruption.

[0101] In step **640**, process **600** may include making available, to a security agent, an indication of at least one operation associated with the kernel function. In some embodiments, the security agent may be an application executing in a user space. For example, step **640** may include making available an indication of operation **402** to security agent **340**. In some embodiments, making available the indication of the at least one operation includes querying the security agent to determine whether the at least one operation violates a security policy. The at least one operation may include a wide variety of operations associated with a kernel function, including, for example, execution of an executable file, writing to at least one file, removing at least one file, or any other operation that may present potential security concerns.

[0102] In step **650**, process **600** may include receiving, from the security agent, a determination of whether the at least one operation associated with the kernel function violates at least one security policy. For example, step **650** may include receiving a result from security agent **340** indicating whether the at least one operation violates a security policy. In some embodiments, process **600** may further include authenticating the security agent. For example, as described above, authenticating the security agent may include verifying a signature of the security agent using a cryptographic key.

[0103] In step **660**, based on the determination indicating the at least one operation does not violate the at least one security policy, process **600** may include allowing the system call to the kernel. For example, step **660** may include returning to operation **402** via operation **414**, as described above. In some embodiments, where the original operation is corrupted, allowing the system call to the kernel may include invoking a detour buffer configured to perform the original instruction of the kernel function. For example, step **660** may include performing one of operations **540** or **550**, as described above. Step **660** may further include jumping to a subsequent noncorrupted operation, such as operation **514**.

[0104] In step **670**, based on the determination indicating the at least one operation violates at least one security policy, process **600** may include performing at least one control action. In some embodiments, the at least one control action may include preventing the at least one operation. Alternatively or additionally, the at least one control action may include various other actions responsive to the violation of the security policy. For example, the control action may include generating an alert or report that the security policy has been violated. As another example, the control action may include revoking or suspending a privilege of an identity, such as identity **112**. The control action may include various other actions, such as terminating an application (e.g., application **350**), disabling access to a resource (e.g., target resource **120**), or the like. In some embodiments, the control action may be performed, at least in part, by security agent **340**.

[0105] FIG. **7** is a flowchart showing an example process **700** for dynamically determining file signature security at the kernel level, consistent with the disclosed embodiments. Process **700** may be performed by at least one processor of a computing device, such as processor **210** described above. It is to be understood that throughout the present disclosure, the term "processor" is used as a shorthand for "at least one

processor." In other words, a processor may include one or more structures that perform logic operations whether such structures are collocated, connected, or dispersed. In some embodiments, a non-transitory computer readable medium may contain instructions that when executed by a processor cause the processor to perform process **700**. Further, process **700** is not necessarily limited to the steps shown in FIG. **7**, and any steps or processes of the various embodiments described throughout the present disclosure may also be included in process **700**, including those described above with respect to, for example, FIGS. **3**, **4**, and **5**A-C.

[0106] In step **710**, process **700** may include identifying, by a system patched by a kernel module, a file signed by a user. For example, this may include files of any type in any format, wherein the file is able to cryptographically store a key base signature. Identifying a file that has been signed may further include looking at the metadata of the file to determine file history and other file details that may be helpful in a security analysis. Step **710** may also include identifying signature data associated with the file. In some embodiments, the file comprises a cryptographic signature generated using a first key that may be a private key associated with the user that corresponds to an asymmetric key pair, and the file further comprises signature metadata, such as a signer identifier, a certificate, a public key, or a key identifier, that enables verification. The private key used to generate the signature may be stored locally in the computing environment in which the signing occurs and may not leave that computing environment. Accordingly, the kernel module may validate the signature using verification data that corresponds to the signer, such as a second key, that may be a public key that corresponds to the first key, certificate, or trust chain information, without requiring access to the private key.

[0107] In step **720**, process **700** may include receiving, by the kernel module, a second key from the user. For example, this may include a public key, which may be assigned to a group of users, a single user, an organization, or the like. The public key may be part of a public-private key pair, such that validation includes comparing the public key provided by the user to the private key that the user uses to sign the file. In some embodiments, the pairing of the public key to private key may allow the user's private key to be kept private while creating an indication of what the system may use to validate that user's signature. In some embodiments, the same user may have multiple different keys assigned to them for different purposes, such that different types of signatures mean different things. For instance, one may be more secure than another for a signature that is more important. When a user has multiple keys that could be used, they may provide the kernel module with the key that is expected to be used for the particular file. In some embodiments, the user may provide the kernel module with multiple keys that could be used to further allow for more validation options. Further, the system may include a user-selectable control configured to enable selective signing of critical files, such that certain files must be signed a certain way. The signature type may also be indicated to the system for validation. Extra keys may be required when a file is marked as critical or the like, such that more validation is required both to sign and to validate the signatures on a file that has been marked as critical.

[0108] In step **730**, process **700** may include validating the signature embedded in the file. As used herein, a signature being "embedded in" a file refers to signature data being stored within the file in a manner that is parsable by the kernel or kernel module, such as within a designated portion of the file format, within an appended data region, or within another structured location associated with the file.

[0109] For example, validating the signature embedded in the file may include some combination of sub-steps of identifying a file (**731**), determining whether the signature is embedded in the file (**732**), validating that the signature corresponds to the second key (**733**), or the like. Step **731** may further allow for validation to occur by ensuring that the file that is being validated is the same file from step **710**. This may include, for example, ensuring that the file matches property and content that is expected for the file is present. The file may be any file type that is able to cryptographically store a signature. Step **732** may further include determining the signature and the properties of the signature of the file that was identified in steps **700** and **731**. In some embodiments, the signature needs to be analyzed to determine who the signature belongs to. Step **733** may further include comparing the information received from the user to the signature identified in step **732**. In some embodiments, the signature may be validated by the public key in order for validation to be completed. In some embodiments, the key may be stored in a secure location, for example in a credential vault. In some embodiments, the validating of step **730** may be performed in response to a kernel-level request to use the file in an executable manner. For example, the validating may be triggered responsive to a request to execute the file, to load the file for execution, or to map the file into memory with execute permissions. Based on a result of the validating, the kernel module may allow the request to proceed when the signature is verified, or may deny the request, generate an alert, or perform another control action when the signature fails verification, is missing, or is otherwise determined to be invalid.

[0110] FIG. **8** is a flowchart showing an example process **800** for dynamically securing kernel-level system functions, consistent with the disclosed embodiments. Process **800** may be performed by at least one processor of a computing device, such as processor **210** described above. It is to be understood that throughout the present disclosure, the term "processor" is used as a shorthand for "at least one processor." In other words, a processor may include one or more structures that perform logic operations whether such structures are collocated, connected, or dispersed. In some embodiments, a non-transitory computer readable medium may contain instructions that when executed by a processor cause the processor to perform process **800**. Further, process **800** is not necessarily limited to the steps shown in FIG. **8**, and any steps or processes of the various embodiments described throughout the present disclosure may also be included in process **800**, including those described above with respect to, for example, FIGS. **3**, **4**, **5**A-C, **6** and **7**. Process **800** is a more complex version of process **700**, wherein elements that are cross-listed with the same number may describe the same techniques disclosed above.

[0111] In step **810**, process **800** may include determining whether tampering to the executable file has occurred, including modifications to the executable file after the signature was generated. For example, this may include analyzing metadata associated with the executable file to determine whether changes have been made after signing, and/or analyzing the executable file content to determine

whether any portion of the executable file covered by the signature has been modified. In some embodiments, changes to the executable file may include changes to the content of the executable file, timestamps, file name, file properties, permissions, ownership, or access control settings. In some embodiments, for a signature to be treated as valid, modifications to the executable file that affect the signed content may be performed prior to generating the signature. In some embodiments, if the executable file is changed after signing in a manner that affects the signed content, the executable file may be re-signed after the changes have been made.

[0112] In step **820**, process **800** may include invalidating the signature on the executable file. For example, in response to determining in step **810** that tampering to the executable file has occurred after the signature was generated, the signature may be treated as no longer valid, even if a verification operation would otherwise indicate that a corresponding key matches. In some embodiments, invalidating the signature may further include updating a status associated with the executable file to indicate the signature is invalid, denying a kernel-level request to execute the executable file, generating an alert, or requesting that a new signature be generated for a non-tampered version of the executable file. In some embodiments, once the signature has been invalidated based on tampering, validation may be restored by re-signing the executable file and repeating process **800**.

[0113] FIG. **9** is a flowchart showing an example process **900** for dynamically securing kernel-level system functions, consistent with the disclosed embodiments. Process **900** may be performed by at least one processor of a computing device, such as processor **210** described above. It is to be understood that throughout the present disclosure, the term "processor" is used as a shorthand for "at least one processor." In other words, a processor may include one or more structures that perform logic operations whether such structures are collocated, connected, or dispersed. In some embodiments, a non-transitory computer readable medium may contain instructions that when executed by a processor cause the processor to perform process **900**. Further, process **900** is not necessarily limited to the steps shown in FIG. **9**, and any steps or processes of the various embodiments described throughout the present disclosure may also be included in process **900**, including those described above with respect to, for example, FIGS. **3**, **4**, **5**A-C, **6**, **7**, and **8**.

[0114] In step **910**, process **900** may include tracking signatures associated with an executable file. For example, this may include determining and storing information associated with an executable file and one or more signatures corresponding to the executable file, Consistent with the discussion above, an executable file may have one or more signatures at any given point including multiple signatures generated by different users and/or multiple signatures generated by the same user over time.

[0115] In some embodiments, tracking includes maintaining signature-related information to support validation, auditing, or security analysis. Tracking may include storing information about each signature when the executable file is analyzed and/or when a signature event is detected.

[0116] In step **920**A-Z, process **900** may include storing, identifying, or accessing signatures for individual users. For example, this may include storing information in a database or similar structure that includes different signatures associated with the executable file. In some embodiments, storing each of the signatures may include storing information indicating which executable file was signed, whether multiple signatures were received for the same executable file, and/or whether signatures were received for different versions or copies of the executable file.

[0117] In some embodiments, there may be one signature **920**A, while in other embodiments there may be a multitude of signatures **920**A-**920**Z, and the disclosed embodiments are not limited to any particular number of signatures.

[0118] In step **930**, process **900** may include tracking of additional metrics associated with each user that has signed. For example, this may include creating a repository for information associated with a user. This may also include determining and storing information associated with an executable file. Consistent with the disclosed embodiments, an executable file may have one or more signatures at any given point. In some embodiments user information may be kept track of. User information may be kept track of because, as discussed above, the volume of users, either from a large volume of different users, or from a large volume of signatures from the same user over time. Tracking may include storing information about each of the signatures at some cadence when the file is analyzed.

[0119] In steps **940**A-**940**Z, information may be tracked or stored based on user. The user that all of the information is stored according to and associated with may be a unique user each time, or associated with a unique signature on the executable file. For example, in some situations the same user may sign the same file more than once. When information is stored, the information may be a combination or sub-combination of time stamp **950**, IP address **960**, network identity **970**, organization details **980**, or additional user data. In some embodiments, time stamp **950** may refer to the time at which the file was signed by that user. In some embodiments, IP address **960** may refer to the IP address of the device that was used to create and execute that user's signature on that particular instance. In some embodiments, network identity **970** may refer to the user's network presence, including details on how they are integrated into or interacting with the system. In some embodiments, organization details **980** may include information about the user's place in the organization, including job title, who they report to, who reports to them, or other organizational details, etc. Other properties may also be stored as associated with the user in step **930**/**940**.

[0120] FIG. **10** is a flowchart showing an example process **1000** for dynamically determining whether an executable file has been signed, consistent with the disclosed embodiments. Process **1000** may be performed by at least one processor of a computing device, such as processor **210** described above. It is to be understood that throughout the present disclosure, the term "processor" is used as a shorthand for "at least one processor." In other words, a processor may include one or more structures that perform logic operations whether such structures are collocated, connected, or dispersed. In some embodiments, a non-transitory computer readable medium may contain instructions that when executed by a processor cause the processor to perform process **1000**. Further, process **1000** is not necessarily limited to the steps shown in FIG. **10**, and any steps or processes of the various embodiments described throughout the present disclosure may also be included in process **1000**, including those described above with respect to, for example, FIGS. **3**, **4**, **5**A-C, **6**, **7**, **8**, and **9**.

[0121] In step **1010**, process **1000** may include determining whether the executable file is unsigned or signed based on an ELF header. For example, this may include analyzing just the ELF header of a file to determine whether or not a signature exists. In a signed file, the ELF header may indicate whether a file has been signed or not. While in some embodiments, the ELF header does not provide enough information to allow for verification or validation, the header may be able to provide a binary determination of whether or not a signature exists. If the validation process is only triggered once a determination that has been made that a signature does exist in the file, computing resources may be saved by only reading through files that do have signatures, as determined by the ELF header.

[0122] In some embodiments, determining whether the executable file is unsigned or signed based on an ELF header may comprise parsing one or more header fields or associated header structures to detect a marker indicative of signature data. For example, the marker may include a designated note entry, a designated section name, a designated program header segment, or other identifier indicating that signature data is present in the file. When the marker is absent, the executable file may be treated as unsigned for purposes of step **1010**.

[0123] In step **1020**, process **1000** may include enabling a kernel to read an entire executable file. For example, this may include reading any type of executable file to determine where the signature may be within the executable file. In some embodiments, when reading the entire executable file, the kernel may be able to identify the signature or signatures within an executable file that require further validation.

[0124] In step **1030**, process **1000** may include verifying the embedded signature. For example, this may include performing cryptographic signature verification operations using a public key that corresponds to the private key used to generate the signature, and determining whether the embedded signature is valid for the executable file. In some embodiments, the private key may be used to sign the executable file and the public key is used to verify the signature.

[0125] It is to be understood that the disclosed embodiments are not necessarily limited in their application to the details of construction and the arrangement of the components and/or methods set forth in the following description and/or illustrated in the drawings and/or the examples. The disclosed embodiments are capable of variations, or of being practiced or carried out in various ways.

[0126] The disclosed embodiments may be implemented in a system, a method, and/or a computer program product. The computer program product may include a computer readable storage medium (or media) having computer readable program instructions thereon for causing a processor to carry out aspects of the present invention.

[0127] The computer readable storage medium can be a tangible device that can retain and store instructions for use by an instruction execution device. The computer readable storage medium may be, for example, but is not limited to, an electronic storage device, a magnetic storage device, an optical storage device, an electromagnetic storage device, a semiconductor storage device, or any suitable combination of the foregoing. A non-exhaustive list of more specific examples of the computer readable storage medium includes the following: a portable computer diskette, a hard disk, a random access memory (RAM), a read-only memory (ROM), an erasable programmable read-only memory (EPROM or Flash memory), a static random access memory (SRAM), a portable compact disc read-only memory (CD-ROM), a digital versatile disk (DVD), a memory stick, a floppy disk, a mechanically encoded device such as punch-cards or raised structures in a groove having instructions recorded thereon, and any suitable combination of the foregoing. A computer readable storage medium, as used herein, is not to be construed as being transitory signals per se, such as radio waves or other freely propagating electromagnetic waves, electromagnetic waves propagating through a waveguide or other transmission media (e.g., light pulses passing through a fiber-optic cable), or electrical signals transmitted through a wire.

[0128] Computer readable program instructions described herein can be downloaded to respective computing/processing devices from a computer readable storage medium or to an external computer or external storage device via a network, for example, the Internet, a local area network, a wide area network and/or a wireless network. The network may comprise copper transmission cables, optical transmission fibers, wireless transmission, routers, firewalls, switches, gateway computers and/or edge servers. A network adapter card or network interface in each computing/processing device receives computer readable program instructions from the network and forwards the computer readable program instructions for storage in a computer readable storage medium within the respective computing/processing device.

[0129] Computer readable program instructions for carrying out operations of the present invention may be assembler instructions, instruction-set-architecture (ISA) instructions, machine instructions, machine dependent instructions, microcode, firmware instructions, state-setting data, or either source code or object code written in any combination of one or more programming languages, including an object oriented programming language such as Smalltalk, C++ or the like, and conventional procedural programming languages, such as the "C" programming language or similar programming languages. The computer readable program instructions may execute entirely on the user's computer, partly on the user's computer, as a stand-alone software package, partly on the user's computer and partly on a remote computer or entirely on the remote computer or server. In the latter scenario, the remote computer may be connected to the user's computer through any type of network, including a local area network (LAN) or a wide area network (WAN), or the connection may be made to an external computer (for example, through the Internet using an Internet Service Provider). In some embodiments, electronic circuitry including, for example, programmable logic circuitry, field-programmable gate arrays (FPGA), or programmable logic arrays (PLA) may execute the computer readable program instructions by utilizing state information of the computer readable program instructions to personalize the electronic circuitry, in order to perform aspects of the present invention.

[0130] Aspects of the present invention are described herein with reference to flowchart illustrations and/or block diagrams of methods, apparatus (systems), and computer program products according to embodiments of the invention. It will be understood that each block of the flowchart illustrations and/or block diagrams, and combinations of

blocks in the flowchart illustrations and/or block diagrams, can be implemented by computer readable program instructions.

[0131] These computer readable program instructions may be provided to a processor of a general purpose computer, special purpose computer, or other programmable data processing apparatus to produce a machine, such that the instructions, which execute via the processor of the computer or other programmable data processing apparatus, create means for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks. These computer readable program instructions may also be stored in a computer readable storage medium that can direct a computer, a programmable data processing apparatus, and/or other devices to function in a particular manner, such that the computer readable storage medium having instructions stored therein comprises an article of manufacture including instructions which implement aspects of the function/act specified in the flowchart and/or block diagram block or blocks.

[0132] The computer readable program instructions may also be loaded onto a computer, other programmable data processing apparatus, or other device to cause a series of operational steps to be performed on the computer, other programmable apparatus or other device to produce a computer implemented process, such that the instructions which execute on the computer, other programmable apparatus, or other device implement the functions/acts specified in the flowchart and/or block diagram block or blocks.

[0133] The flowcharts and block diagrams in the Figures illustrate the architecture, functionality, and operation of possible implementations of systems, methods and computer program products according to various embodiments of the present invention. In this regard, each block in the flowcharts or block diagrams may represent a software program, segment, or portion of code, which comprises one or more executable instructions for implementing the specified logical function(s). It should also be noted that, in some alternative implementations, the functions noted in the block may occur out of the order noted in the figures. For example, two blocks shown in succession may, in fact, be executed substantially concurrently, or the blocks may sometimes be executed in the reverse order, depending upon the functionality involved. It will also be noted that each block of the block diagrams and/or flowchart illustration, and combinations of blocks in the block diagrams and/or flowchart illustration, can be implemented by special purpose hardware-based systems that perform the specified functions or acts, or combinations of special purpose hardware and computer instructions.

[0134] The descriptions of the various embodiments of the present invention have been presented for purposes of illustration, but are not intended to be exhaustive or limited to the embodiments disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art without departing from the scope and spirit of the described embodiments. The terminology used herein was chosen to best explain the principles of the embodiments, the practical application or technical improvement over technologies found in the marketplace, or to enable others of ordinary skill in the art to understand the embodiments disclosed herein.

[0135] It is expected that during the life of a patent maturing from this application many relevant virtualization platforms, virtualization platform environments, trusted cloud platform resources, cloud-based assets, protocols, communication networks, security tokens and authentication credentials, and code types will be developed, and the scope of these terms is intended to include all such new technologies a priori.

[0136] It is appreciated that certain features of the invention, which are, for clarity, described in the context of separate embodiments, may also be provided in combination in a single embodiment. Conversely, various features of the invention, which are, for brevity, described in the context of a single embodiment, may also be provided separately or in any suitable subcombination or as suitable in any other described embodiment of the invention. Certain features described in the context of various embodiments are not to be considered essential features of those embodiments, unless the embodiment is inoperative without those elements.

[0137] Although the invention has been described in conjunction with specific embodiments thereof, it is evident that many alternatives, modifications and variations will be apparent to those skilled in the art. Accordingly, it is intended to embrace all such alternatives, modifications and variations that fall within the spirit and broad scope of the appended claims.

What is claimed is:

1. A non-transitory computer readable medium including instructions that, when executed by at least one processor, cause the at least one processor to perform operations for dynamically validating kernel-level verification of signed binaries, the operations comprising:

identifying, by a system patched by a kernel module, an executable file requested to be executed, wherein the executable file includes a cryptographic signature generated using a first key associated with a user, and wherein the cryptographic signature is embedded in the executable file;

determining, by the kernel module, whether the executable file is signed based on at least one of file metadata, a file format structure, or a designated marker associated with the executable file;

based on determining that the executable file is signed, identifying, by the kernel module, a second key associated with the user, wherein the second key is at least one of: stored in association with the kernel module or is provided to the kernel module by a user-space security agent;

validating the cryptographic signature embedded in the executable file, the validating comprising:

verifying the cryptographic signature using the second key; and

based on a result of the verifying, allowing the request to be executed or denying the request to be executed.

2. The non-transitory computer readable medium of claim **1**, wherein the first key and the second key are part of a public-private key pair.

3. The non-transitory computer readable medium of claim **2**, wherein the first key is stored locally and does not leave a computing environment in which signing occurs.

4. The non-transitory computer readable medium of claim **2**, wherein the second key is stored in association with the kernel module and is used for signature verification.

5. The non-transitory computer readable medium of claim **1**, wherein the operations further comprise detecting tampering to the file, and wherein any modifications to the file invalidate the signature.

6. The non-transitory computer readable medium of claim **1**, wherein the operations further comprise tracking the signature, wherein the executable file is signed multiple times by multiple users using multiple different keys, and tracking the signature comprises maintaining a log of all keys used to sign the executable file together with additional metrics comprising at least one of: a time stamp, an IP address, a network identity associated with the user, or the details of the organization.

7. The non-transitory computer readable medium of claim **1**, wherein the operations are compatible with ELF files.

8. The non-transitory computer readable medium of claim **1**, wherein the determining whether the signature has been embedded in the file further comprises determining whether the executable file is unsigned or signed based on an ELF header.

9. The non-transitory computer readable medium of claim **8**, wherein when the file is determined to be unsigned, the checking whether the signature has been embedded in the file further comprises skipping subsequent steps to reduce computing overhead.

10. The non-transitory computer readable medium of claim **8**, wherein when the executable file is determined to be signed, the checking whether the signature has been embedded in the executable file further comprises:

enabling a kernel to read the entire file; and

verifying the embedded signature.

11. The non-transitory computer readable medium of claim **1**, wherein the system includes a user-selectable control configured to enable selective signing of critical files.

12. A computer-implemented method for dynamically validating kernel-level verification of signed binaries, comprising:

identifying, by a system patched by a kernel module, an executable file requested to be executed, wherein the executable file includes a cryptographic signature generated using a first key associated with a user, and wherein the cryptographic signature is embedded in the executable file;

determining, by the kernel module, whether the executable file is signed based on at least one of file metadata, a file format structure, or a designated marker associated with the executable file;

based on determining that the executable file is signed, identifying, by the kernel module, a second key associated with the user, wherein the second key is at least one of: stored in association with the kernel module or is provided to the kernel module by a user-space security agent;

validating the cryptographic signature embedded in the executable file, the validating comprising:

verifying the cryptographic signature using the second key; and

based on a result of the verifying, allowing the request to be executed or denying the request to be executed.

13. The computer-implemented method of claim **12**, wherein the first key and the second key are part of a public-private key pair.

14. The computer-implemented method of claim **13**, wherein the first key is stored locally and does not leave a computing environment in which signing occurs.

15. The computer-implemented method of claim **13**, wherein the second key is stored in association with the kernel module and is used for signature verification.

16. The computer-implemented method of claim **12**, further comprising detecting tampering to the file, wherein any modifications to the executable file invalidate the signature.

17. The computer-implemented method of claim **12**, further comprising tracking the signature, wherein the executable file is signed multiple times by multiple users using multiple different keys, and tracking the signature comprises maintaining a log of all keys used to sign the executable file together with additional metrics comprising at least one of: a time stamp, an IP address, a network identity associated with the user, or the details of the organization.

18. The computer-implemented method of claim **12**, wherein the operations are compatible with ELF files.

19. The computer-implemented method of claim **12**, wherein the determining whether the signature has been embedded in the executable file further comprises determining whether the file is unsigned or signed based on an ELF header.

20. The computer-implemented method of claim **19**, wherein when the file is determined to be unsigned, the checking whether the signature has been embedded in the file further comprises skipping subsequent steps to reduce computing overhead.

21. The computer-implemented method of claim **19**, wherein when the file is determined to be signed, the checking whether the signature has been embedded in the file further comprises:

enabling a kernel to read the entire file; and

verifying the embedded signature.

22. The computer-implemented method of claim **12**, wherein the system includes a user-selectable control configured to enable selective signing of critical files.

* * * * *