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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
26 June 2025 (26.06.2025)

WIPO | PCT

(10) International Publication Number
WO 2025/137696 A1

(51) International Patent Classification:
G11C 29/04 (2006.01) *G06F 21/62* (2013.01)
G06F 3/06 (2006.01) *H03M 13/05* (2006.01)
H03M 13/03 (2006.01) *G06F 16/13* (2019.01)
G06F 21/64 (2013.01)

(21) International Application Number:
PCT/US2024/061714

(22) International Filing Date:
23 December 2024 (23.12.2024)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
63/614,430 22 December 2023 (22.12.2023) US

(71) Applicant: **NTT RESEARCH, INC.** [US/US]; 940 Stewart Dr., Sunnyvale, California 94085 (US).

(72) Inventors: **BOYLE, Elette**; 940 Stewart Drive, Sunnyvale, California 94085 (US). **KOMARGODSKI, Ilan**; 940 Stewart Drive, Sunnyvale, California 94085 (US). **VAFA, Neekon**; 940 Stewart Drive, Sunnyvale, California 94085 (US).

(74) Agent: **DENARO, James**; P.O. Box 34783, Bethesda, Maryland 20827 (US).

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

(54) Title: MEMORY INTEGRITY VERIFICATION USING HIERARCHICAL DATA STRUCTURES

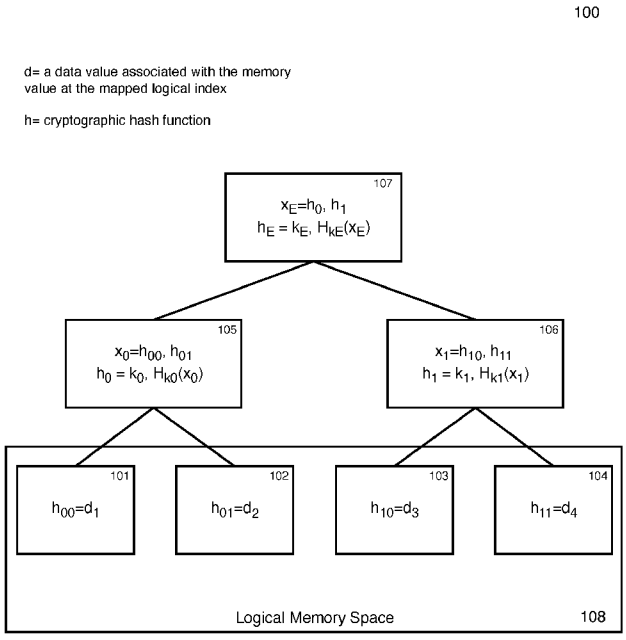

Fig. 1

(57) **Abstract:** A method and system for detecting memory errors and corruption in a computer system using a hash tree data structure. The method involves initializing a hash tree data structure that covers an allocated memory space, with each node in the hash tree representing a block of memory and leaf nodes containing actual memory contents. The physical memory has a physical word size and a logical word size of polylog(n), wherein n represents the logical size of the memory. A cryptographic hash function is used to calculate a hash value for each block of memory, and the calculated hash value is stored in the corresponding node of the hash tree. During a verification phase, the hash values for each block of memory are recalculated and compared with the stored hash values in the hash tree.

WO 2025/137696 A1

[Continued on next page]

TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, WS, ZA, ZM, ZW.

(84) **Designated States** *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, CV, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SC, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, ME, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Published:

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

MEMORY INTEGRITY VERIFICATION USING HIERARCHICAL DATA STRUCTURES

GOVERNMENT LICENSE RIGHTS

[1] This invention was made with government support under DARPA Agreement No. HR00112020023, National Science Foundation Award CNS-2154149, National Science Foundation Award DGE-2141064, and a Simons Investigator award. The government has certain rights in the invention.

TECHNICAL FIELD

[2] The present disclosure relates generally to checking a memory to detect faults in memory operations.

BACKGROUND

[3] Computer systems often rely on memory to store and retrieve data. This memory can be subject to errors and corruption due to various factors such as hardware faults, software bugs, or malicious attacks. Therefore, it is imperative to have mechanisms in place to detect and correct such errors to ensure the reliability and integrity of the data stored in the memory.

[4] One common technique for detecting memory errors and corruption is the use of hash functions. The output, known as the hash value or hash code, is typically a sequence of alphanumeric characters. The primary characteristic of a hash function is that it is deterministic, meaning that the same input will consistently produce the same output. In the context of memory error detection, a hash function can be applied to a block of memory to produce a hash value. This hash value can then be stored and later used to verify the integrity of the memory block. If the memory block is subsequently altered, recalculating the hash value and comparing it with the stored hash value will reveal the alteration.

[5] A hash tree, also known as a Merkle tree, is a data structure that extends the concept of hash functions to a hierarchical level. In a hash tree, each node represents or is related to a block of memory and contains a hash value calculated from its child nodes. The root node of the tree contains a hash value that represents the combined hash values of all the memory blocks. This structure allows for efficient verification of large amounts of data.

[6] However, the use of hash trees for memory error detection requires a substantial amount of computational resources. The process of calculating and comparing hash values can be computationally intensive, especially for large memory spaces. Furthermore, the hash tree itself requires additional memory to store the hash values for each memory block. These requirements can limit the applicability of hash tree-based memory error detection in scenarios where computational or memory resources are constrained.

SUMMARY

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

[8] In general, in a first aspect, the method features online checking of a memory to detect faults in memory operations. This involves referencing a connected physical memory having a physical word size and a logical word size of polylog (n), wherein n represents the logical size of the memory. At a data store positioned at a memory checker module as a proxy between a requester and the physical memory, which may or may not be remote, a d-ary hash function-based tree construction is generated, wherein d equals polylog (n). The tree construction is mapped to a logical memory address space for the physical memory such that each leaf of the tree stores data corresponding to a logical word in the memory. Embodiments of the method may include one or more of the following features. The logical words in the leaves may be copied to their parent nodes in a second level of the tree. At each second level node of the tree, a keyed hash function of the logical words concatenated may be calculated and a corresponding hash value and a hash function's cryptographic keys may be stored. At each node at a third and higher levels of the tree, the results of the hash functions from all of its child nodes may be stored, and a keyed hash function of the stored results from all of its child nodes may be calculated and stored. In another aspect, the method may feature performing a read operation by reading the value of a leaf in the tree associated with an index and reading all nodes from that leaf to the root and verifying that the data value of the parent is consistent with the hash of the child, the hash of the values of the parent are correct, and the root node is identical to the root node in the memory checker's local memory. In yet another aspect, the method may feature performing a write operation with a value for an index, writing the data field at the leaf in the tree associated with the index, and updating all nodes from that leaf to the root. In some implementations, the logical word size may be set to be the physical word size by breaking up the logical word into d chunks in the bottom layer. The memory checker may have a public local state and security may be maintained even if the hashes are available to an adversary. The memory checker may be deterministic, non-adaptive, and operate online. Upon a logical instruction, the memory checker may have query complexity $q = \Theta(\log n / \log \log n)$. The memory checker may be configured to support logical words of size $w\ell = \Theta(w)$, resulting in a bandwidth of $\Theta(\log n / \log \log n)$. The memory checker may be part of the memory, may not require protection of its local state, and the local state may be accessible to an adversary.

BRIEF DESCRIPTION OF THE DRAWINGS

[9] Non-limiting and non-exhaustive examples are described with reference to the following figures.

[10] **FIG. 1** is a schematic representation of a binary tree structure utilized in a memory checker system, illustrating the hierarchical arrangement of memory locations and the aggregation of cryptographic hash values at each level of the tree.

[11] **FIG. 2** illustrates an example architecture for implementing aspects of protected integrity processing according to at least one embodiment, the architecture including a proxy device and a memory system.

[12] **FIG. 3** illustrates a system for ensuring data integrity including a client device and a memory system, in accordance with at least one embodiment.

[13] **FIG. 4** illustrates an example of a computer system architecture that can be used to implement the claimed systems and methods.

[14] **FIG. 5** illustrates further details of the computer system architecture that can be used to implement the claimed systems and methods.

DETAILED DESCRIPTION

[15] The following description sets forth exemplary aspects of the present disclosure. It should be recognized, however, that such description is not intended as a limitation on the scope of the present disclosure. Rather, the description also encompasses combinations and modifications to those exemplary aspects described herein.

[16] In some implementations, a memory checker or verifier may employ a hash tree data structure to detect memory errors and corruption. The hash tree data structure, also known as a Merkle tree, is a tree in which every non-leaf node is labelled with the cryptographic hash of the labels or values of its child nodes. This structure provides an efficient and secure way of verifying the content of large data structures.

[17] The memory checker or verifier creates a hash tree data structure that covers the whole memory space. Each node in the tree represents a block of memory, and the leaf nodes contain the actual memory contents. The memory checker or verifier calculates a hash value for each block of memory and stores it in the corresponding node of the hash tree. The hash value is calculated using a cryptographic hash function, such as SHA-256, that produces a fixed-size output for any input.

[18] During the verification phase, the memory checker or verifier recalculates the hash values for each block of memory and compares them with the stored hash values in the hash tree. If

the hash values match, the memory contents are considered valid. If they don't match, it indicates a memory error or corruption.

[19] The hash tree data structure allows for efficient verification of the memory contents. The memory checker or verifier can traverse the tree from the root node to the leaf node that corresponds to the memory block in question. This reduces the time and computational complexity of the verification process.

[20] If a memory error or corruption is detected, the memory checker or verifier can use the hash tree to locate the exact location of the error. The hash tree provides a hierarchical structure that allows for precise error localization. This technique provides a high level of accuracy and reliability, making it suitable for use in systems and applications where data integrity is paramount.

[21] In some implementations, the memory checker or verifier initiates an initialization process. This process involves the creation of a hash tree data structure that spans the complete memory space. As illustrated in FIG. 1, the hash tree structure 100 is mapped to a logical memory address space 108 for the physical memory. Each leaf of the tree stores data corresponding to a logical word in the memory. For instance, logical words d1 101, d2 102, d3 103, and d4 104 are stored in each leaf of the tree.

[22] In the second level of the tree, the logical words in the leaves 101, 102, 103, and 104 are copied to their parent nodes, specifically nodes 105 and 106. At each of these second-level nodes, a keyed hash function of the concatenated logical words is calculated. The result of this hash function, along with a corresponding hash function's cryptographic secret or key, is stored at the respective node of the tree.

[23] For nodes at the third level and higher levels of the tree, the process is slightly different. At each of these nodes, the results of the hash functions from all of its child nodes are stored. Then, a keyed hash function of the stored hash results from all of its child nodes is calculated. The result of this hash function, along with the corresponding hash function's cryptographic secret or key, is stored at the respective node of the tree. The root of the tree, denoted as xe 107, is the topmost node in this hierarchical structure.

[24] In some cases, a read operation may be performed by reading the value of a leaf in the tree associated with a specific index i, for example d1 101. The memory checker or verifier then reads all nodes from that leaf to the root node xe 107. During this process, the memory checker verifies that: (a) the data value of the parent node is consistent with the hash of the child node; (b) the hash of the values of the parent node are correct; and (c) the root node xe 107 is identical to the root node stored in the memory checker's local memory. By traversing the tree from the

root to the leaf node that corresponds to the memory block in question, the memory checker can precisely identify the location of an error or corruption.

[25] In some variations, the memory checker has a public local state. This means that the local state of the memory checker, which includes the root of the hash tree and other relevant information, is accessible and can be viewed by an adversary. Despite this, the security of the memory checker is maintained even if the hashes are available to an adversary. Even if an adversary has access to the hash values, without the correct input data and hash function keys, they cannot feasibly reverse-engineer the original memory contents or manipulate the memory without detection.

[26] In other variations, the memory checker is deterministic, non-adaptive, and operates online in that the memory checker operates in a fixed, predictable manner and does not adapt its behavior based on the memory contents or the sequence of memory operations. Specifically, for each logical write or read to an index $i \in [n]$, the memory checker performs physical queries at the physical locations exactly according to predetermined sequences W_i and R_i, respectively. This behavior is independent of the local state, contents on the database, or random coins. This deterministic and non-adaptive operation simplifies the design and analysis of the memory checker.

[27] Upon a logical instruction, the memory checker has a query complexity $q = \Theta(\log n / \log \log n)$. In some configurations, the memory checker is designed to support logical words of size $w_\ell = \Theta(w)$, resulting in a bandwidth of $\Theta(\log n / \log \log n)$.

[28] In some implementations, the memory checker or verifier performs a write operation by writing the data field at the leaf in the tree associated with a specific index. For instance, upon a write operation with value v for index i, the memory checker writes the data field v at the leaf in the tree associated with index i. This leaf could be one of the logical words d1 101, d2 102, d3 103, or d4 104. After writing the data field, the memory checker updates all nodes from that leaf to the root xe 107. This process involves updating the hash values stored at each node along the path from the leaf to the root, as illustrated in FIG. 1.

[29] In some configurations, the logical word size is set to be the same as the physical word size by breaking up the logical word into d chunks in the bottom layer of the tree. This is illustrated in FIG. 1, where each leaf of the tree, specifically logical words d1 101, d2 102, d3 103, and d4 104, represents a chunk of the logical word. This configuration allows the memory checker to handle memory blocks of varying sizes efficiently, which can be beneficial in applications that involve large data structures or variable-sized data elements. By breaking up the logical word into d chunks, the memory checker can map each chunk to a separate leaf

node in the tree, thereby ensuring that each leaf node corresponds to a distinct part of the logical word. This configuration also simplifies the process of updating the hash values in the tree during write operations, as each chunk of the logical word can be updated independently. When a write operation is performed, the memory checker writes the data field at the leaf in the tree associated with a specific index. This leaf could be one of the logical words in the tree. After writing the data field, the memory checker updates all parent nodes from that leaf to the root. This process involves updating the hash values stored at each node along the path from the leaf to the root.

[30] Furthermore, this configuration allows the memory checker to efficiently handle large memory spaces. The query complexity of the memory checker, which is the number of physical memory queries performed by the memory checker for each logical memory operation, grows logarithmically with the size of the memory. This ensures that the memory checker can handle large memory spaces efficiently.

[31] While reference herein is made to a memory system, the memory system could be any form of physical memory, data store, or database, whether embodied in a volatile or nonvolatile system, that can be referenced logically, and these terms are used interchangeably.

[32] FIG. 2 illustrates an example architecture 200 for implementing aspects of protected integrity processing according to at least one embodiment. As shown, the architecture includes a proxy device 202 and a memory system 204. Architecture 200 may extend a traditional client-server architecture with a proxy enclave 208 (e.g., a trusted environment), which may intercept all operations (e.g., data requests) issued by a client (e.g., client 206) via a network interface of the proxy device 202. The proxy enclave 208 may intercept all responses from the memory system 204 in order to check integrity of the responses. In this case, the application 210 (which hosts the proxy enclave 208) may manage untrusted tasks, such as socket communication with the memory system 204. The application 210 may also include and/or manage one or more network interfaces of the proxy device 202 which may be utilized to receive client requests from a client 206 and transmit responses from the memory system 204 to the client 206.

[33] The memory system 204 can be hosted on any untrusted machine, and the proxy enclave 208 may be hosted by the application 210 on proxy device 202. In some embodiments, the application 210 and proxy enclave 208 may be co-located with the client 206 or the memory system 204 so long as a chipset of the client 206 or memory system 204 is configured to manage secure memory spaces (e.g., an enclave). The client 206 may establish a secure transport layer security (TLS) channel with the proxy enclave 208 to protect data in transit. The

client 206 may use remote attestation primitives to ensure that the proxy device 202 is genuine (e.g., hosting proxy enclave 208) before establishing the TLS channel.

[34] In at least one embodiment, the proxy device 202 may further include an operating system (OS) 214, a hypervisor 216, a solid state drive (SSD) 218, and chipset 220. Similarly, the memory system 204 may include an OS 222, a hypervisor 224, an SSD 226, and chipset 228.

[35] Utilizing the configuration illustrated in FIG. 2 requires no modifications to the client (e.g., the client 206) or the server (e.g., the memory system 204). This may be especially beneficial given that new database engines or memory systems are developed regularly in industry and clients may be resource constrained. The proxy device 202 may include the following components: enclave-resident code for performing integrity checks, unprotected memory 215 (in some embodiments, containing the tree), and an enclave-resident state (for authenticating the tree and other security records and functions). Given the embodiments described herein, storing the entirety of the proxy's state is not necessary to perform integrity checks.

[36] The proxy device 202 (also referred to as the "proxy") described herein may provide standard operations of a memory, including read and write. While the proxy device 202 may expose these operations (e.g., execute corresponding function calls), it may not implement them entirely. Instead, the proxy device 202 may utilize operations of the memory manager 212, which may implement such operations. The memory manager 212 can manage the data stored within memory 213 (e.g., untrusted memory). One contribution of the proxy device 202 discussed herein is to efficiently check integrity of all responses from the memory system 204.

[37] In some embodiments, the proxy places no trust in the memory system 204. Since the client 206 may be only generating requests, no trust is placed on them either. Thus, in some embodiments the only trusted component is the proxy device 202, which may be hardened using a trusted execution environment.

[38] The illustration of the threat model in FIG. 2 identifies system components that may be under an attacker's control. Except the chipset 220 that runs the proxy enclave 208, the adversary may fully control all hardware in the proxy device 202, including I/O peripherals such as the disk and network. It may be assumed that the attacker is unable to physically open and manipulate the proxy enclave 208. Additionally, it may be assumed that the chipset 220 provides integrity guarantees to the protected enclave memory. The adversary may also control privileged system software, including the OS 222, hypervisor 216, and/or BIOS layers on which the memory system 204 depends.

[39] With reference to FIG. 3, in at least one embodiment, a memory system 302 (e.g., the memory system 204 of FIG. 2) may include a trusted execution environment 300. The functions of proxy device 202 of FIG. 2 may be performed by the trusted execution environment 300. Thus, FIG. 3 depicts an embodiment in which the functionality of proxy device 202 is provided at the memory system 302. The trusted execution environment 300 may include a secure memory space 305 (e.g., the proxy enclave 208 of FIG. 2) that may be managed by a chipset of the memory system 302.

[40] The trusted execution environment 300 may include the integrity checker application 304 and protected code that is executed within the trusted execution environment 300 to perform the integrity checking features discussed herein. In some embodiments, code integrity checker application 304 of the integrity checker application 304 (e.g., the application 210 of FIG. 2) and/or data within the trusted execution environment 300 may be accessed by a processor of the chip set but inaccessible to other systems, devices, or applications other than the integrity checker application 304.

[41] In some embodiments, untrusted data store 310, untrusted operating system (OS) 312, and/or an untrusted hypervisor 314 may be stored within untrusted RAM 318.

[42] The integrity checker application 304 can receive a request (e.g., a query) from client program 319 operating on client device 320. The integrity checker application 304 may forward request (e.g., the query) to the untrusted data store 310 and check for correctness of the results returned from the untrusted data store 310 before forwarding to the client device 320. Since the integrity checker application 304 is trusted, the protections of the trusted execution environment 300 may be utilized to host the code of the integrity checker application 304. For performing integrity checks, the integrity checker application 304 may store and/or maintain an integrity summary which may contain at least a portion (e.g., a root node) of a data structure (e.g., a hash tree) within the trusted executed environment 300. The trusted execution environment 300 may have direct access to the larger untrusted RAM 318. Several optimizations may be implemented based on caching and concurrency, and on standard benchmarks as discussed above.

[43] **Example Computer System**

[44] FIGs. 4 and 5 depict example computer systems useful for implementing various embodiments described in the present disclosure. Various embodiments may be implemented, for example, using one or more computer systems, such as computer system 500 shown in FIG. 4. One or more computer system(s) 500 may be used, for example, to implement any of the embodiments discussed herein, as well as combinations and sub-combinations thereof.

[45] Computer system 500 may include one or more processors (also called central processing units, processing devices, or CPUs), such as a processor 504. Processor 504 may be connected to a communication infrastructure 506 (e.g., such as a bus).

[46] Computer system 500 may also include user input/output device(s) 503, such as monitors, keyboards, pointing devices, etc., which may communicate with communication infrastructure 506 through user input/output interface(s) 502. One or more of processors 504 may be a graphics processing unit (GPU). In an embodiment, a GPU may be a processor that is a specialized electronic circuit designed to process mathematically intensive applications. The GPU may have a parallel structure that is efficient for parallel processing of large blocks of data, such as mathematically intensive data common to computer graphics applications, images, videos, etc.

[47] Computer system 500 may also include a main memory 508, such as random-access memory (RAM). Main memory 508 may include one or more levels of cache. Main memory 508 may have stored therein control logic (i.e., computer software, instructions, etc.) and/or data. Computer system 500 may also include one or more secondary storage devices or secondary memory 510. Secondary memory 510 may include, for example, a hard disk drive 512 and/or a removable storage device or removable storage drive 514. Removable storage drive 514 may interact with a removable storage unit 518. Removable storage unit 518 may include a computer-usable or readable storage device having stored thereon computer software (control logic) and/or data. Removable storage drive 514 may read from and/or write to removable storage unit 518.

[48] Secondary memory 510 may include other means, devices, components, instrumentalities, or other approaches for allowing computer programs and/or other instructions and/or data to be accessed by computer system 500. Such means, devices, components, instrumentalities, or other approaches may include, for example, a removable storage unit 522 and an interface 520. Examples of the removable storage unit 522 and the interface 520 may include a program cartridge and cartridge interface, a removable memory chip (such as an EPROM or PROM) and associated socket, a memory stick and USB port, a memory card and associated memory card slot, and/or any other removable storage unit and associated interface.

[49] Computer system 500 may further include communications interface 524 (e.g., network interface). Communications interface 524 may enable computer system 500 to communicate and interact with any combination of external devices, external networks, external entities, etc. (individually and collectively referenced as remote device(s), network(s), entity(ies) 528). For example, communications interface 524 may allow computer system 500 to communicate with

external or remote device(s), network(s), entity(ies) 528 over communications path 526, which may be wired and/or wireless (or a combination thereof), and which may include any combination of LANs, WANs, the Internet, etc. Control logic and/or data may be transmitted to and from computer system 500 via communications path 526.

[50] Computer system 500 may also be any of a personal digital assistant (PDA), desktop workstation, laptop or notebook computer, netbook, tablet, smartphone, smartwatch or other wearable devices, appliance, part of the Internet-of-Things, and/or embedded system, to name a few non-limiting examples, or any combination thereof.

[51] Computer system 500 may be a client or server computing device, accessing or hosting any applications and/or data through any delivery paradigm, including but not limited to remote or distributed cloud computing solutions; local or on-premises software ("on-premise" cloud-based solutions); "as a service" models (e.g., content as a service (CaaS), digital content as a service (DCaaS), software as a service (SaaS), managed software as a service (MSaaS), platform as a service (PaaS), desktop as a service (DaaS), framework as a service (FaaS), backend as a service (BaaS), mobile backend as a service (MBaaS), infrastructure as a service (IaaS), etc.); and/or a hybrid model including any combination of the foregoing examples or other services or delivery paradigms.

[52] FIG. 5 illustrates an example machine of a computer system 900 within which a set of instructions, for causing the machine to perform any one or more of the operations discussed herein, may be executed. In alternative implementations, the machine may be connected (e.g., networked) to other machines in a LAN, an intranet, an extranet, and/or the Internet. The machine may operate in the capacity of a server or a client machine in a client-server network environment, as a peer machine in a peer-to-peer (or distributed) network environment, or as a server or a client machine in a cloud computing infrastructure or environment.

[53] The machine may be a personal computer (PC), a tablet PC, a set-top box (STB), a Personal Digital Assistant (PDA), a cellular telephone, a web appliance, a server, a network router, a switch or bridge, a specialized application or network security appliance or device, or any machine capable of executing a set of instructions (sequential or otherwise) that specify actions to be taken by that machine. Further, while a single machine is illustrated, the term "machine" shall also be taken to include any collection of machines that individually or jointly execute a set (or multiple sets) of instructions to perform any one or more of the methodologies discussed herein.

[54] The example computer system 900 includes a processing device 902, a main memory 904 (e.g., read-only memory (ROM), flash memory, dynamic random-access memory (DRAM)

such as synchronous DRAM (SDRAM), etc.), a static memory 906 (e.g., flash memory, static random-access memory (SRAM), etc.), and a data storage device 918, which communicate with each other via a bus 930.

[55] Processing device 902 represents one or more processing devices such as a microprocessor, a central processing unit, or the like. More particularly, the processing device may be complex instruction set computing (CISC) microprocessor, reduced instruction set computing (RISC) microprocessor, very long instruction word (VLIW) microprocessor, or processor implementing other instruction sets, or processors implementing a combination of instruction sets. Processing device 902 may also be one or more special-purpose processing devices such as an application-specific integrated circuit (ASIC), a field-programmable gate array (FPGA), a digital signal processor (DSP), network processor, or the like. The processing device 902 is configured to execute instructions 926 for performing the operations and steps discussed herein.

[56] The computer system 900 may further include a network interface device 908 to communicate over the network 920. The computer system 900 also may include a video display unit 910, an alphanumeric input device 912 (e.g., a keyboard), a cursor control device 914 (e.g., a mouse), a graphics processing unit 922, a signal generation device 916 (e.g., a speaker), graphics processing unit 922, video processing unit 928, and audio processing unit 932.

[57] The data storage device 918 may include a machine-readable medium 924 (also known as a computer-readable storage medium) on which is stored one or more sets of instructions 926 (e.g., software instructions) embodying any one or more of the operations described herein. The instructions 926 may also reside, completely or at least partially, within the main memory 904 and/or within the processing device 902 during execution thereof by the computer system 900, where the main memory 904 and the processing device 902 also constitute machine-readable storage media.

[58] In an example, the instructions 926 include instructions to implement operations and functionality corresponding to the disclosed subject matter. While the machine-readable storage medium 924 is shown in an example implementation to be a single medium, the term “machine-readable storage medium” should be taken to include a single medium or multiple media (e.g., a centralized or distributed database, and/or associated caches and servers) that store the one or more sets of instructions 926. The term “machine-readable storage medium” shall also be taken to include any medium that is capable of storing or encoding a set of instructions 926 for execution by the machine and that cause the machine to perform any one or more of the operations of the present disclosure. The term “machine-readable storage

medium" shall accordingly be taken to include, but not be limited to, solid-state memories, optical media, and magnetic media.

[59] Data Integrity Service for Cloud Storage

[60] The systems and methods described herein may be utilized as an additional data integrity service on top of cloud storage services, providing enhanced security and reliability for users storing sensitive or critical data in the cloud. This implementation can be particularly beneficial for organizations dealing with large volumes of data or those operating in regulated industries where data integrity is paramount.

[61] In a cloud storage context, the memory checker module described herein can be implemented as a middleware layer between the client application and the cloud storage provider's API. This middleware may intercept read and write operations to the cloud storage, applying the integrity checking mechanisms before allowing the operations to proceed. The d-ary hash function-based tree construction may be maintained by this middleware, with the root of the tree potentially stored in a secure enclave or trusted execution environment within the cloud infrastructure.

[62] The implementation of the memory checker in a cloud storage context may involve several key components:

[63] 1. Logical to Physical Address Mapping: The memory checker may maintain a mapping between the logical addresses used by the client applications and the physical addresses used by the cloud storage provider. This mapping may be implemented using a distributed hash table or a similar data structure that can efficiently handle large address spaces.

[64] 2. Tree Construction and Maintenance: The d-ary hash function-based tree construction described in the claims may be implemented as a distributed data structure. Each node in the tree may be stored as a separate object in the cloud storage, with pointers to its child nodes. The tree may be updated in a bottom-up manner whenever a write operation occurs.

[65] 3. Caching Mechanism: To improve performance, the memory checker may implement a caching mechanism that keeps frequently accessed nodes of the tree in faster storage tiers or in local memory. This cache may be managed using algorithms such as Least Recently Used (LRU) or Adaptive Replacement Cache (ARC).

[66] 4. Concurrency Control: In a cloud environment with multiple clients accessing the same data, the memory checker may implement concurrency control mechanisms to ensure the integrity of the tree structure during simultaneous read and write operations. This may involve using techniques such as multi-version concurrency control (MVCC) or optimistic concurrency control.

[67] 5. Error Detection and Recovery: When an integrity violation is detected, the memory checker may implement error recovery mechanisms. This may involve rolling back to a previous known good state, or initiating a more comprehensive integrity check of the affected subtree.

[68] 6. Audit Logging: The memory checker may maintain detailed audit logs of all operations, including integrity violations. These logs may be stored in a separate, append-only storage system to ensure their integrity and to facilitate forensic analysis if needed.

[69] Examples of how this system may be applied in various cloud storage scenarios include:

[70] 1. Healthcare Data Management: In a healthcare organization using cloud storage to maintain patient records, the memory checker may be implemented as part of their data management system. When a doctor updates a patient's record, the write operation may be intercepted by the memory checker. The checker may update the corresponding leaf in the hash tree, recalculate the hashes up to the root, and verify the integrity of the entire structure before allowing the write to proceed to the cloud storage. Similarly, when retrieving a patient's record, the read operation may be verified against the hash tree to ensure the data hasn't been tampered with since it was last written.

[71] 2. Financial Transaction Logging: A bank using cloud storage for its transaction logs may implement the disclosed memory checker to ensure that no unauthorized modifications have been made to the logs. Each new transaction may trigger a write operation, which may be verified and incorporated into the hash tree. The memory checker may maintain a sliding window of recent transactions in its cache for quick verification of recent entries. Auditors may then use read operations, verified by the memory checker, to ensure the integrity of the entire transaction history.

[72] 3. Content Delivery Networks: In the context of content delivery networks (CDNs) that use cloud storage, the disclosed memory checker may be used to ensure the integrity of cached content. When a piece of content is updated at the origin, the write operation may update the hash tree. Then, when a CDN edge node retrieves the content, the read operation may be verified against the hash tree, ensuring that the edge node is serving the most up-to-date and unaltered version of the content. The memory checker may implement a hierarchical caching strategy that mirrors the CDN's own caching hierarchy.

[73] 4. Version Control Systems: For cloud-based version control systems, like those used in software development, the disclosed memory checker may provide an additional layer of security. Each commit or merge operation may be treated as a write, updating the hash tree. When developers clone or pull from the repository, the read operations may be verified against

the hash tree, ensuring that the code they're working with hasn't been tampered with. The memory checker may implement a branching strategy that aligns with the version control system's branching model, allowing efficient verification of different branches and versions.

[74] 5. Machine Learning Datasets: In the realm of cloud-based machine learning, where large datasets are often stored and processed in the cloud, the disclosed memory checker may be used to ensure the integrity of both the training data and the trained models. Write operations occurring during the data preprocessing and model training phases may update the hash tree, while read operations during inference or model deployment may be verified against it. This may help prevent attacks that attempt to poison the training data or alter the trained model. The memory checker may implement specialized caching strategies optimized for the access patterns typical in machine learning workflows, such as sequential reads of large datasets during training.

[75] 6. IoT Data Collection: For cloud-based IoT platforms that collect and store data from numerous devices, the disclosed memory checker may be crucial in ensuring the integrity of the collected data. Each data point sent by an IoT device may be treated as a write operation, updating the hash tree. When the data is later analyzed or used to make decisions, the read operations may be verified against the hash tree, ensuring that the data accurately represents what was collected from the devices. The memory checker may implement a time-based partitioning strategy to efficiently handle the high volume of writes typical in IoT scenarios.

[76] 7. Email Services: In the context of cloud-based email services, the disclosed memory checker may be used to provide additional assurance of email integrity. Each new email may be treated as a write operation, updating the hash tree. When a user accesses their inbox, the read operations may be verified against the hash tree, ensuring that no emails have been tampered with or deleted without the user's knowledge. The memory checker may implement a user-based partitioning strategy to allow efficient verification of individual user mailboxes.

[77] 8. Document Collaboration Platforms: For cloud-based document collaboration platforms, the disclosed memory checker may be used to maintain the integrity of shared documents. Each edit to a document may be a write operation, updating the hash tree. When collaborators access the document, the read operations may be verified against the hash tree, ensuring that they're seeing the true current state of the document and that no unauthorized changes have been made. The memory checker may implement a versioning strategy that allows efficient verification of document history and supports features like reverting to previous versions.

[78] 9. Cloud Gaming: In the realm of cloud gaming, where game states are often stored in the cloud, the disclosed memory checker may be used to prevent cheating. Each update to the game

state may be a write operation, updating the hash tree. When the game state is loaded, the read operations may be verified against the hash tree, ensuring that players haven't manipulated their saved games. The memory checker may implement a player-based partitioning strategy to allow efficient verification of individual player states.

[79] 10. Backup Services: For cloud-based backup services, the disclosed memory checker may provide an additional layer of assurance that backups haven't been tampered with. Each backup operation may be a write, updating the hash tree. When a restore operation is performed, the read operations may be verified against the hash tree, ensuring that the restored data is exactly what was originally backed up. The memory checker may implement a time-based and client-based partitioning strategy to allow efficient verification of backups from different time points and different clients.

[80] In all these scenarios, the memory checker may provide an additional layer of security and integrity checking on top of whatever measures the cloud storage provider already has in place. It may allow the client to maintain control over the integrity of their data, even when that data is stored on infrastructure they don't directly control. The use of a hash tree structure may allow for efficient verification of large amounts of data, making it practical even for applications dealing with terabytes or petabytes of information.

[81] Moreover, the memory checker's ability to operate with a public local state, as mentioned in the claims, means that it may function effectively even in scenarios where the hash values might be exposed. This may be particularly useful in distributed or decentralized cloud storage systems, where the hash tree might need to be replicated across multiple nodes for redundancy and performance reasons.

[82] The deterministic, non-adaptive, and online nature of the memory checker, as described herein, may make it particularly suitable for cloud environments where predictability and consistency are important. Cloud services often need to meet strict service level agreements (SLAs), and a memory checker that behaves in a predictable manner can help ensure that these SLAs are met even with the additional integrity checking in place.

[83] The query complexity of $\Theta(\log n / \log \log n)$ mentioned in the claims suggests that the disclosed memory checker may scale efficiently to very large datasets, making it suitable for big data applications in the cloud. For instance, it may be used to maintain the integrity of large-scale data lakes or data warehouses, where petabytes of data might need to be verified.

[84] Trusted Execution Environments

[85] The systems and methods described herein may be implemented within systems utilizing trusted execution environments (TEEs) to enhance security and integrity verification of

external memory. In this configuration, the memory checker may operate on machine memory that is external to the TEE, while the TEE executes the memory checking module's code to verify the integrity of the external memory. This approach may be applied to a wide range of TEE use cases where ensuring the validity of memory outside the trusted environment is necessary, such as in secure enclaves, cloud computing platforms, or IoT devices with limited secure storage.

[86] In some cases, the TEE may provide a secure and isolated execution environment for the memory checker module. The TEE may ensure that the code and data of the memory checker are protected from unauthorized access or tampering, even if the rest of the system is compromised. The memory checker module may be loaded into the TEE during system initialization or on-demand, depending on the specific implementation and security requirements.

[87] The memory checker module executing within the TEE may interact with the external memory through a secure interface provided by the TEE. This interface may allow the memory checker to read from and write to the external memory while maintaining the isolation and security guarantees of the TEE. In some implementations, the TEE may provide cryptographic primitives and secure key storage that the memory checker can use to perform its operations securely.

[88] When implementing the d-ary hash function-based tree construction described in the claims, the TEE may store the root of the tree and other critical data structures within its secure memory. This may provide an additional layer of protection for the most sensitive parts of the memory checker's state. The rest of the tree structure may be stored in the external memory, with the integrity of each node verified using the hash functions and keys stored within the TEE.

[89] In some cases, the memory checker may perform read and write operations on the external memory as follows:

[90] For read operations, the memory checker within the TEE may receive a request to read data from a specific logical address in the external memory. The memory checker may then traverse the hash tree stored in the external memory, starting from the leaf node corresponding to the requested address and moving up to the root. At each level of the tree, the memory checker may verify the integrity of the node by recomputing its hash and comparing it with the stored hash value. The TEE may provide a secure environment for these computations, ensuring that the verification process itself is not tampered with. If all integrity checks pass,

the memory checker may return the requested data to the caller. If any check fails, the memory checker may report an integrity violation.

[91] For write operations, the memory checker within the TEE may receive a request to write data to a specific logical address in the external memory. The memory checker may first update the corresponding leaf node in the hash tree with the new data. Then, it may recompute the hash values for all nodes along the path from the leaf to the root, updating each node in the external memory. The TEE may ensure that these computations and updates are performed securely, without interference from potential attackers. After updating the entire path, the memory checker may update the root hash stored within the TEE's secure memory.

[92] The use of a TEE in conjunction with the memory checker may provide several advantages. First, it may offer strong isolation for the memory checker's code and critical data, protecting them from attacks that may compromise the main operating system or other applications. Second, the TEE may provide secure key storage and cryptographic operations, which may be essential for the hash functions used in the memory checker. Third, the TEE may offer attestation capabilities, allowing remote parties to verify that the memory checker is running in a genuine TEE and has not been tampered with.

[93] In some implementations, the memory checker may leverage the TEE's capabilities to enhance its performance and security. For example, the TEE may provide hardware-accelerated cryptographic operations, which may significantly speed up the hash computations required for the tree structure. The TEE may also offer secure timers and monotonic counters, which the memory checker may use to implement additional security features such as protection against replay attacks.

[94] The memory checker operating within a TEE may be particularly useful in cloud computing scenarios. For instance, in a cloud-based database service, the database engine may run within a TEE, while the actual data is stored in external memory. The memory checker may ensure the integrity of the database contents, protecting against attacks that attempt to modify the data directly in memory. When a query is executed, the memory checker may verify the integrity of the data read from the external memory before it is processed by the database engine within the TEE.

[95] In the context of secure enclaves, such as those provided by Intel SGX, the memory checker may be used to extend the security guarantees of the enclave to a larger amount of memory than can be directly protected by the enclave. The enclave may contain the core logic of an application along with the memory checker, while the bulk of the application's data may be stored in unprotected memory. The memory checker may ensure that any data read from the

unprotected memory has not been tampered with, effectively extending the trust boundary of the enclave.

[96] For IoT devices with limited secure storage, the memory checker operating within a TEE may provide a way to securely use larger amounts of external memory. The device may have a small TEE that runs the memory checker, while the majority of the device's data and code may be stored in external flash memory. The memory checker may ensure the integrity of this external memory, allowing the device to securely boot and operate even if an attacker has physical access to the external memory chip.

[97] In some cases, the memory checker within a TEE may be used to implement secure multi-party computation protocols. Each party may run their part of the computation within a TEE, with the memory checker ensuring the integrity of any data exchanged between parties or stored temporarily in external memory. This may allow for secure collaboration even when the parties do not fully trust each other or the infrastructure on which the computation is running.

[98] The memory checker operating within a TEE may also be applied to secure firmware updates for embedded systems. The TEE may contain the core update logic and the memory checker, while the new firmware image may be stored in external memory. Before applying the update, the memory checker may verify the integrity of the entire firmware image, ensuring that it has not been tampered with during transmission or storage.

[99] In the context of digital rights management (DRM) systems, the memory checker within a TEE may be used to protect media content. The DRM logic may run within the TEE, while encrypted content may be stored in external memory. When playing the content, the memory checker may verify the integrity of the encrypted data before it is decrypted and processed within the TEE, preventing attacks that attempt to modify the content to bypass copy protection.

[100] For blockchain nodes, the memory checker operating within a TEE may provide additional security for the blockchain state. The core consensus and validation logic may run within the TEE, while the blockchain data may be stored in external memory. The memory checker may ensure the integrity of the blockchain data, protecting against attacks that attempt to modify the state directly in memory.

[101] In machine learning applications, the memory checker within a TEE may be used to ensure the integrity of model parameters and training data. The core training or inference logic may run within the TEE, while the large datasets and model parameters may be stored in external memory. The memory checker may verify the integrity of this data as it is read into

the TEE for processing, protecting against attacks that attempt to poison the training data or manipulate the model parameters.

[102] The memory checker operating within a TEE may also be applied to secure logging systems. The core logging logic may run within the TEE, while the log data may be stored in external memory. The memory checker may ensure the integrity and append-only nature of the logs, providing strong guarantees about the authenticity and completeness of the logged information.

[103] In virtual machine (VM) or container environments, the memory checker within a TEE may be used to ensure the integrity of VM or container images. The hypervisor or container runtime may leverage a TEE to run the memory checker, which may verify the integrity of image data stored in external memory before it is loaded and executed.

[104] For secure key management systems, the memory checker operating within a TEE may provide an additional layer of protection for key material stored outside the TEE. The core key management logic may run within the TEE, while encrypted key material may be stored in external memory. The memory checker may ensure the integrity of this encrypted data, protecting against attacks that attempt to substitute or modify the stored keys.

[105] In automotive systems, the memory checker within a TEE may be used to ensure the integrity of critical software components and data. For example, in an autonomous driving system, the core decision-making algorithms may run within a TEE, while large maps and sensor data may be stored in external memory. The memory checker may verify the integrity of this data as it is read into the TEE for processing, ensuring that the autonomous driving decisions are based on untampered information.

[106] The memory checker operating within a TEE may also be applied to secure voting systems. The core vote counting and tabulation logic may run within the TEE, while the individual votes may be stored in external memory. The memory checker may ensure the integrity of the stored votes, providing strong guarantees about the accuracy of the election results.

[107] In all these applications, the memory checker operating within a TEE may provide a powerful tool for extending the security guarantees of the TEE to larger amounts of external memory. By leveraging the isolation and security features of the TEE, the memory checker may offer strong integrity protections even in scenarios where the bulk of the data must be stored outside the trusted environment. This approach may enable the development of secure systems that can handle large amounts of data while still providing robust security guarantees.

[108] Throughout this disclosure, various terms and phrases are used to describe features of the disclosed technology. It is to be understood that these terms and phrases may encompass a variety of meanings and definitions, as is common in the field of technology and patent law. The definitions of these terms may vary depending on the context in which they are used, the specific embodiment being described, or the interpretation of the technology by those skilled in the art.

[109] In various embodiments, certain variable names, symbols, or labels may be used in the claims to represent various elements, components, or steps of the described methods, systems, and apparatuses. These variable names, symbols, or labels are provided for convenience and clarity in describing the claimed subject matter. However, it should be understood that the use of such variable names, symbols, or labels in the claims does not necessarily limit these elements, components, or steps to being the same specific entities described in the specification or in other parts of the disclosure. The variable names, symbols, or labels used in the claims should be interpreted broadly and may encompass various implementations, variations, or equivalents of the described elements, components, or steps, unless explicitly stated otherwise or clearly limited by the context of the claim. As such, the scope of the claims is not confined to the specific examples or embodiments described in the specification, but rather extends to the full breadth of the inventive concepts disclosed herein.

[110] For instance, terms such as "computing device," "processor," "memory," and "network" may refer to a wide range of devices, components, systems, and configurations known in the art, and their specific definitions may differ based on the implementation or design of the system. Similarly, phrases like "securely storing," "computing a vector," and "generating a message" may involve various methods, techniques, and processes that achieve the same or similar outcomes but may be executed in different manners.

[111] It is also to be understood that the use of terms in the singular or plural form is not intended to limit the scope of the claims. For example, the mention of "a computing device" does not preclude the presence of multiple computing devices within a system. Likewise, references to "a network" may include various interconnected networks or a single network comprising multiple segments or layers.

[112] Furthermore, the use of the term "may" in relation to an action or feature indicates that the action or feature is possible, but not necessarily mandatory. This term is used to describe optional or alternative aspects of the disclosed technology that provide flexibility in how the technology may be implemented or utilized.

[113] The definitions provided herein are intended to serve as examples and are not exhaustive. Those skilled in the art may ascribe different meanings to these terms based on the context, the specific technology being described, or the advancements in the field. Therefore, the definitions of the terms and phrases used in this disclosure and the claims are to be interpreted broadly and in a manner consistent with the understanding of those skilled in the relevant art.

[114] The use of the word "a" or "an" when used in conjunction with the claims herein is to be interpreted as including one or more than one of the element it introduces. Similarly, the use of the term "or" is intended to be inclusive, such that the phrase "A or B" is intended to include A, B, or both A and B, unless explicitly stated otherwise.

[115] Reference throughout the specification to "one embodiment," "another embodiment," "an embodiment," and so forth, means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the present disclosure, and may not necessarily be present in all embodiments. Furthermore, the particular features, structures, or characteristics may be combined in any suitable manner in one or more embodiments without limitation.

[116] The use of the terms "first," "second," and the like does not imply any order or sequence, but are used to distinguish one element from another, and the terms "top," "bottom," "front," "back," "leading," "trailing," and the like are used for descriptive purposes and are not necessarily to be construed as limiting.

[117] As used herein, the term "processor" refers to any computing entity capable of executing instructions to perform a specific set of operations, whether implemented in hardware, firmware, software, or any combination thereof. This definition includes, but is not limited to, the following types of processors: Central Processing Unit (CPU), Graphics Processing Unit (GPU), Digital Signal Processor (DSP), Field-Programmable Gate Array (FPGA), Application-Specific Integrated Circuit (ASIC), microcontroller, System on Chip (SoC), Neural Processing Unit (NPU), quantum processor, cloud-based and distributed processors, multi-core and parallel processors, and virtual processors. The term "processor" also encompasses the associated memory hierarchies, including primary memory (such as RAM), secondary storage (such as hard drives and SSDs), and cache memory, which work in conjunction with the processor to store and retrieve data necessary for executing instructions. In this patent application, any reference to a "processor" should be interpreted broadly to include any type of processing unit capable of performing the described functions, regardless of its specific implementation, architecture, or physical form.

[118] As used herein, the term "messages" may refer to any form of data or information that can be processed, transmitted, or stored in a digital format. Messages may include, but are not limited to, arbitrary-length plaintext messages, pre-hashed messages, concatenated messages, binary data, network protocol messages, database records, and time-stamped messages. Messages may be composed of characters, symbols, or binary data and may represent various forms of content such as text, numbers, multimedia, executable code, or any other data that can be digitally encoded. Messages may be used as input for cryptographic functions, such as keyed hash functions, where they are transformed into a fixed-size hash value influenced by a secret cryptographic key. The term "messages" encompasses a wide range of data types and structures, from simple text strings to complex structured data, and may include metadata, headers, footers, or other information that facilitates the processing, transmission, or interpretation of the content. Messages may be generated by users, systems, or processes and may be intended for various purposes, including communication, authentication, verification, logging, or any other function that involves the use of digital data.

[119] The term "database" should be construed to mean a blockchain, distributed ledger technology, key-value store, document-oriented database, graph database, time-series database, in-memory database, columnar database, object-oriented database, hierarchical database, network database, or any other structured data storage system capable of storing and retrieving information. This may include traditional relational database management systems (RDBMS), NoSQL databases, NewSQL databases, or hybrid database systems that combine multiple database paradigms. The database may be centralized, distributed, or decentralized, and may employ various data models, indexing strategies, and query languages to organize and access the stored information. It may also incorporate features such as ACID (Atomicity, Consistency, Isolation, Durability) compliance, eventual consistency, sharding, replication, or partitioning to ensure data integrity, availability, and scalability. The database may be hosted on-premises, in the cloud, or in a hybrid environment, and may support various access methods including direct queries, API calls, or event-driven architectures.

[120] The terms "connected," "coupled," or any variant thereof, mean any direct or indirect connection or coupling between two or more elements, and may encompass the presence of one or more intermediate elements between the two elements that are connected or coupled to each other.

[121] The description of the embodiments of the present disclosure is intended to be illustrative, and not to limit the scope of the claims. Many alternatives, modifications, and variations will be apparent to those skilled in the art. A number of implementations have been

described. Nevertheless, it will be understood that various modifications may be made without departing from the spirit and scope of the disclosure. Accordingly, other implementations are within the scope of the following claims.

CLAIMS

1. A method for checking a memory to detect faults in memory operations, the method comprising:

referencing a connected physical memory having physical word size and a logical word size of polylog(n), wherein n represents the logical size of the memory;

at a data store positioned at a memory checker module as a proxy between a requester and the physical memory, generating a d-ary hash function-based tree construction, wherein d = polylog(n);

mapping the tree construction to a logical memory address space for the physical memory such that each leaf of the tree stores data corresponding to a logical word in the memory;

copying the logical words in the leaves to their parent nodes in a second level of the tree;

at each second level node of the tree, calculating a keyed hash function of the logical words concatenated and storing a key for the hash function and the result of the hash function; and

at each node at a third and higher levels of the tree, storing the results of the hash functions from all of its child nodes, and calculating a keyed hash function of the stored results from all of its child nodes and and storing a key for the hash function and the result of the hash function.

2. The method of claim 1, further comprising performing a read operation by reading the value value of a leaf in the tree associated with index i and reading all nodes from that leaf to the root and verifying that:

(a) the data value of the parent is consistent with the hash of the child;
(b) the hash of the values of the parent are correct; and
(c) the root node is identical to the root node in the memory checker's local memory.

3. The method of claim 1, further comprising performing a read operation through the memory checker by:

receiving an index of a logical memory location $i \in [n]$ from the requester;

generating a corresponding read request to the remote physical memory;

in the tree, reading the d leaves corresponding to the d parts of logical index i, starting at node $u = v$ and going to $u = \varepsilon$;

at each u, the memory checker computes $H_{ku}(x_u)$, wherein H_k is a universal one-way hash function with key k and x is a concatenation of the data from the children nodes stored in the memory of that node;

checking that it is consistent with h_u, and checking that x_u contains the right value $h_{u||j}$ for children j it has read;

checking whether the value of h_ε it has read from the memory is consistent with its local state, which has an uncorrupted copy of h_ε; and

if all checks pass, the memory checker returns $data_i$ to the requester, otherwise, the memory checker determines a failure state or returns $\perp$.

4. The method of claim 1, wherein upon a write operation with value v for index i, writing the data field v at the leaf in the tree associated with index i, and updating all nodes from that leaf to the root.

5. The method of claim 1, further comprising a write operation through the memory checker by:

receiving an index of a logical memory location $i \in [n]$;

writing to the d leaves corresponding to the d parts of logical index i so that $x_{v||j} = 0^{2d\lambda}$ and $h_{v||j} = data_{v||j}$ for all children $j \in [d]$;

performing reads and writes up the path from v to the root ε to recompute hashes, starting at node u = v and going to $u = \varepsilon$, whereby at each u, the memory checker updates x_u with its new entry (or entries, for the case where u = v), samples fresh $k_u \sim \{0, 1\}^\lambda$, and writes $h_u = (k_u, H_k(x_u))$;

checking whether the value of h_ε it has read from the public database is consistent with its local state, which has an uncorrupted copy of h_ε; and

if all checks pass, the memory checker updates its local state to the new value of h_ε, otherwise, the memory checker determines a failure state or returns $\perp$.

6. The method of claim 1, further comprising setting the logical word size to be the physical word size by breaking up the logical word into d chunks in the bottom layer.

7. The method of claim 1, wherein the memory checker has a public local state and security is maintained even if the hashes are available to an adversary.

8. The method of claim 1, wherein the memory checker is deterministic, non-

adaptive, and operates online.

9. The method of claim 8, wherein deterministic and non-adaptive means that if there exist fixed sequences $W_1, \cdots, W_n \in [m]^{qW}$ and $R_1, \cdots, R_n \in [m]^{qr}$ such that each logical write or read to index $i \in [n]$ performs physical queries at the physical locations exactly according to the sequence W_i (or R_i, respectively), regardless of the local state, contents on the database, or random coins.

10. The method of claim 1, wherein upon a logical instruction, the memory checker has query complexity $q = \Theta(\log n / \log \log n)$.

11. The method of claim 1, wherein the memory checker is configured to support logical words of size $w_\ell = \Theta(w)$, resulting in a bandwidth of $\Theta(\log n / \log \log n)$.

12. The method of claim 1, wherein the memory checker is part of the memory, does not need to protect its local state, and the local state is accessible to an adversary.

13. A system for checking a memory to detect faults in memory operations, the system comprising:

a client device;

a memory system; and

a proxy device that stores instructions for:

referencing a connected physical memory having physical word size and a logical word size of polylog(n), wherein n represents the logical size of the memory;

at a data store positioned at a memory checker module as a proxy between a requester and the physical memory, generating a d-ary hash function-based tree construction, wherein d = polylog(n);

mapping the tree construction to a logical memory address space for the physical memory such that each leaf of the tree stores data corresponding to a logical word in the memory;

copying the logical words in the leaves to their parent nodes in a second level of the tree;

at each second level node of the tree, calculating a keyed hash function of the

logical words concatenated and storing a key for the hash function and the result of the hash function; and

at each node at a third and higher levels of the tree, storing the results of the hash functions from all of its child nodes, and calculating a keyed hash function of the stored results from all of its child nodes and storing a key for the hash function and the result of the hash function.

14. The system of claim 13, further comprising performing a read operation by reading the value of a leaf in the tree associated with index i and reading all nodes from that leaf to the root and verifying that:

(a) the data value of the parent is consistent with the hash of the child;

(b) the hash of the values of the parent are correct; and

(c) the root node is identical to the root node in the memory checker's local memory.

15. The system of claim 13, further comprising performing a read operation through the memory checker by:

receiving an index of a logical memory location $i \in [n]$ from the requester;

generating a corresponding read request to the remote physical memory;

in the tree, reading the d leaves corresponding to the d parts of logical index i, starting at node $u = v$ and going to $u = \varepsilon$;

at each u, the memory checker computes $H_{ku}(x_u)$, wherein H_k is a universal one-way hash function with key k and x is a concatenation of the data from the children nodes stored in the memory of that node;

checking that it is consistent with h_u, and checking that x_u contains the right value $h_{u||j}$ for children j it has read;

checking whether the value of h_ε it has read from the memory is consistent with its local state, which has an uncorrupted copy of h_ε; and

if all checks pass, the memory checker returns $data_i$ to the requester, otherwise, the memory checker determines a failure state or returns $\perp$.

16. The system of claim 13, wherein upon a write operation with value v for index i, writing the data field v at the leaf in the tree associated with index i, and updating all nodes from that leaf to the root.

17. The system of claim 13, further comprising a write operation through the memory checker by:

receiving an index of a logical memory location $i \in [n]$;

writing to the d leaves corresponding to the d parts of logical index i so that $x_{v||j} = 0^{2d\lambda}$ and $h_{v||j} = data_{v||j}$ for all children $j \in [d]$;

performing reads and writes up the path from v to the root ε to recompute hashes, starting at node $u = v$ and going to $u = \varepsilon$, whereby at each u, the memory checker updates x_u with its new entry (or entries, for the case where $u = v$), samples fresh $k_u \sim \{0, 1\}^{\lambda}$, and writes $h_u = (k_u, H_k(x_u))$;

checking whether the value of h_ε it has read from the public database is consistent with its local state, which has an uncorrupted copy of h_ε; and

if all checks pass, the memory checker updates its local state to the new value of h_ε, otherwise, the memory checker determines a failure state or returns $\perp$.

18. The system of claim 13, further comprising setting the logical word size to be the physical word size by breaking up the logical word into d chunks in the bottom layer.

19. The system of claim 13, wherein the memory checker has a public local state and security is maintained even if the hashes are available to an adversary.

20. The system of claim 13, wherein the memory checker is deterministic, non-adaptive, and operates online.

21. The system of claim 20, wherein deterministic and non-adaptive means that if there exist fixed sequences $W_1, \cdots, W_n \in [m]^{qW}$ and $R_1, \cdots, R_n \in [m]^{qr}$ such that each logical write or read to index $i \in [n]$ performs physical queries at the physical locations exactly according to the sequence W_i (or R_i, respectively), regardless of the local state, contents on the database, or random coins.

22. The system of claim 13, wherein upon a logical instruction, the memory checker has query complexity $q = \Theta(\log n / \log \log n)$.

23. The system of claim 13, wherein the memory checker is configured to support logical words of size $w_\ell = \Theta(w)$, resulting in a bandwidth of $\Theta(\log n / \log \log n)$.

24. The system of claim 13, wherein the memory checker is part of the memory, does not need to protect its local state, and the local state is accessible to an adversary.

100

d= a data value associated with the memory value at the mapped logical index

h= cryptographic hash function

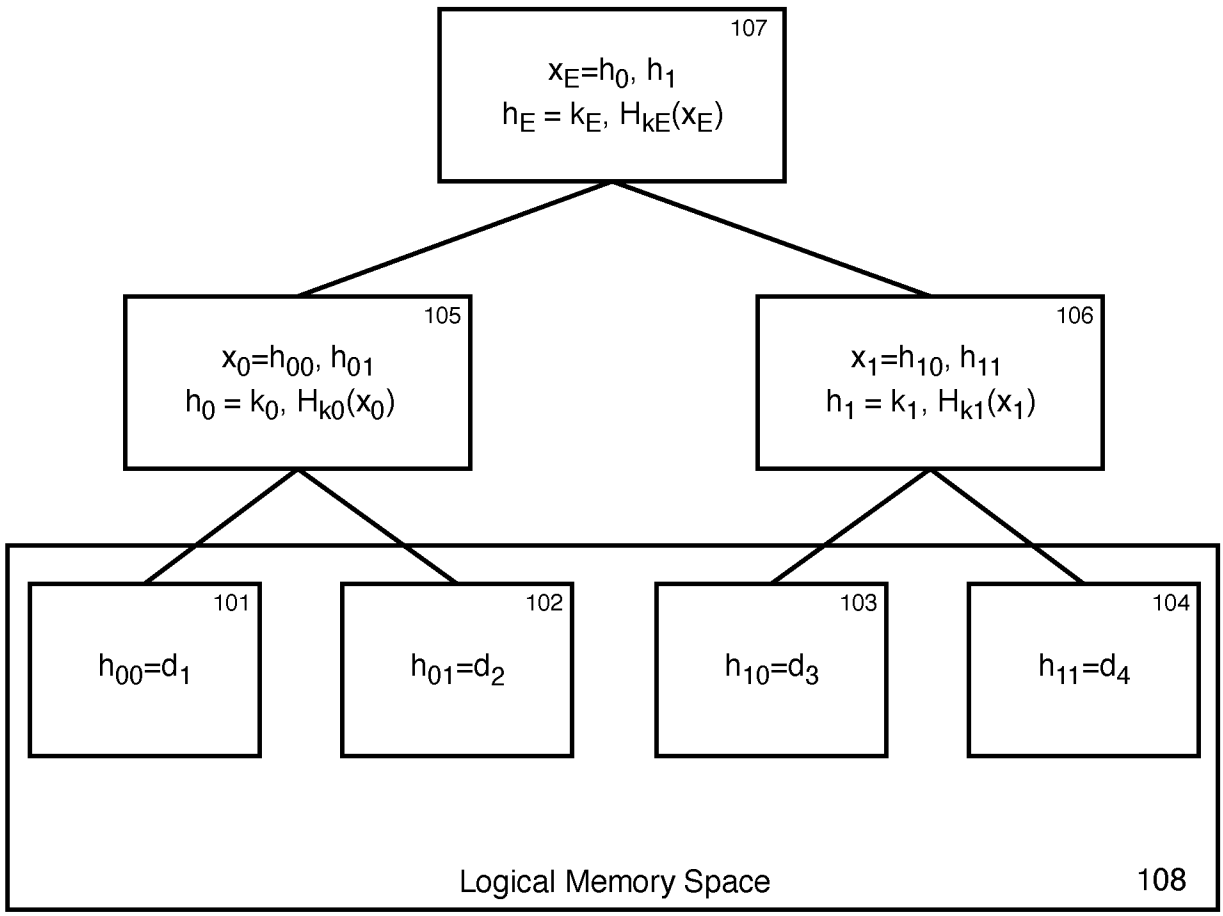

Fig. 1

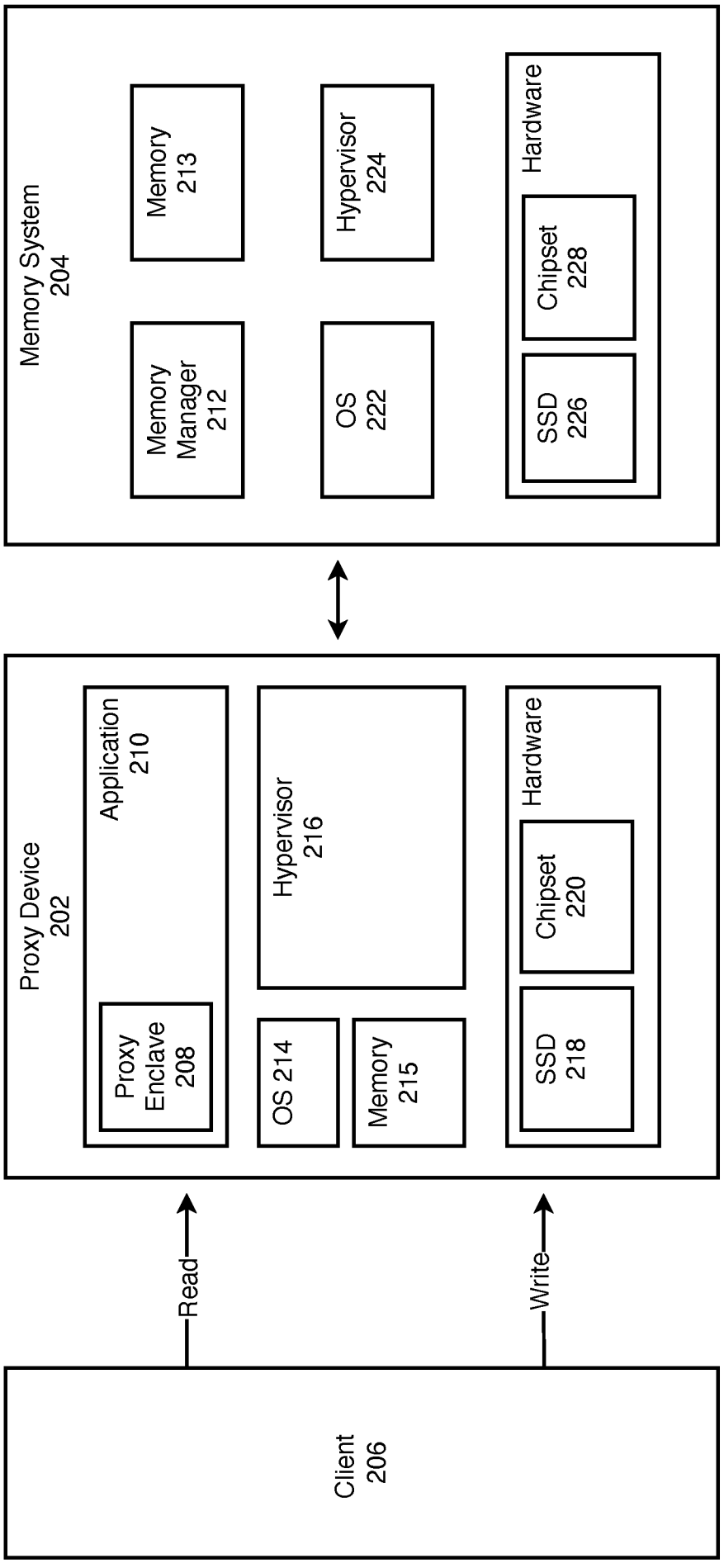
200
Memory System
204
Memory
213
Hypervisor
224
Hardware
Memory
Manager
212
OS
222
Chipset
228
SSD
226
Proxy Device
202
Application
210
Proxy
Enclave
208
Hypervisor
216
Hardware
Chipset
220
OS 214
Memory
215
SSD
218
Read
Write
Client
206

Fig. 2

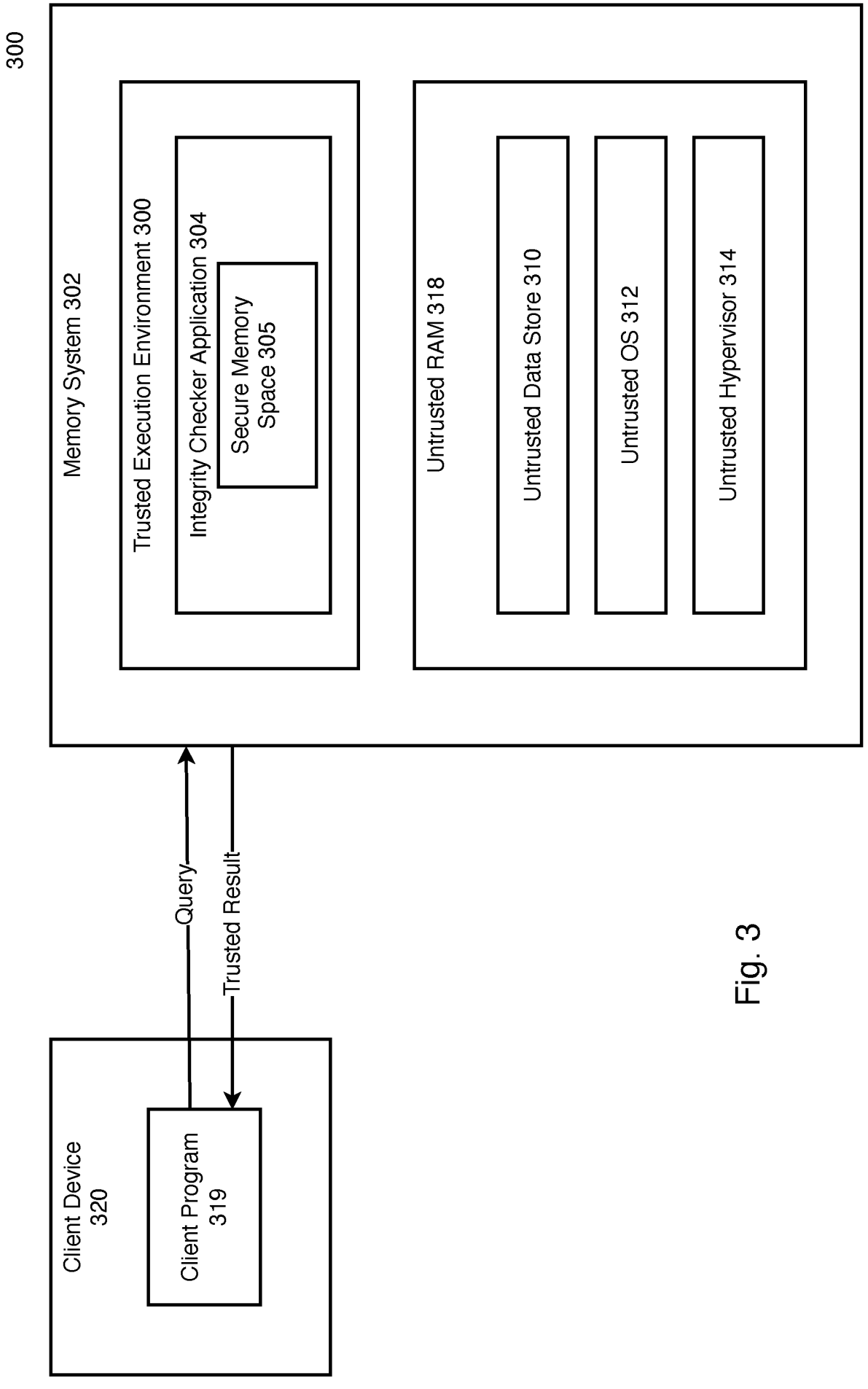
300
Memory System 302
Trusted Execution Environment 300
Integrity Checker Application 304
Secure Memory Space 305
Untrusted RAM 318
Untrusted Data Store 310
Untrusted OS 312
Untrusted Hypervisor 314
Query
Trusted Result
Client Device 320
Client Program 319

Fig. 3

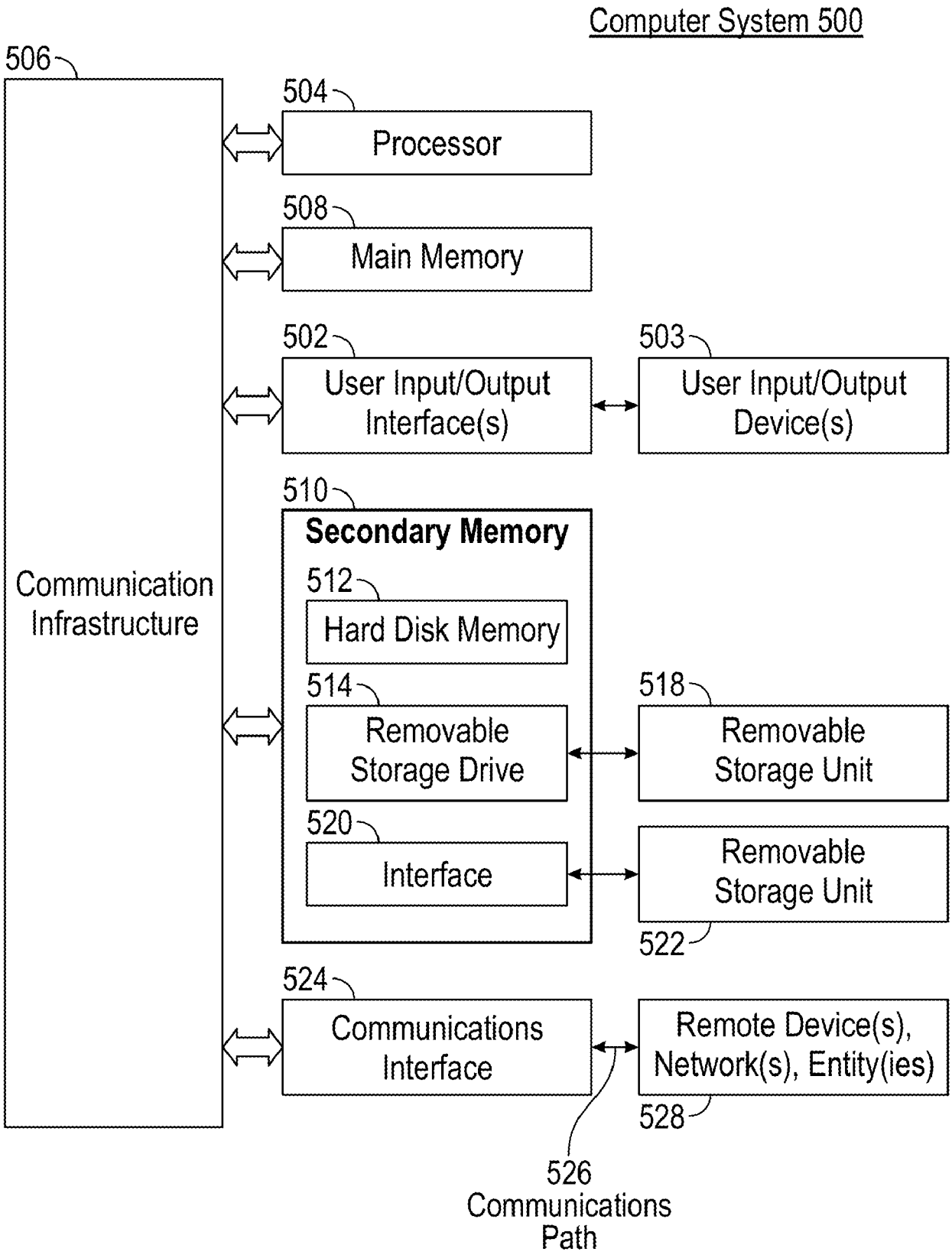
Computer System 500
506
Communication Infrastructure
504
Processor
508
Main Memory
502
User Input/Output Interface(s)
503
User Input/Output Device(s)
510
Secondary Memory
512
Hard Disk Memory
514
Removable Storage Drive
518
Removable Storage Unit
520
Interface
Removable Storage Unit
522
524
Communications Interface
Remote Device(s), Network(s), Entity(ies)
528
526
Communications Path

Fig. 4

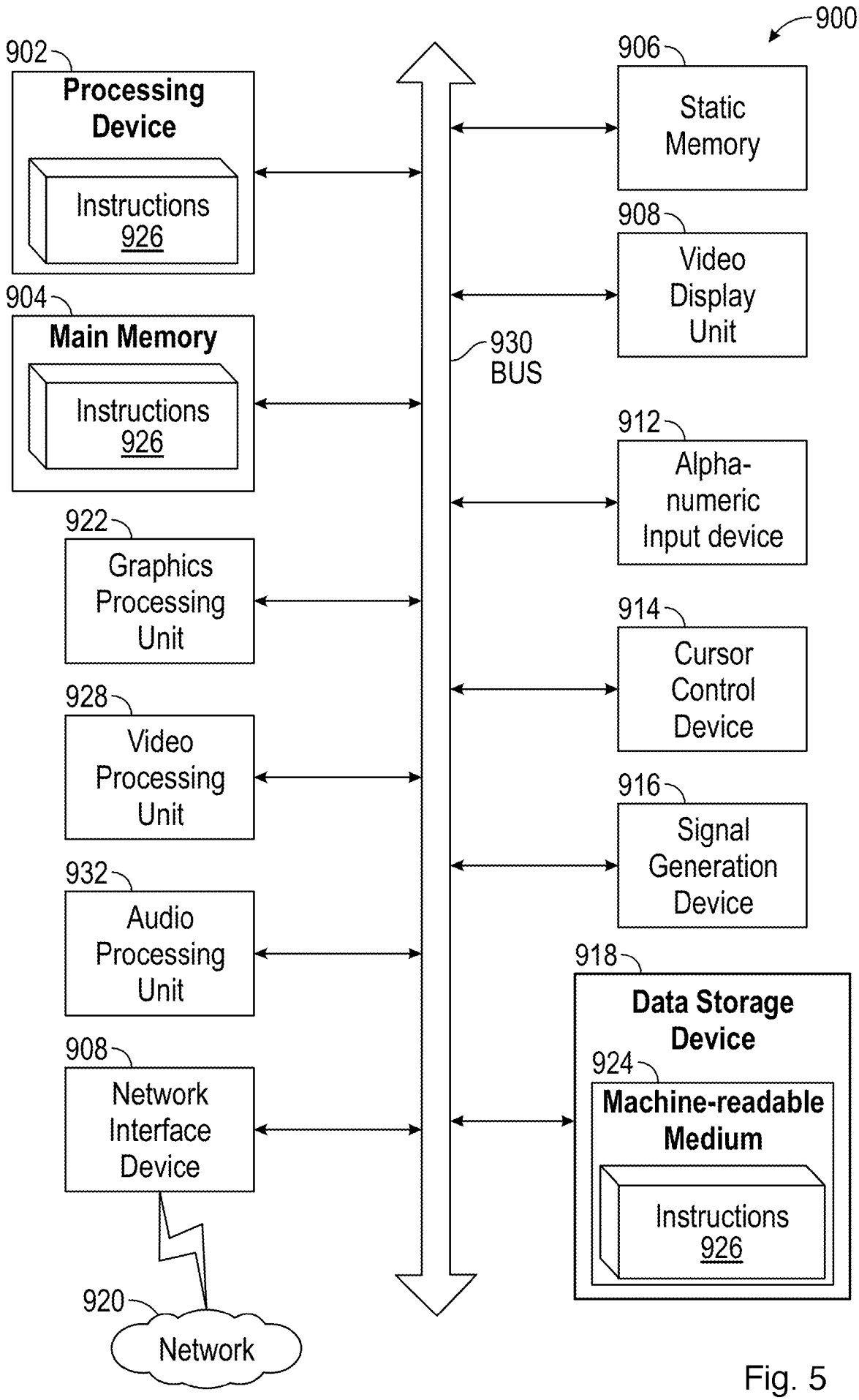
900
902
Processing Device
Instructions 926
904
Main Memory
Instructions 926
922
Graphics Processing Unit
928
Video Processing Unit
932
Audio Processing Unit
908
Network Interface Device
920
Network
930
BUS
906
Static Memory
908
Video Display Unit
912
Alpha-numeric Input device
914
Cursor Control Device
916
Signal Generation Device
918
Data Storage Device
924
Machine-readable Medium
Instructions 926

Fig. 5

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2024/061714

A. CLASSIFICATION OF SUBJECT MATTER

IPC: ***G11C 29/04*** (2025.01); ***G06F 3/06*** (2025.01); ***H03M 13/03*** (2025.01); ***G06F 21/64*** (2025.01); ***G06F 21/62*** (2025.01); ***H03M 13/05*** (2025.01); *G06F 16/13* (2025.01)

CPC: ***G11C 29/04; G06F 3/0619; H03M 13/6572; G06F 21/64; G06F 21/62; H03M 13/05; H03M 13/03;*** *G06F 16/13*

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

See Search History Document

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

See Search History Document

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

See Search History Document

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 10,515,049 B1 (Tahoe Research Ltd) 24 December 2019 (24.12.2019) entire document, especially Abstract and Fig. 2; col 5, ln 57-67	1-24
A	US 2020/0057664 A1 (Intel Corporation) 20 February 2020 (20.02.2020) entire document, especially Abstract and para [0073], [0451], [0454]	1-24
A	US 2013/0091405 A1 (Von Thun et al.) 11 April 2013 (11.04.2013) entire document	1-24
A	US 2006/0069966 A1 (Liu) 30 March 2006 (30.03.2006) entire document	1-24
A	US 2014/0047296 A1 (Motabar et al.) 13 February 2014 (13.02.2014) entire document	1-24
P,A	US 2024/0089113 A1 (NXP USA Inc) 14 March 2024 (14.03.2024) entire document	1-24

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

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

Date of the actual completion of the international search	Date of mailing of the international search report
23 February 2025 (23.02.2025)	**03 March 2025 (03.03.2025)**

Name and mailing address of the ISA/US	Authorized officer
COMMISSIONER FOR PATENTS **MAIL STOP PCT, ATTN: ISA/US** **P.O. Box 1450** **Alexandria, VA 22313-1450** **UNITED STATES OF AMERICA**	**KARI RODRIQUEZ**
Facsimile No. **571-273-8300**	Telephone No. **PCT Help Desk: 571-272-4300**

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
P,A	US 2024/0267213 A1 (NTT Research Inc) 08 August 2024 (08.08.2024) entire document	1-24
A	Edward Suh et al. "Hardware Mechanisms for Memory Integrity Checking." In: researchgate.net, December 2002, [online] [retrieved on 22 February 2025] Retrieved from the Internet [URL: https://www.researchgate.net/publication/2565834_Hardware_Mechanisms_for_Memory_Integrity_Checking] entire document	1-24
A	Mathialagan et al. "MacORAMa: Optimal Oblivious RAM with Integrity." In: Cryptology {ePrint} Archive, Paper 2023/083, [Shortened-pages 1-49] February 25, 2023, [online] [retrieved on 22 February 2025] Retrieved from the Internet [URL: https://eprint.iacr.org/2023/083] entire document	1-24