Architectural Support for High Speed Protection of Memory Integrity and Confidentiality in Multiprocessor Systems

Weidong Shi[†] Hsien-Hsin S. Lee[†‡] Mrinmoy Ghosh[‡] Chenghuai Lu[†]

College of Computing[†]
School of Electrical and Computer Engineering[‡]
Georgia Institute of Technology
Atlanta, GA 30332-0280
{shiw,lulu}@cc.gatech.edu[†] {leehs,mrinmoy}@ece.gatech.edu[‡]

ABSTRACT

Recently there is a growing effort in both the architecture and the security community to create a hardware solution for authenticating system memory. As shown in the previous work, hardware-based memory authentication will become a vital component for creating future trusted computing environments and digital rights protection. Almost all these prior work have focused on authenticating memory exclusively owned by a single processing element. However, in today's computing platforms, memory is often shared by multiple processing elements that support a shared system memory with a snooping cache coherence protocol. Authenticating shared memory is a new challenge to memory protection. In this paper, we present a secure and fast architecture for authenticating shared memory. In terms of incorporating memory authentication into the processor pipeline, we propose a new scheme called *Authentication Speculative Execution*. Unlike the prior approaches, our scheme does not compromise security for performance. The novel ASE scheme is not only secure as it is combined with a one-time-pad (OTP) based memory encryption but also efficient to tolerate authentication latency by executing unauthenticated instructions speculatively. Results using modified RSIM running SPLASH2 benchmark show only 5% overhead in performance on dual and quad processor platforms. Furthermore, ASE shows 80% better performance on average over conservative non-speculative execution based authentication schemes. The scheme is of practical use for both multiprocessor systems and uni-processor systems where memory is shared by one main processor and other co-processors on the system bus.

1. INTRODUCTION

Recently, there has been intensive research in the area of trusted computing facilitated by hardware based authentication and decryption/encryption [15, 8, 12]. The effort of putting security features to hardware platforms and micro-architecture holds great promises to address many security issues that have haunted computing industry for decades including digital rights protection, anti-reverse engineering, software confidentiality, secure distributed computing, and virus protection to name just a few. Among many such architectures, hardware based memory authentication is often an essential and absolutely necessary component. Software based memory authentication methods, no matter how carefully designed, always have the vulnerability of executing altered codes or accessing altered data driven by malicious purposes such as bypassing security/copyright checking. Virus can spread also due in part to the fact that a modified code image can be loaded and executed without being detected. Software and OS-based authentication on either code or data before the program execution can help to reduce the risk but cannot eliminate the vulnerability completely. For example, a simple software solution is to allow the OS to schedule a process read from the disk only after the code image of the process was authenticated. This solution enhances security but cannot prevent attackers from tampering the codes on the fly after loaded.

There has been a number of papers published recently in the architecture community addressing the problem of providing a secure computing environment where memory is authenticated with hardware support [4, 11, 12]. The challenge of memory authentication in the architecture design is to enable a highly efficient memory authentication at low cost without compromising security. However, most solutions proposed thus far assume that the memory is exclusively "owned" by one processing element (often the main processor). Inside the processor, memory is authenticated on per process basis with a memory authentication signature computed for each process's virtual space. This signature is generally the root of a authentication tree. Such strong process isolation (on both the inter- and intra- processor levels) prevents the signature from being shared by multiple processors. When inter-processor memory sharing is inevitable, a copy from one processor's authenticated domain to another's is required. Such copying operations often require re-authentication of the shared memory by the destination processor. For multiprocessor (MP) systems, it is not a trivial task to synchronize and maintain authentication signatures for frequently shared data without significantly degrading system performance. Worse yet, it is difficult to achieve integrity protection of memory shared among multiple processors because of potential replay attack on either the shared bus or the shared physical memory. This means that all the existing approaches of hardware based memory authentication are inapplicable to the scenarios of MP memory protection.

In this paper, we present a fast and low overhead solution to authenticate the shared memory of an MP system. Through securing every component along the path from a computing device to another computing device or the commonly shared memory, a chain of authentication is constructed. The chained authentication scheme is capable of preventing most software based and hardware based attacks on the memory system and the shared data path among processors. Such a secure memory environment facilitates high speed secure data sharing for both MP systems and some uni-processor systems that demand high performance secure data communication between the core processor and other peripheral agents on the system bus. The scheme also optionally provides high performance protection on information confidentiality for shared data.

In addition, the paper addresses for the first time, the issue of how to tie the result of memory authentication securely into the processor pipeline design. We investigated and compared three alternative designs regarding how results of authentication is used — *authentication in-order execution* (AIOE), *authentication speculative execution* (ASE), and *lazy authentication execution* (LAE). Under *authentication in-order execution*, when either instruction or data fetch incurs a cache miss and causes information fetched from the memory, the processor pipeline stalls until the newly fetched instruction or data is fully authenticated[1]. For *authentication speculative execution* (ASE), the processor pipeline resumes execution immediately after the fetched information is decrypted before the authentication completes. In other words, instructions awaiting either un-authenticated data or results computed based on unauthenticated data can be speculatively issued and executed but not allowed to retire until both the instruction itself and its dependent data are authenticated. Furthermore, bus cycles are not granted to memory accesses that are not considered secure or *authentication safe*. A memory access is not considered *authentication safe* if, 1) it tries to write un-authenticated results back to memory; 2) it reads/writes to a memory address generated from un-authenticated data; 3) it fetches instructions from memory based on control flow determined by un-authenticated data. Such memory accesses are called *authentication unsafe* accesses. An *authentication unsafe* access becomes *authentication safe* only after all the data it depends on is authenticated. *Lazy authentication* (LAE) is a weak authentication scheme that only authenticates fetched data and instructions in groups over a relatively large time span in the magnitude of tens of thousands of cycles.

The main contributions of the paper are:

- A unified fast and secure means for authenticating memory for both uni-processor and multiprocessor systems. The approach relies on division of labor and distributes security workloads to both secure processors and a secure memory controller (*North Bridge*) thus requires a light weight secure processor design. The approach detects not only software based tampering of data but also physical attacks including replay attacks in the shared memory.
- An innovative secure multiprocessor bus protocol for authenticating coherent bus transactions.
- A fast memory authentication approach based on *OTP (one-time-pad)* and *authentication speculative execution* to tolerate the latency of memory authentication for both processor-to-processor and memory-to-processor accesses.
- A secure authentication mechanism that is not only fast, but also *authentication safe*, and supports precise interrupts for security exceptions. Despite being fast, it does not trade security for performance as is the case with *lazy authentication* schemes like LHash [11].

[1]Note that (1) pipeline has to stall for decryption if the fetched information is encrypted; (2) in an out-of-order machine, other instructions having no dependency on the missing instruction or data can be issued and executed

The rest of the paper is organized as follows. Section 2 presents the challenges and related work of memory authentication. Section 2 addresses the security risks and performance implications for shared memory authentication. It also presents assumptions of our targeted platforms. Section 3 presents our solution to authenticating shared memory for multiprocessors. Performance were evaluated and analyzed in Section 4. Section 5 concludes the paper.

2. CHALLENGES IN SHARED MEMORY PROTECTION

In this section, we discuss many basic issues associated with shared memory protection at high level. It presents the basic platform architecture our solution is targeted for. It also answers the questions such as the rationale of why shared memory needs protection and shows the types of attacks our solution is aimed to prevent and detect.

Threat of Physical Attacks: History shows that when it comes to break security measures in a commodity computing platform, attackers often are not only well motivated but also very knowledgeable and possess the required skills to build customized hardware to break the security protection in any imaginable way [7]. In order to crack out the protected secret, attackers may dump all the bus transactions on the system/peripheral buses, construct customized spoofing device or hardware, exploit the coherence snooping bus protocol by injecting artificial bus signals, replay bus transactions, spoof, alter or replay RAM contents on the fly through hardware, and so on. Even though software based protections or light weight hardware based protection such as TCG [6] provide some protection using minimal silicon resources, it is almost impossible for them to survive from these physical attacks.

Efficiency and Security: Almost all the recently proposed security computing platforms with hardware-based memory protections assume that everything in such a system is insecure except the main processor with built-in security support [15, 8, 12]. Under such assumptions, many proposed protection solutions often have all the hardware security features including memory authentication implemented in the main processor. Gassend et al. [4], the CHTree authentication scheme constructs a m-ary hash tree for protecting the memory integrity of an application process under a uni-processor environment. As shown, CHTree slows down the execution by around 20% with a 2MB L2 and incurs 33% memory space overhead. The LHash scheme in [11] was proposed to improve the authentication speed. This scheme logs memory operations and performs integrity checks only when a large number of memory operations is accumulated. According to our definition, the LHash is a type of *lazy authentication* technique. However there are potential security risks associated with *lazy authentication*, especially when used together with OTP for memory protection.

Multiprocessor Domain: Furthermore, all the existing solutions are designed for uni-processor memory protection and assume that the security boundary lies at the interface between a secure processor and its system bus. Such a centralized view fits with a uni-processor platform but does not

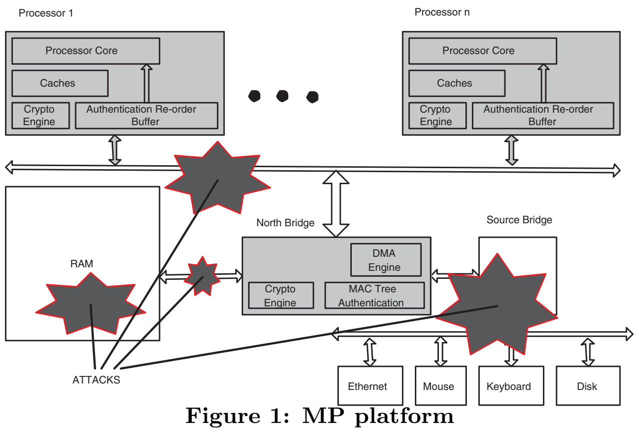


Figure 1: MP platform

apply to an MP system. A simple way to extend the existing solutions to the MP scenario is to have a separate copy of the memory for each processing element[2] (PE) and have the secure OS to copy the data from one trusted domain to another using protected message passing mechanism whenever needed. This will however significantly increase the delay of inter-processor communication and substantially undermines performance of MP applications.

Replay Attacks: One of the major challenges of designing a secure MP platform is how to prevent and detect replay attacks. There are two types of replay attacks: 1) replay logged bus transactions, including both cache-to-cache and memory-to-cache bus transactions; 2) replay information stored in the physical RAM. Under the system where the memory is exclusively owned by a single PE (i.e. core processor), replay attack can be prevented using a hardware implementation of a Merkle hash tree or a MAC tree inside the PE. But such solutions do not work when the shared memory can be updated by multiple PEs. If each PE maintains its own authentication tree, either Merkle hash tree or MAC tree, to verify and synchronize the root signatures of these authentication trees across multiple processors could be cumbersome and lead to significant performance impact on frequent inter-processor communication.

Distributing Secrets: Another challenge of designing a tamper-resistant MP system is how to distribute and share secret information among processors and devices. Such shared information may include symmetric cryptographic keys, shared sequence number, etc. Distributing and sharing secrets is a unique problem to MP shared memory protection.

Architectural Support for Security Primitives: Yet another challenge in MP protection is with respect to interoperability. Protections provided by a MP tamper resistant system can be best viewed as security primitives analogous to other architecture supports designed for synchronization and consistency. These security primitives themselves cannot guarantee security requirements from being violated. But they can be used by properly developed secure operating systems and secure applications to construct a software environment in which memory integrity and secrecy can be guaranteed.

In summary, shared memory authentication is a unique problem. Existing uni-processor techniques based on Merkle / MAC tree cannot resolve this issue without tremendous modification. To enable a fast, secure, and unified solution for authenticating shared memory, we propose a distributed scheme where both the main processor(s) and the chipset contribute to create a secure environment for trusted software execution. Our shared memory authentication scheme is universal for both MP shared memory systems and uni-processor systems with a memory shared by a core processor and other peripherals/agents. Figure 1 illustrates the platform architectures to which our solution is applicable.

Instead of having each PE to use its own Merkle hash/MAC tree, our solution enables shared memory protection through a centralized MAC tree based authentication implemented in the memory system with a secure MP coherent bus protocol. The shared memory protection is achieved by securing the data path from each PE to the shared system memory. With the MAC memory authentication tree embedded in the North Bridge (i.e. the memory controller), the data path between the memory controller and the physical RAM is secured. Moreover, the secure MP bus protocol provides a trusted and authenticated environment for both cache-to-cache and memory-to-cache bus transactions. Untrusted devices cannot commit any bus transaction to the protected shared memory and any attempt to replay past authenticated bus transactions can also be detected. Since both the data path from each PE to the system memory and the data paths among PEs are protected, secure and authenticated sharing of the system memory is provided. Different from the previous approaches that put all the hardware resources for memory authentication into one single secure processor, our solution provides a trusted environment for MP shared memory through securing the platform.

3. SECURITY MODEL FOR SHARED MEMORY

In this section, we present the detailed security model and architectural support for shared memory authentication. We only focus on symmetric multiprocessor (SMP) systems where a coherent snoopy bus and a large physical RAM are shared by a number of processors. It is straightforward to extend the solution to situations where each processor maintains a local memory. The proposed MP shared memory protection scheme can be used to ensure both integrity and confidentiality for MP shared memory. We first present a MP platform oriented security model. Then, architectural supports for the security model and a latency-tolerating technique called *authentication speculative execution* are proposed.

3.1 Multiprocessor security model

Figure 1 shows target architecture of MP systems. A split-transaction, cache coherent system bus connects each processor to the shared memory. The security model holds no specific assumptions about the coherent bus protocol, neither is it tied to any particular MP system. Examples of applicable MP systems include SGI's POWERpath-2, Alpha's MP protocol, and Intel Xeon based MP systems. To simplify the discussion, the scheme to be presented uses a split-transaction, SGI Powerpath-2 like cache-coherency protocol [3]. Applying the scheme to other MP system should be straightforward with little changes. The shaded blocks in the system are trusted and protected components. The dark stars denote points of potential attacks.

[2]A processing element (PE) can be a core processor, a co-processor, or a peripheral in a uniprocessor system or a processor in an MP system. We generalize such a device as a PE.

The proposed security model provides a system level specification for constructing a trusted environment for MP software execution. It addresses four issues — management of protected MP processes, distribution and sharing of secrets, protection on integrity and secrecy, and software distribution. The security model assumes the existence of a secure OS kernel [4]. A secure OS kernel is a set of trusted core OS services. These services are executed in a trusted domain. The secure kernel is verified by secure BIOS during system boot [1].

3.1.1 Process control

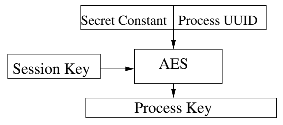
One basic and essential protection on a multi-tasking system is *process or task isolation.* Different process should not be allowed to access other processes' protected domain. To achieve *process isolation*, two conditions must be satisfied. Firstly, each process must be uniquely identified. Secondly, unique per-process cryptographic information must be used for protecting integrity and confidentiality of each process's memory. The traditional *process id* (pid) is not a good choice because the likelihood of reusing a pid is very high. Here we define a unique 128-bit number, *process uuid*, a universal unique identifier to uniquely identify a process. The *process uuid* is obtained from a random number generator. The process uuid itself is not considered as secret and can be securely shared among multiple processors during initialization by the secure kernel. Process uuid is treated as process context and is protected against tampering during context switch.

Secret padding and keys used for authenticating or encrypting memory information of each process is derived from each process' uuid. A new privileged instruction which is used only by the secure kernel is introduced to set up process uuid, called *set_uuid*. Execution of the instruction includes several steps. One step involves that the processor assigns the *process uuid* to an internal uuid register and computes a process key as described in Figure 2(a). The session key shown in the figure is created at boot time and described in detail in Section 3.1.3. Secret Constant is an initial value, same for all Processing Elements (PE's) and known only to each PE and the secure memory controller. Other steps of the instruction *set_uuid* are described later in section Section 3.2.2. relate to how the *process uuid* is shared by other devices attached to the shared bus.

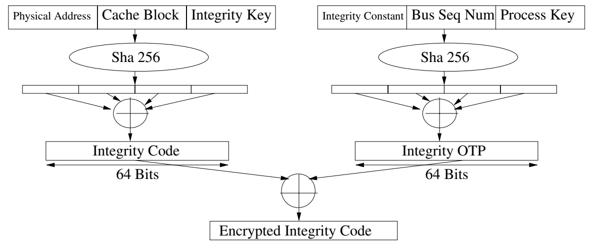
3.1.2 Integrity and confidentiality protection

Integrity code is also referred to as message authentication code (MAC), a well established technique for guaranteeing data integrity by verifying whether a piece of received or retrieved data was tampered during transmission or storage. For the purpose of memory integrity protection and authentication, before a PE stores a chunk of data to the insecure memory, it will compute an integrity code using a MAC generation algorithm and keep it alongside the data. In the case of digital rights protection or secure software execution, the data itself may or may not be encrypted depending on the security requirement. Later, when the same PE or any other PE attempts to access the data, the integrity of the data will be verified by re-generating the integrity code of the retrieved data using the same MAC algorithm and comparing it against the stored one. Any tampering to the stored data will be detected when a mismatch occurs.

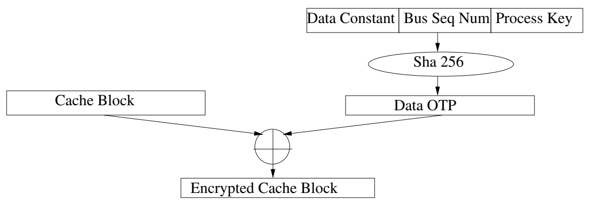
In our shared memory authentication scheme, each PE (including the North Bridge) is responsible for computing the integrity code for data to be stored to the memory or re-



(a) Process Key Generation



(b) Integrity Code Generation (*Integrity Key* and *Bus Sequence Number* generation are described in Figure 4))

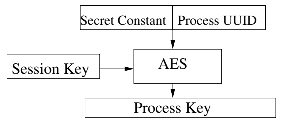


(c) Data Encryption

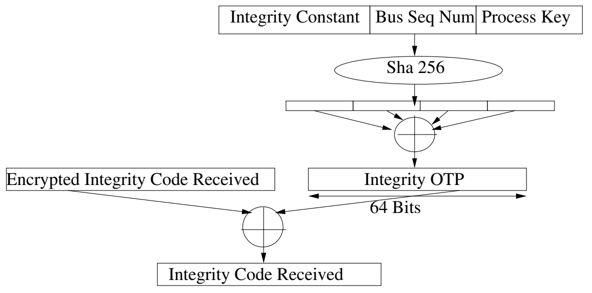
Figure 2: Security Operations on Each Cache Block Evicted from Processor or Transmitted as Coherence Miss

quested by other PEs. When the data is shared by multiple PEs, the *key* along with other necessary information for generating/verifying the integrity code must be shared among all the involved PEs. The integrity code is encrypted using OTP when it is transmitted through the shared system bus. A *one-time-pad* (OTP) is uniquely computed using a confidential shared bus sequence number which is tracked by all the PEs on the system bus. The sequence number is incremented by all the devices attached to the system bus after each bus transaction. For protecting confidentiality of either information stored to the memory and coherence response to other processor's request, the data is also encrypted using another OTP also computed based on the shared bus sequence number.

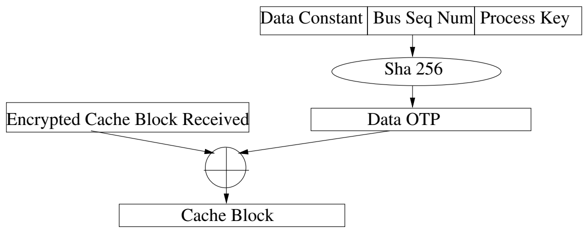
Figure 2 shows needed operations for a cache block that is either dirty-evicted from the protected domain or requested by other processors. The operations illustrated are conducted by a PE on each protected cache block to be written to the system bus. They involve, generation of the process key using the process uuid (Figure 2(a)), generation of the encrypted integrity code using the generated process key, bus sequence number, the data to be written and the integrity key (Figure 2(b)) and finally encryption of the data using the bus sequence number and the process key (Figure 2(c)). SHA256 [9] and AES128 [2] are hash and encryption standards. Integrity key is a 256-bit secret shared by the units attached to the shared system bus. Session key is an AES key uniquely initialized every time after the system is started. Distribution of the shared secrets such as the sequence number and the session key is addressed in Section 3.1.3. For two blocks next to each other in the figures implies they are concatenated. The $\oplus$ stands for



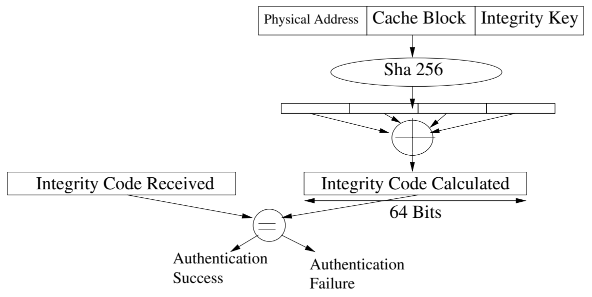
(a) Process Key Generation



(b) Reading Received Integrity Code with Process Key



(c) Decrypting Data with Process Key



(d) Authentication of Data

Figure 3: Security Operations on Each Cache Block Received

XOR. Both the integrity key and the sequence number are hidden from software access and can not be accessed externally. Similarly, computed data such as *integrity_code* and the *process_key* are also hidden from software access. Only encrypted integrity codes and encrypted cache blocks are observable as they are transmitted over the shared bus. All the PEs share the same constant values which are burnt inside each PE. Most of the shared secrets, such as the session key, the integrity key, and the sequence number are not fixed constants. They are uniquely assigned each time after the system is booted using approaches described in Section 3.1.3.

Integrity verification and decryption of received cache block (coherence reply and memory read) are shown in Figure 3. Reading involves, computation of the process key (Figure 3(a)), decrypting the received encrypted integrity code and encrypted data using the process key (Figure 3(b) and Figure 3(c)) and finally recomputing the integrity code from the decrypted data to compare with the received integrity code for authentication (Figure 3(d)).

The security model shown in Figure 2 and Figure 3 minimizes the performance critical path between encryption and decryption. The encryption_OTP is pre-computed. In the best scenario, the interval of transferring an encrypted cache block consists of only time of a XOR operation on the sender, transmission delay, and another XOR operation on the receiver. Authentication requires much more time because integrity code has to be computed before transmission and verified after the cache block is received.

3.1.3 Secrets distribution and sharing

How to securely distribute and share secret such as the keys, the padding, the sequence number, and etc., is a major challenge for designing a secure distributed system. Obviously, the secret can not be broadcast as plaintext over the system bus. Integrity of the shared secret also has to be maintained so that it can not be forged. Furthermore, the shared secret such as the session keys, the sequence number must be different each time the machine is rebooted to prevent replay attack. Here we present a novel and efficient way for distributing secret information across multiple processors connected by a shared bus.

Similar to a regular symmetric multiprocessor system, one processor has to be designated as the boot processor to bring up the system. This processor will execute its secure BIOS and boot into a secure OS. The uniqueness of our solution is that the shared secrets themselves are not transmitted, instead they are computed by each involved processor in a secure way based on information that can be openly shared.

During boot time, the bootstrap processor broadcasts the range of physical memory to be protected to all the processing units and the memory controller. It could be a portion of the entire physical address space or all the physical RAM space. After that, it starts key generation. During key generation, each unit attached to the shared bus is granted bus cycles in turn to broadcast a random 64-bit number. Then each unit concatenates all the random numbers it collects from the bus including the one it broadcasts and computes a hash value using some hash function. The hash result is truncated into a 128-bit AES session key. Then, all the shared secrets including the shared bus sequence number, the process key, the integrity key are all synchronously computed based on the session key as shown in Figure 4.

Both the *secret_hash_key* and the *secret_constant* are constants permanently burnt into the processor chip, and the memory controller during manufacturing. They are secrets stored in the chip inaccessible by either software running inside or any device from outside. After a device creates all the required keys and numbers, it will enter a synchronization barrier. After all the devices on the shared bus complete key generation and enter the synchronization barrier, regular bus and memory transaction are resumed with the appropriate protection specified in this paper.

The session key and bus sequence number are not tied to a particular process and are not considered part of a process context. But a process is free to specify whether segments/pages of its virtual memory should be mapped to protected physical memory. Note that a different session key is generated each time after the system is freshly rebooted.

3.1.4 Software distribution and platform key

The cryptographic protection proposed is "self-contained" because the session key, the root of all the keys, the sequence

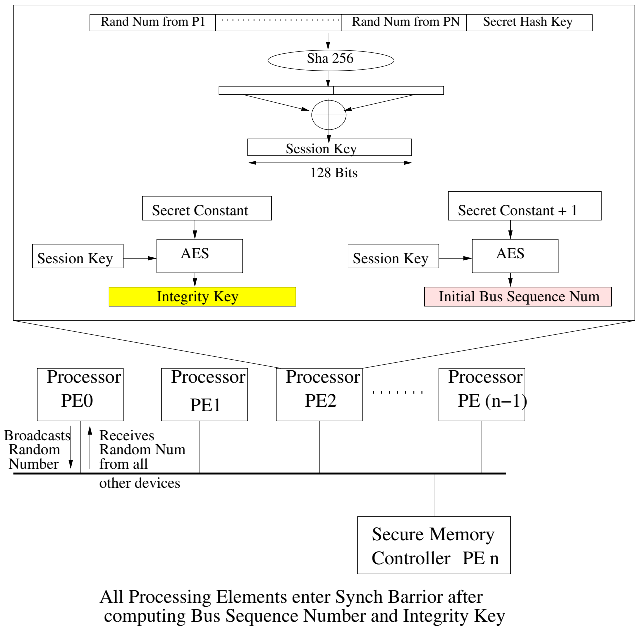


Figure 4: Processor Initialization and Distribution of Shared Secrets

number, etc, are not constants and will be modified each time after reboot. Software vendors are not able to generate the integrity code or OTPs used in the protection because they do not have the session key. To execute software either encrypted or authenticated by vendors in the mode with the proposed protection, conversion from the vendor protected domain to the MP platform protected domain is required. This is achieved through a platform key. A platform key is a pair of public-private keys with private key permanently burnt into the MP's chipset. Vendors encrypt the symmetric cryptographic key used to encrypt/authenticate a software with the public platform keys. When the software is copied to memory from its disk image, it will be decrypted, authenticated using the keys set by the software vendors, and then re-encrypted using the methods described in Figure 2 and Figure 3. As we will describe next, this conversion does not necessarily require processor involvement and can be performed in high speed with security enabled DMA engines.

3.2 Architectural Support of MP Security

In this section, we present a detailed architecture model for implementing the MP security model described in the previous section. The implementation must be efficient and high performance while not compromising security. There are three security enabled platform architecture components, the shared system bus, the memory controller, and the secure processors. Extra security related functionality has to be added to these components to support the proposed MP security model. Furthermore, new techniques must be invented to minimize the performance impact of security verification. For MP systems and benchmarks, authentication latency is an even more significant performance influencing factor because integrity code of coherent response of cache-to-cache communication has to be computed, transmitted, re-computed, and verified. To tolerate the latency of integrity checking, we propose two new techniques. First, we propose a split transaction bus model for data and its integrity code. Second, *authentication speculative execution* is proposed to further hide the latency of authentication in the secure processor.

3.2.1 *Secure symmetric coherent bus protocol*

The purpose of the secure multiprocessor bus protocol is to prevent spoof and replay attack on the shared coherent MP bus. It plays an essential role for providing a chain of authentications for both cache-to-cache and memory-to-cache accesses. Although the principle of how we secure the MP bus is in fact not tied to any MP coherence bus protocol, but for the sake of discussion, we restrict the design to a four-state coherence protocol similar to SGI POWERpath-2 with cache-to-cache transfer triggering a write-back to memory. Each cache has four states; invalid, exclusive, dirty exclusive, and shared. Each transition between states is either initiated by the processor or by a coherent transaction. A duplicate set of cache tags [3] is maintained by each processor interface ASIC and bus arbitration is done in distributed manner. Similar to POWERpath-2, every bus transaction requires five clock cycles. A system wide bus controller logic executes the same five-state machine synchronously: arbitration, resolution, address, decode, and acknowledge.

All the devices on the shared multiprocessor bus including the memory controller share the same secret 64-bit bus transaction sequence number described in Figure 2 and Figure 3. Since all the bus transactions are visible to all the units attached to the snoopy MP bus, it is straightforward for a unit on the bus to update and keep track of the sequence number. After a bus transaction completes, every unit on the bus *increments* its copy of the sequence number internally. The sequence number is initialized during system boot as shown in Figure 4. The number is kept as secret by all the involved devices and never transmitted in either plaintext or ciphertext on the bus.

One unique performance feature of our secure bus is the split transaction of data and its integrity code. We may infer from Figure 2 and Figure 3, that integrity code generation and verification is the critical path of the MP security model. To minimize the impact of authentication on performance, our secure bus model allows data block and its integrity code transmitted separately. For coherent response, a cache block can be transmitted first followed by the encrypted integrity code after it is computed. The un-authenticated data will be used by the processor pipeline of the destination processor speculatively. We call this scheme, *authentication speculative execution* (ASE). After the integrity code is finally received and verified, completed instructions using the un-authenticated data can be retired and stalled memory operations using address generated using the un-authenticated data can be issued. Section 3.2.3 details the ASE scheme.

Note that the 64-bit bus sequence is good enough for security protection. This is because for a bus running at speed of hundreds of MHz or a few GHz, it would take hundreds of years if not thousands for a 64-bit sequence number to wrap-around.

3.2.2 *Secure memory system*

This section describes the architecture of the secure memory system consisting of a memory controller with an integrated security engine, and a number of physical RAM chips. We should note that for using integrity code alone is not sufficient for verifying memory integrity because a hacker can

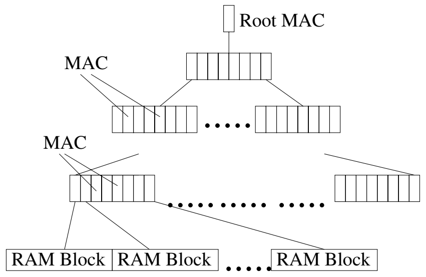
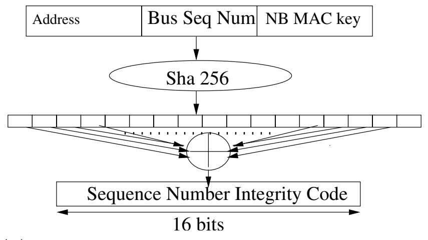
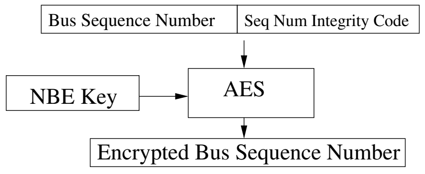


Figure 5: MAC Tree



(a) Sequence Number Integrity Code Generation



(b) Encrypting Bus Sequence Number with Sequence Number Integrity Code

Figure 6: Bus Sequence Number Encryption

replace new data along with its MAC with old stale data and MAC. Such replay attacks cannot be detected without using techniques such as Merkle hash tree or MAC tree. When the memory is not shared by multiple processors, all integrity protection features can be implemented in the main processor [4, 11, 12]. But this solution is inapplicable to MP shared memory systems.

The primary goal of the security engine embedded in the North Bridge memory controller is to detect alternation or replay of data stored in the system memory. It is another critical component in the chain of shared memory authentications. A simple solution is to have either Merkle hash tree or MAC tree implemented in the memory controller. Note that the integrity code itself is not transmitted in plaintext over the MP bus and is unknown to the hackers. In our architecture a MAC tree is employed to provide both security and speed for memory authentication. Organization of the MAC tree is shown in Figure 5. A leaf node represents an individual integrity code and each internal node denotes a MAC of all the children nodes.

The detailed operation of a memory write is as follows. After the integrity of received data is verified, the memory controller will update the MAC tree by substituting the new integrity code (after it is XORed with the integrity OTP) into the tree. Then it will send the data with the encrypted integrity code to the memory. To be able to verify the integrity code later, the memory controller will also store the encrypted bus sequence number to the memory. Each bus sequence number can be encrypted using AES as shown in Figure 6. Both the *NB_MAC_key* and the *NB_E_key* (where NB stands for North Bridge and E stands for encryption) used during encryption are secret information maintained by the North Bridge itself.

To improve performance, the bus sequence numbers for frequent data blocks are cached inside the North Bridge. This can speed up integrity verification for data retrieved from the physical RAM.

Upon receipt of a read request, the memory controller will fetch both the data and the associated encrypted integrity code from the physical RAM. The corresponding encrypted bus sequence number will also be retrieved if it is not cached in the North Bridge. The authentication mechanism will extract the original integrity code using the approach detailed in Figure 3. To verify whether the integrity code and the data is a replay, it is inserted into the MAC tree. Starting from the bottom of the tree, recursively, a new MAC is computed and compared with the cached internal MAC tree node. If a match is found, the integrity code is verified valid. Since it is impractical to cache all the internal nodes of the MAC tree, many internal MAC tree nodes have to be stored in the insecure system memory and brought into the memory controller when needed. To prevent from leaking sensitive information and jeopardizing security, confidentiality of the internal MAC tree node has to be maintained. This is achieved by encrypting the internal MAC tree node using 128-bit AES encryption scheme.

Memory latency is another critical issue for high performance. Our secure memory system is specifically designed for reducing memory latency at the same time without losing security protection. The following techniques provided by the proposed architecture are for memory latency reduction.

- The integrity OTP and data OTP are pre-computed. To speed up the process of verifying retrieved integrity code from memory, the North Bridge prefetches the encrypted bus sequence number. If addresses of future memory accesses can be predicted or speculated, the integrity and data OTPs for these addresses can be pre-computed.
- Both the MAC tree node and the bus sequence number for frequent data blocks are cached to improve memory access speed.

The secure memory controller (North Bridge) is the center of the proposed MP security model. Beside the MAC tree, it also shares secrets with other processors on the shared bus, maintains platform key pairs described in the security model, and transforms protected information from the software vendor's domain to the platform's domain.

The memory controller holds several security oriented registers. Fixed addresses are assigned to these registers and access to these registers must be performed through protected bus transactions. First, there is a *process uuid register* in each processor. During execution of a *set_uuid* instruction by a secure processor, the processor also issues a secure write access to the corresponding uuid register in the North Bridge so that a process key can be derived by the memory controller. Upon receipt of a new uuid value, the memory controller computes its version of the current process key based on Figure 2. Similar to the uuid register, there are North Bridge registers assigned for holding software vendor keys encrypted by the platform's public key. The North Bridge can extract the vendor keys using the private platform key. When the vendor keys are enabled, transactions from peripheral devices such as disks, network devices are first verified or decrypted using the vendor keys and then converted using the process key and the integrity key understandable by the secure processors based on Figure 2. The platform key pair is permanently burnt in the North Bridge.

Note that the secure OS kernel always resides in a separately protected memory space. Access to the secure kernel uses a different process key and *set_uuid* is not needed when application switches to secure kernel mode. The memory range of the secure kernel is also maintained by the North Bridge. The uuid registers and encrypted vendor key registers in the North Bridge reside in the secure kernel memory space. Another important security role served by the North Bridge is the conversion of memory protection when data is transmitted from peripherals such as disk to the physical memory. The conversion mechanism supports both DMA based and processor based memory operations. For DMA, the involved secure processor initializes both the related uuid register, and the software vendor key register first, then starts the DMA engine. The memory controller will automatically verify and convert protections from vendor's domain to the platform's domain for every chunk of data written to the memory. Similarly, when results in the memory are DMAed to the peripherals, they can be optionally converted back to the software vendor's domain. Both operations can be achieved with support of the security DMA engine without increasing workload on the secure processor.

3.2.3 Authentication speculative execution

As aforementioned, there are three alternatives of incorporating results of integrity verification into processor pipeline — AIOE, ASE, and LAE. The pros and cons of each approach are listed in Table 1. To achieve a balance between security and performance, ASE is definitely the best compromise. Detailed discussion of vulnerabilities of LAE is outside the scope of this paper.

Using ASE, data and its integrity code can be transmitted separately. Each data transaction on the MP bus is tagged with a transaction number. The upper bound of the tag is the maximum number of outstanding bus transactions supported. A bus transaction is not considered complete before its integrity code is received. For both inter-processor communication and regular memory fetch, data can be transmitted and processed even before integrity code arrives, hence not stalling the pipeline. For inter-processor communication, integrity code has to be computed by the source processor, transmitted, and verified by the destination processor. For a memory fetch, integrity code has to be verified through the MAC tree. Data transactions with either un-verified integrity or missing integrity code are kept in a structure we call the *Sequential Authentication Buffer* (SAB) which is illustrated in Figure 7. SAB is an on-chip buffer that keeps read transactions with outstanding integrity codes or transactions with unverified integrity codes in the sequence they were triggered in the system. There is an "authenticated_bit" associated with each SAB entry. This bit is set when the integrity of a read transaction is verified. SAB broadcasts the index of an authenticated entry to a bus shared by memory load/store queues and the re-order buffer. Even though the transactions can be authenticated in any order SAB entries have to be broadcasted sequentially which justifies it being called a *sequence* buffer. Both load/store queues and re-order buffer have an extra SAB tag

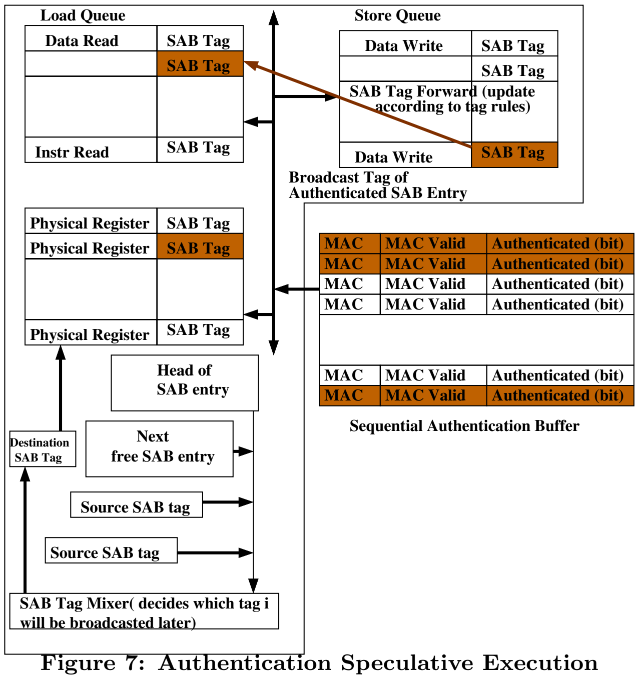


Figure 7: Authentication Speculative Execution

field. If the value stored in the tag field is zero, it means that value held in the queue or the re-order buffer is either authenticated, or produced using authenticated data. If the SAB tag value is a positive number i, it means that data associated with the queue or re-order buffer is not considered authenticated until integrity of the transaction held in entry i of the SAB is verified. When a load/store queue or reorder buffer snoops a broadcast of index i from the SAB, any entry tagged with index i is reset to zero.

An ASE processor allows the use of un-authenticated data for speculative execution. However, the in-flight results will be tagged with its corresponding entry index in the SAB. For example, consider an instruction "load r2, [addr]" which uses data fetched from [addr] that misses the cache. Therefore, data of [addr] will be fetched from memory and the transaction allocates an entry with an index i in the SAB. Then the entry holding r2 value in the physical register file will be tagged with i. The instruction is also inhibited from being retired from the processor pipeline. However, the dependent instructions using r2 as an input operand are allowed to be issued and executed. For example, assume that the next instruction is "add r3, r3, r2", where source r3 contains an unauthenticated input value and tagged with a value k. After the add instruction is completed, the destination r3 will be tagged with either i or k depending on which one will be broadcasted by SAB later. If i would be broadcasted later than k, then r3 will be tagged with i, otherwise, tagged with k. The pseudo-code in Figure 8 shows how to decide which tag would be broadcasted later given two source tags, the SAB tag to be broadcasted next (*SABhead*), and a pointer to the next free SAB entry (*next_free_SAB*). Implementation of the SAB tag updating rules called ARB Tag Mixer is done in hardware.

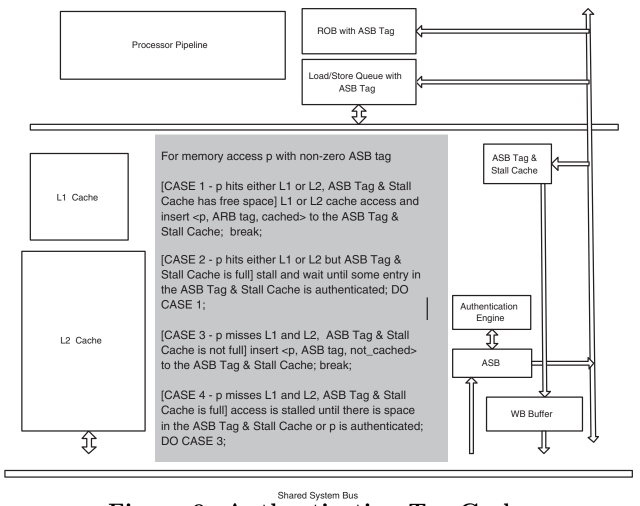
When un-authenticated data is to be stored to the memory hierarchy or data/ instruction needs to be fetched from memory hierarchy using address computed from unauthenticated data, it is allowed to be issued to the memory hier-

Table 1: Pros and Cons of AIOE, ASE, and LAE

Scheme	Pros	Cons
AIOE	secure	slow, support precise interrupt
ASE	secure	support precise interrupt, faster than AIOE, allow split transaction of data and integrity code
LAE	fastest	insecure, worst for protection using one-time-pad, no precise interrupt for authentication failure

```
tag_mixer (SABhead, next_free_SAB, src1_tag, src2_tag){
    if (SABhead < next_free_SAB) {
        return MAX(src1_tag, src2_tag);
    }else {
        if both src1_tag1 and src2_tag >= SABhead
            or both src1_tag1 and src2_tag <= next_free_SAB
            return MAX(src1_tag, src2_tag);
            if (src1_tag1 < =src2_tag) return src1_tag;
            else return src2_tag;
        }
}
```

Figure 8: Compute a new SAB tag



For memory access p with non-zero ASB tag

[CASE 1 - p hits either L1 or L2, ASB Tag & Stall Cache has free space] L1 or L2 cache access and insert <p, ARB tag, cached> to the ASB Tag & Stall Cache; break;

[CASE 2 - p hits either L1 or L2 but ASB Tag & Stall Cache is full] stall and wait until some entry in the ASB Tag & Stall Cache is authenticated; DO CASE 1;

[CASE 3 - p misses L1 and L2, ASB Tag & Stall Cache is not full] insert <p, ASB tag, not_cached> to the ASB Tag & Stall Cache; break;

[CASE 4 - p misses L1 and L2, ASB Tag & Stall Cache is full] access is stalled until there is space in the ASB Tag & Stall Cache or p is authenticated; DO CASE 3;

Figure 9: Authentication Tag Cache

archy. Load/store queue of memory units are also extended with an SAB tag field. When a piece of data is forwarded from the store queue, its SAB tag kept in the store queue is also "forwarded" to update the destination register's SAB tag.

If the access < addr, SAB tag i > has to be issued to the memory and hits either L1 or L2 cache and the authentication tag i can be cached somewhere, execution can be still continued as normal. Next time, the data in addr is fetched into some physical register again, the cached SAB tag i will also be retrieved and used to update the register's SAB tag field. But, if the access is *authentication unsafe* and triggers an L2 miss, it will be stalled for the purpose of security protection. The stalled access together with its authentication tag i will be maintained in a cache, called *Authentication Stall Cache* (ASC). ASC maintains all the L2 miss memory accesses that are *authentication unsafe* and can not be issued to the bus because of unauthenticated SAB tags.

The *Authentication Stall Cache* (ASC) also receives SAB broadcast. If a broadcasted tag is equal to the tag stored in a ASC entry, the corresponding stalled memory access can be removed from the cache and ready to be granted bus cycles for the memory request.

The above description requires that authentication tag be cached alone with the data in the on-chip cache. This can be achieved by adding a field to each cache line. However, since the interval between the time that an un-authenticated data is received or produced and the time when it is authenticated (all its sources are authenticated) is relatively short, it is more efficient to disassociate SAB tag and the data to have a separate small tag cache for storing authenticated tags. Furthermore, the tag cache can be merged with the *Stall Cache* to be a unified SAB Tag & Stall Cache as shown in Figure 9.

Instruction fetch is stalled when its execution or fetch depends on unauthenticated data, such as conditional branches. The instruction is tagged with SAB tag of data sources, and is stalled until the data sources are verified. Under ASE, if instruction fetch hits on-chip instruction caches, it can be optionally speculatively executed but results produced by the instruction will be tagged with SAB tag associated with the instruction. However, if the conditional branch triggers an L2 miss, it will be stalled in the ASC until the data source it depends on is authenticated.

3.3 Security Analysis

This section provides a security analysis for both the MP bus protocol and the secure memory system. The objective of MP shared memory authentication is to prevent unauthorized alternation and replay of coherent response and data stored in the shared memory. There are a number of techniques adversaries can try to exploit our system and we will show that none of them will succeed in breaking the proposed protection mechanism. First, a hacker may try to forge the integrity code. This clearly does not work since each unit can verify the integrity code and our integrity code is generated using a strong cipher. The integrity key is a secret and it is re-generated after the machine is rebooted. Second, a hacker may try to replay both data and the associated integrity code on the MP bus, this will fail because every bus transaction is protected by a sequence number that does not stay the same. Third, a hacker may try to do replay attack on the memory. This would not succeed neither because of the MAC tree protection. Since both the secure MP bus protocol and the secure memory system use OTP, people may suspect that attacker can launch known plain text attacks. This is also impractical because plain text attacks requires that attacker knows the integrity code. In our scheme, integrity code is kept secret. It is not accessible either by software or external devices. Only encrypted integrity code is transmitted and stored in the physical RAM. Moving memory block and its encrypted MAC around does not work neither because the encrypted MAC is generated using address as part of the input. Replay data written by a different process will also fail because the MAC is encrypted by an OTP generated by a process key which is unique to each process. Finally, security privileged instructions such as *setup_uuid* can be only used inside the secure kernel. User program is not allowed to use these instructions thus preventing spoofing of a process uuid.

4. PERFORMANCE ANALYSIS

4.1 Memory overhead

Table 2: Memory Overhead

Structure	Size (bytes)
SAB(32 entries)	32*(8b MAC+1b cntrl)=288
SAB Tag & Stall Cache (64 entries)	(4b addr+1b tag+1b cntrl)*64 =384

Table 3: Applications and input parameters

Application	Parameters
lu	256 by 256 matrix, block 8
radix	512K keys
water	343 molecules
quiksort	32768
mp3d	5000
fft	65536

The proposed memory protection scheme needs additional die space to implement security related tables or caches such as SAB, SAB tag cache, sequence number cache, etc. These tables or caches often contain only small number of entries and in lots of cases can be merged with other related structures. For example, SAB can be merged with the existing table for tracking outstanding bus transactions. Table 2 lists the memory overhead of required on-chip structures.

The sequence numbers associated with each cache line size RAM block and the intermediate nodes of MAC tree are stored in the RAM. The space needed is approximately 1/(m-1) of the RAM size with an m-ary balanced MAC tree. For example, for a 256-bit cache line with a 64-bit sequence number and MAC, the RAM overhead is about 25% of the protected RAM space. Note that the scheme allows only portion of the whole RAM protected. It is up to the system on how the protected physical memory is allocated. The caches implemented in the memory controller are sequence number cache and MAC tree caches for the frequent sequence numbers and MAC tree nodes. They are typically small from 8KB to no more than 32KB.

4.2 Simulation Environment

For characterizing and evaluating our proposed MP system, we use RSIM [10] as our infrastructure to simulate a 4-node MP system. Each node includes a MIPS R10000 like out-of-order processor, L1, and L2 cache. We modified the simulator to support the SGI POWERpath-2 MP coherent bus protocol and a shared main memory. Secure snoopy bus protocol, memory authentication, and authentication speculative execution are all implemented into the

Parameters	Values
CPU	4-issue per cycle
reorder buffer	64 instructions
load/store queue	64 instructions
L1 cache	8-Kbyte, directly mapped
L2 cache	4-way, Unified, 32B line, 256KB
L1 Latency	1 cycle
L2 Lat (256KB)	3 cycles
Memory Latency	X-5-5-5 (cpu cycles) X depends on mem page status
Memory Bus	200 MHz, 8B wide
SHA-256 Latency	180ns
AES Latency	180ns
Bus Sequence # Cache	2Way; 8KB, 32KB; 64B line
MAC tree	2Way; 8KB,32KB; 64B line

Table 4: Processor model parameters

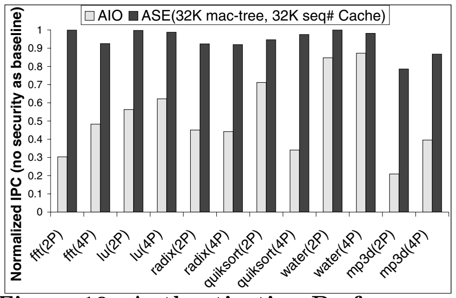


Figure 10: Authentication Performance

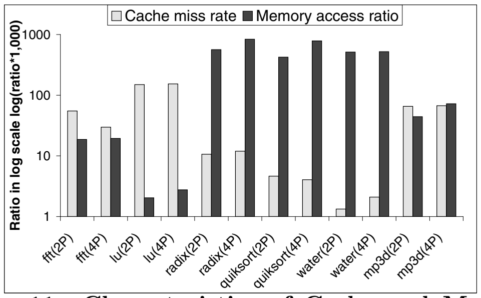


Figure 11: Characteristics of Cache and Memory References

RSIM simulator. To characterize the memory transactions more accurately, we integrated an accurate DRAM model [5] based on the PC SDRAM specification. SPLASH-2 benchmark suite [14] was used. The SPLASH-2 benchmark applications [14] and its input parameters use in this study are listed in Table 3. Table 4 lists the basic processor configuration parameters used throughout the experiments unless otherwise specified. The latencies of Mac tree and sequence number caches were obtained using CACTI [13].

4.3 Performance

4.3.1 Authentication Performance

Figure 10 compares the performance of different authentication execution schemes. It shows IPC normalized to baseline[3] under two scenarios, AIOE, and ASE. We tried two MP settings, 2P and 4P systems because dual and quad processor platforms are the most popular choices for today's commercial workstations. The data indicates that ASE is much faster than AIOE and incur very little performance degradation. The average performance degradation for ASE is less than 5% for both 2P and 4P systems. The performance improvement of ASE over AIOE on average is about 80% for both 2P and 4P systems. Figure 11 shows two important profiling results of the benchmarks, combined L1/L2 cache misses and proportion of memory references with respect to the total number of instructions executed. It will be used later for explaining some applications' characteristics.

Furthermore, we evaluated the effect of SAB tag & stall cache. There are several conditions, an ideal tag cache (always hit), a 32 entry tag cache (8-way, 4 set), a 64 entry tag cache (8-way, 8 set), and no tag cache under a quad processor setting. The results are shown in Figure 12. The IPCs were normalized to those of ideal tag cache. As indicated, tag cache can improve performance for some benchmarks es-

[3]Baseline has zero security protection of any kind.

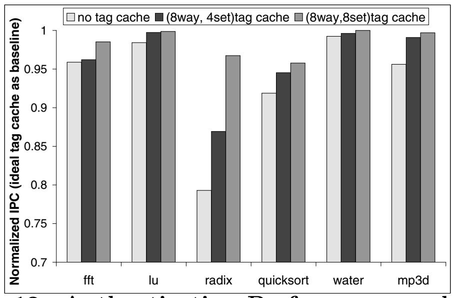


Figure 12: Authentication Performance under Different Tag Cache Setting, 32K MAC tree & 32 K sequence number caches, Processors=4

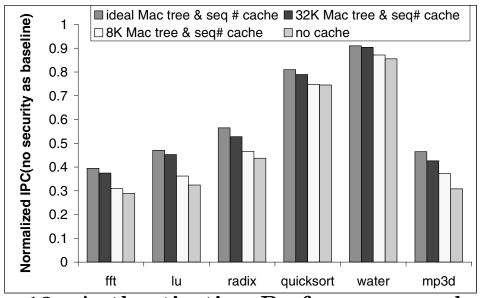


Figure 13: Authentication Performance under Different North Bridge Cache Resources, Processors=4

pecially quicksort, radix, and mp3d. As shown in Figure 11, these three benchmarks are memory intensive. The average improvement is about 5%, and more than 10% for some benchmarks compared to the system with no tag cache.

Another factor of authentication performance is the amount of cache resources in the North Bridge. Results in Figure 13 show the effects of MAC tree and sequence number cache size using IPC normalized to an ideal MAC tree and sequence number cache. On average, a 32KB MAC tree and sequence number caches can deliver an IPC only 3% less than that with an ideal MAC tree and sequence number cache. The performance of 8KB MAC tree and sequence number cache and no cache are 6% and 26% respectively.

4.3.2 Encryption Performance

The protection scheme supports information stored in the RAM optionally encrypted. However, the latency overhead is un-balanced. For cache-to-cache access, the overhead is very small because the data is encrypted and decrypted by two XOR operations given the OTP is pre-computed. For memory-to-cache access, the overhead is bigger because the OTP can not be pre-computed (it can be done under prefetch but we did not evaluate its impact in this paper). However, the memory controller can start the OTP computation as soon as the request is received and the associated sequence number is cached in the sequence number cache. Figure 14 gives results showing categorizations of L2 misses, in which, most of the benchmarks except for a few like "radix" show more cache-to-cache accesses than memory accesses. The behavior of the "radix" benchmark may be explained by the fact that it actually does a radix sort on a huge array of non-negative numbers. The huge number of memory accesses in "radix" may be explained as capacity misses for the large array. We may also observe from the fig-

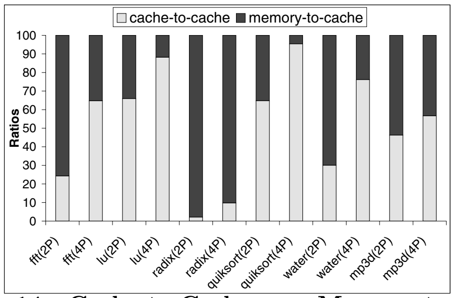


Figure 14: Cache-to-Cache vs. Memory-to-Cache Access

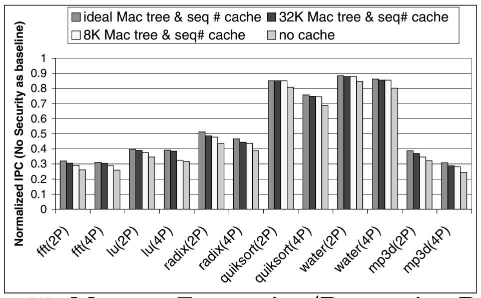


Figure 15: Memory Encryption/Decryption Performance

ure that cache to cache accesses for a four processor system is larger than for a two processor system. This is because of that for a four processor system there is more data in the caches which causes more communication among them.

Since the sequence number cache plays a critical role in determining the latency of encrypted memory data, we evaluated encryption protection performance under different sequence number cache sizes, no sequence number cache, 8K, 32K, and ideal. Each cache line of the sequence number cache is 64 bytes long and holds eight 64-bit sequence numbers. Figure 15 shows normalized IPC results. As the results indicated, using encryption will slow down the performance significantly for some benchmarks. With any caches, the averaged performance is about 45% of the baseline. The sequence number cache can hide some of the latency overhead. A larger sequence number cache will deliver better IPC results. One way to reduce the overhead of encryption is to increase the operating frequency of the North Bridge. We evaluated this option by running the encryption logic at 400Mhz. Normalized IPC results are shown in Figure 16. Under the assumption of ideal MAC tree and sequence num-

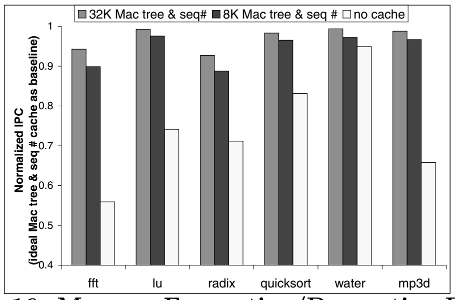


Figure 16: Memory Encryption/Decryption Performance at 400MHz, Processors=4

ber caches, the slow down is about 40%. Intergrating the North Bridge with the processor chip has the same effect with the North Bridge operates at the processor clock rate.

5. CONCLUSIONS

This paper proposed a unified hardware-based memory protection scheme for both uni-processor and MP platforms. It addresses the issue of protecting memory shared in an MP system.

At the same time when our scheme was developed, uni-processor based Loghash [4, 11] integrity protection was also extended to the scenario of memory protection in multi-processor environment. However, as pointed out in the paper, Loghash based scheme authenticates large number of memory accesses collectively thus is much less secure. Furthermore, loghash does not support precise interrupts on authentication exceptions. Last, loghash requires replicated authentication tree implemented in each processor therefore causing significant amount of area overhead than our approach, which requires only one authentication tree implemented in memory controller.

Our scheme on the other hand is not only able to protect the memory integrity but also the confidentiality of an MP shared memory. Different from the previous endeavors on uni-processor memory protection, our scheme achieves memory security in a platform distributed manner and relies on a light-weight secure processor implementation. Unlike previous approach that trades security for performance, the novel *authentication speculative execution* (ASE) is both secure to be combined with a one-time-pad (OTP) based memory protection and is effective in hiding authentication latency. Simulation results show a 5% performance overhead on a quad processor platform and 80% improvement over the *authentication in-order execution* (AIOE).

6. ACKNOWLEDGEMENTS

This research was supported by NSF Grants CCF-0326396 and CNS-0325536. The authors would also like to thank Tao Zhang for his early discussion on protection issues for shared memory MP systems.

7. REFERENCES

[1] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 65. IEEE Computer Society, 1997.

[2] Federal Information Processing Standard Draft. Advanced Encryption Standard (AES). National Institute of Standards and Technology, 2001.

[3] M. Galles and E. Williams. Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor. In *Technical report, Silicon Graphics Computer Systems,*, Mountain View, 1994.

[4] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and Merkle Trees for Efficient Memory Integrity Verification. In *Proceedings of the 9th International Symposium on High Performance Computer Architecte*, 2003.

[5] Matthias Gries and Andreas Romer. Performance Evaluation of Recent DRAM Architectures for Embedded Systems. In *TIK Report Nr. 82, Computing Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich*, November 1999.

[6] The Trusted Computing Group. https://www.trustedcomputinggroup.org/home.

[7] A. Huang. Keeping Secrets in Hardware the Microsoft XBOX Case Study . *MIT AI Memo*, 2002.

[8] D. Lie, C.Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectual Support For Copy and Tamper Resistant Software. In *Proceedings Of The 9th international Conference On Architectual Support For Programming Languages and Operating Systems*, 2000.

[9] National Institute of Science and Technology. FIPS PUB 180-2: SHA256 Hashing Algorithm.

[10] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM Reference Manual. Version 1.0. In *Department of Electrical and Computer Engineering, Rice University. Technical Report 9705*, 1997.

[11] E. G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S.Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings Of the 36th Annual International Symposium on Microarchitecture*, December 2003.

[12] E. G. Suh, D. Clarke, M. van Dijk, B. Gassend, and S.Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing . In *Proceedings Of The International Conference on Supercomputing*, 2003.

[13] S. Wilton and N. Jouppi. Cacti: An enhanced cache Access and Cycle Time Model . In *IEEE Journal of Solid-State Circuits,vol. 31, no. 5*, pages 677–688, May 1996.

[14] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages pages 24–36, Italy, 1995.

[15] Jun Yang, Youtao Zhang, and Lan Gao. Fast Secure Processor for Inhibiting Software Piracty and Tampering. In *36th Annual IEEE/ACM International Symposium on Microarchitecture*, December, 2003.