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

Weidong Shi H.H. Sean Lee Mrinmoy Ghosh
Chenghuai Lu Tao Zhang [*]
801 Atlantic Drive
Atlanta, GA 30332-0280
Georgia Institute of Technology
shiw,leehs,mrinmoy,lulu@cc.gatech.edu

ABSTRACT

Recently there is a growing interest in both the architecture and the security community to create a hardware based solution for authenticating system memory. As shown in the previous work, such silicon based memory authentication could become a vital component for creating future trusted computing environments and digital rights protection. Almost all the published work have focused on authenticating memory that is exclusively owned by one processing unit. However, in today's computing platforms, memory is often shared by multiple processing units which support shared system memory and snoop bus based memory coherence. Authenticating shared memory is a new challenge to memory protection. In this paper, we present a secure and fast architecture solution for authenticating shared memory. In terms of incorporating memory authentication into the processor pipeline, we proposed a new scheme called *Authentication Speculative Execution*. Unlike the previous approach for hiding or tolerating latency of memory authentication, our scheme does not trades security for performance. The novel ASE scheme is both secure to be combined with one-time-pad (OTP) based memory encryption and efficient to tolerate authentication latency. Results using modified rsim and splash2 benchmarks show only 5% overhead in performance on dual and quad processor platforms. Furthermore, ASE shows 80% performance advantage on average over conservative non-speculative execution based authentication. The scheme is of practical use for both symmetric multiprocessor systems and uni-processor systems where memory is shared by the main processor and other co-processors attached to 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 [11, 6, 4]. 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 proposed recently, hardware based memory authentication is often an essential and absolutely necessary component. Software based memory authentication, 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/copy right 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 operating system based authentication on either code or data before the program execution can help to reduce the risk but would not eliminate the vulnerability completely. For example, a simple software based solution is to allow the OS to schedule a process read from the disk only after the code image of the process has been authenticated. This solution enhances security but would not prevent attackers from modifying codes on the fly after it is 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 [3, 6, 7]. The challenge of memory authentication in the architecture design is to find out an efficient way that high speed memory authentication can be achieved at low cost without compromising security. However, most solutions proposed thus far assume that the memory is exclusively "owned" by one computing unit (often the main processor). Inside the processor, memory is authenticated on per process basis with a root memory authentication signature computed for each process's virtual space. Such strong process isolation (on both the inter- and intra- processor levels) prevents the root 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 root signatures for frequently shared memory information without significantly degrading system

[*]This tech report is a revision of a draft submitted to a computer architecture conference, March 21, 2004 with the same title.

performance. Worse yet, it is difficult to achieve integrity protection of memory shared among multiple processors because of potential re-play attack on either the shared bus or the shared physical memory. This means that all the existing approaches of hardware based memory authentication are not applicable to the scenario of multiprocessor memory protection.

We present a fast and low overhead solution to authenticate MP shared memory. 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 the kind of uni-processor systems that demand high performance secure data communication between the main processor and other processor-like peripherals that attach to the system bus. The scheme also optionally provides high performance protection on information confidentiality of shared data.

Furthermore, 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 using either un-authenticated data or results computed based on unauthenticated data can be speculatively issued and executed but is not allowed to retire until both the code itself and all the data it depends on 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 read/write 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 magnitute of tens of thousands of cycles.

The security and performance implication of the three design choices have not been clearly discussed previously in the literature and we advocate in this paper the advantage of ASE for both security and performance reasons. Although LAE could deliver the best performance but it is weak in security and when combined with a stream cipher, it leads to significantly less secure systems subject to many potential attacks. ASE achieves reasonable performance at the same time does not sacrifice security. Furthermore, ASE supports precise interrupts for authentication exception, which is not possible for a LAE based design.

[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 main contributions of the paper are summarized as follows.

- A unified fast and secure way for authenticating memory for both symmetric multiprocessor and uni-processor systems. The approach relies on division of labor and distributes security workload 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 re-play attacks on the shared memory.
- An innovative secure multiprocessor bus protocol for authenticating coherent bus transactions.
- A fast memory authentication approach based on stream ciphers 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 [7].

The rest of the paper is organized as follows, the next section presents the previous related work on memory authentication for memory exclusively owned by one processor. After that, section 3 addresses the security risks and performance implications associated with shared memory authentication. It also presents assumptions of the targeted platforms of our solution. Then in the next section, we present our main solution for authenticating memory shared symmetrical multiple processors. Section 4 shows performance evaluation and results. Finally, we conclude the paper in section 5.

2. SHARED MEMORY PROTECTION

In this section, we address many basic issues associated with shared memory protection at a high level. It presents the basic platform architecture our solution is targeted for. It also answers the questions such as why shared memory needs protection and shows the types of attacks our solution is aimed to prevent. It describes the main rationale of our solution and paves the road for the detailed discussion in the next section.

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 skill to build customized hardware to break the security protection in any imaginable way [10]. 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 fake bus signals, re-play bus transactions, spoof, alter or re-play RAM contents on the fly through hardware. Although software based protections or light weight hardware based protection such as TCPA [1] provide some protection using minimal silicon resources, it is almost impossible for them to survive from the kind of hardware attacks like the ones mentioned above.

Since the whole system is open to physical attacks from the hackers, almost all the recently published systems for hardware based memory protection assume that everything in a computing platform is insecure except the main processor with build-in security support[11, 6, 4]. Based on such assumptions, many proposed protection solutions often have all the hardware security features including memory authentication implemented in the main secure processor. Solutions proposed by Gassend et al. [3], the GHTree authentication scheme constructs a m-ary hash tree for protecting the integrity of virtual space of an application process in a single processor platform. As shown by the results, GHTree will incur about 20% execution overhead with a 2MB L2 cache and 33% memory overhead. To improve the performance of memory authentication, a LHash scheme [7] was proposed. The scheme logs memory operations and performs integrity checks only when a large number of memory operations are accumulated. Results indicate that LHash out-performs CHTree only when a large number of memory accesses are aggregated and authenticated together. According to our definition, LHash is a type of *lazy authentication* technique. Our study of attacks on tamper resistant system show that there are some potential security risks associated with *lazy authentication*, especially when it is used together with stream cipher for memory protection.

Furthermore, all the existing solutions are designed for uniprocessor memory protection and assume that the boundary between the protected secure domain and the insecure domain lies at the interface between the secure processor and the system bus. Such centralized view fits with the uniprocessor platform but does not apply to the multiprocessor systems. A simple way to extend the existing solutions to the multiprocessor scenario is to have a separate copy of the memory for each processing unit and have the secure OS to copy the data from one unit's trusted domain to another unit's domain using protected message passing mechanism when it is needed. This will however significantly increase the delay of inter-processor communication and greatly undermines performance of multiprocessor applications.

One of the challenges of designing a secure multiprocessor system is how to prevent and detect re-play attacks. There are two types of re-play attack, 1) re-play logged bus transactions, including both cache-to-cache and memory-to-cache bus transactions and 2) re-play information stored in the physical RAM. Under the system where the memory is exclusively owned by a single processing unit, re-play attack can be prevented using a hardware implementation of Merkle hash tree or a MAC tree inside the processing unit. But such solution does not apply when memory can be updated by multiple processing units/devices. If each processing unit 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 is not trivial and can have significant performance impact on frequent inter-processor communication.

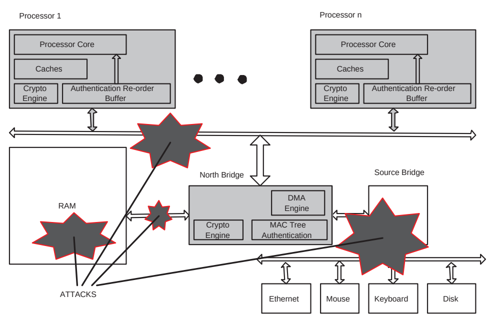


Figure 1: MP platform

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

A third challenge is on inter-operability of MP tamper resistant system with MP operating system. Protections provided by a MP tamper resistant system can be best viewed as security primitives analogous to other architecture support designed for synchronization and consistency. These security primitives themselves can not guarantees security requirements from being violated. But they can be used by properly developed secure operating system and secure application to construct a software environment that memory integrity and secrecy can be guaranteed.

To conclude, shared memory authentication is a unique problem of memory authentication and can not be satisfyingly solved by the existing single processor based Merkle hash/MAC tree approaches without significant modification. A fast, secure, and unified solution for authenticating memory is essential for designing future high performance tamper resistant system.

Different from the centralized approach, our new scheme tries to tackle the problem in a distributed fashion 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 targeted for both multiprocessor systems where a single system memory is shared by multiple processors through a shared coherent system bus and regular single processor system where system memory is shared by a main processor with other co-processors that attach to the system bus. Figure 1 illustrates the type of platform architecture our solution is applicable.

Instead of having each processing unit/device to use its own

Merkle hash/MAC tree, our solution solves the problem of shared memory protection through a centralized MAC tree based authentication implemented in the memory system with a secure multiprocessor coherent bus protocol. It is designed to provide a trusted environment for MP shared memory by securing the data path from each processing unit to the shared system memory. With the MAC memory authentication tree embedded in the *north bridge*, the data path between the memory controller and the physical RAM is secured. Furthermore the secure MP bus protocol provides a trusted and authenticated environment for both cache-to-cache and memory-to-cache bus transactions. Un-trusted devices cannot complete bus transactions to the protected shared memory and any attempt to re-play past authenticated bus transactions can also be detected. Since both the data path from each processing unit to the system memory and the data paths among processing units are protected, secure and authenticated sharing of the system memory becomes possible. Different from the previous approaches that put all the hardware resources for memory authentication into 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 detailed security model and architecture support for shared memory authentication. We only focus on the symmetric multiprocessor system where a coherent snoop 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. Such efforts are not made in this paper and the topic may be a subject of future study. The proposed MP shared memory protection scheme can be used to ensure both integrity and confidentiality for MP shared memory. In this section, we first present a platform oriented security model for multiprocessor system. Then, we propose architecture supports for the security model and *authentication speculative execution* as an authentication latency tolerating technique.

3.1 Symmetric multiprocessor security model

Figure 1 shows target architecture of MP systems. A split-transaction, cache coherent system bus provides interconnect for each processor to the memory system. 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's Xeon processor based MP systems. To simplify discussion, the scheme is presented using a split-transaction, SGI Powerpath-2 like cache-coherency protocol [8]. 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.

The security model presented in this section provides a system level specification for constructing a trusted environment for MP software execution. It addresses four issues, management of protected MP process, distribution and sharing of secret, protection on integrity and secrecy, and software distribution. The security model assume the existence of a secure OS kernel [3]. 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 [2].

3.1.1 process control

One basic and essential protection on a multi-task system is *process or task isolation.* Different process should not be allowed to access other process's protected domain. To achieve *process isolation*, two conditions must be satisfied. First, each process must be uniquely identified. Second, unique per process cryptographic information must be used for protecting integrity and confidentiality of each process's memory. The traditional process id is not a good choice because the likelihood of reusing a process id is very high. In this paper, we define a unique 128-bit number, *process uuid, universal unique identifier* to uniquely identify a process. The *process uuid* can be obtained either from a random number generator or derived from process id by encrypting it through a session key. Secret padding and keys used for authenticating or encrypting memory information of each process is derived from each process's uuid. The *process uuid* itself is not considered as secret and can be securely shared among multiple processors. Process uuid is treated as process context and is protected against tamper during process context switch.

A new privileged instruction is introduced to setup 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. Other steps relate to how the *process uuid* is shared by other devices attached to the shared bus. They are described later. *set_uuid* is a privileged instruction used by the secure kernel during process context switch.

3.1.2 integrity and confidentiality protection

Integrity code is also referred to as message authentication code (MAC), which is a well established technique to guaranteeing data integrity by verifying whether a piece of received or retrieved data was tampered or not during transmission or storage. For the purpose of memory integrity protection and authentication, when a processing unit wants to store a chunk of data to the insecure memory, it will compute an integrity code using a MAC generation algorithm and store the result integrity code 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 processing unit or any other processing unit wants to access the data, it will verify the integrity of the data by re-computing the integrity code of the retrieved data using the same MAC algorithm and compare the result with the retrieved integrity code. Alternation on either the integrity code or the data itself during storage will result into a mismatch of the two integrity codes.

In our shared memory authentication scheme, each processing unit is responsible for computing the integrity code for data to be stored to the memory or requested by other pro-

```
Write_Cache_Block():
variables:
    input cache_block$_{addr}$; // cache block
    input physical_addr; // addr of the cache block
    input process_uuid; // uuid for the executing process
    output encrypted_integrity_code$_{addr}$;
    output encrypted_cache_block$_{addr}$;
    static bus_sequence_num;
    static integrity_key;
operations:
    integrity_code$_{addr}$ =
        XOR_Truncate$_{64bit}$(SHA256(physical_address
        ||cache_block$_{addr}$||integrity_key));
    process_key =
        AES$_{session\_key}$(secret_const||process_uuid);
    integrity_OTP =
        XOR_Truncate$_{64bit}$(SHA256(const_integrity
        ||bus_sequence_num||process_key));
    encrypted_integrity_code$_{addr}$ =
        integrity_OTP$\oplus$integrity_code$_{addr}$;
    encryption_OTP =
        SHA256(const_encryption||bus_sequence_num||process_key));
    encrypted_cache_block$_{addr}$ =
        encryption_OTP$\oplus$cache_block$_{addr}$;
```

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

cessor units. When the data is shared by multiple units, the key along with other necessary information for generating/verifying the integrity code must be shared among all the involved processing units. The integrity code is encrypted using stream cipher when it is transmitted through the shared system bus. An *one-time-pad* (OTP) is uniquely computed using a confidential shared bus sequence number kept track by all the units attached to 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 reply to other processor's request, the data can be optionally encrypted using another OTP also computed based on stream cipher and the shared bus sequence number. figure 2 shows operations on a cache block that is either evicted from the secure processor or requested by other processors.

The operations are conducted on each protected cache line that is to be written to the system bus. XOR_Truncate$_x$ is a function that splits input into multiple x-bit chunks and xored them together to generate a x-bit output. SHA256 [12] and AES128 [5] 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 a 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. The symbol || in figure 2 stands for concatenation operation and $\oplus$ stands for XOR (exclusive or) operation. Both the integrity key and the sequence number are hidden from software access and can not be accessed externally neither. Similarly, computed data such as integrity_code and the process_key are also hidden from software access. Only encrypted integrity code and cache block are observable because they are transmitted over the shared bus.

Note that 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 approach described in the next section. Integrity verification and decryption of received cache

```
Read_Cache_Block():
variables:
    input physical_addr;
    input process_uuid;
    input encrypted_integrity_code$_{addr}$;
    input encrypted_cache_block$_{addr}$;
    output cache_block$_{addr}$;
    output authenticated;
    static bus_sequence_num;
    static integrity_key;
operations:
    process_key =
        AES$_{session\_key}$(secret_const||process_uuid);
    integrity_OTP =
        XOR_Truncate$_{64bit}$(SHA256(const_integrity
        ||bus_sequence_num||process_key));
    received_integrity_code$_{addr}$ =
        integrity_OTP$\oplus$encrypted_integrity_code$_{addr}$;
    encryption_OTP =
        SHA256(const_encryption||bus_sequence_num||process_key));
    cache_block$_{addr}$ =
        encryption_OTP$\oplus$encrypted_cache_block$_{addr}$;
    integrity_code$_{addr}$ =
        XOR_Truncate$_{64bit}$(SHA256(physical_address
        ||cache_block$_{addr}$||integrity_key));
    if (received_integrity_code$_{addr}$ == integrity_code$_{addr}$)
        authenticated = true;
    else
        authenticated = false;
```

Figure 3: Security Operations on Each Cache Block Received

block (coherence reply and memory read) are shown in figure 3. Output bit *authenticated* indicates whether integrity of the read cache block can be verified. It is set if integrity of the received cache block can be verified.

As shown in figure 2 and figure 3, the designed security model minimizes the performance critical interval between encryption and decryption. The encryption_OPT can be pre-computed. In the best scenario, the interval of transferring an encrypted cache block consists of only time of a XOR operation on the send side, transmission delay, and another XOR operation on the receive side. 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 distribute and share secret

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 broadcasted 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 not be the same each time the machine is rebooted to prevent re-play attack. In this paper, 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.

First, during boot time, the bootstrap processor broadcasts the range of physical memory to be protected to all the processing units and memory controller. It could be only a

```
Distribute_Shared_Secret():
variables:
    input device_id;
    output bus_sequence_num;
    output integrity_key;
    output session_key;
    local random_numuber_array[MAX_NUM_DEVICES];
    local counter;
operations:
    counter = 0;
    while (1) {
        if (counter!=device_id) {
            random_number_array[counter] =
            random_number_broadcasted_by_device$_{counter}$;
        } else {
            broadcast a random number;
            random_number_array[device_id] =
            the number broadcasted;
        }
        counter++;
        break if all the devices are granted chance to broadcast;
    }
    random_string = random_number_array[0] ||
        random_number_array[1] ||
        ... ||
        random_number_array[TOTAL_NUM_DEVICES];
    session_key =
        XOR_Truncate$_{128bit}$(SHA256(random_string
        ||secret_hash_key));
    integrity_key=AES$_{session\_key}$(secret_constant);
    bus_sequence_num=AES$_{session\_key}$(secret_constant+1);
    enter barrier;
```

Figure 4: Processor Initialization and Distribution of Shared Secrets

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 multiprocessor bus is granted in turn bus cycles to broadcast a random 64-bit number. Then each unit concatenates all the random number it collects from the bus including the one it broadcasts and feeds the result into a 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 computed based on the session key (figure 4).

Both the secret_hash_key and the secret_constant are constants permanently burnt into the processor chip, and memory controller during manufacture. They are secrets stored in the chip un-accessible 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 symmetric multiprocessor 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 a MP system is freshly rebooted.

3.1.4 software distribution and platform key

The cryptographic protection proposed in this paper is "self-contained" because the session key, the root of all the keys, the sequence number, etc, is not a constant and modified each time after the system is rebooted. Software vendors are not able to generate the integrity code or OTPs used in the protection because they don't known 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 multiprocessor 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 MP 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 in the next subsection, this conversion does not necessarily require processor involvement and can be performed in high speed with security enabled DMA engines.

3.2 Architecture Support of MP security model

In this subsection, we present a detailed architecture model for implementing the MP security model described in the previous section. There are two important issues that have to be kept in mind during implementation. One is that the implementation has to be secure. The second is that the implementation must be efficient and high performance under the condition that security is not compromised. 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 reply of cache-to-cache communication has to be computed, transmitted, recomputed, and verified. To tolerate the latency of integrity checking, we proposed two new techniques described in this subsection. First, we proposed a split transaction bus model for data and its integrity code. Second, we proposed *authentication speculative execution* to further tolerate 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 re-play attack on the shared coherent MP bus. It plays an essential role for providing a chain of authentication 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 with any MP coherence bus protocol, but for the sake of discussion, we restrict the design to a 4 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. Transition between cache states is caused by actions initiated by the processor or by coherent transactions appearing on the bus. Duplicate set of cache tags [8] is maintained by the processor interface ASIC and bus arbitration is done in distributed manner. Similar to POWERpath-2, every bus transaction consists of five clock cycles. System wide bus controller logic executes the same five-state machine synchronously: arbitration, resolution, address, decode, and acknowledge.

All the devices connecting to the shared multiprocessor bus including the memory controller share the 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 snoop 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 according to figure 4. The number is kept as secret by all the involved devices and never transmitted in either plaintext or ciphertext across the bus. Aside from the sequence number, the integrity key, the session keys are also shared by all the devices attached to the shared bus.

One unique performance feature of our secure bus is split transaction of data and its integrity code. As shown in figure 2 and figure 3, integrity code computing and verification is the slowest part of MP security model. To minimize the impact of authentication on performance, our secure bus model allows data block and its integrity code separately transmitted. For coherent reply, 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 or 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 enough for security protection. This is because for a bus running at speed of hundreds of MHz or a few GHz, it would take at least hundreds of years if not thousands for a 64-bit sequence number to wrap-around.

3.2.2 secure memory system

Note that using integrity code alone is not sufficient for verifying memory integrity because a hacker can replace new data along with its MAC stored to the memory with old staled data and MAC. Such re-play attack is not detectable without using techniques such as Merkle hash tree or MAC tree. When memory is not shared by multiple processors, all integrity protection features can be implemented in the main processor [3, 6, 7]. But this solution can not be applied to MP shared memory systems.

The secure memory system consists of a memory controller with an integrated security engine, RAM DIMMs, and a number of physical RAM chips. The primary goal of the security engine embedded in the *north bridge* memory controller is to detect alternation or re-play of data stored in the system memory. It is another critical component in the chain of shared memory authentication. 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. A MAC tree can be 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 is as follows. After integrity of received data

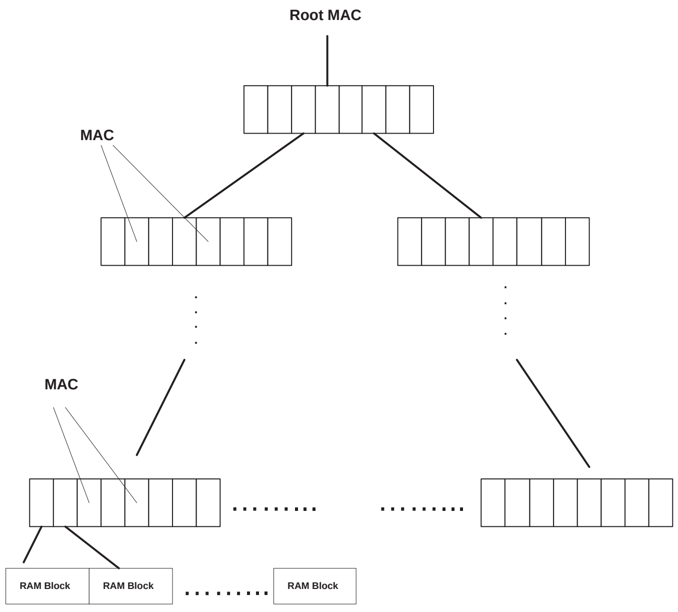


Figure 5: MAC tree

```
Sequence Number Encryption():
variables:
        input bus_sequence_num;
        input address;
        output encrypted_bus_sequence_num;
        static sequence_MAC_key;
        static sequence_key;
        local counter;
operations:
        sequence_number_integrity_code$_{addr}$ =
            XOR_Truncate$_{16bit}$(SHA256(address
             ||bus_sequence_num||sequence_MAC_key));
        encrypted_bus_sequence_num =
             AES$_{sequence\_key}$(bus_sequence_num
             ||sequence_number_integrity_code$_{addr}$);
```

Figure 6: Bus Sequence Number Encryption

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 encrypted bus sequence number to the memory. Each bus sequence number can be encrypted using AES as shown in figure 6. Both the sequence_MAC_key and the sequence_key used during encryption are secret information maintained by the memory controller.

To improve performance, the bus sequence numbers for frequent data blocks can be cached inside the *north bridge* memory controller. This will 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 re-play, 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, validity of the integrity code is verified. Since it is impossible 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 they are 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 plays a critical role in high performance computing and should be minimized whenever it is possible. Our secure memory system is specifically designed for reducing memory latency at the same time without loss of security protection. The features provided by the proposed architecture to reduce memory latency are summarized as follows:

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

The secure memory controller is the center of the proposed MP security model. Beside the MAC tree, it also shares secrets with other processors attached to 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 conducted through protected bus transactions. First, there is a process uuid register associated with 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 according to 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 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 according to figure 2. The platform key pair is permanently set in the *north bridge* during manufacture.

It is important to point out that the secure OS kernel always resides in a separately protected memory space. Access to

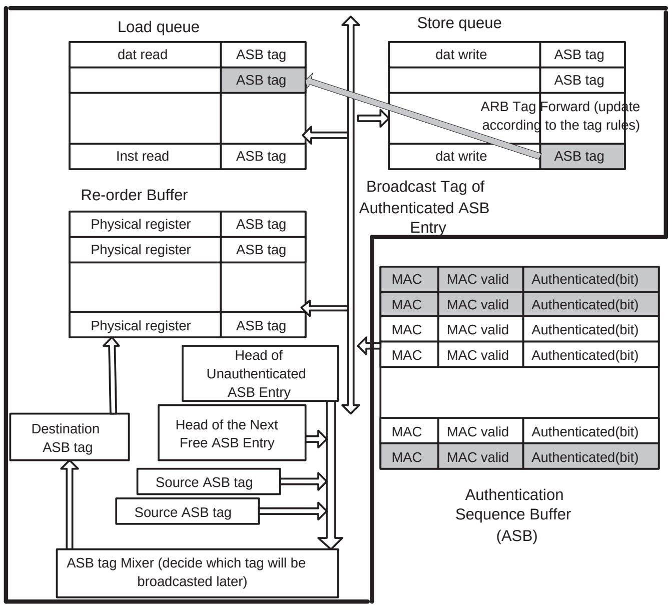


Figure 7: Authentication Speculative Execution

the secure kernel uses a different process key and *set_uuid* is not required when application switches to secure kernel mode. A copy of the memory range of 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 secure memory controller 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 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.

Under 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 com-

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

plete if its integrity code has not been received. For both inter-processor communication and regular memory fetch, data can be transmitted and processed even before integrity code arrives, hence the pipeline execution will not be stalled. Note that 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 *Authentication Sequence Buffer* (ASB) which is illustrated in figure 7. ASB 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 one "authenticated_bit" associated with each ASB entry. This bit is set when the integrity of a read transaction is verified. ASB broadcasts the index of an authenticated entry to a bus shared by memory load/store queues and the re-order buffer. We should note that though the transactions can be authenticated in any order ASB 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 ASB tag 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 ASB 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 ASB is verified. When a load/store queue or reorder buffer snoops a broadcast of index i from the ASB, any entry tagged with index i is reset to zero.

An ASE processor allows the usage of un-authenticated data for continuing program execution. However, the uncommitted results will be tagged with its corresponding entry index in the ASB. For example, let us consider an instruction "load r2, [addr]" which uses data fetched from [addr], but misses the cache. Therefore, data of [addr] will be fetched from memory and the transaction allocates an entry with an index i in the ASB. 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 ASB later. If i would be broadcasted later than k, then r3 will be tagged with i, otherwise, r3 will be tagged with k. The pseudo-code in figure 8 shows how to decide which tag would be broadcasted later given two source tags, the ASB tag to be broadcasted next(ASBhead), and a pointer to the next free ASB entry (next_free_ASB). Implementation of the ASB

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

Figure 8: Compute a new ASB tag

tag updating rules is done in hardware, and called ARB Tag Mixer.

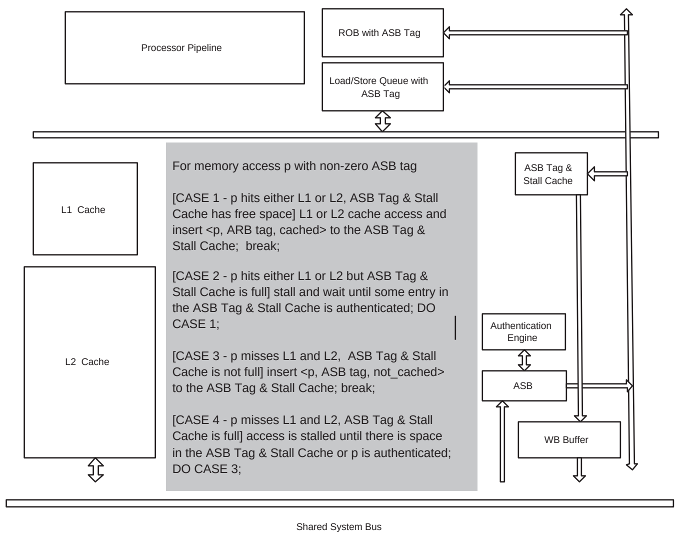
When un-authenticated data is to be stored to the memory hierarchy or data/instruction needs to be fetched from memory hierarchy using address computed from un-authenticated data, it is allowed to be issued to the memory hierarchy. Load and store queue of memory units are also extended with an ASB tag field. When a piece of data is forwarded from the store queue, its ASB tag kept in the store queue is also "forwarded" to update the destination register's ASB tag.

If the access $<$ addr, ASB 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 ASB tag i will also be retrieved and used to update the register's ASB tag field. But, if the access is *authentication unsafe* and triggers a 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 ASB tags.

The *Authentication Stall Cache* (ASC) also receives ASB 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.

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 ASB 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 ASB 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 ASB tag of data sources, and



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

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 ASB tag associated with the instruction. However, if the conditional branch triggers a 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 of both the MP bus protocol and the secure memory system. The objective of MP shared memory authentication is to prevent unauthorized alternation and re-play of coherent reply and data stored in the shared memory. There are a number of techniques adversaries can try to break our system and we will show that none of them will succeed in breaking the proposed protection mechanism. First, hacker may try to forge the integrity code. This clearly would not work because 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 a machine is rebooted. Second, hacker may try to re-play both data and the associated integrity code on the MP bus, this would fail because every bus transaction is protected by a sequence number that does not stay the same. Third, hacker may try to do re-play 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 stream cipher, 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 as a 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. Re-play data written by a different process will also fail because the MAC is encrypted by OTP generated using a process key unique to each process. Finally, security privileged instructions such

Table 2: Memory Overhead

Structure	Size (bytes)
`ASB(32 entries)`	32*(8b MAC+1b cntrl)=288
`ASB Tag & Stall Cache64 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

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

The proposed memory protection scheme described needs additional memory space to implement security related tables or caches such as ASB, ASB 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, ASB 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. As we use 256-bit cache line and 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

Table 3 shows the MP applications we used in the study and the parameters used for simulation. The applications are from the SPLASH-2 benchmark suite [14].

For charaterizing and evaluating our proposed scheme for an MP system, we use RSIM [15] as our infrastructure to simulate a 4-node MP system. Each node includes a MIPS R1000 like out-of-order processor, L1, and L2 cache. We modified the simulator in order to support the SGI POWERpath-2 MP coherent bus protocol and a shared main memory. Secure snoop bus protocol, memory authentication, and *authentication speculative execution* are all incorporated into the RSIM simulator. To characterize the memory transactions more accurately, we integrated an accurate DRAM model [9] based on the PC SDRAM specification. SPLASH-2 benchmark suite [14] was used. The SPLASH-2 bench-

Parameters	Values
CPU	4-issue per cycle
reorder buffer	64 instructions
load/store queue	64 instructions
L1 cache	8-Kbyte, directly mapped
L2 cache	4W, 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

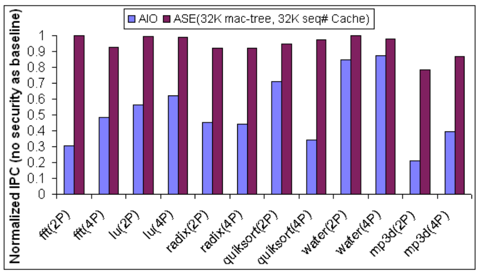


Figure 10: Authentication Performance

mark 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*

First, we compared the performance of different authentication execution schemes in figure 10. The figure shows IPC normalized to baseline (baseline corresponding to the results under no security protection of any kind) 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 creating commodity workstations today. The data indicates that ASE is much faster than AIOE and incur very small 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 executed instructions. Based on the figure, we can find out applications that have high cache miss rates or applications that have lots of memory references.

Furthermore, we evaluated the effect of ASB tag & stall cache. There are several conditions, an ideal tag cache (always hit), a 32 entry tag cache (8-way, 4 set), 64 entry tag cache (8-way, 8 set), and no tag cache under a quad pro-

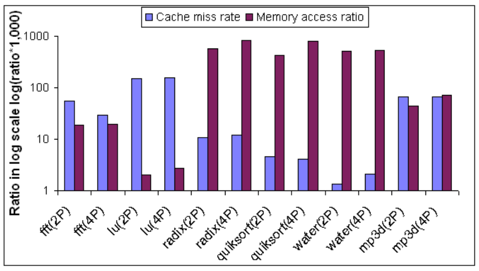


Figure 11: Characteristics of Cache and Memory References

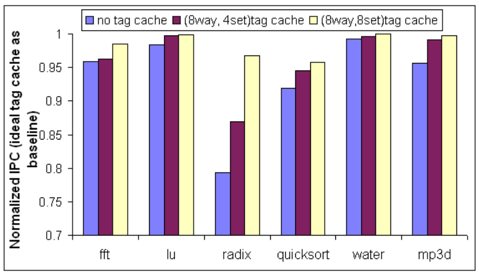


Figure 12: Authentication Performance under Different Tag Cache Setting, 32K mac tree & 32 K sequence number caches, Processors=4

cessor setting. The results are displayed in figure 12. The figure shows IPC normalized to the results of ideal tag cache. As suggested by the results, tag cache can improve performance for some benchmarks especially quicksort, radix, and mp3d. According to 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 with results under ideal MAC tree and sequence number cache. On average, 32KB MAC tree and sequence number caches show IPC 97% of IPC obtained with ideal MAC tree and sequence number caches. The performance of 8KB caches and no cache are 94% and 74% respectively.

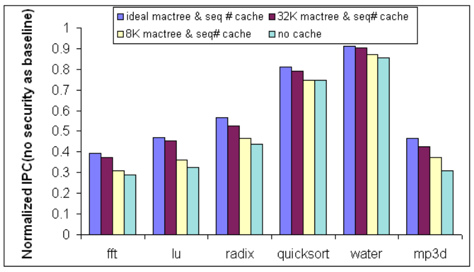


Figure 13: Authentication Performance under Different North Bridge Cache Resources, Processors=4

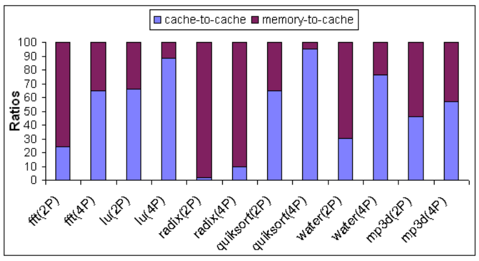


Figure 14: Cache-to-Cache vs. Memory-to-Cache Access

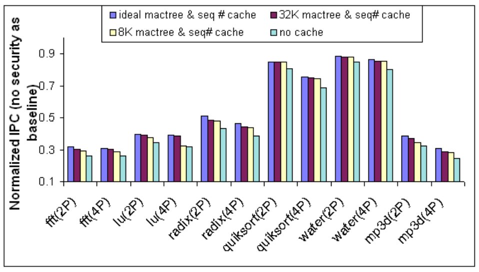


Figure 15: Memory Encryption/Decryption Performance

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 stream key is pre-computed. For memory-to-cache access, the overhead is bigger because the stream key can not be pre-computed (it can be done under pre-fetch but in this paper, we don't assume the existence of pre-fetch). However, the memory controller can start computing of the stream key 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. As suggested by the data, most of the SPLASH/SPLASH2 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 figure that cache to cache accesses for a four processor system is larger than for a two processor system. This may be explained by considering the fact that for a four processor system there is more data in the caches which causes more communication among them.

Since sequence number cache plays a critical role in determining 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 64bytes long and holds 8 64bit sequence numbers. Figure 15 shows normalized IPC results.

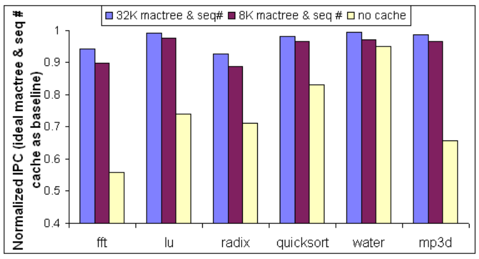


Figure 16: Memory Encryption/Decryption Performance Under Higher Clock-rate, Processors=4

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. Sequence number cache can hide some of the latency overhead. Larger sequence number cache has better IPC results. One way to reduce the overhead of encryption is to run the north bridge at higher clock rate. 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 number caches, the slow down is about 40%. Intergrading 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 scheme of memory protection for both uni-processor and multiprocessor platforms. It is the first paper to address the issue protecting memory shared in a MP system. The scheme is not only able to protect memory integrity but also confidentiality of 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 weighted secure processor implementation. Unlike previous approach that trades security for performance, the novel *authentication speculative execution* (ASE) is both secure to be combined with one-time-pad (OTP) based memory protection and efficient to hide authentication latency. Results using rsim and splash2 benchmarks show only 5% overhead in performance on a quad processor platform and 80% improvement over AIOE.

6. REFERENCES

[1] The Trusted Computing Platform Alliance. http://www.trustedpc.com. 2003.

[2] 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.

[3] D.Clarke M.van Dijk B.Gassend, G.E. Suh 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.

[4] M.Mitchell P.Lincoln D. Boneh J.Mitchell D. Lie, C.Thekkath 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, 2000.

[5] Federal Information Processing Standard Draft. Advanced encryption standard (aes). national institute of standards and technology, 2001.

[6] Blaise Gassend Marten van Dijk Srini Devadas Edward Suh, Dwaine Clarke. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Int'l Conference on Supercomputing*, 2003.

[7] B. Gassend M.van Dijk E.G.Suh, D.Clarke and S.Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings Of the 36th Annual International Symposium on Microarchitecture*, December, 2003.

[8] 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.

[9] 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.

[10] A. Huang. Keeping secrets in hardware the microsoft xbox case study. *MIT AI Memo*, 2002.

[11] Lan Gao Jun Yang, Youtao Zhang. Fast secure processor for inhibiting software piracty and tampering. In *36th Annual IEEE/ACM International Symposium on Microarchitecture*, December, 2003.

[12] National Institute of Science and Technology. Fips pub 180-2: Sha256 hashing algorithm.

[13] N. Jouppi S. Wilton. 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] Evan Torrie Jaswinder Pal Singh Steven Cameron Woo, Moriyoshi Ohara 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] Parthasarathy Ranganathan Vijay S. Pai and Sarita V. Adve. Rsim reference manual. version 1.0. In *Department of Electrical and Computer Engineering, Rice University. Technical Report 9705*, 1997.