Authentication Control Point and Its Implications For Secure Processor Design

Weidong Shi
Motorola Application Research Lab
Motorola, Inc.
Schaumburg, IL 60196
larry.shi@motorola.com

Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
leehs@gatech.edu

ABSTRACT

Secure processor architecture enables tamper-proof protection on software that addresses many difficult security problems such as reverse-engineering prevention, trusted computing, secure mobile agents by providing a secure computing environment that is resistant to both physical tampering and software exploits. Two essential features offered by a secure processor are software encryption for protecting software privacy and integrity verification for preventing tampering of the protected software. Despite a number of secure processor designs have been proposed, the delicate relationship between privacy and integrity protection in the context of modern out-of-order processor design is not well understood. This paper aims to remedy this research deficiency by evaluating different designs that integrate software decryption and integrity verification into an out-of-order pipeline. Our paper provides an in-depth analysis of the security and performance trade-offs, implications of several designs in the context of memory fetch side-channel exploits. Among the evaluated spectrum of design alternatives are (1) authentication-then-issue, (2) authentication-then-commit, (3) authentication-then-write, (4) authentication-then-fetch, and (5) authentication-then-commit + address obfuscation. Performance of various designs was evaluated using a cycle based processor model and SPEC 2000 benchmark suite.

1. INTRODUCTION

According to the US Department of Defense, 80% mission critical information are stored in various memory devices of high tech military or communication systems. Those systems, when falling into the hands of hostile parties, the critical information stored in their memory devices may be eavesdropped or disclosed. Secure processors such as those described in [5, 12, 13, 21, 22, 27, 28, 25] come to the rescue because they provide tamper-resistant protection for information stored in regular memory devices and support a tamper-proof computing environment. In addition, a general-purpose secure processor can also create a strong trusted computing environment, enabling digital rights protection.

Secure processors offer two main cryptographic services for protecting both the static and dynamic image of valuable software or data stored in regular memory devices: encryption and authentication. Encryption protects privacy of information while authentication protects information integrity and detects tampering for the protected data. At the first glimpse, the task of integrating a cryptographic engine into an out-of-order high performance processor for information security may seem deceptively straightforward. Nevertheless, in-depth investigation reveals that many design and security issues have not been fully understood based on existing literature. Particularly, the role of integrity protection and its relationship with privacy protection in the context of secure processor design was not sufficiently addressed.

Due to the complexity and performance consideration, it is common to disassociate decryption and authentication operations in a secure processor architecture, e.g. issuing decrypted instructions before the completion of verifying their authenticity. Justification of such disassociation becomes even stronger because the latency of authenticating fetched data is often significantly longer than the latency of decrypting them when a performance optimized decryption design is applied for privacy protection. This type of disassociation has been proposed or suggested in many published secure processor designs [12, 18, 23] with little or no investigation on the security implications of decoupling decryption and authentication.

In this paper, we explore and scrutinize the design space of decryption and authentication disassociation with the objective to find a solution that is secure, fast, and simple to implement. We investigated and evaluated five designs, (1) *authen-then-issue*, (2) *authen-then-commit*, (3) *authentication -then-write*, (4) *authen-then-fetch*, and (5) *address obfuscation plus authen-then-commit*. Under *authen-then-issue*, a secure processor does not issue instructions or operands whose integrity has not been fully verified. *Authen-then-issue* is a conservative solution that allows almost no decryption and authentication disassociation. Although it is secure, the downside is that it may incur significant performance overhead. Under *authen-then-commit*, a secure processor speculatively issues unverified instructions and data to the pipeline and commits finished instructions only after both the instruction itself and its operands are authenticated. Another design is *authen-then-write*. Under *authen-then-write*, all permanent changes to memory state have to be made based on the results derived from the authenticated instructions and operands. In contrast to *authen-then-write*, under *Authen-then-fetch*, a secure processor allows bus cycles to be granted to a memory fetch only if all the instructions and data that the memory fetch depends on due to data or control dependency were authenticated. In addition, we also examine address obfuscation and its relation to the aforementioned authentication designs.

The contributions of this paper are:

- Identify and discuss the security risks and implications of decoupling decryption and authentication in the context of using memory fetch as a side-channel for violating content confidentiality and privacy;
- Propose a range of design choices that allow different degree of decryption and authentication disassociation;
- Evaluate the security trade-off, design complexity and performance of the proposed range of designs that de-

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

couple decryption and authentication.

The rest of the paper is organized as follows. Section 2 discusses secure processor architectures. Section 3 describes several exploits based on the memory fetch side-channel. In Section 4, we present a range of design choices and address each's pros and cons. It also describes architecture details of how to integrate authentication mechanism into a processor pipeline. The performance of the proposed designs are evaluated in Section 5, followed by related work in Section 6. Finally, Section 7 concludes this work.

2. SECURE COMPUTING MODEL

A secure processor architecture ensures that applications will be executed in a tamper-proof manner. To achieve these goals, two techniques are commonly employed. The first one is data encryption. Encryption, for protecting *confidentiality* and privacy, aims to eliminate the possibilities of disclosure of sensitive data. As the data or software stored in the off-chip memory are encrypted, adversaries should be unable to reveal them even if the contents of the external memory could be eavesdropped via exposed interfaces. Secure processor can choose any standard encryption mode for memory encryption. Since memory decryption has a significant overhead, encryption modes that allow parallel decryption or pre-computation are preferred over the encryption modes that demand serial decryption operations. Recently, a variety of *counter mode* based techniques have been applied to secure processor designs [19, 23, 27].

The second technique is integrity verification or also called authentication. Integrity verification, achieved by employing *Message Authentication Code (MAC)* [15], guarantees the detection of any unauthorized modification of the program or data. The MACs are stored along with each data block such as a cache line. On-die caches are considered part of the trusted environment, hence, the instructions and data can be kept in plaintext when residing inside the on-chip caches during execution. There are several approaches and standards for generating a MAC, e.g. HMAC [10], CBC-MAC [2], etc. It is up to the discretion of the secure processor's designer to choose the one that is most efficient and secure.

In general, in a secure processor that uses optimized decryption mode for decrypting fetched instructions or data, the authentication process may take longer time than the decryption because in theory a secure processor can only start verification after data is fetched from memory. In contrast, under some encryption modes such as counter mode, decryption can start in many cases when a memory fetch address is generated [19, 23, 27]. This creates a latency gap between the time when the instructions and data are decrypted and the time when they are authenticated or verified. Note that some authentication schemes such as CBC-MAC have both long decryption latency and authentication latency. Though those schemes have a narrower gap between the decryption latency and the authentication latency, they are less favorable because the ability to decrypt critical words fetched from the memory within a minimal latency plays a critical role in the performance of a secure processor. Table 1 shows the latency gap between decryption and authentication under two different memory protection schemes, [Counter mode + HMAC] and [CBC + CBC MAC], where N is the number of 128-bit data chunks in a cache line and n ($>=0$) is the index of a data chunk. Decryption latency denotes the latency of the decryption cipher used such as the AES.

3. MEMORY FETCH AS INFORMATION DISCLOSING CHANNEL

A typical secure processor may encrypt information stored in an external memory device. But, memory fetch address shown on the front side interface between a secure processor and the memory device is usually not encrypted. Encrypting fetch address is neither practicable nor necessary because as long as the system still uses regular DRAM modules, an adversary can always eavesdrop at the final interface where fetch addresses themselves are not encrypted. Since a secure processor discloses fetch address in plaintext, an adversary may comprise information privacy by tricking the system to send sensitive information as fetch addresses. In fact, there are many possible hacking scenarios that allow an adversary to achieve such a feat.

3.1 Threat Model

Previous research on memory fetch security adopts a natural execution threat model where memory fetch trace (both data and code fetch) created by natural execution of a program may comprise a side-channel for leaking sensitive information due to possible partial reconstruction of program control flow [29, 24]. However, there are some more serious security risks if the disclosed fetch addresses are induced by tampered program or data.

For attaining high performance, a modern out-of-order processor often speculatively issues and executes instructions, speculatively fetches instructions or data. These speculative execution techniques when combined with secure processor design can lead to overly aggressive disassociation of the decryption and the authentication, which may induce potential new security risks. An adversary may deliberately alter software or data in some specific way and the altered software or data, if executed or used speculatively, may disclose sensitive information via the memory fetch addresses. Shi et al. in [20] documents some exploit scenarios that could occur when authentication is not performed in a timely fashion. In this paper, we further demonstrate the necessity of using decrypted yet authenticated information with scrupulous caution. To be self-contained, we briefly describe some vulnerabilities associated with loose coupling between the decryption and the authentication in the following subsections. Additional details can be found in [20].

Most standard encryption modes including CBC mode, counter mode and many others are malleable. A malleable encryption mode is a mode where flipping certain bits in a ciphertext will induce certain bits of the decrypted plaintext to flip as well. In the counter mode encryption, flipping a particular ciphertext bit will make the same bit of the corresponding decrypted plaintext to flip. In the CBC mode, flipping a ciphertext bit will induce plaintext bit flip at certain offset depending on the encryption unit size of the underlying cipher. Through bit flipping, an adversary may transform a piece of protected program or data in such a way that the altered code or data may either directly or indirectly disclose sensitive information as fetch addresses. It is a standard practice to provide non-malleability to a malleable encryption mode by adding authentication [3]. However, disassociation or decoupling of the decryption and the authentication in an out-of-order processor may leave large enough security holes for the hackers. The authentication may lag behind the decryption with a latency ranging from tens of cycles to even hundreds. This provides a security-blank execution window where tens of unverified instructions or more may be speculatively executed without their integrity completely verified. In general, those speculatively executed instructions are not allowed to modify processor and memory state before they reach the commit stage. Unfortunately, memory fetches are not *architecture state change* operations. A standard processor will grant bus cycles to speculative memory fetches before the commit stage. This makes memory fetch a *side-channel* for leaking

	Decryption Latency	Authentication Latency
Counter Mode+HMAC	MAX(memory fetch latency, decryption latency)	memory fetch latency + HMAC hash latency
CBC + CBC MAC	memory fetch latency+decryption latency*(n+1)	memory fetch latency+decryption latency*N

Table 1: Latency Gap Between Decryption and Integrity Verification (approximate estimate)

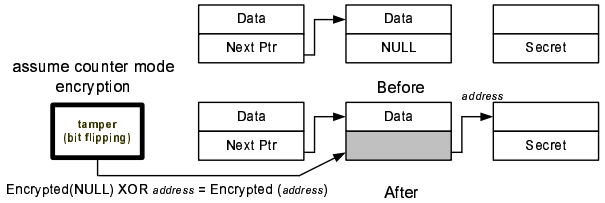


Figure 1: Point Conversion Exploit

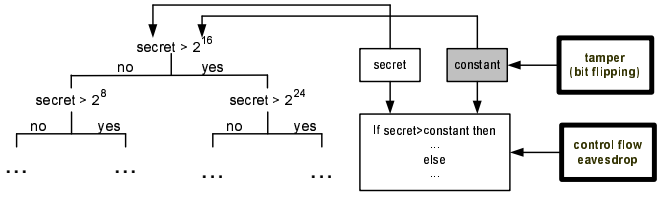


Figure 2: A Binary Search Exploit

out sensitive information.

3.2 Side-Channel Exploits via Fetch

Now we briefly describe some of the exploits of memory fetch side-channel for breaking data confidentiality protection provided by a secure processor. It is worth pointing out that the list given below is by no means comprehensive.

3.2.1 Pointer Conversion

The basic idea of *pointer conversion* exploit is to convert encrypted sensitive data into pointers such that its value will be automatically disclosed when the pointers are dereferenced. There may exist many variations of this exploit. Here we give only one example, called *"linked list attack"* to illustrate how to recover encrypted data by only altering program data. Linked list is commonly used in programming. One property of linked list is that the last node is always terminated with a NULL. Assume that an adversary knows where the linked list ends (the last node). An adversary may use input manipulation or control flow reconstruction based on fetch trace to either force a linked list to end at some known location or discover when and where a linked list terminates. Further assume that there is a secret data value x stored in memory location l, which the adversary wants to discover. The adversary can alter the NULL pointer into l - *node size* + *4* so that the secret data becomes a node pointer as shown in Figure 1. To convert a counter-mode encrypted NULL pointer into an encrypted pointer value pointing to l - *node size* + *4* requires only one simple XOR operation of the encrypted NULL pointer with the address as shown in Figure 1. When the linked list is traversed, the program will try to use the secret data as a node pointer (i.e. an address) and issue a corresponding memory load that reveals its value as a (bogus) fetch address. There are many ways for an adversary to recover memory locations of sensitive data. For example, an adversary may run a local copy of the same system and discover the likely position where sensitive data may be stored [11]. An adversary may also use control flow leaked from the memory fetch to find out likely memory locations of sensitive data.

3.2.2 Binary Search

If there is some comparison that compares some secret data with some constant stored in the memory and the constant value is known by an adversary. The adversary may launch so called *binary search* exploit. The adversary may alter the constant value in the power-of-2 basis and eavesdrop how the modification will affect the comparison result (control flow). If the secret data is 32-bit long, according to the principle of binary search, at most $\log_2(2^{32}) = 32$ trials will be enough to recover the sensitive data. Figure 2 illustrates such an exploit.

Binary search exploit requires tampering of constant or constant string values or any other value that is used for comparison with a secret. To launch bit flipping attack on constant or strings, an adversary must first recover their plaintext values. This will not be too difficult because lots of strings or constants are either outputs that can be eavesdropped or input supplied by the users. For example, constant zero is frequently used for testing and comparison. If a program tests a secret value against value zero, an adversary could launch aforementioned binary search by tampering the zero value that the secret is compared with.

3.2.3 Disclosing Kernel

Disclosing kernel is a short piece of malicious code that may disclose possibly arbitrary data to a side-channel. The simplest *disclosing kernel* comprising only two RISC instructions is one that loads some arbitrary data into a secure processor, then uses the data as a fetch address. An adversary may insert a *disclosing kernel* into either the code space or data space by tampering either code or data. Then by altering one more instruction or function return address, the adversary may hijack program control to the *disclosing kernel*. A slightly sophisticated version of *disclosing kernel* such as the one that has a loop can potentially disclose the entire application's memory space to a side-channel.

Inserting a *disclosing kernel* into an application's code space is in fact much easier than people thought. An adversary needs to first guess or recover a short sequence of encrypted instructions' plaintext whose size is large enough to hold the *disclosing kernel*. Then by applying two XOR operations — *disclosing kernel XOR encrypted instruction sequences XOR recovered/guessed instruction plaintext*, the adversary will be able to embed the disclosing kernel. Injecting or embedding *disclosing kernel* requires recovery or guess of a short piece of code or data, which is not hard at all because RISC instructions even in encrypted format are highly predictable. One source of predictability comes from invariant code sequence. Compiler always does code generation in a predictable way. The binary codes produced by a compiler comprise many short code sequences that are either invariant or predictable. Such invariant or predictable short code sequence can be found in a program's entry point, function epilogue or prologue, compiled loop structure, etc. An adversary can replace one of those predictable or invariant code sequences with a *disclosing kernel*.

Inserting a *disclosing kernel* into an application's data space could also be very simple because of the existence of frequent data values [26]. Research shows that a large percentage of data values are zeros. An adversary may have a very high success rate of inserting a disclosing kernel into a counter mode encrypted memory space by simply XORing

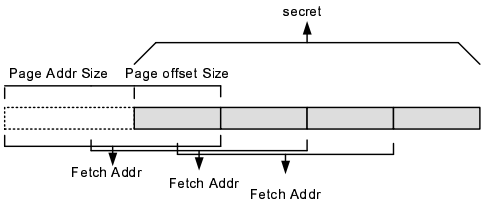


Figure 3: Shift Window

```
/* given base virtual page address 0x01ebc000 is valid */
R0 ← LOAD[p] /* load secret into R0 */
Loop:
  R1 ← R0 & 0x000000ff
  R1 ← R1 | 0x01ebc000
  LOAD [R1] /* disclose 8 bits */
  R0 ← R0 >> 8 /* right shift 8 bits */
  JMP Loop
```

Figure 4: Example of Disclosing Kernel

the *disclosing kernel* with ciphertext whose plaintext is most likely to be zeros. If a *disclosing kernel* is inserted into data space, the adversary needs to hijack the program control flow into the *disclosing kernel*.

Finally, a disclosing kernel can alternatively output sensitive data to a standard output channel such as I/O ports instead of using it as a fetch address. The difference between those two types of exploits is that output to an output channel is generally considered as operations that have to wait until the completion of integrity verification. This means that *authen-then-commit* could be sufficient to prevent a disclosing kernel from outputting sensitive data to an I/O channel.

3.3 Impact of Virtual Memory Translation

The aforementioned exploits represent an ideal situation where sensitive data always show up directly as memory fetch addresses. Most high performance processors support virtual memory. A piece of sensitive data when used as fetch address may map into invalid address space and trigger memory translation exception. It is noteworthy that many embedded processors do not use virtual memory for a variety of reasons such as meeting real-time constraints. Those exploits can thus be applied straightforwardly. On the other hand, the aforementioned exploits such as *pointer conversion* and *disclosing kernel* are still effective when a secure processor uses virtual memory address translation. First, an adversary may try to disable virtual address translation if possible. Second, an adversary may try to tamper the address translation table to fool the system and avoid translation fault. Third, many processors throw exception and log the faulty address. For example, the MS Windows throws a window that displays the invalid address to the user. If this is the case, an adversary can recover the sensitive data easily by reading the log or the displayed address. If all the above do not work, an adversary can use the following two techniques.

3.3.1 Shift window and page address mask

For many processors, the page size is at least 4KB. This means that the lower 12 bits of a 32-bit address will be unaffected by address translation. An adversary can use a shift window to recover 12 bits at a time. Figure 3 illustrates this technique. For the higher-order bits, the adversary can mask them out to make sure that the result can always be translated. Given that a piece of sensitive data is first loaded into a register, the adversary can mask out or transform the page address before the data is used as fetch address. For instance, assume that there is a 32-bit secret, 0xdeadbeef, stored at byte location p, and a 24-bit data, 0x000102, stored at byte location p-3. Further assume that the data are encrypted using counter mode and the corresponding ciphertext for the 32-bit secret is 0xa07613ec and the corresponding ciphertext for the 24-bit data is 0xbc6911. If an adversary can modify encrypted instructions by tampering invariant code sequences such as program or function prologue or epilogue, they can use the disclosing kernel in Figure 4 to disclose the 32-bits secret through 4 load instructions. Note that in an out-of-order processor using authen-then-commit, the example disclosing kernel could be speculatively executed and all the loads may be carried out even though the disclosing kernel itself fails integrity verification.

3.3.2 Brute force or random page address tampering

If all the previously described techniques fail, an adversary can resort to brute-force or random page address tampering. For example, in the *pointer conversion* exploit, an adversary may either randomly or systematically flip ciphertext bits that map to a page address. Assume that the page address size is 20 bits and the application has 100MB memory space mapped with virtual address translation. Using brute force or random page address tampering, the adversary on average has a chance of one out of about 40 (2^{20}/(100MB/4KB)) trials to have some random data correctly translated. Considering the disproportional distribution of frequent or predictable values, an adversary can speed up the process by exploiting frequent or predictable values.

4. AUTHENTICATION ARCHITECTURE

It is not the objective of this paper to discuss how to design or choose a MAC standard for integrity protection. This issue has been addressed in other recent works [22, 23]. Rather, the main focus of this paper is to investigate different schemes of integrating the integrity verification logic into an out-of-order processor pipeline. In this section, we will provide detailed information on how to tie the authentication results with instruction execution and discuss the security and performance implications for each method in the context of memory fetch side-channel exploits. We treat the specific MAC verification logic as a black box which returns a binary verification result for each piece of fetched data.

4.1 Authentication Queue

Figure 5 illustrates a block diagram of an out-of-order processor pipeline augmented with integrity verification and cryptographic functionality. For every block of code or data fetched from the memory, the secure processor decrypts the information and verifies its integrity. We assume that authentication in general is falling behind decryption. For each fetched block of data or code, a secure processor sends a request to a component, called authentication queue. An integrity verification unit reads request from the queue, authenticates data in the requesting order and returns the authentication result. Since each request is associated with a queue entry, the entry index provides a way for identifying authentication requests. There is a register called *LastRequest Register* that always points to the index of the most recent authentication request.

4.2 Authentication Architecture

In this subsection, we will explore four design alternatives that integrate authentication results with the instruction execution.

4.2.1 Authen-then-issue

Authen-then-issue is a conservative approach. According to *authen-then-issue*, the fetched instructions can be issued

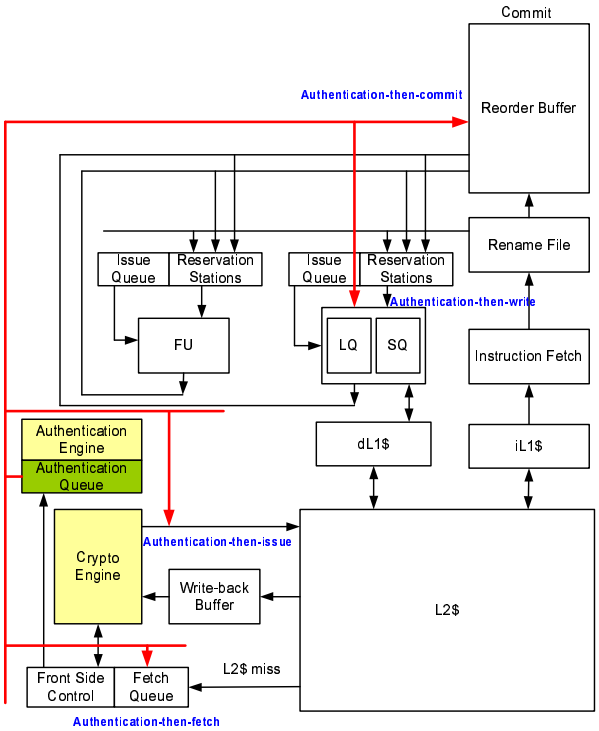


Figure 5: Secure Processor Block Diagram

and the fetched data can be used as operands only after integrity verification is completed. This approach is simple to implement and can prevent all the runtime side-channel exploits as described earlier. However, it gains security at a significant cost on performance because in this case integrity verification is on the critical path.

4.2.2 Authen-then-write

Authen-then-write is the most optimistic (or the least restricted) of the four. In which, integrity of memory state is guaranteed if for every piece of data written to the memory, the secure processor is certain that the data is generated based on the authenticated code and data. Detailed implementation of this approach may vary.

In one implementation, when a STORE instruction is ready for issuing (all its operands are fetched), the secure processor will read the *LastRequest Register* and associate the index value as a tag of the STORE instruction, called "authentication tag". The store value that should be written back to memory will remain in the store queue until it receives a broadcast result that indicates that the authentication request referenced by the store's authentication tag has been successfully verified. This ensures that the secure processor only updates cache and memory with values produced by the authenticated code and data. Note that authentication engine sends verification result in the natural request order. Upon the receipt of a matching authentication tag, the secure processor can be certain that all the codes and data before the waiting store are also authenticated.

Under *authen-then-write*, the secure processor guarantees that at any moment, the information stored in the memory can be trusted in the sense that they are produced based on authenticated code and data. However, this approach does not prevent any of the aforementioned active exploits of memory fetch side-channels. It does not guarantee privacy of software and data from being disclosed through the side-channel exploits.

4.2.3 Authen-then-commit

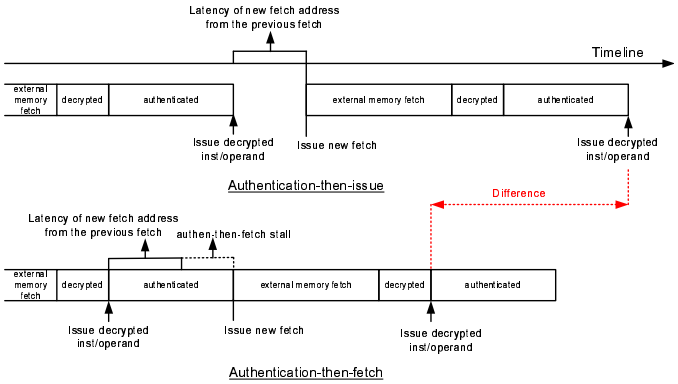


Figure 6: Timeline of Authen-then-fetch vs. Authen-then-issue

In modern out-of-order processors, instructions wait in a the Re-Order Buffer for handling precise exception and mis-speculation before they are committed. According to *authen-then-commit*, a secure processor will not commit an instruction until both the instruction itself and its operands are authenticated. In some aspects, *authen-then-commit* is similar to *authen-then-write*. For example, *Authen-then-commit* also guarantees that the secure processor only updates memory states based on authenticated code and data. However, there are several fundamental differences between these two approaches. First, *authen-then-write* deals with only memory state, sometimes only the external memory state while *authen-then-commit* handles both the memory state and the processor state. Second, *Authen-then-commit* provides precise exception for authentication exceptions while *authen-then-write* does not. Though seemingly a very sound design, *authen-then-commit* does not prevent any of the aforementioned memory fetch side-channel exploits either because it allows unverified memory fetches to be speculatively issued.

4.2.4 Authen-then-fetch

According to *authen-then-fetch*, a secure processor must not grant bus cycles to an external memory fetch until certain authentication conditions are met. Those include data fetches issued by fetch instructions and instruction fetches triggered by program execution. For data fetch, *authen-then-fetch* requires three conditions to be satisfied before the data fetch can be issued to the bus: 1) the fetch instruction itself be authenticated; 2) the fetch address be authenticated; 3) all the data or previously executed instructions for which the data fetch has data or control dependency be authenticated. Those include data and instructions for computing the fetch address. For instruction fetch triggered by a control transfer instruction, *authen-then-fetch* requires that, 1) the control transfer instruction be authenticated; 2) all the data or previously executed instructions that the control transfer has data or control dependency with be authenticated. To reach a particular data fetch or instruction fetch point, there is an execution path, also called program slice, that includes all the previous dependent instructions. In other words, according to *authen-then-fetch*, a secure processor is allowed to issue fetch address on the front side bus only after all the instructions and data involved in the path or program slice have been authenticated.

A precise implementation of this approach requires dynamic tracking of control and data dependency that may be too complex and expensive. Fortunately, there are many alternate implementations that sufficiently satisfy all the requirements of *authen-then-fetch* without resorting to depen-

0-7695-2732-9/06 $20.00

dency tracking. In one variation, memory fetch is not issued until the secure processor drains the authentication queue which we call *drain-authen-then-fetch*. For a new memory fetch, the secure processor stops sending more authentication requests to the authentication queue, waits for the current authentication to be drained, issues the memory fetch afterward, and then resumes sending more authentication requests. Alternatively, a secure processor may associate the current value of the "LastRequest Register" with each issued instruction. If the instruction triggers a memory fetch, the fetch is stalled until the authentication queue has completed authentication of the associated request. Figure 6 uses an example to illustrate the difference between *authen-then-issue* and *authen-then-fetch*. The example shows two external memory fetches where the second fetch is based on the first fetch. There is an assumed fixed latency between the time when the data/instruction based on the first fetch result is issued to the pipeline and the time when the second fetch address is ready. The dotted line highlights the time difference between the two schemes and shows the latency advantage of the *authen-then-fetch* over the *authen-then-issue*. Note that under the *authen-then-fetch* policy, a new external memory fetch only stalls on already issued, recently decrypted instructions or operands. Instructions or data that are decrypted after the new memory fetch is created or outstanding external memory fetches will have no latency impact on this new memory fetch.

4.3 Fetch Address Obfuscation

Fetch address obfuscation [4, 6, 29, 30], in general, includes techniques that eliminate the fetch address side-channel or obscure the disclosed fetch addresses. Perfect address obfuscation is hard to achieve because of the high cost associated with address obfuscation [6]. In [6], an approximate address obfuscation solution was described with a very high memory bandwidth requirement [4]. Since an ideal address obfuscation scheme is hard to establish, solutions providing only limited address obfuscation are recently proposed [4, 29].

Address obfuscation is related to the authentication architecture because it tries to eliminate the memory fetch side-channel. Nonetheless, address obfuscation will not replace authentication architecture because integrity verification is indispensable for a tamper-evident, tamper-proof design. For instance, address obfuscation alone will not prevent a disclosing kernel from leaking sensitive data to an I/O port. This suggests that it be used together with *authen-then-commit* to offer sufficient security protection. The availability of address obfuscation will certainly improve a secure processor's strength against certain malicious tampering on privacy but it should be kept in mind that address obfuscation is not a panacea for all the exploits, neither is it absolutely needed for preventing active exploits/tampering based on memory fetch side-channel.

In addition to authentication architecture, address obfuscation such as the one in [29] requires some extra significant memory or area overhead such as memory randomization or memory re-shuffle cache. Address obfuscation based on memory randomization or re-shuffle also has significant impact on how an OS loads software or swaps memory pages since it will very likely destroy memory operation locality and coherence. In certain cases, address obfuscation may increase the number of page faults by several factors [4].

Table 2 compares different authentication architectures in several aspects. It is recommended that *authen-then-fetch* be used together with *authen-then-commit* to guarantee authenticated memory/processor state and to support precise exceptions on security and authentication related faults.

5. PERFORMANCE ANALYSIS

Parameters	Values
Frequency	1.0 GHz
Fetch/Decode width	8
Issue/Commit width	8
L1 I-Cache	DM, 16KB, 32B line
L1 D-Cache	DM, 16KB, 32B line
L2 Cache	4way, Unified, 64B line, write back cache, 256KB and 1MB
L1 Latency	1 cycle
L2 Latency	4 cycles(256KB), 8 cycles(1MB)
I-TLB	4-way, 128 entries
D-TLB	4-way, 128 entries
RUU	128, 64 entries
Memory Bus	200MHz, 8B wide
Memory Latency	X-5-5-5 (core clocks) X depends on page status
CAS latency	20 mem bus clocks
Precharge latency (RP)	7 mem bus clocks
RAS-to-CAS (RCD) latency	7 mem bus clocks
Decryption latency	80ns

Table 3: Processor model parameters

5.1 Simulation Framework

Our simulation work is based on SimpleScalar running Alpha binaries compiled with -O3 option. Each benchmark is fast-forwarded according to SimPoint [17] and then simulated for 400 million instructions in performance mode. During fast-forwarding, L1 and L2 caches were warmed up. Table 3 summaries the processor parameters. We integrated a more accurate DRAM model [7] to improve the system memory modeling, in which bank conflicts, page miss, row miss, page miss are all modeled following the PC SDRAM specification. Eighteen SPEC2000 INT and FP benchmarks with high L2 misses and memory throughput requirements were used for evaluation.

5.2 Implementation

The latency of decryption or integrity verification varies substantially depending on many factors such as encryption mode, cipher, authentication scheme, process technology, architecture design, etc. To best justify our performance conclusions, we use reference implementations. In simulation study, we conduct sensitivity study to capture different variations and design scenarios.

5.2.1 Cipher

The Rijndael cipher can process data blocks of 128, 192, or 256 bits by using key lengths of 128, 196 and 256 bits. It is based on a round function, which is iterated 10 times for a 128-bit length key, 12 times for a 192-bit key, and 14 times for a 256-bit key. For high throughput and high speed hardware implementation, Rijndael is often unrolled and with each round pipelined into multiple pipeline stages (4-7) to achieve high decryption/encryption throughput [14, 9, 8]. The minimal total area of unrolled and pipelined Rijndael is about 100K - 150K gates to achieve 15-20Gbit/sec throughput [9]. Based on a synthesized Verilog implementation, each decryption round of pipelined AES-Rijndael takes around 5-6nsec using 0.18μm standard cell library. In this study, unless specified, the default reference latency is 80ns for 256-bit Rijndael.

5.2.2 Encryption Mode

Encryption mode has a major impact on secure processor performance. A few recent published works provided detailed evaluation and comparison of ECB, CBC, and counter mode [19, 22, 27]. In all the studies, counter mode delivers the best performance because it allows pre-computation of decryption pads in parallel with memory fetch. Our reference implementation of encryption mode is based on [19].

Table 2: Characteristics Comparison of Different Schemes

	prevent active fetch address side-channel disclose	precise exception (authentication exceptions)	authenticated memory state	authenticated processor state
Authen-then-issue	√	√	√	√
Authen-then-write			√	
Authen-then-commit		√	√	√
Authen-then-fetch plus commit	√	√	√	√
Address obfuscation plus commit	√	√	√	√

5.2.3 Integrity Verification

MAC-based integrity verification is a standard operation. In the reference implementation, we use the standard HMAC [10] for protecting integrity of data blocks stored in the external RAM. The default MAC size is 64 bits (truncated HMAC MAC). The reference HMAC uses standard SHA-256 algorithm [16]. The simulation study is based on a Verilog implementation of SHA-256, synthesized using Synopsys. The latency for this design is 74ns for 512 bits of padded input (padding with the required padding in SHA-256).

In addition to per-data block based integrity verification, a secure processor sometimes also applies a hash tree or MAC tree [13, 22] for preserving the overall memory space integrity and preventing replay attacks of data blocks. This causes substantially additional amount of latency overhead. The CHTree scheme in [22] constructs an m-ary hash tree where m is the number of child nodes per parent node has and is equal to the size of the cache line divided by the size of hash values.

5.2.4 Address Obfuscation

The implementation of address obfuscation is based on a revised model of [29]. Each time, a cache line is written to the external memory, its memory location will get re-mapped or reshuffled. When a cache line is fetched from the external memory, the secure processor will look up a re-map cache to retrieve the current re-mapped location of the memory cache line. To prevent adversaries from reconstructing re-mapped locations, re-map data are encrypted when they are stored in the external memory device or evicted from the on-chip re-map cache. Note that both instructions and data blocks are re-mapped.

5.3 Performance Analysis

In this section we summarize performance results. The results were collected on two L2 cache sizes, 256KB and 1MB.

5.3.1 Performance of Authentication Architectures

Figure 7(a) and Figure 7(b) show the normalized IPC of six schemes of using the authentication results under 256KB L2 for both the SPEC2000 integer and floating benchmark programs. The IPC is normalized against the IPC of a baseline situation of decryption only without integrity verification. The results suggest that *authen-then-issue* and *authen-then-commit + address obfuscation* have the worst performance where the average IPC for *authen-then-issue* is about 87% of the baseline IPC and the average IPC of *authen-then-commit + address obfuscation* is about 86% against the baseline of no address obfuscation. For some benchmark programs such as ammp, bzips, mgrid, twolf, and vpr, their IPCs under *authen-then-issue* are below 80% of the their respective baselines'. In contrast, *authen-then-write* shows the best performance. On average, IPC under *authen-then-write* is more than 98% of the baseline IPC, which means less than 2% performance penalty due to integrity verification. The next one is *authen-then-commit*, its average IPC is more than 96% of the baseline IPC. For most benchmark programs, IPC under this scheme is over 90% of the baseline IPC except mgrid whose normalized IPC is 86%. Under

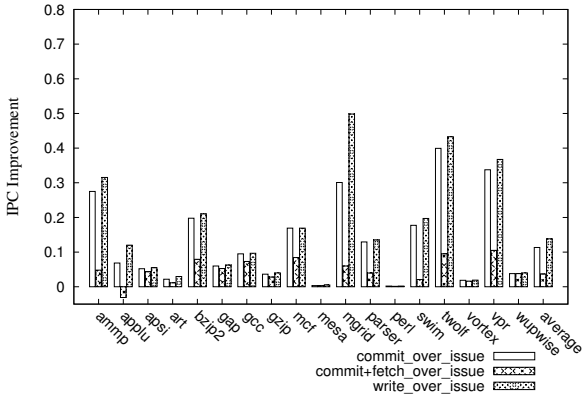


Figure 8: Comparison of IPC Speedup Over Authen-then-issue, 256KB L2

authen-then-fetch, the average IPC is 92% of the baseline IPC. Combination of *authen-then-commit* and *authen-then-fetch* yields average IPC performance of 90% of the baseline.

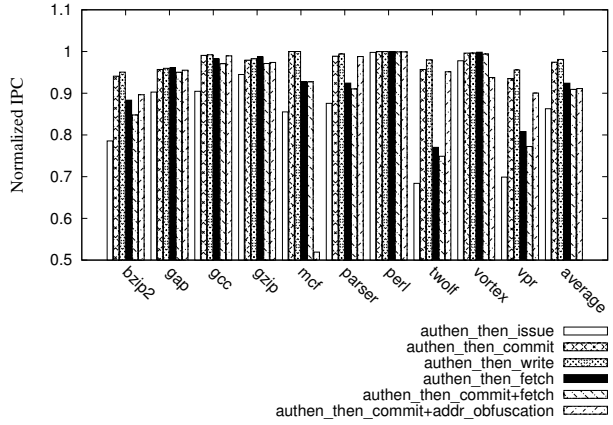
Figure 7(c) and Figure 7(d) show the normalized IPC performance with a 1MB L2 cache. Since there are fewer memory accesses when the L2 size quadruples, the performance impact of different schemes is less than the scenario of 256KB L2. However, the performance ranking of the six schemes is almost the same where the *authen-then-issue* and the *authen-then-commit + address obfuscation* policies have the lowest performance and the *authen-then-write* has the highest performance.

As discussed earlier, *authen-then-issue* and *authen-then-commit + address obfuscation* are relatively more secure. Figure 8 compares *authen-then-commit*, *authen-then-write*, and *authen-then-commit plus authen-then-fetch* with *authen-then-issue* by showing the IPC speedup of the three schemes over IPC of *authen-then-issue*. The results indicate that on average, *authen-then-commit* improves IPC by 12%. For four benchmark programs, the improvement is over 20% and three of them show more than 30% improvement. Six benchmark programs show an improvement from 10% to 20%. For *authen-then-write*, the performance improvement on average is about 14%. However, as discussed before, both *authen-then-commit* and *authen-then-write* are less secure than *authen-then-issue*. In contrast, combination of *authen-then-commit* and *authen-then-fetch* provides much better security and it does not suffer from the previously described exploits just like *authen-then-issue*. For five benchmarks, *authen-then-commit plus authen-then-fetch* provides about 10% performance improvement over *authen-then-issue*.

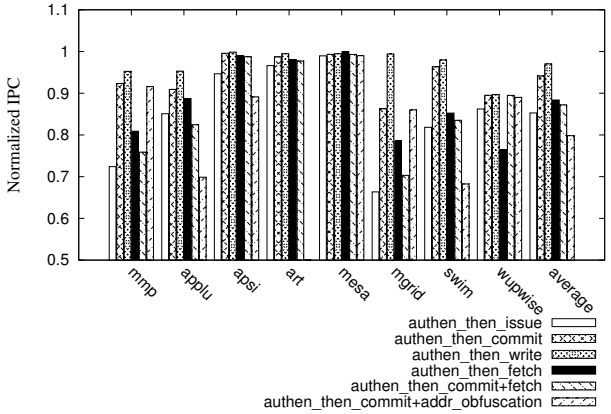
Figure 9 shows that the impact of re-map cache size when address obfuscation is applied together with *authen-then-commit*. As expected, IPC improves with the size of re-map cache.

5.3.2 RUU Size

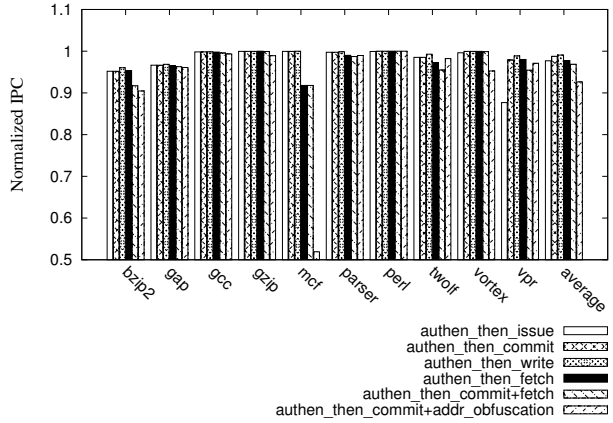
The size of Register Update Unit (RUU) may have some impact on performance of the studied schemes. To under-



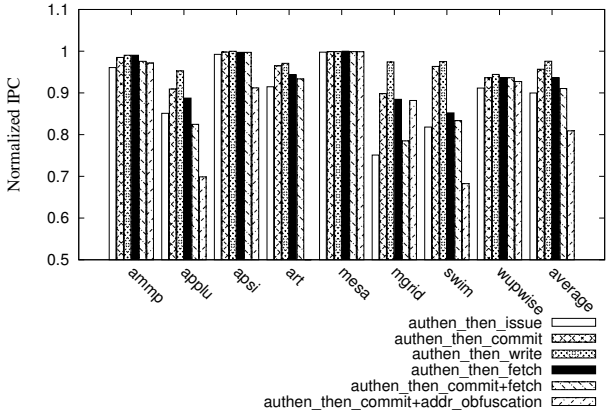
(a) SPEC2000 INT (256KB L2)



(b) SPEC2000 FP (256KB L2)



(c) SPEC2000 INT (1MB L2)



(d) SPEC2000 FP (1MB L2)

Figure 7: Normalized IPC Under Different Authentication Schemes, 256KB Re-map Cache for Address Obfuscation (Baseline: decryption only with no authentication)

stand its sensitivity, we reduce the number of the RUU entries by half. Figure 10 shows the results. The results indicate the same performance pattern. The performance rank of four schemes from the lowest to the highest are, *authen-then-issue*, *authen-then-commit plus authen-then-fetch*, *authen-then-commit*, and *authen-then-write*. Figure 11 shows the IPC improvements of the *authen-then-commit* and the *authen-then-commit plus authen-then-fetch* over the *authen-then-issue*. The *authen-then-commit plus authen-then-fetch* policy shows a performance improvement about 10% for 5 benchmark programs while the *authen-then-commit* scheme improves IPC in the range from 10% to 50% for 10 benchmark programs.

5.3.3 Impact of Hash Tree Authentication

Hash or MAC tree can prevent replay attacks. One side-effect of using hash or MAC tree such as the CHTree approach is the additional latency overhead for integrity verification. We evaluate five authentication schemes under a hash tree authentication using the implementation described in [22]. Our implementation performs the verification of the internal hash tree nodes concurrently when it is allowed. Authenticated and verified tree nodes are cached inside the processor using a dedicated 8KB hash tree cache. Figure 12 shows the normalized IPC of *authen-then-issue*,*authen-then-write*, *authen-then-commit*, *authen-then-fetch* and *authen-then-commit plus authen-then-fetch*. Again, the average performance results indicate similar ranking of performance among the five schemes with *authen-then-issue* being the slowest scheme and *authen-then-write* being the fastest scheme. However, the performance difference among *authen-then-write*, *authen-then-commit* and *authen-then-fetch* become very small due to significantly increased authentication latency. For almost every benchmark, the normalized IPC decreases because of the potential lengthy processing of hash tree nodes. Figure 13 shows the performance improvements of the *authen-then-commit* and the *authen-then-commit plus authen-then-fetch* over the *authen-then-issue* policy. For *authen-then-commit*, 7 benchmark programs show their performance improved from 10% to 35%. For the *authen-then-commit plus authen-then-fetch*, 5 benchmark programs have their

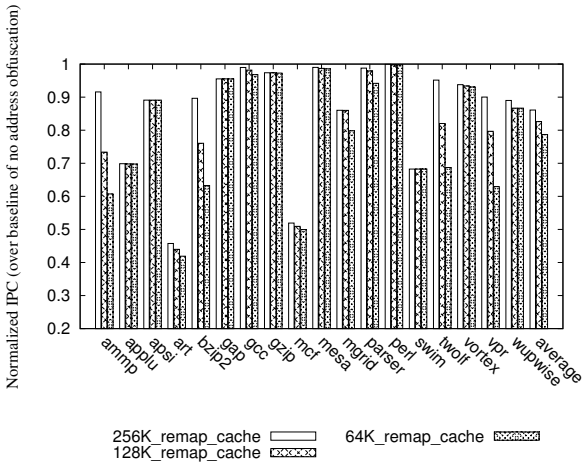


Figure 9: Normalized IPC for Three Settings of Address Obfuscation Re-map Cache Size, 256KB L2

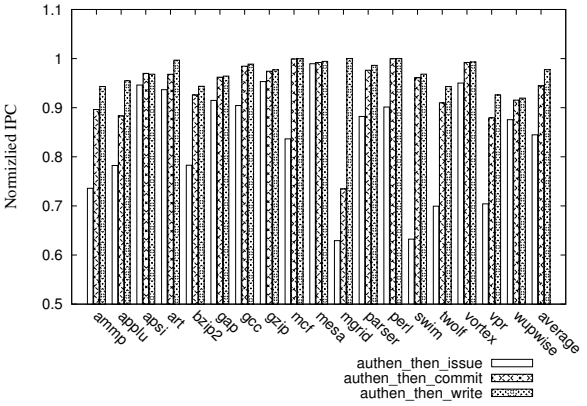


Figure 10: Normalized IPC with 64-entry RUU (256KB L2)

performance improved by more than 10%.

6. RELATED WORK

To prevent adversaries from compromising trusted applications is becoming a major challenge not only to software developers but also to processor architects and system designers. To combat malicious exploits, alliance such as Trusted Computing Group (TCG) [1] was formed across a variety of industry segments to tackle the information security issues. In mission critical and military embedded applications, requirements for protecting software privacy and integrity are even more strenuous where reverse engineering and hardware tamper pose inherent threats.

XOM [12] pioneered the design of a security processor architecture to protect trusted software from physical tamper. The security of the XOM architecture is achieved by a tamper-resistant processor design. Suh et al. improved XOM's approach with the AEGIS security processor architecture design [22] in which both privacy and integrity of applications are protected. They also described the implementation of a secure environment with a block cipher encryption and a CHTree integrity checking scheme [5]. Later, Lu et al. proposed an improved integrity checking scheme for achieving better performance in [13].

In [23], the performance of confidentiality protection was improved by replacing the block cipher encryption with a counter mode class encryption. Yang et al. proposed a similar scheme in [27] for performance improvement for protecting privacy. In addition, Shi et al. exploits the tem-

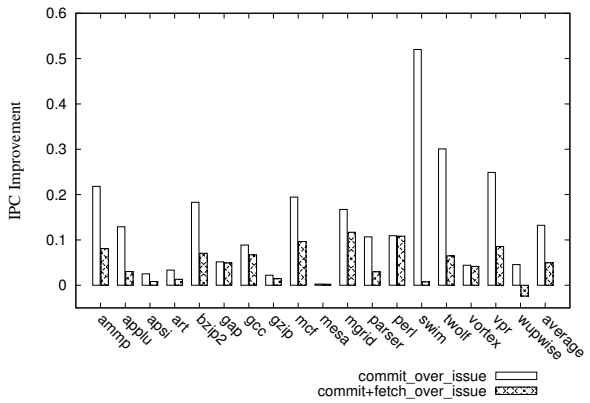


Figure 11: Comparison of IPC Speedup Over Authen-then-issue Under Different Authentication Schemes (256KB L2, 64-Entry RUU)

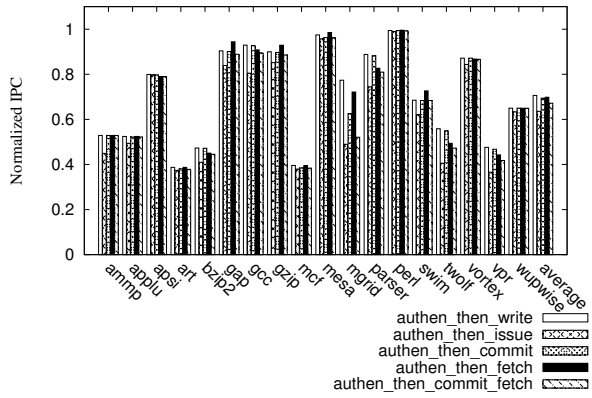


Figure 12: Normalized IPC Under Different Authentication Schemes, Memory Authentication Tree (Baseline: decryption only with no authentication)

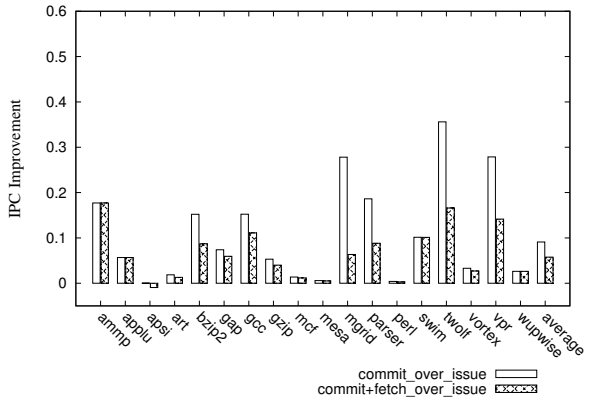


Figure 13: Comparison of IPC Improvement Over Authen-then-issue, Memory Authentication Tree

0-7695-2732-9/06 $20.00

poral and spacial localities exhibited by software execution and devised prediction techniques [19] to improve performance for counter mode encryption. More recently, Yan et al. in [25] evaluated and compared from performance aspect several types of integrity checking schemes, particularly, the *lazy authentication* and the *authentication then commit*. The *lazy authentication* scheme as defined in [20, 25] is a weak integrity verification scheme. It verifies integrity of code or data as a group over a large, vulnerable time window (weaker than any approach described in this paper). Performance comparison between *lazy authentication* and *authentication then commit* was presented in [25]. The issues of the *lazy authentication* was addressed in [20] from the security aspect. The risks of losing privacy through the memory fetch side-channel in security processor design are also discussed in [20].

7. CONCLUSIONS

In this paper, we explored the design spectrum and analyzed their implications on performance and security when integrating integrity verification and decryption into an out-or-order processor pipeline for providing a secure computing environment. We provided an in-depth analysis of the risks associated with memory fetch side-channel in the context of a secure processor design. We first classify the types of authentication integration. Then, based on both security analysis and performance evaluation, we show that the *authen-then-issue* policy and the *authen-then-commit + address obfuscation* policy provide better protection against the side-channel exploits but has the worst performance overhead. *Authen-then-write* guarantees the integrity of processing results stored in an un-trusted external memory. Furthermore, *authen-then-commit* ensures the integrity of both memory and processor states at any moment and supports precise interrupt for security exceptions. But neither approach can completely prevent violation of software and data confidentiality through runtime exploits of memory fetch side-channel. In conclusion, the analysis and the results of this paper provides valuable risk assessment and the design trade-off information for guiding the design of a tamper-proof secure processor.

8. ACKNOWLEDGMENT

This research was supported by NSF Grants CCF-0326396 and CNS-0325536 and a DOE Early CAREER PI Award.

9. REFERENCES

[1] Trusted Computing Group. http://www.trustedcomputinggroup.org/.

[2] John Black and Phillip Rogaway. CBC MACs for Arbitrary-Length Messages: The Three-Key Constructions. In *J. Cryptol.*, volume 18, pages 111–131, 2005.

[3] Danny Dolev, Cynthia Dwork, and Moni Naor. Non-malleable Cryptography. In *Proceedings of the 23rd ACM Symposium on Theory of Computing*, 1991.

[4] Lan Gao, Jun Yang, Marek Crobak, Youtao Zhang, San Nguyen, and Hsien-Hsin S. Lee. A Low-cost Memory Remapping Scheme for Address Bus Protection. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniuqes*, 2006.

[5] 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 Int'l Symp. on High Performance Computer Architecture*, 2003.

[6] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

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

[8] Alireza Hodjat and Ingrid Verbauwhede. Minimum Area Cost For a 30 to 70 Gbits/s AES Processor. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI '04), pp. 498-502.*

[9] Alireza Hodjat and Ingrid Verbauwhede. Speed-Area Trade-off for 10 to 100 Gbits/s. In *37th Asilomar Conference on Signals, Systems, and Computer*, 2003.

[10] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. United States, 1997. RFC Editor.

[11] Abhishek Kumar. Discovering Passwords In the Memory. *http://www.infosecwriters.com/text_resources/.*

[12] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support For Copy and Tamper Resistant Software. In *Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems*, 2000.

[13] Chenghuai Lu, Tao Zhang, Weidong Shi, and Hsien-Hsin S. Lee. M-TREE: A High Efficiency Security Architecure for Protectign Integrity and Privacy of Software. *Journal of Parallel and Distributed Computing for a special issue on Security in Grid and Distributed Systems*, (66):1116–1128, 2006.

[14] M. McLoone and J. V. McCanny. High performance single-chip fpga rijndael algorithm implementations. In *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, pages 65–76. Springer-Verlag, 2001.

[15] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., 1996.

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

[17] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[18] Weidong Shi, Hsien-Hsin S. Lee, Mrinmoy Ghosh, and Chenghuai Lu. Architectural Support for High Speed Protection of Memory Integrity and Confidentiality in Multiprocessor Systems. In *IEEE Parallel Architecture and Compilation Techniques*, 2004.

[19] Weidong Shi, Hsien-Hsin S. Lee, Mrinmoy Ghosh, Chenghuai Lu, and Alexandra Boldyreva. High Efficiency Counter Mode Security Architecture via Prediction and Precomputation. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 14–24, 2005.

[20] Weidong Shi, Hsien-Hsin S. Lee, Chenghuai Lu, and Mrinmoy Ghosh. Towards the Issues in Architectural Support For Protection of Software Execution. *SIGARCH Comp. Arch. News*, 33(1):6–15, 2005.

[21] Weidong Shi, Chenghuai Lu, and Hsien-Hsin S. Lee. Memory-centric Security Architecture. In *2005 International Conference on High Performance Embedded Architectures and Compilers*, 2005.

[22] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *17th Annual ACM International Conference on Supercomputing*, 2003.

[23] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srini Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *36th Annual International Symposium on Microarchitecture*, 2003.

[24] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security, Dec 2004.

[25] Chenyu Yan, Brian Rogers, Daniel Englender, Yan Solihin, and Milos Prvulovic. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In *Proc. of the International Symposium on Computer Architecture*, 2006.

[26] Jun Yang and Rajiv Gupta. Frequent Value Locality and Its Applications. In *ACM Transsactions on Embedded Computing Systems*, volume 1, pages 79–105, 2002.

[27] Jun Yang, Youtao Zhang, and Lan Gao. Fast Secure Processor for Inhibiting Software Piracy and Tampering. In *Proc. of the Int'l Symp. on Microarchitecture*, 2003.

[28] Xiangyu Zhang and Rajiv Gupta. Hiding Program Slices for Software Security. In *Proceedings of the 2003 Internal Conference on Code Genration and Optimization*, pages 325–336, 2003.

[29] Xiaotong Zhuang, Tao Zhang, Hsien-Hsin S. Lee, and Santosh Pande. Hardware Assisted Control Flow Obfuscation for Embedded Processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 292–302, 2004.

[30] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. HIDE: an Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2004.