Covert and Side Channels due to Processor Architecture*

Zhenghong Wang and Ruby B. Lee
Department of Electrical Engineering, Princeton University
{zhenghon,rblee}@princeton.edu

Abstract

Information leakage through covert channels and side channels is becoming a serious problem, especially when these are enhanced by modern processor architecture features. We show how processor architecture features such as simultaneous multithreading, control speculation and shared caches can inadvertently accelerate such covert channels or enable new covert channels and side channels. We first illustrate the reality and severity of this problem by describing concrete attacks. We identify two new covert channels. We show orders of magnitude increases in covert channel capacities. We then present two solutions, Selective Partitioning and the novel Random Permutation Cache (RPCache). The RPCache can thwart most cache-based software side channel attacks, with minimal hardware costs and negligible performance impact.

1. Introduction

Covert channels and side channels are two types of information leakage channels. A covert channel uses mechanisms that are not intended for communications, e.g., writing and checking if a file is locked to convey a "1" or "0". In a covert channel [1], an insider process leaks information to an outsider process not normally allowed to access that information. The insider (sending) process could be a Trojan horse program previously inserted stealthily into the computer. An outsider (receiving) process need only be an unprivileged process.

In a physical side channel attack, unconventional techniques are used to deduce secret information. Typically, the device has been stolen or captured by the adversary who then has physical access to it for launching a physical side-channel attack. Traditional side channel attacks involved differential power analysis [2-5] and timing analysis [6-10]. Different amounts of power (or time) used by the device in performing an encryption can be measured and analyzed to deduce some or all of the key bits. The number of trials needed in a power or timing side channel attack could be much less than that needed in mathematical cryptanalysis.

In this paper, we consider software side channel attacks. In these attacks, a victim process inadvertently assumes the role of the sending process, and a listening (attacker) process assumes the role of the receiving process. If the victim process is performing an encryption using a secret key, a software side channel attack allows the listening process to get information that leads to partial or full recovery of the key. The main contributions of this paper are:

- Identification of two new covert channels due to processor architecture features, like simultaneous multi-threading (SMT) and speculation.
- Showing that covert channel capacities have increased by orders of magnitude.
- Analysis of cache-based side channel attacks.
- Insufficiency of software isolation approaches for mitigating information leakage through processor-based covert and side channels.
- Selective partitioning solution for SMT-based covert channels.
- Novel Random Permutation Cache (RPCache) solution that can thwart cache-based software side channel attacks.

Section 2 describes the threat model. Section 3 illustrates the problem with real attacks and analysis of newly identified cache side channels. Section 4 shows the insufficiency of software solutions, motivating the need for hardware solutions to a hardware-induced problem. Section 5 provides our Selective Partitioning solution. Section 6 presents our novel Random Permutation Cache solution, and experimental results on its performance and security. Section 7 reviews related work and section 8 presents our conclusions.

* This work was supported in part by DARPA and NSF Cybertrust 0430487, and NSF ITR 0326372.

2. Threat model

The threat model is that of an adversary whose goal is to learn information that he has no legitimate access to. Within the computer system, an adversary is one or more unprivileged user processes.

Since our focus in this paper is on the impact of processor architecture features on the problem, we assume that the critical modules of the software system (like the OS kernel and the modules enforcing security policies) are free of software vulnerabilities. Other software modules, such as the guest OS in a Virtual Machine or the application software, may have security flaws that allow a cooperating process of the adversary, e.g. an insider, to gain access to the information. In this case, we assume that appropriate security policies are enforced so that the cooperating process is isolated from the adversary. In this paper we consider software attacks and do not consider physical attacks like bus probing and power analysis.

3. Processor covert and side channels

3.1. New SMT/FU covert channel

Simultaneous Multi-Threaded (SMT) processors [15] run many processes concurrently, sharing almost all processor resources in a very tightly coupled way. In particular, the concurrent threads share a pool of functional units (FUs) that are allocated dynamically to each process every cycle. By contending for the shared FUs, one process can interfere with another, leading to covert channels. Though in principle this is a typical covert *timing* channel [13], it is orders of magnitude faster than traditional covert channels.

insider	observer

`int bit;`
`…`
`do {`
`  bit = get_bit();`
`  if ( bit == 1 )`
`  MULTIPLY();`
`  else`
`  NULL(); }`
`while ( !TX_end() );`
	`int time, dt;`
	`…`
	`time = 0;`
	`do {`
	`  dt = time;`
	`  RUN();`
	`  time = get_time();`
	`  STORE(time-dt); }`
	`while ( !RX_end() );`

Figure 1. Pseudocode for SMT/FU channel

Consider a system which contains two processes that are not allowed to communicate at all with each other. The insider (sender) process can modulate the use of functional units, e.g. the multipliers, to send information to the receiver process. Figure 1 shows the pseudocode for both processes. To send a '1' bit, the insider calls MULTIPLY() to execute a fixed number of instructions which try to use up all the integer multipliers. It calls NULL(), which executes several hundred NOP instructions, to send a '0' bit. The observer senses the modulated signal by comparing its progression with a timer T. By calling RUN(), he executes integer multiply instructions at a constant rate. When a '1' is sent, most multipliers are used by the insider and the observer can detect this because his execution will be slowed down.

We implemented this channel on a Pentium-4 processor with hyper-threading (Intel's SMT), which supports only two simultaneous threads [16]. Figure 2 shows an example of the received waveform. The bit string shown is transmitting "10101010…"

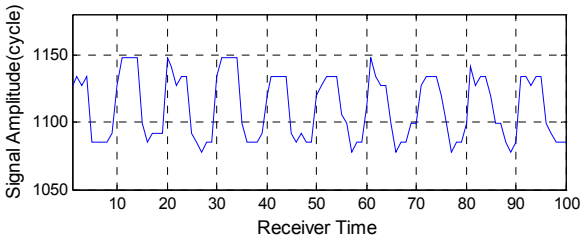


Figure 2. Observed signal waveform

3.2. SMT/cache side channel

In an SMT processor, caches are also shared. An attacker can run a receiver (or observer) process simultaneously with the victim process on an SMT processor. This enables observation of the victim process's cache usage at run time. In [12], Percival demonstrated an attack on RSA using this approach. The attack is simple: the attacker accesses an array of his own data repeatedly so that he occupies all cache lines. During the execution of the victim process, i.e. the RSA encryption process in this case, if the encrypting process accesses a cache line, the attacker's data will be evicted. The next time the attacker accesses his data corresponding to this cache line, he will experience a cache miss. By measuring his memory access time, the attacker can detect such cache misses. The attacker therefore can learn the victim process's cache access pattern, based on which he can determine when multiplication and squaring operations used in RSA encryption occur in the victim process. He can also learn which table entry is accessed during a key-dependent table lookup in RSA. The attacker then can recover the RSA key of the victim process, based on the observed cache usage information.

In [22] Osvik et al. applied this approach to AES and demonstrated how easy it is to recover the key. They showed that after just 800 writes to a Linux dm-crypt encrypted partition, the full AES key can be recovered.

3.3. Statistical cache side-channel

In non-SMT processors, cache-based software side channel attacks are also possible. Bernstein's attack on AES [11] illustrates such an attack. The victim is a software module that can perform AES encryption for a user. The module is a "black box" and the user is only able to choose the input to the AES software module and measure how long it takes to complete the encryption. He found that for most software AES implementations running on modern processors, the execution time of an encryption is input-dependent and can be exploited to recover the secret encryption key.

Attack Description: The attack consists of three steps.

1. *Learning phase*: Let the victim use a known key ***K***; the attacker generates a large number, N, of random plaintexts ***P***. He then sends the plaintexts to the cipher program (a remote server in [11]) and records the encryption time for each plaintext. He uses the algorithm shown in Figure 3 to obtain the timing characteristics for ***K*.**
2. *Attacking phase*: Repeat the same operation in the learning phase except that an unknown key ***K'*** is used. Note that the input set is randomly generated and not necessarily the same as used in step 1.
3. *Key recovery*: Given the two sets of timing characteristics, use the correlation algorithm shown in Figure 4 to recover the unknown key ***K'***. Function *findMax*() searches for the maximum value in the input array and returns its index.

In Figure 3, ***P*** denotes a plaintext block that will be encrypted with the secret key ***K***. p_i and k_i denote the i-th byte of ***P*** and ***K***, $i \in [0,15]$ (We assume 128-bit AES in this example; hence both the block to be encrypted, ***P***, and the key, ***K***, are 16 bytes long). A large number of random plaintext blocks ***P*** are encrypted with the same key ***K***. For each byte p_i in the plaintext blocks, find the encryptions where $p_i = 0$, and calculate $t_{avg}^i(0,\boldsymbol{K})$ which is the average of the execution times of the AES encryptions where the i^{th} plaintext byte is 0, using key ***K***. Repeat for $p_i = 1, 2, \ldots, 255$. This can be plotted as a timing characteristic chart for byte i using key ***K*** as shown in Figure 5(a), where i=0. (This is obtained in our experiments over a Pentium M processor with Cygwin/OpenSSL0.9.7a). The x-axis represents the value of the plaintext byte p_i, from 0 to 255, and the y-axis the average encryption time (minus a fixed offset). Sixteen such charts are plotted, one for each byte position of the plaintext blocks. These 16 charts together represent the timing characteristic, denoted $t_{avg}^i(j,\boldsymbol{K})$, for $0 \le i \le 15$ and $0 \le j \le 255$, of AES encryption on a given platform for a given ***K***.

```
For key K:
For s = 1 to N do begin
    Generate a random 128-bit Plaintext block, Ps;
    Ts = time taken for AES encryption of Ps using K;
end;
For i = 0 to 15 do begin
    For j = 0 to 255 do begin
        count = 0;
        For s = 1 to N do begin
            If pi = j then
                TSUMi(j) = TSUMi(j) + Ts;
                count = count+1;
            end;
            tavg^i(j,K) = TSUMi(j)/count;
    end;
end;
```

Figure 3. Timing characteristic generation

```
For i = 0 to 15 do begin
  For j = 0 to 255 do begin
    Corr[j] = Σ_{m=0}^{255} [ t^i_avg(m,K) • t^i_avg(m ⊕ j, K') ]
  end;
  ki' = findMax(Corr);
end;
```

Figure 4. Key-byte searching algorithm

Experiments show that the average execution time $t_{avg}^i(j,\boldsymbol{K})$ is pretty much fixed for a given system configuration. Furthermore, it is found that when a different key ***K'*** is used, the timing charts roughly remain the same except that the locations of the bars in the charts are permuted, as shown in Figure 5 (a) and (b). More specifically, the following equation holds:

$$t_{avg}^i(p_i, \boldsymbol{K}) = t_{avg}^i(p'_i, \boldsymbol{K'}) \text{ if } p'_i \oplus k'_i = p_i \oplus k_i \quad (1)$$

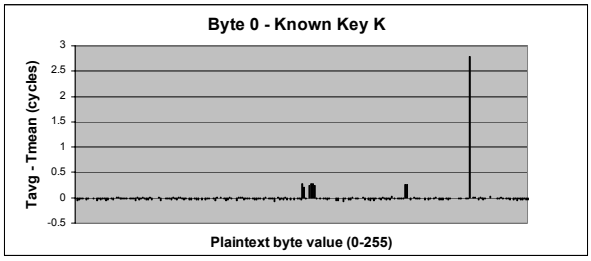
where $\oplus$ is the bit-wise XOR operation, and k_i and k'_i are the i-th byte of ***K*** and ***K'*** respectively.

Attack Analysis: We try to explain why (a) there are high bars corresponding to certain x-values in Figure 5, and (b) why these same peaks occur, but are permuted, when different keys are used.

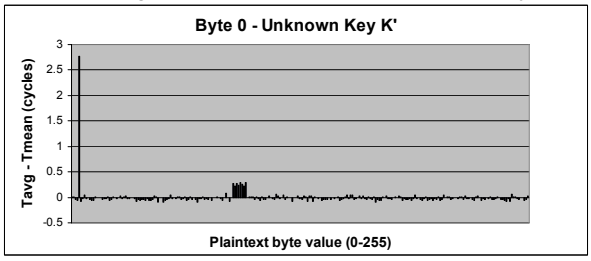
(a) Table lookups are intensively used in various AES implementations. During the AES encryption, the tables will gradually be loaded into the cache when table entries are actually used. If there are no other cache accesses in addition to these table lookups, eventually the tables should all be loaded into the cache and there will not be any cache misses, if the cache is large enough. In reality, however, there are wrapper and other background processes that cause cache accesses which evict some entries of the tables out of the cache. Moreover, some of these cache accesses evict cache lines at fixed locations. The index used in table lookup in the first round is $p_i \oplus k_i$. Given a key K, some values of p_i will cause the table lookup to access those evicted cache lines and will experience a cache miss that leads to longer execution time (on

average). These can be seen as high bars in the timing charts at the x-axis values corresponding to these p_i.

(b) When a different key ***K'*** is used, since the index used for a table lookup is the XOR of the plaintext byte and the key byte, another set of plaintext values p'_i that satisfies $p_i \oplus k_i = p'_i \oplus k'_i$ will generate the same index for table lookups using p_i and k_i, and cause accesses to the same cache lines. This explains why their average encryption times are about the same, as in equation (1). Since Figure 5 (a) and (b) is plotted in terms of p_i and p_i', respectively, they have similar peaks, but in different locations. In Bernstein's attack, the key recovery step exploits this fact: $t_{avg}^{i}(j, \boldsymbol{K}) = t_{avg}^{i}(j \oplus k_i \oplus k'_i, \boldsymbol{K'})$ that is derived from equation (1). Since k_i is known, the attacker can try all 256 possible values of k'_i to permute the timing charts obtained in the attacking phase. The correct value of k'_i should make the permuted chart most similar to the one obtained in the learning phase. The similarity is quantitatively measured via correlation, using the algorithm shown in Figure 4.



(a) Timing characteristic chart for a known key K



(b) Timing characteristic chart for a different key K'

Figure 5. Timing characteristic charts of byte 0

3.4. New speculation-based covert channel

While the previous examples leak out information due to contention for shared resources (either cache or functional units), we have identified a different type of covert channel based on exposing events to unprivileged software that were previously not visible to it (e.g., exceptions). This has happened recently in some processors supporting speculative execution.

To hide the long latency that a load instruction may introduce, control speculation in IA-64 allows a load instruction to execute speculatively [14]. IA-64 adds a one-bit flag, the NaT bit, to each general-purpose register. If the speculative load instruction (ld.s) would cause an exception, the NaT bit of the target register is set, but the exception is not taken right away. Control speculation allows deferral of the exception, allowing the program itself to handle the exception when necessary. In current Itanium processors, TLB misses or TLB access bit violations are typical examples of ld.s exceptions which can be deferred. In addition, speculative loads may also be deferred by hardware based on implementation-dependent criteria, such as the detection of a long-latency cache miss. Such deferral is referred to as *spontaneous deferral* [14].

Such a mechanism, however, can be exploited to facilitate information leakage. For example, in the cache-based side channel attacks described earlier, cache misses are detected by measuring cache access timing. However, if spontaneous deferral is implemented in a future version of the Itanium processor such that cache misses can be deferred, the observer can observe the cache miss using control speculation. He can access a cache line using the ld.s instruction to check the NaT bit of the target register. If the NaT bit is set, a cache miss is detected. In contrast to timing measurement which suffers from noise, this mechanism is like a noiseless channel.

Similar methods can be used to detect exceptions such as a TLB miss. This is particularly useful when an insider is available. The insider can choose any exception sources available, not limited to cache or TLB misses. To encode a bit, the insider makes certain changes in the system such that later on, when the observer executes the speculative instruction, these changes will cause a deferred exception which sets the NaT bit. The observer can then see the bit sent by checking the NaT bit.

3.5. Data rates of covert channels

Table 1 shows the data rates of the new processor-based covert channels. To make the data comparable, we implement the SMT/FU channel on the same processor as used in [12], i.e., a 2.8GHz Pentium-4 processor with hyper-threading. The rate of the SMT/Cache channel reported in [12] is approximately 400 Kilobytes per second, or 3.2 Mbps (Megabits per second). We measured the rate of the SMT/FU channel as approximately 500 Kbps (Kilobits per second), which can be even higher if further optimized. We estimated the rate of the speculation-based channel based on a processor model with settings typical of an Itanium (IA-64) processor (1GHz clock rate; a 16-way

2MB cache with 128-byte cache lines.) A conservative estimation of this rate is about 200Kbps.

In contrast, traditional OS-based covert channels, e.g., the Inode table channel and the upgraded directory channel, exploit shared resources at system level. The resulting rates are much lower, e.g., around 50bps and 0.5bps respectively with typical data in the 1990's [13]. Even if we assume a linear increase in such OS covert channel rates with a 100X increase in processor clock rate, the processor-based covert channels are still orders of magnitude faster than the traditional OS-based covert channels.

Table 1. Data rates of new covert channels

SMT/cache	SMT/FU	Control Spec.
~3.2Mbps	~500Kbps	~200Kbps

4. Insufficiency of software isolation

Software isolation methods providing Mandatory Access Control (MAC) and Virtual Machine (VM) technology may erroneously lull us to think that they also prevent information from being leaked out. Unfortunately, without being aware of these attacks, software isolation can be completely ineffective.

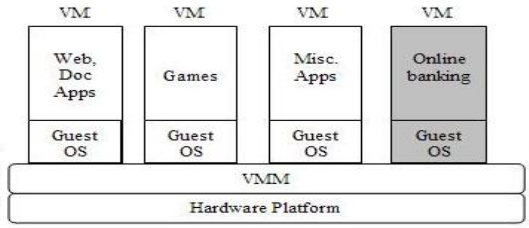


Figure 6. A VMM based system.

Figure 6 illustrates a recent trend towards implementing Virtual Machines, managed by a Virtual Machine Monitor (VMM), e.g., Terra [17], Xen [18]. A VM could be an open-box one (shown in white), which is allowed to communicate with other VMs via legitimate communications channels, or a closed-box one (shown in gray), which is completely isolated from other VMs. Security policies need to be established and enforced by the VMM. Such a system architecture can provide many desirable properties other than isolation, e.g., extensibility, compatibility and security.

As an example, consider an online banking application running in one of the closed-box VMs. Since it involves the use of important secrets such as the user's password, cryptographic keys, bank account information, etc., it is isolated from all other VMs and is only allowed to communicate with the authenticated bank server. The underlying VMM enforces security policies which disallow any form of communications between the closed-box VM and all other VMs. This ensures that the adversary outside the closed-box VM has no access to the user's secrets. Even if there is an insider in the closed-box VM, e.g. a Trojan horse or a backdoor in the banking application itself, and it gains access to the secrets, it has no way to distribute them outside of the VM, except to the trusted bank server.

While this sounds safe at first glance, such software isolation can be broken by exploiting certain processor architecture features. As described in section 3, a recent attack [12] on a hyper-threading processor allows a user process to extract the RSA key of another process which is performing RSA encryption. No special equipment is needed in the attack and the attack does not even require any software flaw for exploitation. The spy process only needs to execute a series of memory accesses and observe the timing while the victim process is running on the same processor. A VMM system running on top of a SMT processor therefore is vulnerable to this attack. An adversary outside the closed-box VM can steal the RSA keys involved in the online banking transaction.

A very important observation here is that unlike other security problems, the information leakage mechanism shown above does not break any protection mechanisms and can escape detection. Even with perfect access control, information flow monitoring and auditing, information can still be leaked out by exploiting processor architecture features, without being detected. In the next two sections, we propose two solutions to mitigate SMT-based covert channel attacks and cache-based software side-channel attacks.

5. Selective partitioning solution

The first general solution approach is to minimize resource sharing, and hence prevent interference between processes. The SMT/FU covert channel exploits the sharing of functional units by multiple simultaneously active threads. A straightforward way to block such a channel is to disallow any other processes from running when a protected thread is scheduled for execution. This strict partitioning can have severe performance consequences. This can be meliorated by allowing protected processes to be executed with other processes, only disallowing the simultaneous execution of processes that should be isolated from each other. We refer to this as a *selective partitioning* solution. It is similar to a "Chinese Wall" separation policy, but at the hardware thread level.

Selective partitioning can be implemented in software, e.g., by having the OS enforce this restriction

in process scheduling. This applies scheduling restrictions only to critical processes that operate on sensitive data. These processes may be cryptographic routines, which tend to have small working sets compared to other applications. A typical mix of applications used in ordinary PC systems consists of mostly non-critical processes.

Selectively partitioning can also be based on leveraging existing hardware mechanisms. For the hardware solution, we leverage the "fairness control" mechanism implemented in some SMT processors to prevent overuse of certain shared resources by a process. For example, in Intel's hyper-threading processor, the allocator has fairness control logic for assigning buffers to micro-ops from different logical processors and the instruction scheduler has fairness control on the number of active entries in each scheduler's queue. Our hardware solution proposes that such fairness control logic can be leveraged to mitigate covert channels as well. For the SMT/FU covert channel we identified in section 3, the existing fairness control logic in the instruction scheduler can be modified slightly so that, when necessary, it allocates a fixed number of entries for each process in each queue. This would minimize the interference between the two concurrent processes running on the chip.

The performance degradation of the system with our selective partitioning solution can be estimated as:

$$R = (1-p)R_0 + p \cdot R_1 \qquad (3)$$

where R_0 is the throughput when there is no scheduling restriction, R_1 the throughput when a critical process is running on the processor, and p is the probability of the occurrence of such restricted execution. The relative throughput therefore can be written as:

$$\frac{R}{R_0} = (1-p) + \frac{R_1}{R_0} p = 1 - (1 - \frac{R_1}{R_0}) p = 1 - \alpha p \qquad (4)$$

To estimate the performance degradation incurred by Selective Partitioning, we first estimate the coefficient α in (4), by performing a preliminary test on a HT processor. We wish to compare the performance of a system with HT enabled versus with HT disabled (simulating the restricted execution, i.e. only one process can execute at a time). With HT enabled, we observed up to 30% performance increase in terms of overall system throughput, though in a few occasions we also observed performance degradation. In general, we found that in most cases, the relative throughput of HT-disabled vs. HT-enabled system is in the range of 0.75-0.95, or equivalently, α is in the range of 0.05-0.25. Figure 7 shows the performance degradation curve when the probability of restricted scheduling p changes from 0 to 1. In the worst case, when α equals 0.25 and p equals 1, the performance degradation is approximately 25%. In typical cases, p is likely to be in the range of 0.1 to 0.15, in which case the performance degradation is less than 5%.

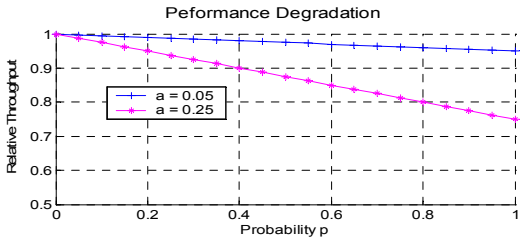


Figure 7. Performance of selective partitioning

6. Random permutation cache solution

A second general solution is to use "signal randomization". Any signal sent by the sending process is randomized with respect to a receiving process. Our solution is to use *different memory-to-cache mappings for processes* that need to be isolated from others. Other processes cannot deduce what cache index bits are used by a process when the mapping is unknown. Furthermore, this mapping should not be fixed since the attacker may be able to learn the mapping by doing a number of experiments.

Changing the memory-to-cache mapping for each process can be implemented by a variety of mechanisms, such as XORing the cache index bits with a random number or hashing the cache index bits. XOR and hash-based mapping are simple to implement, but may not provide enough randomness. Rather, we propose to use random permutation that gives the best 1-to-1 random mapping. This can be achieved with one level of indirection: keep a table that contains the permuted index for each original cache index. When accessing the cache, the original index is used to look up this table for the corresponding permuted index, which is then used to access the cache. This extra level of indirection for Level 1 data cache accesses is costly in terms of cycle-time latency or cycles per access. Below, we show how we achieve random permutation mapping without an extra level of indirection for table lookup and without lengthening the cache access time.

6.1. Low-overhead RPCache implementation

Figure 8 shows the functional block diagram of a generic cache. During a cache access, a portion of the N bits of the effective address, used to index the cache, is sent to the address decoder. The decoder outputs 2^N word lines and in each access only one of them is

driven high to select the cache set that contains the data being accessed. For each word line there is a comparison module which compares the effective address A with the current word line number k. The cache set is selected only when these are equal. We can add an N-bit register, which we denote a Permutation Register (PR), for each word line which contains a permuted cache index, feeding this into the comparison module instead of the original constant for the word line number k. We call such a set of permutation registers a Permutation Register Set (PRS). By changing the contents of the PRS, arbitrary cache index mapping can be achieved.

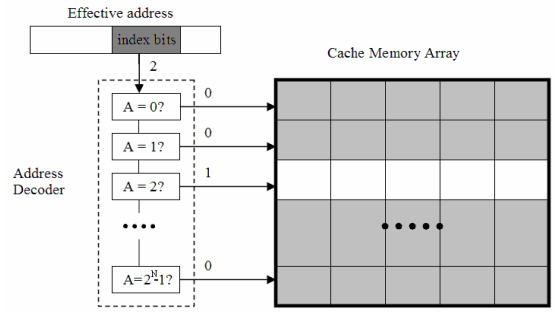


Figure 8. A generic cache architecture.

In real implementations, the comparators in the address decoder for the cache are not implemented as separate units. Also, the fact that A is compared to a constant word line number k is exploited to simplify circuit design. The only difference between using the variable contents of a Permutation Register (in our RPCache) rather than a constant word line number k is that fixed connections between a grid of wires in the address decoder circuit are replaced with switches. Though the drain capacitance of the switches increases the load in the address decoder circuits, proper circuit design can easily overcome this problem with no extra delay introduced.

To prevent the attacker from learning the memory-to-cache mapping via experiments, the mapping needs to be gradually changed. This can be implemented by swapping cache sets, two at a time. To change the cache mapping, two permutation registers of the PRS are selected and their contents swapped. The corresponding cache lines are invalidated, which triggers the cache mechanism to write back any "dirty" cache lines, i.e., cache lines that have newer data written in it since they were brought into the cache from memory. Subsequent accesses to these invalidated cache sets will miss in the cache, which will degrade performance. However, as we will show in section 6.4, the performance impact is very small.

After investigating different swapping policies, including periodically swapping, we have found an optimal swapping strategy from the information-theoretic perspective. This is based on realizing that only cache misses that cause replacements give side-channel information, so no swapping needs to be done when there are no such cache misses. Upon a cache replacement, we swap the cache index of the cache set that contains the incoming cache line with any one of the cache sets with equal probability. So for any cache miss that the receiver process detects, it can be caused by a victim process's cache access to any one of all cache sets with equal probabilities. Hence, when the receiver process detects a cache miss, he cannot learn anything about the cache locations used by the sender process. This cache swapping policy also incurs very little performance overhead, as shown later.

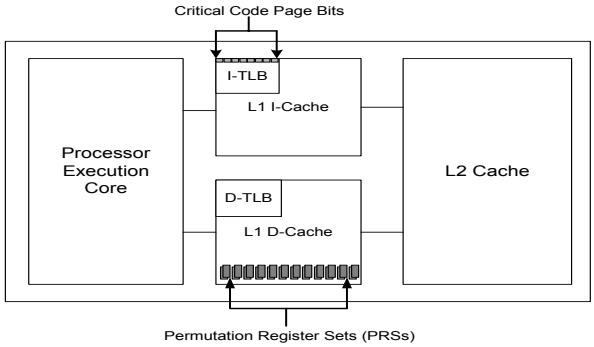


Figure 9. A processor with RPCache

6.2. RPCache architecture

The RPCache requires permutation registers (PR), one for each set of the L1 (Level 1) Data cache. In a direct-mapped cache, there is one PRS register for each cache line. The processor can contain one or more sets of such permutation registers. There is also a new bit per Instruction Translation Look aside Buffer (ITLB) entry that we call the Critical Code Page (CCP) bit.

6.3. RPCache usage model

Each PRS set may be associated with a segment of code that needs to be protected. This may be a whole process, or a critical part of a process, e.g., the crypto-related shared library calls. When such a segment of code is executing on the processor, the corresponding PRS is used to permute the index to the cache.

The CCP bit in an ITLB entry indicates if the code on that page needs to be protected. When an instruction is fetched for execution, the CCP bit in the

corresponding ITLB entry is checked. If it is set, the cache access of a load or store instruction will go through the PRS mapping, otherwise the cache access will use the original cache index. Critical processes and the critical segments of a process are marked.

The PRS can be managed solely by hardware. During a context switch, the old PRS values are simply discarded and a new set of values are generated (if the ITLB of the new process has its CCP bit set). Dirty cache lines may need to be written back, because next time this process is swapped in, it will use a different PRS to index the cache. It has to get a copy of the data from the next level cache or from the main memory, and the freshness of the data must be ensured. For a write-through cache, however, no such overhead is necessary since the next level cache always has the latest data.

The PRS can also be maintained by the OS. Upon each context switch, the PRS of the process that is swapping out should be saved as part of the context and the OS should load the PRS values of the incoming process to the on chip PRS registers.

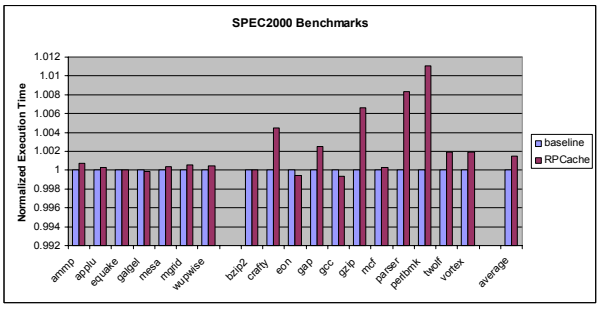


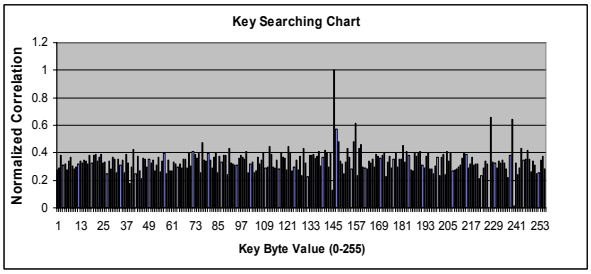
Figure 10. RPCache Performance

6.4. Performance evaluation

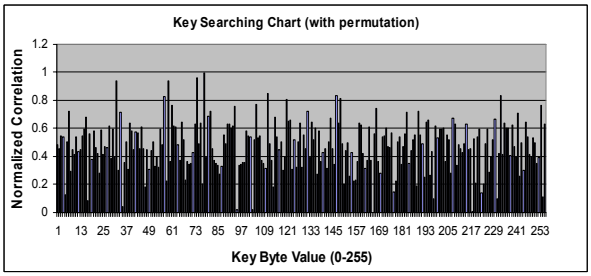
The RPCache may introduce extra overhead when a change of the cache permutation mapping occurs. This may be during a context switch or when two cache sets are swapped. For a context switch, the overhead is insignificant relative to the time between two context switches. For the hardware-managed swapping, we swap just the contents of the pair of PRS registers while invalidating their associated cache lines.

To evaluate the performance degradation, we run a set of SPEC2000 benchmarks on the Simplescalar simulator [25]. Figure 10 shows the normalized execution time. The baseline machine has a 2-way set-associative write-back L1 data cache. The data marked with "RPCache" is generated on a machine using the cache set swapping scheme where the two cache sets to be swapped are invalidated with dirty lines written back to the next cache level.

The performance degradation is very low: 1.1% worst case (perlbmk) and only 0.15% on average. This appears to be because cache misses and cache line replacements normally occur infrequently. The performance degradation is mainly due to the extra cache misses caused by the invalidation of the cache lines. However, since each time the invalidated sets are only two out of all cache sets, the resulting performance degradation is also insignificant.



(a) Key searching chart without cache-index permutation



(b) Key searching chart with cache-index permutation

Figure 11. Key searching charts

6.5. Security analysis

Three of the attacks described in sections 3 involve cache-based information leakage.

Bernstein's statistical cache attack: We first evaluate the effectiveness of the RPCache in mitigating Bernstein's statistical cache attack and illustrate this in Figure 11. We simulate the effect of cache permuting by swapping the memory blocks of the AES tables once per 25 AES encryptions. This is roughly equivalent to swapping two cache sets per 10,000 cycles. In Figure 11, the x-axis is the key value that is used in the key-byte searching algorithm described in Section 3. The y-axis is the normalized correlation of the two timing characteristic charts given the corresponding x value as the guessed key byte. The higher the y value, the more the two charts match. The x value that has the highest correlation value is the discovered key-byte. Figure 11(a) shows a distinct high y value for a given x value (x=143), leading to a likely key-byte match. In Figure 11(b), the same key is used, but with the RPCache remapping, the correlation

is rather random, resulting in non-recovery of the key byte. In fact, the highest peaks are actually misleading, since they indicate values that are not the correct key-byte. This shows the effectiveness of our RPCache remapping in thwarting Bernstein's attack.

SMT/Cache side channel attack: in the SMT/Cache side channel attack, the receiver process can directly detect the memory locations used by the sender (victim) process by detecting cache misses. The RPCache thwarts this attack since each time the receiver process observes a cache miss, the swapping policy ensures that this cache miss can be caused by a victim process's cache access to any one of the cache sets with equal probability. Therefore no information is gleaned about which cache line was accessed by the victim (sender) process. This effectively stops SMT/Cache side channel attacks.

Speculation-enhanced cache attack: the control-speculation mechanism, described in section 3.4, can provide a more reliable way to detect cache misses. However, this will not help if our RPCache is used. Recall that RPCache does not prevent the attacker from detecting cache misses. Instead, RPCache makes the victim process's cache accesses unrelated to the attacker's cache miss pattern, and hence no information can be inferred even if the attacker can accurately detect cache misses.

7. Related work

Past work on covert channels analyzed system specifications and implementations for illegal information flows [13]. Past work on side-channel attacks focused on differential power [2-5] and timing [6-10] analysis. Cache-based side channel attacks were studied in [26-27]. Some side-channel attacks were reported recently which allow complete key recovery during AES and RSA encryption [11][12][22].

The control flow information leakage problem due to the exposure of address traces on the system memory bus is studied in [21][24]. Both proposed probabilistic approaches for hiding the real access sequence. Our work is different since we focus on information leakage caused by *resource interference*, e.g., cache interference, which has fundamentally different assumptions. Our solutions are also very different. In [23], an approach different from ours, i.e. cache partitioning, was proposed to mitigate one type of cache-based side channel attacks. This incurs performance penalties and also requires changes in the ISA, compiler and operating system. Other relevant work include special purpose secure devices such as the IBM 4758 cryptographic coprocessor [20].

8. Summary and conclusions

Information leakage attacks that exploit processor architecture features are particularly dangerous for many reasons. They dramatically increase the bandwidth and reliability of covert and side channels, and they exist even when strong software isolation techniques are present.

Unlike traditional covert channels, processor-based covert channels are much faster and more reliable. They are much faster because microprocessors operate at the highest clock rate in the system and resource sharing can be very tightly coupled, as we showed for Simultaneous Multi-Threaded (SMT) processors. Processor-based covert channels also result in more reliable covert channel communications since the global on-chip clock makes the synchronization easier. In fact, we showed that the data rate of these processor-based covert channels can be orders of magnitude larger than traditional covert channels.

We showed that processor-based covert channels are not prevented (or even impacted) by strong software isolation architectures like Virtual Machine technology with secure Virtual Machine Monitors or secure hypervisors. In fact, the software trends toward portable design methodology and virtualization techniques both try to "hide" the hardware, making most of the system design and development independent of hardware. This can be very dangerous, since it can lead to oblivion of the serious and growing threat of hardware processor-based information leakage. Also, since covert channels and side channels only rely on legitimate use of the system and do not directly access secrets, the system can not detect the existence of such an attack even if perfect access control, monitoring and auditing mechanisms are implemented.

This paper demonstrated the information leakage problem at the processor architecture level, with detailed covert and side channel examples. We identified two new convert channels based on highly-touted processor features, viz., Simultaneous Multi-Threading and Speculative Execution. We also provided detailed analysis of why the two recently publicized cache-based side channel attacks work.

We then suggested two solutions, Selective Partitioning and the novel RPCache. Selective partitioning by software (or hardware) can prevent the SMT/FU covert channel problem. Here, we estimated the performance degradation at no more than 25%, with an expected value of less than 10%. Our RPCache proposal uses an efficient cache-index randomization solution that thwarts software cache-based side channel attacks. Performance degradation is less than

1%, and the solution is transparent to software. We believe this solution can defeat most cache-based attacks that try to find which cache locations are used by another process.

In conclusion, hardware processor-based covert channels and cache side channels can be very dangerous. Mitigation of these at the hardware level is often necessary since these hardware-based channels are not impacted by even strong software isolation mechanisms. We hope to have alerted both the security and computer architecture communities to this processor-induced information leakage threat. Future work will attempt to identify more processor features that facilitate covert channels and side channels and will study other solutions for this growing threat.

References

[1] Butler W. Lampson, "A note on the confinement problem", *Communications of the ACM*, v.16 n.10, pp.613-615, Oct. 1973.

[2] C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis", *Advances in Cryptology - CRYPTO '99*, vol. 1666 of Lecture Notes in Computer Science, pp.388-397, Springer-Verlag, 1999.

[3] J.-S. Coron and L. Goubin, "On Boolean and arithmetic masking against differential power analysis", *Cryptographic Hardware and Embedded Systems* (*CHES 2000*), vol. 1965 of Lecture Notes in Computer Science, pp. 231-237, Springer-Verlag, 2000.

[4] M. Joye, "Smart-card implementations of elliptic curve cryptography and DPA-type attacks", *Smart Card Research and Advanced Applications VI*, pp.115-125, Kluwer Academic Publishers, 2004.

[5] J. Waddle and D. Wagner, "Towards efficient second-order power analysis", *Cryptographic Hardware and Embedded Systems* (*CHES 2004*), vol. 3156 of Lecture Notes in Computer Science, pp. 1-15, Springer-Verlag, 2004.

[6] P. Kocher, "Timing attacks in implementations of Diffie-Hellman, RSA, DSS, and other systems", *Proceedings Crypto '96, Lecture Notes in Computer Science*, vol. 1109, Springer-Verlag, pp. 104-113.

[7] Werner Schindler, "A Timing Attack against RSA with the Chinese Remainder Theorem", *CHES 2000*, pp.109-124, 2000.

[8] Werner Schindler, "Optimized Timing Attacks against Public Key Cryptosystems", *Statistics and Decisions*, 20:191-210, 2002.

[9] David Brumley and Dan Boneh, "Remote Timing Attacks are Practical", *Proceedings of the 12th USENIX Security Symposium*, pp.1-14, August 2003.

[10] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestre, J.-J. Quisquater, and J.-L. Willems, "A practical implementation of the timing attack", *Proc. CARDIS 1998, Smart Card Research and Advanced Applications*, 1998.

[11] D.J. Bernstein, "Cache-timing Attacks on AES", http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[12] C. Percival, "Cache Missing for Fun and Profit", http://www.daemonology.net/papers/htt.pdf.

[13] National Computer Security Center, "A Guide to Understanding Covert Channel Analysis of Trusted Systems", NCSC-TG-30, November 1993, http://www.radium.ncsc.mil/tpep/library/rainbow.

[14] Intel Itanium Architecture Software Developer's Manuals Volume 1-3, http://www.intel.com/design/itanium2/documentation.htm.

[15] Dean Tullsen, Susan Eggers, and Henry Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[16] D.T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J.A. Miller, M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture", *Intel Technology Journal, vol.6, issue 1*, pp.4-15, 2002.

[17] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing", *Proceedings of the 19th ACM Symposium on Operating System Principles*, pp. 193-206, Oct 2003.

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, "Xen and the Art of Virtualization", *Proceedings of the 19th ACM Symposium on Operating System Principles*, pp. 164-177, Oct 2003.

[19] A. Baratloo, T. Tsai, and N. Singh, "Transparent Run-Time Defense Against Stack Smashing Attacks", *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[20] The IBM 4758 PCI cryptographic coprocessor, available at http://www-03.ibm.com/security/cryptocards.

[21] X. Zhuang, T. Zhang, and S. Pande, "HIDE: an infrastructure for efficiently protecting information leakage on the address bus", *ACM 11thInternational Conference on Architecture Support for Programming Language and Operating Systems*, 2004.

[22] D. A. Osvik, A. Shamir and E. Tromer, "Cache attacks and Countermeasures: the Case of AES", *Cryptology ePrint Archive*, Report 2005/271, 2005.

[23] D. Page, "Partitioned Cache Architecture as a Side-Channel Defense Mechanism", *Cryptology ePrint Archive*, Report 2005/280, 2005.

[24] Oded Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs", *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, May 1987.

[25] T Austin, E. Larson, and D. Ernst, "Simplescalar: an infrastructure for computer system modeling," IEEE computer, 35(2), Feb 2002.

[26] Daniel Page, "Theoretical use of cache memory as a cryptanalytic side-channel", *technical report CSTR-02-003*, Department of Computer Science, University of Bristol, 2002.

[27] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, Hiroshi Miyauchi, "Cryptanalysis of DES implemented on computers with cache," *Proc. CHES 2003*, LNCS 2779, 62-76, 2003.