Noname manuscript No.
(will be inserted by the editor)

Covert Timing Channels exploiting Cache Coherence Hardware: Characterization and Defense

Fan Yao · **Miloš Doroslovački** ·
Guru Venkataramani

Received: date / Accepted: date

Abstract Information leakage of sensitive data has become one of the fast growing concerns among computer users. With adversaries turning to hardware for exploits, caches are frequently a target for timing channels since they present different timing profiles for cache miss and hit latencies. Such timing channels operate by having an adversary covertly communicate secrets to a spy simply through modulating resource timing without leaving any physical evidence.

In this article, we demonstrate a new vulnerability exposed by cache coherence protocols where adversaries could manipulate the coherence states on certain cache blocks to alter cache access timing and communicate secrets illegitimately. Our threat model assumes the trojan and spy can either exploit explicitly shared read-only physical pages (e.g., shared library code), or use memory deduplication feature to implicitly force create shared physical pages. We demonstrate a template that adversaries may use to construct covert timing channels through manipulating combinations of coherence states and data placement in different caches. We investigate several classes of cache coherence protocols, and observe that both directory-based and snoopy protocols can be subject to covert timing channel attacks. We identify that the root cause of the vulnerability to be the existence of access latency difference for cache lines in read-only cache coherence states: *Exlusive* and *Shared*. For defense, we propose a slightly modified cache coherence scheme that will enable the last level cache to directly respond to read data requests in these read-only coherence states, and avoid any latency difference that could enable timing channels.

Fan Yao
The George Washington University
E-mail: albertyao@gwu.edu

Miloš Doroslovački
The George Washington University
E-mail: doroslov@email.gwu.edu

Guru Venkataramani
The George Washington University
E-mail: guruv@gwu.edu

Keywords cache coherence protocols; covert timing channels; information leakage; hardware security; hardware defense

1 Introduction

Information leakage attacks, that maliciously tap insider processes and secretly communicate sensitive data, are an increasingly growing threat for computer users. Covert channels are one such class of attacks, where a trojan process, that has access rights to user's sensitive profile information, secretly communicates such data to a spy process despite the underlying system security policy explicitly prohibiting any such communication [1]. It is important to note the trojan will not be able to directly communicate secrets to the outside entities with the system auditors monitoring for any such activity. Therefore, they rely on covert channels to reveal secrets to the spy. In contrast to side channels where a victim process unwittingly exposes sensitive application profile to the spy monitoring its activity, covert channels work by intentional collusion between two malicious processes, namely the trojan and spy [2].

Among several types of covert channels, timing attacks are practically very difficult to catch since the trojan and spy communicate simply by manipulating the access timing to shared hardware resources without leaving any physical trace of an attack [3]. Caches, in particular, are a widely exploited resource since they present the largest attack surface to adversaries. To manipulate the cache access latencies, prior works have shown how to intentionally creating cache hits and misses by either generating contention in a pre-determined cache set [4–7] or through invalidating shared cache lines through cache cleansing instructions such as *clflush* [8]. Recently, Irazoqui et al. [9] demonstrated a cache side channel that exploit different latencies due to remote cache hit (i.e., cache hit in another socket kept coherent with the requesting socket) and DRAM accesses. We note that these prior works rely on access latency difference between DRAM vs. caches, and as such, *do not demonstrate the vulnerability of hardware coherence protocol and its states*.

In contrast to several cache-based covert timing channels, our recent work [10] has demonstrated that cache coherence protocols, which are a widely supported performance feature in most modern multi-core processors, can be vulnerable to information leakage attacks. We have shown that the differences in timing profiles on cache block accesses in *Exclusive (E)* and *Shared (S)* coherence states may present a significant vulnerability that can be taken advantage by malicious entities for covert timing channel exploits.

This article offers new technical contributions over our prior work in two major aspects: First, we have systematically analyzed the vulnerability of several classes of cache coherence protocols (directory-based, snoopy with inclusive and non-inclusive) caches, and have presented our insights. Second, we have proposed a defense mechanism that proposes a secure cache coherence scheme with modest changes to existing coherence protocols, thereby, effectively closing the latency gap between cache accesses to read-only states, namely E and S.

We note that prior works have studied defense techniques to thwart timing channels on individual caches [11–14]. These techniques either work by preventing the ad-

versaries from cache evictions with significant hardware modifications (e.g., random cache replacement) or through partitioning the cache among various user domains. Although they are shown to be effective for several existing cache timing channel attacks, these prior mechanisms are not designed to counter the attacks that manipulate *cache lines in distinct coherence states and in different caches across several cores*. Thus, they may be ineffective in defending against the vulnerabilities studied in this article. We note that the root cause for the newly demonstrated vulnerability is the differing latency profiles for cache coherence transactions associated with different read-only states. Therefore, to protect the cache coherence fabric and to avoid adversely affecting the latency-critical read operations, we propose a defense scheme that eliminates the latency differences among the corresponding read transactions. The modified cache coherence scheme is able to defend against the exploits involving cache coherence states.

In summary, the contributions of our work are as follows:

1. We present a new study that shows the vulnerabilities exposed by hardware cache coherence protocols, where adversaries exploit coherent cache blocks in *exclusive* and *shared* states present in *different* levels of the cache hierarchy.

2. We show six ways to exploit the read latency differences for cache blocks, stemming from combinations of coherence states and cache levels, and demonstrate them on real machines.

3. We propose a new defense against timing channels in cache coherence hardware through small changes to read operations on exclusive state cache blocks, that effectively eliminates the latency differences between read transactions to E and S coherence states. Our evaluations show that our proposed hardware changes incur minimal overhead (less than 0.5%) for benign multi-threaded applications.

2 Background

2.1 Covert Channels

According to Trusted Computer System Evaluation (TCSEC) [1] developed by US Department of Defense, covert channel is defined as a communication channel that is exploited by a process to perform information transfer in a manner that violates the system's security policy. Specifically, timing channels work by allowing a trojan process to signal information to a spy process by modulating its own use of system resources in such a way that the change in response time observed by the spy would provide information. As per TCSEC criteria, low bandwidth covert channels typically present lower threat than higher bandwidth channels. This is because, adversaries typically incur high price on lower bandwidth channels with lower returns in terms of information gain. For example, TCSEC points out that the malicious entity obtains very little useful information on covert channels with bandwidths below 0.1 bits/sec. TCSEC classifies a high bandwidth covert channel to have a minimum rate of 100 bits/sec based upon measurements from several different computer systems.

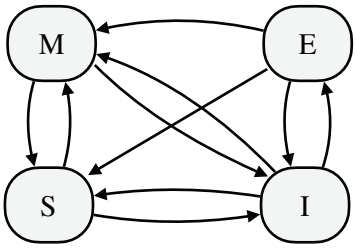


Fig. 1: State transitions for the MESI protocol

2.2 Cache Coherence

Almost every modern processor family from leading hardware vendors implement variants of *MESI* cache coherence protocol to realize coherence among data in private caches [15, 16]. The MESI protocol has four stable states, namely: (i) *Modified* (M) state, where the data block is present only in one private cache and has been recently updated by the owner core in comparison to the value in main memory. Note that the current owner has both read and write permissions. (ii) *Shared* (S) state, where the cache block is potentially present in more than one private cache and has not changed by any core recently compared to the value in main memory. The current core only has read permission on the block. (iii) *Exclusive* (E) state, where the cache block is present only in the current owner cache, and the data has not changed recently compared to the data contents in main memory. The current owner core has read permissions. However, since the cache block is present only in the current cache, when a write operation need to be performed, E state lets the owner core to acquire write permissions and upgrade to M state without the need to generate invalidation messages to other cores. Also, read misses to this block by other cores will downgrade the coherence state in the current core to S state. This coherence state offers performance advantage by enabling rapid transitions to M or S from the E state depending on the memory operation. (iv) *Invalid* (I) state, where the cache block is invalid, and does not have read or write permissions. Figure 1 illustrates the state transitions for the four coherence states in MESI-based protocols. Several design options are available for implementing cache coherence protocols [17]. In a snooping protocol, cache controllers send coherence requests by broadcasting the messages to all other cache coherence controllers. In contrast, directory-based protocols offer higher scalability by allowing directed, unicast messages between cache controller and the memory controller or directory. Besides, modern processors also feature multiple levels of caches to improve the caching performance. In inclusive cache hierarchy, lower-level caches (e.g., shared caches) always store the cache blocks that are currently cached in upper-level caches (e.g., private caches). Non-inclusive cache does not enforce this requirement, i.e., lower-level caches may replace blocks that are still currently cached in upper-level caches. In exclusive cache hierarchy, the lower-level caches are designed to not cache any of the memory blocks that already reside in upper-level caches.

3 Memory Sharing in Server Systems

To manipulate cache coherence states, the trojan and the spy should first have shared physical memory such that certain cache blocks are shared and coherence is enforced by the hardware. Prior techniques [8,9] have shown their timing channel implementations using explicitly shared library code or data between trojan and spy. Coherence protocols would maintain states on such blocks to keep a coherent view of the program memory that is supported by the underlying hardware. We note that this memory sharing method can be used in our attack model as well. However, we note that this explicitly shared memory setup will imply that we assume the trojan (with access to sensitive user data) and the spy (that can't access sensitive data as per the system security policy) to have explicitly shared memory (e.g., shared library). Such prerequisite may be difficult to achieve in systems where strict isolation policies are enforced.

A more sophisticated attacker might circumvent the explicit sharing requirement by exploiting a feature called memory deduplication, sometimes also referred to as Kernel Same-page Merging (KSM). KSM is a kernel feature inside the OS that allows the system to share identical memory pages (i.e., pages with the same memory contents) between different processes. This feature is routinely used to enhance system performance by avoiding the need to duplicate physical memory pages that hold identical data. In current Linux systems, KSM is a kernel thread that periodically scans the entire memory to identify identical memory pages and make them to be candidates for merging. At the end of KSM scan, a single physical copy of the page is retained and all of the duplicated pages are updated to point to this single physical page in the application's page table. The physical pages belonging to the duplicate pages are then released back to the system that can be used later for storing more physical pages with distinct memory contents. The single physical copy (at the end of the merging process) is marked as *copy on write* and resides in *read-only* sharing mode. In other words, write operations to these read-only shared pages are not possible since the kernel will separate them into two separate pages if one of the sharers happen to modify the contents of the page. KSM has been widely adopted in most modern server-class systems to avoid unwanted memory duplication and improve the page miss rate by reducing the memory footprint in virtualized environments [18,19].

In covert channels, the trojan and spy may exploit this feature to force create shared memory without explicitly having to share any data between them. In particular, the trojan and spy can generate physical pages with identical bit patterns known to both of them ahead of time. KSM scans the memory spaces belonging to processes in the order of their starting times (earliest first). To avoid noise from external processes that may accidentally have the exact same bit patterns, the trojan and spy can go through a trial communication phase where they perform a series of cache flushes and reloads on this page to make sure that no other process is currently sharing this page as a result of KSM scan. If an external sharing of this page is detected (through timing measurements), the trojan and spy may repeat creating shared memory through KSM using another set of identical bit patterns known to both of them.

4 Threat Model

We assume that the trojan and spy share (*one or more*) multi-core processors. The trojan has access to secretive information and it desires to transmit the secrets to the spy. The spy process who does not have accesses to the secrets, will then cause damages to users by exfiltrating the information to outside world (e.g., identify theft). The trojan can be an application that is downloaded from untrusted sources. Note that the trojan is unable to send secretive data to the outside by itself since it either lacks sufficient permissions to do so or its transmission activity can possibly be identified and suspended by the system confinement mechanism at the software level. Meanwhile, due to existing isolation techniques such as sandboxing, the trojan and spy processes are explicitly prohibited from any form of direct communication. Such settings exist currently in a variety of real-world scenarios, especially in multi-tenant cloud environment where virtual machines from multiple users can be co-scheduled on the same physical machine to increase resource utilization.

The trojan and spy are able to create shared DRAM pages by either explicitly mounting shared libraries or by leveraging KSM to silently merge two identical pages. We note that the latter method is more stealthy since it does not explicitly request memory sharing between the trojan and spy. Also, KSM is widely used in various server systems to improve memory usage efficiency, which is less likely to be disabled due to its high performance advantages. We further assume that the trojan is capable of spawning multiple threads that would run on multiple cores either within the same socket or across multiple sockets. Through this capability, the trojan can intentionally modulate the cache access timing through placing the shared data block in different coherence states and possibly in various levels of the memory hierarchy (local processor's caches, another processor's caches in a multi-processor). The pattern of timing differences between cache block accesses in different coherence states and locations enables a spy to infer trojan's transmission. To synchronize, the trojan and spy can initiate a pre-transmission process where the trojan and spy checks if a pre-determined series of activities (such as flushes) are observed on the shared cache blocks [10].

5 Coherence States and Latency

In order to understand the variations in cache access latency and the effects of cache coherence states, we perform experiments that load (read) data in specific cache coherence states (S or E) from various cache locations (local and remote caches with respect to the requesting core). We construct a micro-benchmark with process threads that could be pinned to either one or multiple cores. Each requestor thread periodically issues load operations to local and/or remote cache blocks that are in one of the two coherence states: S or E[1]. In this study, we use a dual-socket Intel Xeon X5650 server, each with 6 cores running at 2.67 GHz frequency. Each processor has a 32 KB

[1] Note that other coherence states such as M may also exhibit different latency profiles. However, change to M state will require writes to the cache blocks. Since writes to shared memory will annul silent page sharing (created using KSM), we do not consider these other states.

Cache/Coherence State	Min Latency	Average Latency	Max Latency
Local Exclusive	116	**124**	128
Local Shared	92	**96**	101
Remote Exclusive	244	**248**	253
Remote Shared	220	**228**	236

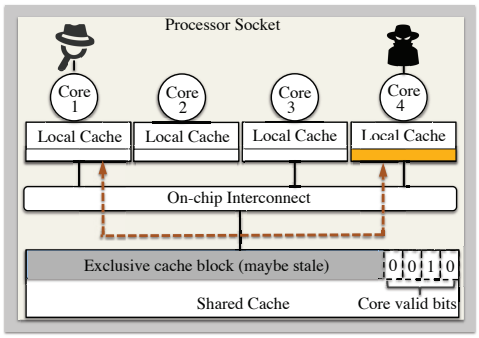
Table 1: Load operation latency (CPU Cycles) in various (location, coherence state) combinations. Location is with respect to Spy.

private L1, 256 KB private L2 caches, 12 MB shared L3 cache within each socket. All of the caches are kept coherent using hardware cache coherence. Our experiments were conducted on a typical desktop server system with a representative mix of workloads such as browser, dropbox, code editors running alongside our micro-benchmark as we made our measurements.

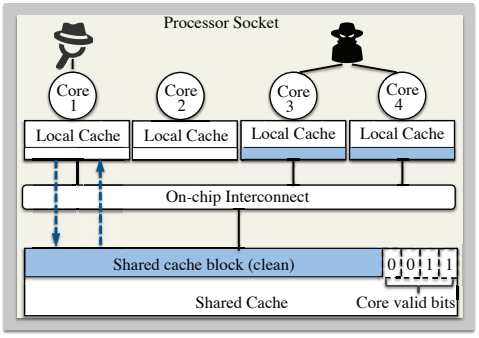
For our measurements, we generate 1,000 memory read (load) operations for each combination pair of (location, coherence state), and time these loads using *rdtsc* instruction. We note that coherence transactions are generated in each case. For example, in *Local Shared* configuration, the requested data is a local L2 cache miss and is fetched from L3 cache in the same (local) processor socket where the data is present in the *S* state. In *Local Exclusive*, the requested data is local cache miss and is fetched from another core's L1 or L2 cache belonging to the local socket where the data is present in the *E* state. Similarly, in *Remote Shared*, the requested data is present in the S state in the L3 cache of a different (remote) processor socket. In *Remote Exclusive*, the requested data is present in the *E* state in a L2 cache belonging to a remote socket. Table 1 shows the minimum, average and maximum access latencies for cache blocks in the various (location, coherence state) combination pairs. Our results show that these combination pairs show distinct bands of cache access latency distributions. We observe that accessing a cache block in the *E* state incurs longer latency than accessing a data block in *S* state (e.g., 124 cycles for accessing local E state block and 98 cycles for local S state data block) triggered by cache lookup in different coherence states (described in Section 6). Similar latency difference could also be observed for accessing blocks in remote caches as well. Our experiments demonstrate that the latency values are contained within a relatively narrow range for each configuration, and the ranges corresponding to different configurations are sufficiently distinct from each other. This clearly demonstrates that the malicious attackers may exploit such latency differences between these combination pairs to achieve their timing channel implementation.

6 Exploiting Cache Coherence

Having seen the differences in cache access latency profiles for different coherence states, we investigate practical ways that the trojan and spy processes can implement timing channels.



(a) Cache block in E state



(b) Cache block in S state

Fig. 2: Trojan (on the right side) explicitly controlling Cache Coherence States as E or S by running on one or two cores within the multi-core processor. The dotted lines show the service path of the spy (on the left side) for a data block residing in E and S states respectively.

6.1 Intra-socket Cache Coherence

Figure 2 shows an illustration of the attack using intra-socket coherence. Here we assume a multi-core processor where each core has a private write-back cache kept coherent using a variant of MESI protocol, and all of the cores have access to a shared last level cache (LLC).

For a read miss in the private cache, the request is first sent to the shared LLC. The LLC maintains the *core valid bits* vector for each cache block indicating the private caches that have a copy of the cache block [20]. An *1* bit value shows that the corresponding core stores that block currently, and a *0* shows that the corresponding core does not have it.

When the count of *1*'s in the vector is more than one, it shows that two or more sharers exist for this block. That is, the cache block is in the *S* state. Since the LLC has a clean data copy, it can directly service the cache miss request from the requesting core.

When the count of *1*'s in the *core valid bits* vector is equal to one, it shows that only one cache currently has the block. This means that the cache may have the block in the *E* or *M* coherence states. Due to the fact that most existing cache coherence protocols allow silent upgrade of cache line from E state to M state, the LLC copy of the block is *possibly stale*. Therefore, when a subsequent request for the same cache block arrives, to avoid sending possibly stale data back to the requestor, the LLC forwards the cache request to the owner. The owner cache responds to the requesting core with the latest copy of the cache block, and downgrades itself to the *S* state. The owner also performs a write-back to the LLC to leave a clean copy for future read misses on this block. At the end of this transaction, note that the *core valid bits* vector is updated to reflect the new sharer (the requesting core), and the count of *1*'s (sharer caches) increases to two.

When the count of *1*'s in the *core valid bits* vector is equal to zero, it denotes that none of the caches currently have the block. If the LLC has a clean copy of the data (i.e., cache valid is 1), the LLC can service the miss request. Otherwise, the miss request is forwarded to the lower level memory, e.g., DRAM. This case does not generate any coherence activity.

In order to communicate covertly, the trojan has to place a cache block, B (that can be read by the spy as well) in either of *S* or *E* coherence states, and let the spy observe B's access latency. The trojan spawns two reader threads on two different cores, and lets both of these trojan threads access B such that the LLC will record at least two *1*'s in its *core valid bits* vector. When the spy generates a read miss on B, its miss is serviced by the LLC since a clean copy will be available there.

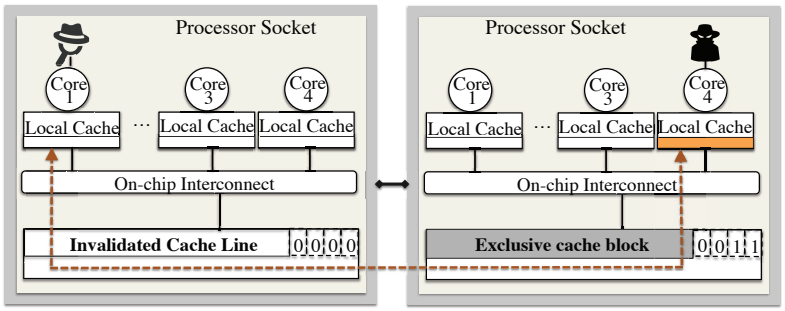
On a similar note, to intentionally place B in *E* coherence state, B will be flushed from all coherent caches. The trojan spawns one reader thread, that will then place a read miss for B. The LLC's *core valid bits* vector will record that only one sharer exists for B. When the spy generates a read miss on B, its miss will routed to the trojan's local (private) cache. The spy's read miss on a cache block in E state creates a different latency profile compared to a read miss on B that is in *S* state.

6.2 Inter-socket Cache Coherence

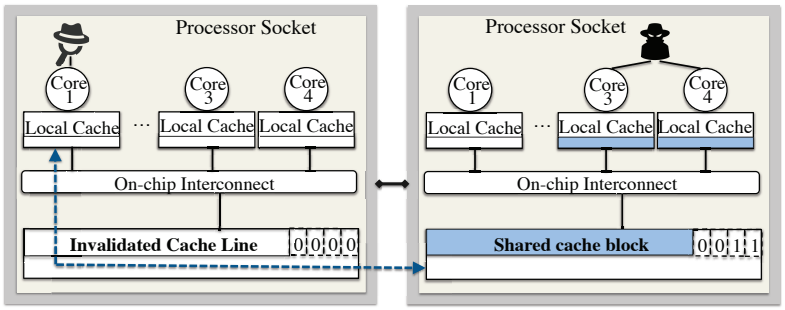
Inter-socket cache coherence has been implemented through high speed point-to-point links, e.g., AMD's HyperTransport bus [16], and the Intel's Quick Path Interconnect [15]. Such high speed links provide for efficient data sharing between the sockets including the ability to maintain coherence between the caches.

The inter-socket coherence works similar to the intra-socket cache coherence (see Section 6.1) with slight modifications to how the data miss requests are handled. When a core requesting a cache block B generates a read miss and the corresponding core's LLC does not have B, the read miss request is sent to other remote sockets first instead of DRAM.

If B is in *S* state in a remote socket, then a clean copy of B is present in the corresponding remote LLC. The data reply is sent back from this remote LLC to the requesting core's LLC that is then propagated up the memory hierarchy to the requestor core. If B is in *E* state in a remote socket, then the corresponding remote



(a) Cache block in E state



(b) Cache block in S state

Fig. 3: Trojan (on the right side) explicitly controlling Cache Coherence States as E or S by running on one or two cores within the multi-socket, multi-core processor. The dotted lines show the service path of the spy (on the left side) for a data block residing in E and S states respectively.

LLC routes the data miss request up to the remote core, which then responds with the data reply. The remote core then downgrades its cache copy to *S* state.

Similar to covert timing channels exploiting on-chip coherence states, the trojan-spy pair can exploit block B's presence in *E* or *S* states in remote caches and the resulting access timing differences. Figure 3 shows an illustration of this scenario. To explicitly place a block B in *S* state on a remote cache, all existing copies of B must be flushed from all of the caches (through *clflush* or an equivalent instruction, or through eviction of all the ways in the set [7]). The trojan spawns two threads of itself on one of the sockets participating in hardware cache coherence, and places a block in *S* state similar to how we described for the intra-socket scenario (see Section 6.1). On a different socket, the spy spawns its thread, and generates a read miss to B to observe its access latency. To explicitly place B in *E* state on a remote cache, all existing copies of B are flushed. The trojan spawns its thread on one of the coherent sockets, and places the block in *E* state similar to how we described for the intra-socket scenario (see Section 6.1). On a different socket, the spy spawns its thread, and generates a read miss to B in order to observe its latency.

Algorithm 1: Trojan Communication Protocol

```
   Input: read-only cache block: B, Txbit[], CS_c, CS_b;
 1 //CS_c is the coherence state used in bit communication;
 2 //CS_b is the coherence state used for bit boundary;
 3 spawn trojan threads;
 4 synchronize with spy using shared cache block, B;
 5 //B could be created implicitly via KSM or through explicitly
 6 //shared data or library code;
 7 //Spy-trojan communication protocol defines three counters:
 8 //C_1, C_0 and C_b for communicating 1, 0 and boundary respectively;
 9 i = 0;
10 while Txbit[i] != -1 do
11     Repeat C_b times: put B in CS_b state;
12     if Txbit[i] == 1 then
13         Repeat C_1 times: put B in CS_c state;
14     else
15         Repeat C_0 times: put B in CS_c state;
16     i++;
```

7 Construction of Timing Channels exploiting Coherence States

We show a template for constructing trojan and spy that can be eventually integrated into a real-world adversaries to exfiltrate sensitive secrets. For example, let us say that a spy has the capability to observe any communication transmitted over a public network between two processes with access to sensitive information. Even if the the spy cannot communicate directly with either of these entities, nor may it be able to decipher the communicated bits especially when the communication is encrypted. However, a malicious insider trojan (that has access to secrets) could covertly communicate with the spy. Specifically, the trojan could transmit the secrets (such as cryptographic keys) stealthily to the spy by modulating accesses to the coherent caches on shared physical memory blocks.

7.1 Trojan and Spy

The trojan and spy pick a (location, coherence state) combination pair to modulate timing and communicate bits (1 or 0), and another distinct (location, coherence state) combination pair to separate individual bit transmission (i.e., to say that a bit transmission has finished and another will start after the boundary). These two combination pairs are denoted as CS_c and CS_b respectively, where c stands for communication and b denotes boundary. Correspondingly, we assume that the bands of cache access latency values T_c and T_b are already known to the trojan and spy through self-measurements on cache hardware (similar to our experimental results in Figure 1). Within the bit transmission period, the trojan and spy will also know how many consecutive times a block B will be seen in CS_c state to distinguish between the transmission of bits '1' and '0', denoted by C_1 and C_0 respectively. We note that having

Algorithm 2: Spy Communication Protocol

```
   Input: read-only cache block: B, Tvalues[]=-1;
 1 //Two access latency bands, T_c and T_b;
 2 //T_s is the sampling interval;
 3 //wait for the trojan to begin transmission;
 4 synchronize with trojan using shared cache block, B;
 5 //B could be created implicitly via KSM or through
 6 //explicitly shared data or library code;
 7 while true do
 8     flush B from cache;
 9     //wait for T_s sec until trojan has an opportunity to reload;
10     load B and time the load (T);
11     if T is within T_b then
12         //transmission has started;
13         break;
14 //reception period
15 i = 0;
16 while true do
17     flush B from cache;
18     //wait for T_s for trojan to reload;
19     load B and time the load (T);
20     record T into Tvalues[i++];
21     if T is outside of T_c and T_b for N consecutive times then
22         //N is defined by the trojan and spy;
23         break;
24 //translation period (interpret 1's and 0's)
25 read Tvalues[] vector from index 0 to N;
26 i = 0; j = 0; count[] = 0;
27 while Tvalues[i]! = -1 do
28     Repeat until Tvalues[i] is within T_b band: i++;
29     bit_c = 0;
30     Repeat until Tvalues[i] is within T_c band: bit_c++; i++;
31     count[j++] = bit_c;
32 //Thold, Threshold separates C_1 and C_0 and helps decipher bits;
33 j = 0;
34 while count[j] != 0 do
35     if count[j++] > Thold then
36         //Infer that the transmitted bit is 1;
37     else
38         //Infer that the transmitted bit is 0;
```

distinct communication and boundary values remove the need for synchronization on each bit transmission.

Algorithm 1 shows the trojan's implementation. The trojan is assumed to have multi-threads that can *explicitly* control the placement of blocks in S or E state either locally or remotely. For every '1' bit to be transmitted, the trojan puts the cache block in CS_c coherence state for C_1 times, and for every '0' bit transmission, the trojan places the cache block in CS_c for C_0 times. Between every bit transmission, the trojan places the cache block in CS_b for C_b times to denote bit boundaries.

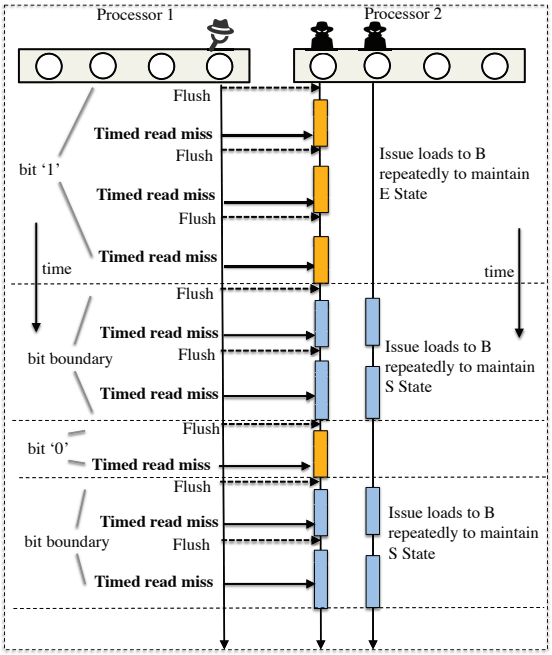


Fig. 4: Illustrative example of '1' and '0' transmission protocol between trojan(s) and spy.

Cache Location and Coherence State for bit communication and boundary	Notation	Number of Trojan threads
(Local Exclusive, Local Shared)	$LExcl_c - LShared_b$	2 (local)
(Remote Exclusive, Remote Shared)	$RExcl_c - RShared_b$	2 (remote)
(Remote Exclusive, Local Exclusive)	$RExcl_c - LExcl_b$	2 (1 local, 1 remote)
(Remote Exclusive, Local Shared)	$RExcl_c - LShared_b$	3 (2 local, 1 remote)
(Remote Shared, Local Exclusive)	$RShared_c - LExcl_b$	3 (1 local, 2 remote)
(Remote Shared, Local Shared)	$RShared_c - LShared_b$	4 (2 local, 2 remote)

Table 2: Trojan implementation along with states used for bit communication and boundary. 'Remote' and 'Local' are with respect to the spy's location.

Algorithm 2 shows spy's implementation. The spy is a single-threaded observer that times the cache block accesses using repeated patterns of flushes and reloads on them. We see that the spy has three phases: 1. Polling for start of transmission by repeated flush and reload of a shared block B. 2. Reception of transmitted bits by timing each access to B, and recording latencies into *Tvalues[]* vector. 3. Translation of *Tvalues[]* by accumulating the consecutive T values belonging to the same band, and distinguishing them into bits '1' and '0', and 'bit boundaries'.

Figure 4 gives a diagrammatic illustration of an *example communication protocol between the trojan and spy*. In this example, the trojan is located in a processor different from that of the spy (Note that the trojan and spy may potentially be in the

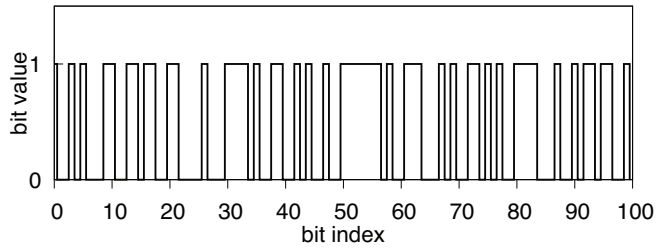


Fig. 5: Bit pattern (100 bits) covertly transmitted by the trojan.

same processor as well). The trojan modulates the cache access timing for the spy by placing B in *E* state when it wants to transmit a bit, and through placing B in *S* state to indicate boundaries between bits. The trojan spawns 2 threads on the remote socket, and issues load requests to B from just one thread to explicitly place it in *E* state and issues load requests to B from both threads to explicitly place it in *S* state. In particular, between cache block flushes initiated by the spy, the trojan places B in *E* state for 3 consecutive times to signal a '1' bit, and places B in *E* state for just 1 time to signal a '0' bit. For bit boundaries, the trojan places B in *S* state for 2 consecutive times between flushes initiated by the spy.

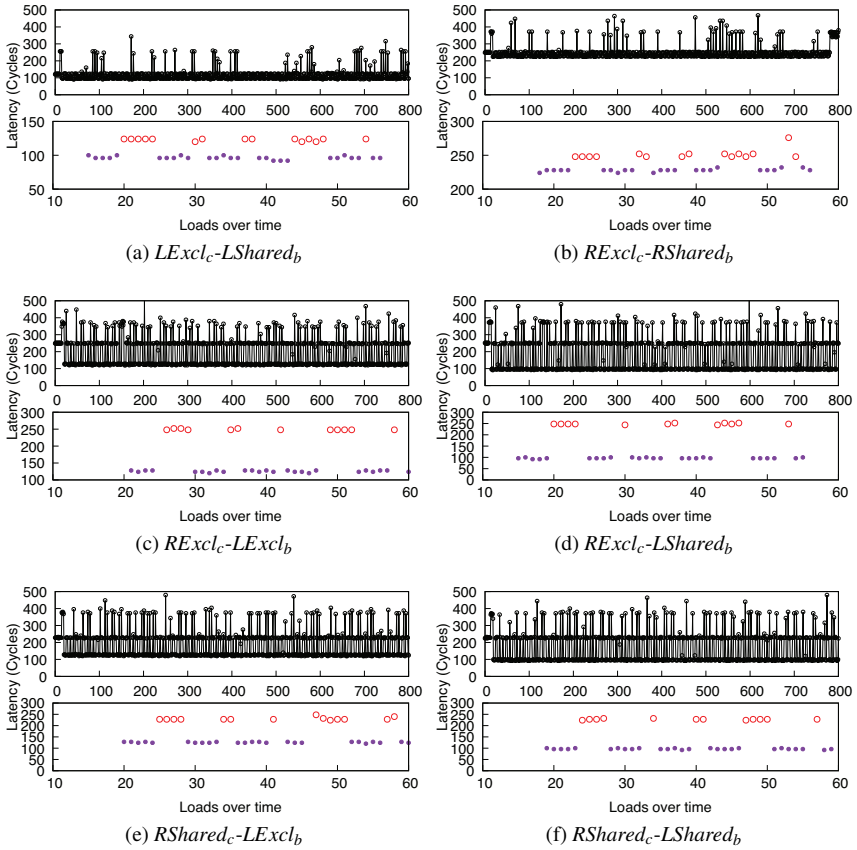
Table 2 shows 6 cases where the trojan and spy use two distinct (location, coherence state) combination pairs for bit transmission and bit boundary identification. The location identifiers 'local' and 'remote' are with respect to the spy, since it measures the load latencies and deciphers the bit values/boundaries on its end.

8 Timing Channel Demonstration

We conduct experiments on a Intel Xeon X5650 2-socket server with a total of 12 cores, the configuration described in Section 5. We pin the trojan and spy threads onto specific cores using the *sched_setaffinity* API. All of the reported load latencies were obtained by inserting the *rdtsc* instruction. We implement the 6 attack scenarios listed in Table 2 and study their bandwidths.

8.1 Trojan-Spy Covert Communication

We show the bit pattern that the trojan intends to covertly communicate with the spy in Figure 5. Figure 6 shows the profile of load latencies as observed by the spy on its side. For each combination pair of (location, coherence state), we show two sets of results: the top portion shows the load latencies observed throughout the entire reception period, and bottom portion shows a magnified view illustrating the communication of the first five bits in the top figure for clarity. In this magnified view, we observe that for each '1' bit transmitted, the spy observes the load latency in the T_c band, corresponding to CS_c, for four or five consecutive times (each dot in the figure denotes a 'timed' load operation); for each '0' bit transmitted, the spy observes



(a) $LExcl_c$-$LShared_b$ (b) $RExcl_c$-$RShared_b$

(c) $RExcl_c$-$LExcl_b$ (d) $RExcl_c$-$LShared_b$

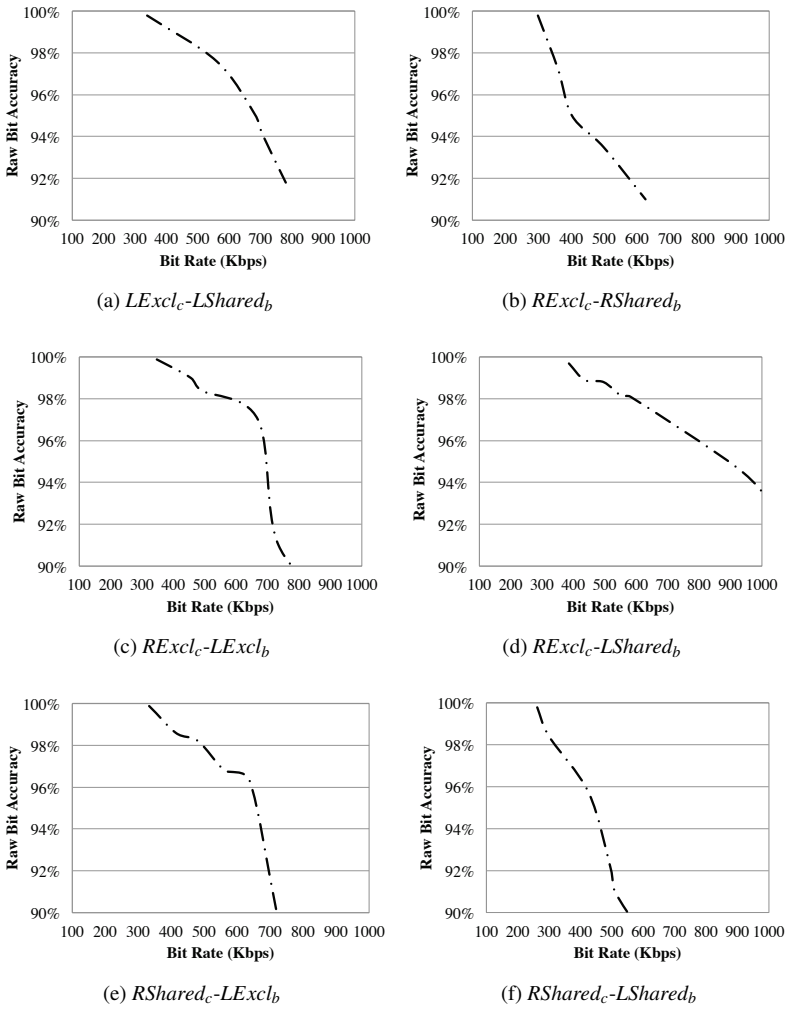
(e) $RShared_c$-$LExcl_b$ (f) $RShared_c$-$LShared_b$

Fig. 6: Bit Reception by the Spy through measuring load latency (in CPU cycles). The top portion in each subfigure shows the entire reception period, and the bottom portion shows a magnified view for the reception of first five bits.

load latency in the T_c band for one or two consecutive times (See discussion in Section 7.1). These are shown as red dots in the bottom portion of each figure. Similarly, the boundary between bit values are deciphered by the spy when it observes load latency in the T_b band, corresponding to CS_b, for four to five times consecutively. Our experiments show that the spy is able to correctly decipher the transmitted bits for all 6 attack scenarios with 100% accuracy.

8.2 Transmission Bandwidth vs. Raw Bit Accuracy

To study the raw bit accuracy with increasing transmission bit rates between the trojan and spy, we perform experiments by tuning two knobs: 1. Reduce the number of consecutive caching operations for shared blocks that communicate bit values and



(a) $LExcl_c$-$LShared_b$ (b) $RExcl_c$-$RShared_b$

(c) $RExcl_c$-$LExcl_b$ (d) $RExcl_c$-$LShared_b$

(e) $RShared_c$-$LExcl_b$ (f) $RShared_c$-$LShared_b$

Fig. 7: Raw bit accuracy as captured by the spy with increase in transmission rates.

boundaries, i.e., values of C_1, C_0 and C_b. 2. Reduce the interval between shared cache block loads by the spy, i.e., the value of T_s. Algorithms 1 and 2 for further details on these parameters. Figure 7 shows our results. In this study, we note that there are 3 possibilities for raw bit error on the reception side: 1. certain bits may be lost, 2. extra bits may be added due to duplication (very rare and we did not observe any such occurrence in our experiments), and 3. certain bits may be flipped (1 mis-interpreted as 0, or vice versa). Accuracy is defined as the ratio of number of raw bits correctly received by the spy to total number of raw bits transmitted by the trojan. As we increase the bit rate to beyond 500 Kbps, we see that most cases experience a rapid drop in raw bit accuracy. However, there are two exceptions: 1. $RExcl_c - LExcl_b$ be-

gins with a high initial bit rate of over 400 Kbps and declines to below 90% accuracy only beyond 800 Kbps. 2. $RExcl_c - LShared_b$ shows high immunity and a good raw bit accuracy of 96% even at 800 Kbps. We note that the *effective 'information bit' accuracy rates* can be kept potentially high by leveraging higher raw bit transmission rates especially when the underlying transmission protocol incorporates error correcting codes. Methods to recover information bits due to omission and bit flips is a well studied topic [21], and is outside the scope of our work.

9 Vulnerability Analysis on Variants of Coherence Protocols

Coherence Protocol and Cache Inclusiveness	*Exclusive* Cache Block	*Shared* Cache Block
Snoopy, Inclusive	Requestor→Owner cache → Requestor	Requestor→LLC →Requestor
Snoopy, Non-inclusive	Requester→Owner cache →Requestor	Requestor→MemCtrl →Requestor
Directory, Inclusive	Requestor→Directory →Owner cache→Requestor	Requestor→LLC →Requestor
Directory, Non-inclusive	Requestor→Directory →Owner cache→Requestor	Requestor→MemCtrl →Requestor

Table 3: Sequence of coherence controllers that interact in order to service the cache blocks in *E* and *S* state under different classes of cache coherence protocols. 'LLC' and 'MemCtrl' denote Last Level Cache and Memory Controller respectively.

The processors used in our evaluation deploy a variant of directory-based coherence protocol (with LLC's core-valid-bits) that directs the coherence messages to specific cores in order to service these cache misses. In this section, we systematically study the coherence state vulnerabilities in different variants of coherence protocols.

Two factors play a key role in determining the cache access latency, namely the family of coherence protocols and the inclusiveness property of caches. Different coherence protocol implementations have varying set of transactions when accessing a cache block in a specific coherence state. When a requestor cache controller issues a cache miss request for a block, coherence messages are sent over the interconnection fabric. The owner (e.g., cache controllers, coherence directory or memory controller), that *owns* the requested block, will respond with the data reply. Generally, for an E state cache block, the cache controller that currently holds the private copy of the date is designated as the owner, and responds with the data reply. Differently, the cache directory (usually, LLC) or the memory controller typically own cache blocks in S state. Table 3 illustrates the sequence of coherence controllers associated with

servicing data miss requests on E- and S-state cache blocks in the four variants of cache coherence protocols[2].

For snoopy protocols run on inclusive caches, read operations on E-state blocks will involve snooping into the private owner cache, while reads on S-state blocks are satisfied by the lower level LLC that has a clean copy of the cache block and acts as the owner [17]. Although both coherence transactions need two hops, they involve different paths. As a result, the cache access latencies for S and E states can be distinguished by adversaries that are monitoring for such information. Similarly, for directory-based protocols run on inclusive caches, a read on E state cache blocks requires a coherence request first sent to the directory module (that is typically maintained in the LLC on many modern processors). The directory then forwards the request to the owner cache, which will subsequently respond with the data. On the other hand, reads on S state cache blocks are replied by the LLC. These two coherence transactions differ in the hops traversed, which results in the distinct latency bands as demonstrated in Table 1.

Additionally, the cache inclusion policy influences read operations to S-state cache blocks. Specifically, for inclusive caches, the LLC always owns the S-state blocks and will respond to cache controller directly with the data. In non-inclusive caches, the memory controller is set to own the cache block and accordingly requests to S-state blocks will be serviced by the main memory since the LLC may not potentially have a copy. Such design decisions are made to avoid multiple data transfers from the various sharers.

In summary, we note that all four variants of cache coherence protocols can be vulnerable to exploits due to differences in their timing profile (as discussed in Section 6).

10 Securing Cache Coherence Protocols

As we know, E-state aims to reduce the coherence transactions and the corresponding latency for writes that immediately follow the read operation to that memory block. Most existing cache coherence protocols allow cache blocks to transition from E to M state without initiating coherence transactions (silent upgrade). Due to this optimization, the cache directory and memory controller will not own these E-state blocks and assume that their data copies are stale.

As discussed in Section 9, across all the four variants of cache coherence implementations, private caches claim ownership of E-state blocks. These design considerations result in latency differences corresponding to the read-only coherence states, namely E and S, and potentially enable construction of timing channels using read-only states.

[2] When the LLC holds a copy of the cache block, the coherence transactions on non-inclusive caches are similar to that of exclusive caches. Therefore, the coherence transactions we listed for non-inclusive caches in Table 3 may be applied to a strictly exclusive cache hierarchy as well.

10.1 Modifying E→M Transition

To remove the read latency differences between E and S states, a potential solution is to service the read requests to cache blocks in read-only states (E and S) uniformly by the directory (or memory controller). This means that all E→M upgrade requests from the cores should be forwarded onto the directory or memory controller every time. This makes the LLC to be aware of the precise coherence state for the corresponding data block (i.e., helps distinguish E vs. M). The LLC can then store the correct coherence state for that block. This enables the LLC to have ownership of E-state cache blocks, since they are guaranteed to be clean until being notified by the owner core. Also, the read requests to both E- and S- state blocks can now be serviced by the LLC, and the *read* timing difference between these two states for an external requestor will be zero.

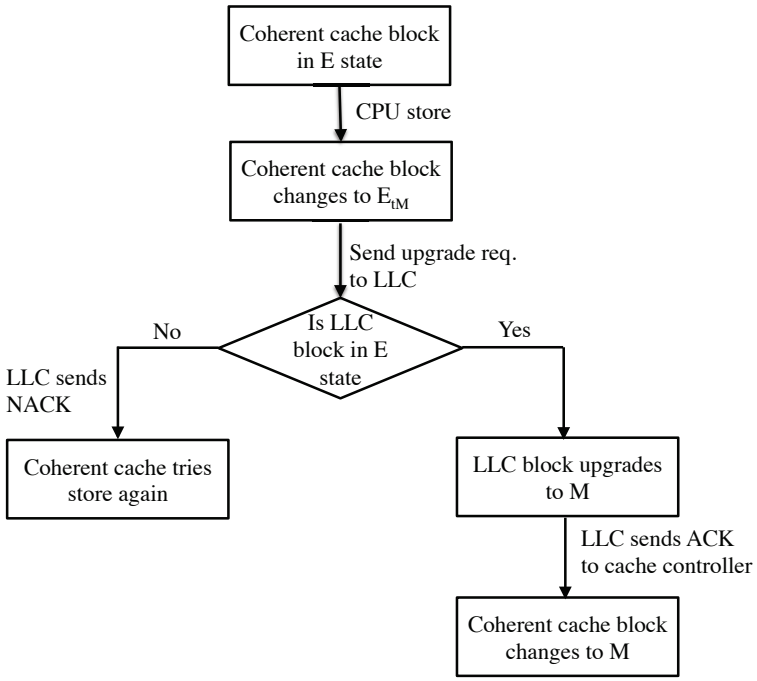


Fig. 8: Handling E→M transition in directory-based protocols. *Coherent Cache* denotes private caches kept coherent using the coherence protocol hardware.

Modifications needed in Directory-based protocols. In order to make LLCs own E- and S- state cache blocks, directory-based protocols need an additional transient coherence state. Figure 8 shows our solution approach. Upon receiving a write command on E-state blocks in the private coherent caches, the corresponding coherence state transitions to E_{tM}. The write upgrade request is then forwarded to the LLC that maintains the directory information corresponding to the cache block. If the current

Coherence State	Min Latency	Average Latency	Max Latency
Exclusive & Shared	119	**119.4**	120

Table 4: Load operation latency (Cycles) for S- and E- state blocks within the socket using the modified directory-based protocol.

coherence state in the LLC is also E, then the LLC modifies its coherence state to M, approves the E→M, and forwards the acknowledgment to the requestor core. Otherwise, the write request is denied, and the requestor core re-initiates the write operation all over again by sending invalidation requests to other cores.

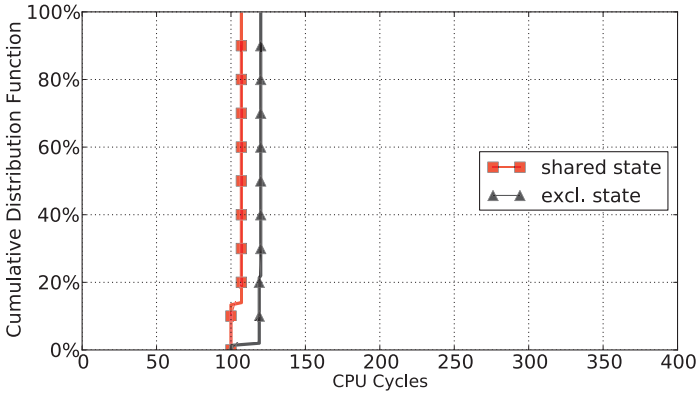
Modifications needed in Snoopy-based protocols. In snoopy protocols, write upgrade requests are typically sent over the system bus when transitioning from S to E state. Since memory controller monitors all of coherence traffic on the system bus, we note that the following modifications can be made to avoid read latency differences between E and S-state blocks. 1. All read requests to E- and S-state blocks can be replied directly by the memory controller. 2. The upgrade miss requests can be issued for E-state blocks for E→M transitions, instead of S-state blocks for S→E transitions.

10.2 Latency Profiles with the Modified Coherence Protocol

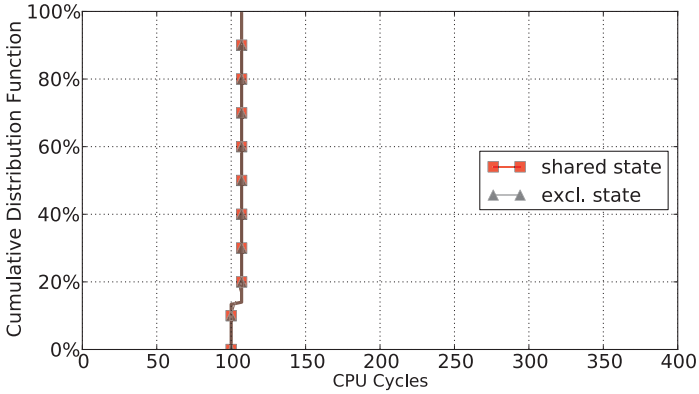
To evaluate the effectiveness of our proposed mechanism, we model the modified E→M coherence transition and measure the latency profiles using Gem5, a cycle-accurate full system simulator [22]. We configure Gem5 with eight x86 cores, and use a minimal Linux distribution with kernel version 2.6.32. Each core has a 32 KB private L1 and all cores share a 2MB L2 cache. The microbenchmark (described in Section 5) is used to profile the cache access latencies. Figure 9 shows the CDFs of cache access latencies for the E- and S- state cache blocks under the original coherence protocol and the modified coherence protocol with changes to E-state cache blocks. We can see that under the original MESI-based protocol, the latency profiles for E and S cache blocks accesses are easily distinguishable as the distributions form two narrow bands that do not overlap (Figure 9a). Figure 9b demonstrates the same latency profiles in the modified protocol that aims to close the latency gap between the E and S cache block accesses. In fact, the two distribution are exactly the same as the E and S cache block accesses now involve the same coherent transactions. Table 4 lists the latency statistics including minimum, average and maximum latencies for accessing E- and S-state blocks. Obviously, an attacker would not be able to build a covert channel by manipulating the two latency values.

10.3 Implications on Application Performance

The modifications to the cache coherence protocol require additional messages sent to the directory or memory to upgrade cache blocks from E to M as writes to E-state block will be blocked before the upgrade transaction is completed. This may



(a) Original protocol



(b) Modified protocol

Fig. 9: Distributions of latencies for accessing E- and S- state cache blocks under original MESI protocol and the modified protocol with changes to E-state cache blocks

potentially affect the application performance. To evaluate the performance overhead involved in the modified coherence protocol, we run several multi-threaded PARSEC benchmarks [23] that have various levels of cache coherence activities. Each benchmark is configured to run with four threads. Figure 10 shows the performance overheads in terms of execution time for each benchmark's region of interest (ROI). Notably, we observe less than 0.5% overhead for these applications. The performance impact is negligible for the following two reasons: First, the number of stores to E-state cache blocks is only a relatively small portion of all store instructions; Second, the additional transaction for the write to E-state block is lightweight as it only involves notifications to the last level cache, unlike writes to S-state blocks that typically generate *getM* requests as well as invalidation messages [17]. Specifically, we see that *blackscholes* has very few stores to E-state blocks and our mechanism only incurs less than 0.01% overhead. On the other hand, *fluidanimate* performs a consid-

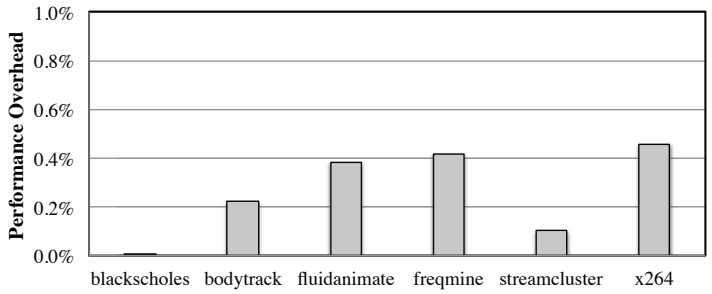


Fig. 10: Performance overhead for the modified cache coherence protocol in PARSEC benchmarks

erable number of writes to E-state blocks (that introduce longer latencies) and as well as many reads to remote E-state blocks (that have reduced delays), and the overall influence of the modified coherence protocol is less than 0.4%.

Moreover, our secure cache coherence is designed to protect the systems where untrusted processes are running on the same machine and are sharing copy-on-write pages (e.g., through KSM). We note that under this context, the latency of E-block upgrade is essentially hidden by the latency of copying the physical page during the first write operation to copy-on-write page. To avoid performance slowdown in regular applications, a simple switch between the performance version (unmodified protocol) and secure version (our modified protocol) can be designed in hardware such that we can achieve trade off between performance and security. When the switch is enabled, E→M transitions will undergo additional steps before actual transition that are described in our Section 10.1.

11 Related Work

Prior work have demonstrated timing channels that exploit cache hardware [8,7,24,6, 25,26], functional units [27,28], processor-memory bus [29], frequency settings [30] and branch predictors [31,32]. Jiang et al. [33] showed how to construct a side channel that recovers AES encryption keys on GPUs. Most of these prior works rely on modulating the access timing behavior of a specific hardware resource that may potentially be addressed through *carefully monitoring that unit*, and if it is feasible, by isolating or disabling them. Differently, our work characterizes and studies the defense against an attack that leverages the hardware cache coherence protocol operating on multiple caches and coherence states.

Yao et al. [26] showed how to construct covert timing channels by leveraging non-uniform memory access latencies in multiple sockets. In contrast to this attack, the proposed covert timing channel exploits combinations of coherence states and cache location in order to construct timing channels. Also, almost all of the prior adversary models [9,26] rely on user-initiated shared cache-blocks (via shared system libraries). We show a more sophisticated adversary model where shared physical memory could also be created using memory deduplication mechanisms such as KSM.

Many prior works have studied the detection and defense techniques for covert/side channel attacks. Demme et al. [34] introduced a metric to quantify the difficulty level to exploit a system for side channels. Wang et al. [12] proposed secure hardware cache designs with partition-locking and random permutation. Venkataramani et al. [2,35,36] have proposed techniques that detect contention-based timing channels in functional units and caches. Hunger et al. [37] also studied contention-based cache timing channels and proposed anomaly-based detection. Yan et al. [38] build a record and replay framework that detects covert timing channels by analyzing the difference of caches miss patterns observed from the record and replay runs. In terms of defense, Liu et al. [13] studied mechanisms that use cache allocation technology (CAT) to create secure cache partitioning to prevent information leakage from victim processes. Prefetch-guard [39] leverages hardware prefetchers to obfuscate the latency measurements for the trojan process in covert timing channels. Several other works have proposed mechanisms that offer memory safety protection using hardware support for memory access monitoring and tainting [40,41]. These mechanisms can be effectively leveraged to protect systems from covert storage channels. To defend against memory-based timing channels, Ferraiuolo et al. [42] designed a secure memory scheduling algorithm. Camouflage [43] reshapes the timing of memory requests and responses to a deterministic distribution that eliminate memory access pattern snooping by malicious users. Recent work [44] leverages computation logic in emerging memory technology to cryptographically obfuscate memory addresses and memory bus timing to defend against memory bus attacks. We note that these prior defense mechanisms are not designed to protect system-wide coherence protocols that span multiple caches and take advantage of their coherence mechanisms.

12 Conclusions

In this article, we presented an important vulnerability exposed by a frequently-used performance enhancing feature in many modern multi-core and multi-socket systems, namely cache coherence protocols. We showed how adversaries could exploit cache coherence states and construct covert timing channels in order to illegally exfiltrate sensitive secrets to untrusted parties. We demonstrated six practical cases for covert timing channels on real-world commercial processors. In contrast to prior works, we assume a broader adversary model where the trojan and spy can either exploit explicitly shared read-only physical pages, or use memory deduplication feature to implicitly force create shared physical pages. We demonstrate how adversaries can manipulate combinations of coherence states and data placement in different caches to construct timing channels. Moreover, we performed vulnerabilities analysis on multiple variants of coherence protocols and showed that all the major variants are subject to the attacks under study. More importantly, we demonstrated our defense mechanism with slight modifications to read operations on exclusive cache blocks. This removes the read latency difference between read-only coherence states, and obstructs the adversaries from taking advantage of these states to implement their timing channels. We evaluated the performance impact of our proposed changes to the cache coherence protocols for multi-threaded benign applications. The experimental results

revealed that our modified coherence protocol introduces minimal performance overhead.

Acknowledgment

This material is based on work supported by the US National Science Foundation under CAREER Award CCF- 1149557 and CNS-1618786, and Semiconductor Research Corp. (SRC) contract 2016-TS-2684. Any opinions, findings, conclusions, or recommendations expressed in this article are those of the authors, and do not necessarily reflect those of the NSF or SRC.

References

1. Department of Defense Standard, *Trusted Computer System Evaluation Criteria*. US Department of Defense, 1983.
2. G. Venkataramani, J. Chen, and M. Doroslovacki, "Detecting hardware covert timing channels," *IEEE Micro*, vol. 36, pp. 17–27, Sept 2016.
3. A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, L. T. X. Phan, M. Sherr, and W. Zhou, "Detecting covert timing channels with time-deterministic replay," in *USENIX Symposium on Operating Systems Design and Implementation*, pp. 541–554, 2014.
4. O. Acıiçmez, B. B. Brumley, and P. Grabher, "New results on instruction cache attacks," in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 110–124, Springer, 2010.
5. O. Aciiçmez, "Yet another microarchitectural attack: exploiting I-cache," in *Proceedings of ACM Workshop on Computer Security Architecture*, pp. 11–18, ACM, 2007.
6. Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of L2 cache covert channels in virtualized environments," in *Proceedings of ACM Workshop on Cloud Computing Security*, pp. 29–40, ACM, 2011.
7. F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proceedings of IEEE Symposium on Security and Privacy*, pp. 605–622, IEEE, 2015.
8. Y. Yarom and K. Falkner, "Flush+ reload: a high resolution, low noise, L3 cache side-channel attack," in *USENIX Security Symposium*, pp. 719–732, 2014.
9. G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Proceedings of ACM Asia Conference on Computer and Communications Security*, pp. 353–364, ACM, 2016.
10. F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, pp. 168–179, IEEE, 2018.
11. F. Liu and R. B. Lee, "Random fill cache architecture," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pp. 203–215, IEEE, 2014.
12. Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 494–505, 2007.
13. F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, pp. 406–418, IEEE, 2016.
14. Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: secure dynamic cache partitioning for efficient timing channel protection," in *Proceedings of IEEE Design Automation Conference*, pp. 1–6, IEEE, 2016.
15. "Intel QuickPath Architecture," 2012. `http://www.intel.com/pressroom/archive/reference/whitepaper_QuickPath.pdf`.
16. P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache hierarchy and memory subsystem of the AMD Opteron processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, 2010.
17. D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.

18. C. A. Waldspurger, "Memory resource management in VMware ESX server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.
19. A. Barresi, K. Razavi, M. Payer, and T. R. Gross, "CAIN: silently breaking ASLR in the cloud," in *USENIX Workshop on Offensive Technologies*, 2015.
20. "Using Intel VTune Amplifier," 2013. `https://goo.gl/E9Fp2m`.
21. R. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962.
22. N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.
23. C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of ACM International Conference on Parallel Architectures and Compilation Techniques*, pp. 72–81, ACM, 2008.
24. D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security Symposium*, pp. 897–912, 2015.
25. T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of ACM Conference on Computer and Communications Security*, pp. 199–212, ACM, 2009.
26. F. Yao, G. Venkataramani, and M. Doroslovacki, "Covert timing channels exploiting non-uniform memory access based architectures," in *Proceedings of ACM Great Lakes Symposium on VLSI*, pp. 155–160, ACM, 2017.
27. O. Aciicmez and J.-P. Seifert, "Cheap hardware parallelism implies cheap security," in *Proceedings of IEEE Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 80–91, IEEE, 2007.
28. D. Evtyushkin and D. Ponomarev, "Covert channels through random number generator: Mechanisms, capacity estimation and mitigations," in *Proceedings of ACM Conference on Computer and Communications Security*, pp. 843–857, ACM, 2016.
29. Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: high-speed covert channel attacks in the cloud," in *USENIX Security Symposium*, pp. 159–173, 2012.
30. M. Alagappan, J. J. Rajendran, M. Doroslovacki, and G. Venkataramani, "DFS covert channels on multi-core platforms," in *Proceedings of IEEE International Conference on Very Large Scale Integration*, IEEE, 2017.
31. O. Aciiçmez, c. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of ACM Symposium on Information, Computer and Communications Security*, pp. 312–320, ACM, 2007.
32. D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Understanding and mitigating covert channels through branch predictors," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 1, p. 10, 2016.
33. Z. H. Jiang, Y. Fei, and D. Kaeli, "A complete key recovery timing attack on a GPU," in *Proceeding of IEEE International Symposium on High Performance Computer Architecture*, pp. 394–405, IEEE, 2016.
34. J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: a metric for measuring information leakage," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 106–117, 2012.
35. J. Chen and G. Venkataramani, "An algorithm for detecting contention-based covert timing channels on shared hardware," in *Proceedings of ACM Workshop on Hardware and Architectural Support for Security and Privacy*, ACM, 2014.
36. J. Chen and G. Venkataramani, "Cc-hunter: Uncovering covert timing channels on shared processor hardware," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pp. 216–228, IEEE Computer Society, 2014.
37. C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari, "Understanding contention-based channels and using them for defense," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, pp. 639–650, IEEE, 2015.
38. M. Yan, Y. Shalabi, and J. Torrellas, "ReplayConfusion: Detecting cache-based covert channel attacks using record and replay," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pp. 1–14, IEEE, 2016.
39. H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani, "Prefetch-guard: Leveraging hardware prefetchers to defend against cache timing channels (short paper)," in *Proceedings of IEEE Symposium on Hardware Oriented Security and Trust*, IEEE, 2018.

40. G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Memtracker: An accelerator for memory debugging and monitoring," *ACM Transactions on Architecture and Code Optimization*, 2009.
41. J. Shen, G. Venkataramani, and M. Prvulovic, "Tradeoffs in fine-grained heap memory protection," in *Proceedings of ACM Workshop on Architectural and System Support for Improving Software Dependability*, ACM, 2006.
42. A. Ferraiuolo, Y. Wang, D. Zhang, A. C. Myers, and G. E. Suh, "Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, pp. 382–393, IEEE, 2016.
43. Y. Zhou, S. Wagh, P. Mittal, and D. Wentzlaff, "Camouflage: Memory traffic shaping to mitigate timing attacks," in *Proceedings of International Symposium on High Performance Computer Architecture*, pp. 337–348, IEEE, 2017.
44. A. Awad, Y. Wang, D. Shands, and Y. Solihin, "Obfusmem: A low-overhead access obfuscation for trusted memories," in *Proceedings of ACM International Symposium on Computer Architecture*, pp. 107–119, ACM, 2017.