Are Coherence Protocol States Vulnerable to Information Leakage?

Fan Yao, Miloš Doroslovački and Guru Venkataramani
Department of Electrical and Computer Engineering,
The George Washington University, Washington, DC, USA
Email: {albertyao, doroslov, guruv}@gwu.edu

***Abstract*—Most commercial multi-core processors incorporate hardware coherence protocols to support efficient data transfers and updates between their constituent cores. While hardware coherence protocols provide immense benefits for application performance by removing the burden of software-based coherence, we note that understanding the security vulnerabilities posed by such oft-used, widely-adopted processor features is critical for secure processor designs in the future.**

In this paper, we demonstrate a new vulnerability exposed by cache coherence protocol states. We present novel insights into how adversaries could cleverly manipulate the coherence states on shared cache blocks, and construct covert timing channels to illegitimately communicate secrets to the spy. We demonstrate 6 different practical scenarios for covert timing channel construction. 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 (e.g., shared library code), 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. We also explore how adversaries could exploit multiple caches and their associated coherence states to improve transmission bandwidth with symbols encoding multiple bits. Our experimental results on commercial systems show that the peak transmission bandwidths of these covert timing channels can vary between 700 to 1100 Kbits/sec. To the best of our knowledge, our study is the first to highlight the vulnerability of hardware cache coherence protocols to timing channels that can help computer architects to craft effective defenses against exploits on such critical processor features.

***Keywords*-coherence protocols; covert timing channels; information leakage; hardware security**

I. Introduction

Cyber attacks, that exploit malicious insiders and exfiltrate secret information, are a growing concern for computer users. Covert channels are one such class of insider threats, where a trojan process, that has access to sensitive user-related information (e.g., user's personal data), secretly exfiltrates the data to a spy process *even when* the underlying system security policy explicitly prohibits any such communication [1]. Also, note that the trojan cannot directly reveal secrets to the outside world (since system security auditors can be easily catch such activity), and has to rely on covert modes of operation to exfiltrate 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.

Among several covert channel implementations, timing-based attacks are extremely stealthy since the trojan and spy communicate simply by manipulating the access timing to shared resources. These covert timing channels leave no physical trace of an attack, and are extremely difficult to detect just through forensic examination of the system state [2]. Recently, Flush+Reload [3] demonstrated a side channel that exploits the latency difference between cache hits and DRAM accesses. Irazoqui et al. [4] further extend this work and demonstrate a potential side channel by exploiting 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.*

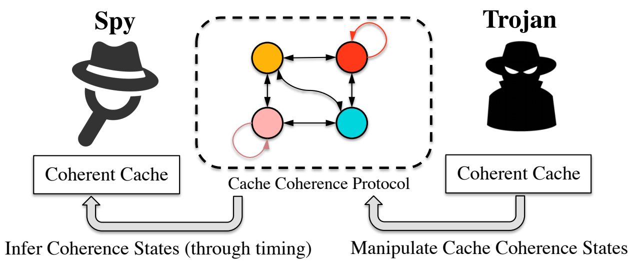

Spy
Trojan
Coherent Cache
Cache Coherence Protocol
Coherent Cache
Infer Coherence States (through timing)
Manipulate Cache Coherence States

Fig. 1: Illustration of a trojan and spy exploiting cache coherence states

In this paper, we *systematically unravel* the vulnerability exposed by *cache coherence states* to covert timing channels (Figure 1). For the first time, we show how *exclusive* and *shared* coherence states may present a significant vulnerability that can be taken advantage by adversaries for covert timing channel construction purposes. In contrast to prior works, we assume a *broader adversary model* where the trojan and the spy can force create coherence transactions through either explicitly created read-only shared physical pages (e.g., shared library code) or implicitly created shared physical memory pages through a OS feature named Kernel Same Page Merging (KSM). Our study presents *novel* insights into the behavioral characteristics of a class of covert timing channels that exploit coherence states, their peak bandwidths, and transmission rates in the presence of external noise.

We note that our study highlights the need to understand the vulnerability exposed by cache coherence protocols, and motivates further studies on how to design performance-friendly defenses against such covert timing channels. Further, we demonstrate attacks using multi-bit symbols through exploiting multiple cache access latency values resulting from combinations of coherence states and cache locations.

Our study is significant for several key reasons. Most modern processor vendors offer support for hardware-based cache coherence, and this trend is likely to stay for the foreseeable future [5]. While hardware cache coherence protocols improve system performance by obviating the need for applications to explicitly maintain data coherence, they could inadvertently possess certain characteristics prone to security breaches. Therefore, understanding the vulnerability exposed by such oft-used hardware features is critical to improving the overall system security. It is worth noting that covert channels, that exploit specific hardware units or accelerator structures, *can be defended by either closely auditing their usage or disabling them altogether where possible*. In sharp contrast, hardware cache coherence-based threats pose challenges due to the following reasons: 1. Coherence protocol encompasses caches from multiple levels and possibly different sockets as well, and hence the attack surface that can be exploited by the adversary is significantly large, 2. Unlike specific hardware functional units, cache coherence mechanisms cannot be simply deactivated since most currently popular and legacy software programming models depend on them. Therefore, our work necessitates a more careful understanding of how coherence protocols could be exploited that could help system designers to devise smarter ways to defend against such channels.

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

1) We present *novel* insights into the vulnerabilities exposed by hardware cache coherence protocols. In particular, we show how the adversaries could exploit coherent cache blocks in *exclusive* and *shared* states present in *different* levels of the cache hierarchy.

2) We illustrate 6 different ways in which the trojan could exploit the read latency differences for cache data blocks, arising from various combinations of coherence states and data locality. We demonstrate the feasibility of covert transmission between trojan and the spy through implementing all of these combinations on real processors.

3) We show experimental results to: a) Analyze the bandwidth capacities of various timing channel implementations, b) Explore multi-bit symbol transmission that encode data by exploiting the latency differences stemming from multiple combinations of coherence states and cache locations (effectively increasing the achievable peak bandwidth from 700 Kbits/sec with binary encoded symbols to about 1100 Kbits/sec with multi-bit encoding). c) Understand the behavior of timing channels in the presence of external noise, and study the effect on their transmission rates when error correction mechanisms are incorporated.

II. Background

A. Covert Channels

Trusted Computer System Evaluation Criteria (TCSEC or Orange Book developed by US Department of Defense) [1] defines covert channel as any communication channel that can be exploited by a process to transfer information in a manner that violates the system's security policy. Among the various types of covert channels, 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. TCSEC notes that covert channels with low bandwidths represent a lower threat than those with higher bandwidths. This is because, lower bandwidth channels become increasingly more expensive for the adversary with diminishing returns in terms of information gain (e.g., the adversary gets almost no useful or meaningful information on covert channels with bandwidth rate of 0.1 bits/sec or below). Based on measurements from several different computer systems, TCSEC classifies a high bandwidth covert channel to have a minimum rate of 100 bits/sec.

B. Cache Coherence

Most modern processors, including Intel Xeon and AMD Opteron families, support *slight* variants of *MESI* cache coherence protocol to preserve data coherence in private caches [6], [7]. The MESI protocol has four states, namely:

1) *Modified* (M) state, where the cache block is present only in one private cache and is dirty, i.e., the data has been modified compared to the value in main memory. This also implies that the current core/processor has write permission to modify the block. 2) *Shared* (S) state, where the cache block is present in more than one private cache and is clean, i.e., the data matches the value in the main memory. This implies that current CPU only has read permission to the block. 3) *Exclusive* (E) state, where the cache block is present only in the current cache, but is clean, i.e., the data matches the contents in main memory. In this state, the current CPU only has read permissions. However, since the cache block is present only in the current cache, it lets the owner CPU to acquire write permissions and upgrade to M state without the need to generate invalidation requests to other cores. Also, any read misses to this block by other cores will downgrade the coherence state in the current CPU to S state. This dual-intent coherence state improves the performance by enabling quick transitions to M or S from the E state depending on the memory operation. 4) *Invalid* (I) state, where the cache block is invalid, and does not have read or write permissions.

Depending on the processor family, there are other specialized cache coherence states to further optimize performance. For instance, the Intel Xeon processor family implements the MESIF protocol, where the F state is meant to designate the processor that will forward the cache block to the requestor. AMD processor family implements MOESI, where O state is created to designate the owner processor after a modified cache block transitions to shared state. This avoids write-back operations to memory whenever the modified blocks are shared between processors. We note that *such additional states simply serve to improve performance, and do not fundamentally add new functionality to M, E, S, or I states*. For clarity, we do not consider such performance-optimizing coherence states in our current work.

Furthermore, Intel Xeon and AMD Opteron support cache coherence across multiple sockets (processors) through high speed links between the processors [6], [7], [8]. This enables multiple processors (multiple CPUs) to share data among them using the underlying coherence protocol.

III. Attack Model

We assume that the trojan and spy are running on the same machine that features *one or more* multi-core processors. In contrast to the attack model assumed by Irazoqui et al. [4] that requires multiple sockets, our demonstrated attacks can work equally well when the trojan and spy are co-located on the same socket. In order to control coherence states, we 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 put a block in shared (S) state by explicitly making two trojan threads load a memory block. The trojan intentionally modulates the cache access timing through placing the shared data block in different coherence states and possibly in different levels of the memory hierarchy (local processor's caches, another processor's caches in a multi-processor). The pattern of timing differences between shared block accesses in different coherence states and locations enables a spy to infer the transmitted bit(s) from the trojan.

We note that trojan just cannot directly reveal secrets since system security auditors can detect such activity and prevent a trojan from doing damages to sensitive user information. Therefore, we assume that a malicious trojan would actively seek to covertly communicate with a spy and exfiltrate information through covert means.

Also, as software confinement mechanisms continue to improve and provide stronger isolation guarantees between processes, hardware structures and their associated mechanisms will be natural targets for covert timing channels. In this vein, we illustrate the vulnerabilities exposed by cache coherence protocols that provide for timing differences in cache accesses depending on the coherence state.

IV. Sharing Physical Memory

Prior to constructing timing channels, the trojan and the spy should first have shared physical memory such that timing of accesses to these addresses can be manipulated. As noted in Section III, prior techniques [3], [4] have shown their timing channel implementations by explicitly sharing library code and data between the trojan and spy. In effect, the coherence protocols would maintain states on such blocks to keep a coherent memory view supported by the underlying hardware. Our attack model would work with a similar setup. However, this setup could imply that we assume the trojan (with access to sensitive data) and the spy (that can't access sensitive data) to have shared code or data, which could be difficult in systems where strict isolation guarantee policies are enforced.

We note that a more agile adversary could circumvent the explicit code or data sharing requirement by exploiting a feature called memory deduplication supported by the OS.

Kernel Same Page Merging (or 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 and avoid having to duplicate physical memory pages holding identical data. In current Linux systems, the KSM is a kernel thread that periodically scans the entire memory to identify identical memory pages and make them to be candidates for merging. After the merging process is over, a single physical copy of the page is kept and all of the duplicate copy pages are updated to point to this single physical page in the 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, preventing any unexpected direct communication between the sharer processes. This feature is widely adopted to compact memory, avoid unnecessary memory duplication and reduce memory page misses [9], [10].

From the above description, it is clear that the processes may begin to share pages *unknowingly behind the scenes* due to the OS merging of identical physical pages belonging to different processes. This feature can be now exploited by the trojan and spy to force create shared memory even without explicitly having to share any library code or data between them. In particular, since covert channels are created by colluding parties, the trojan and spy could intentionally generate physical pages with identical bit patterns known to both of them ahead of time. KSM scans the process memory spaces 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 will have to 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 memory deduplication. If an external sharing of this page is detected (via timing measurements), the trojan and spy may repeat creating shared memory through deduplication using another set of identical bit patterns known to both of them.

V. Coherence States and Latency

To understand the effect of cache coherence states and the corresponding cache access latency, we perform experiments that load (read) data in specific cache coherence states (S and E) from specific cache locations (local and remote caches with respect to the requestor). We construct a micro-benchmark with threads that could be pinned to either one or multiple

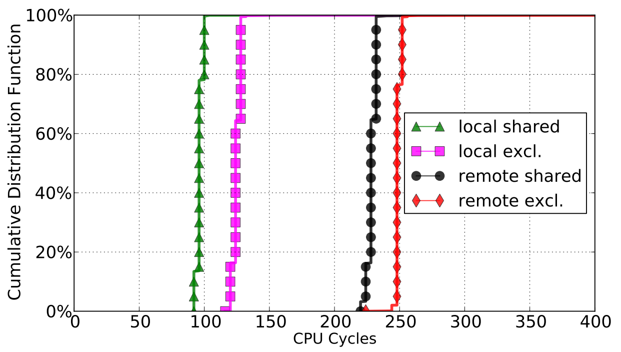
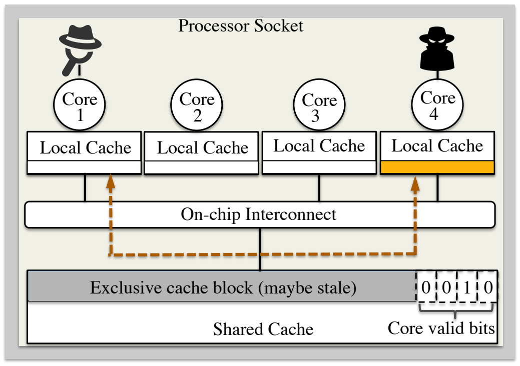


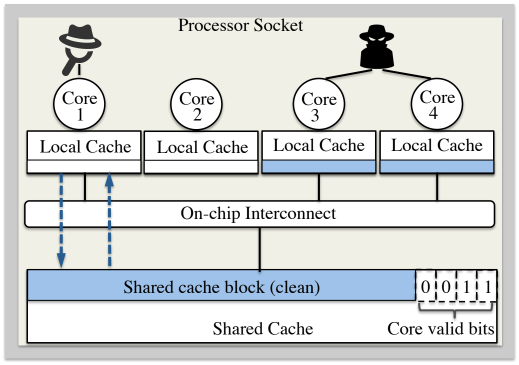
Fig. 2: Load operation latency in various (location, coherence state) combinations.

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. 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 private L1, 256 KB private L2 caches, 12 MB shared L3 cache within each socket. All of the caches are kept coherent in hardware. Our experiments were conducted on a system with a representative workload for a typical desktop server (i.e., applications such as browser, dropbox, code editors were running alongside our code 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) chip 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 chip 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 chip. In *Remote Exclusive*, the requested data is present in the *E* state in a L2 cache belonging to a remote chip. Figure 2 shows the cumulative distribution function (CDF) for the various (location, coherence state) combination pairs. Our results show that these combination pairs show distinct bands of latency distributions. We observe that accessing a cache block in the *E* state incurs longer latency than accessing 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 VI). 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 band for each configuration, and the bands corresponding to different configurations are sufficiently distinct from each other. This clearly demonstrates the viability of exploiting the latency difference between these combination pairs to implement timing channels.



(a) Cache block in E state



(b) Cache block in S state

Fig. 3: Trojan 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 for a data block residing in E and S states respectively.

VI. Exploiting Cache Coherence

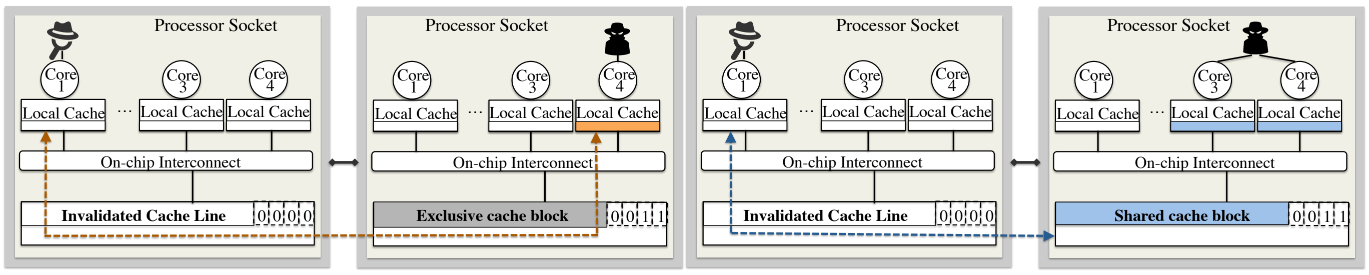
In this section, we show some practical ways that the trojan and spy processes can exploit cache latency differences to exfiltrate sensitive data.

A. On-chip Cache Coherence

Figure 3 shows an illustration of the attack using on-chip 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).

During a read miss in the private cache, the miss request is first sent to the shared LLC. The LLC maintains the *core valid bits* vector for each block that denotes which of the coherent private caches have a copy of the cache block [11]. An *1* bit value indicates that the corresponding core *caches* that block currently, and a *0* indicates that the corresponding core does not have that block.

If the total number of *1*'s in the vector is greater than one, it indicates that two or more sharers exist for this block. In other words, this denotes that the cache block is in the *S* state in the memory subsystem, and the cache copy in the LLC is *clean*. Since the LLC has a clean data copy, it can directly service the cache miss request from the requesting core.



(a) Cache block in E state (b) Cache block in S state

Fig. 4: Trojan 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 for a data block residing in E and S states respectively.

If the total number of *1*'s in the *core valid bits* vector is equal to one, it indicates that only one cache currently has the block (i.e., owns the block). Note that this cache may have the block in the *E* or *M* coherence states. Also, this may mean that the LLC copy of the block is *possibly stale*, since the current owner could have modified the block contents during cache residency. To avoid sending possibly stale data back to the requesting cache, 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 total number of *1*'s (sharer caches) increases to two.

If the total number of *1*'s in the *core valid bits* vector is equal to zero, it indicates 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 (using rdtsc or an equivalent instruction). The trojan spawns two reader threads on two different cores, and lets both of these trojan threads access the cache block 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.

Similarly, 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.

B. Inter-chip Cache Coherence

Many well-known family of processors provide inter-socket cache coherence through high speed point-to-point links, e.g., AMD's HyperTransport bus [7], and the Intel's Quick Path Interconnect [6]. 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 on-chip cache coherence (see Section VI-A) with slight modifications to how the data miss requests are routed. 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 LLC routes the data miss request up to the current owner (remote) core, which then responds with the data reply. The current owner (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 4 shows an illustration of this exploit. 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 [12]). 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 on-chip scenario (see Section VI-A). On a different socket, the spy spawns its thread, and generates a read miss to B to observe its access latency. To explicitly place a 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 on-chip scenario (see Section VI-A). On a different socket, the spy spawns its thread, and generates a read miss to B in order to observe its latency.

VII. Timing Channel Construction

In this section, we describe the trojan and spy construction process, and show how they would exploit cache access

Algorithm 1: Trojan Communication Protocol

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

Algorithm 2: Spy Communication Protocol

```
Input: read-only cache block: B, Tvalues[]=-1;
//Two access latency bands, $T_c$ and $T_b$;
//$T_s$ is the sampling interval;
//wait for the trojan to begin transmission;
synchronize with trojan using shared cache block, B;
//B could be created implicitly via KSM or through
//explicitly shared data or library code;
while true do
    flush B from cache;
    //wait for $T_s$ sec until trojan has an opportunity to reload;
    load B and time the load (T);
    if T is within $T_b$ then
        //transmission has started;
        break;
//reception period
i = 0;
while true do
    flush B from cache;
    //wait for $T_s$ for trojan to reload;
    load B and time the load (T);
    record T into Tvalues[i++];
    if T is outside of $T_c$ and $T_b$ for N consecutive times then
        //N is defined by the trojan and spy;
        break;
//translation period (interpret 1's and 0's)
read Tvalues[] vector from index 0 to N;
i = 0; j = 0; count[] = 0;
while Tvalues[i]! = -1 do
    Repeat until Tvalues[i] is within $T_b$ band: i++;
    $bit_c$ = 0;
    Repeat until Tvalues[i] is within $T_c$ band: $bit_c$++; i++;
    count[j++] = $bit_c$;
//Thold, Threshold separates $C_1$ and $C_0$ and helps decipher bits;
j = 0;
while count[j] != 0 do
    if count[j++] > Thold then
        //Infer that the transmitted bit is 1;
    else
        //Infer that the transmitted bit is 0;
```

latency differences created by combination pairs of cache location and coherence state associated with the cache block.

We illustrate a template for the trojan and spy that can be eventually integrated into a *real-world adversarial setting* designed to exfiltrate sensitive secrets. For example, let us consider a scenario where a spy process has the ability to observe encrypted communication transmitted over a public network between two processes with access to sensitive information. As per the system security policy, the spy cannot directly communicate with either of these entities due to it being on lower security stratum, nor can it decipher the communicated bits without knowing the decryption key. However, a malicious insider trojan (that has access to secrets) could collude with the spy to circumvent the system security and communicate secrets covertly as follows:

1) To compromise symmetric cryptography techniques (e.g., AES, DES), a trojan transmits symmetric encryption/decryption key covertly to the spy through modulating accesses to the coherent caches on shared physical memory blocks. With the already captured encrypted text and the now-obtained decryption key, the spy could *covertly* receive the message without any direct communication with the trojan.

2) To compromise asymmetric encryption standards (e.g, RSA), a trojan and the spy intentionally sign up for the RSA service under the pretense of encrypting their own texts. Since trojan-spy share the same coherence fabric, the trojan could covertly transmit its decryption key through modulating accesses to the coherent cache. The spy could decrypt the encrypted text and gather sensitive data.

A. Pre-transmission

In order to construct the convert timing channel using cache coherence states, there are two considerations: 1. shared physical read-only memory between trojan and spy to enable covert communication (Section IV), 2. synchronization between trojan and spy prior to transmission.

In our experiments, read-only shared memory is implicitly created through KSM when the trojan and spy intentionally write identical data to their individual pages with a deterministic, pseudo-random number generator function that begins with the same seed. Specifically, the trojan and spy create shared memory as follows: 1. Allocate memory through system calls such as *alloc()* and populate them with identical contents. The allocated pages with similar content are merged through invoking the system call *madvice()*. 2. The trojan and spy will wait for a certain period (e.g, 30 seconds) for the merging process to be complete. We note that this creation of shared memory needs to be done exactly *once* prior to entire trojan-spy communication. In very rare occasions, where a third independent process has its page merged with the memory page that is actively utilized by trojan/spy for covert communication, we have to discard such a page, and create another shared page that will be uniquely accessed by *just* the trojan and spy without external interference. Such situations can be prevented by creating a *spare* shared page initially, thus avoiding any necessity to re-invoke KSM.

For synchronization, the trojan issues flush of the shared cache block and then reloads the same block continuously for a number of times (e.g., about 20 in our experiments), and

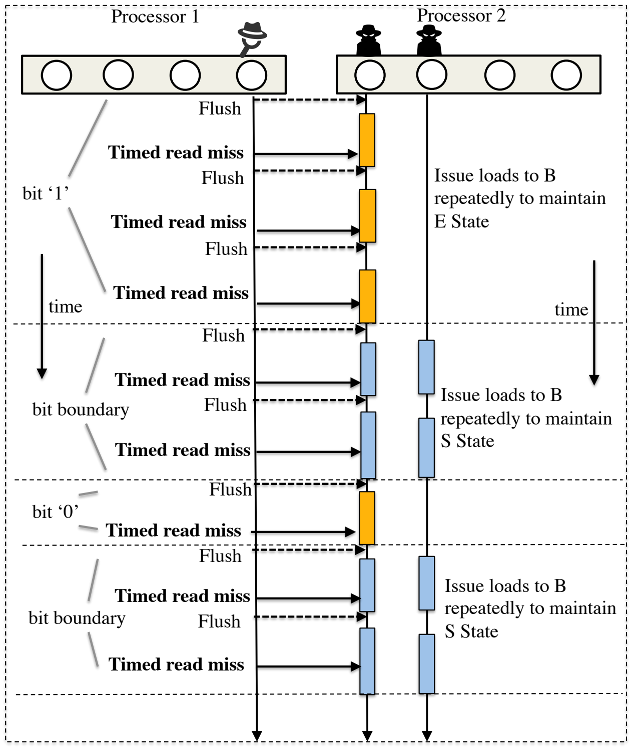


Fig. 5: Illustrative example of '1' and '0' transmission protocol between trojan(s) and spy.

the spy periodically issues load instruction to the same cache block. The trojan and spy time their respective load instruction latency. Synchronization is considered complete when the trojan observes a series of long latencies because of having to load data from memory, and when the spy notices a sequence of latencies eventually converging to a stable band of values. This process indicates that the trojan and spy uniquely share a block and that the spy is able to decipher the block's presence in trojan's cache through timing the cache block accesses. The actual transmission can start once the synchronization is successful. Our experiments show that it takes, on average, 90 milli-seconds for trojan-spy synchronization. We note that this step needs to be performed prior to covertly transmitting the first bit or after every OS context switch that involves either the trojan or the spy.

B. Trojan and Spy

To implement covert timing channels using coherence states, 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 delineate bit transmission *boundaries* (i.e., to say that a bit transmission has ended and another will start at the end of 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 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 I: Trojan implementation along with states used for bit communication and boundary. 'Remote' and 'Local' are with respect to the spy's location.

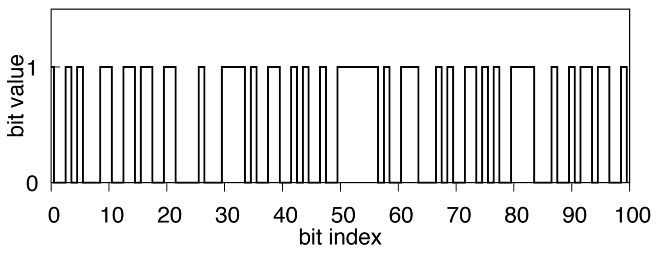


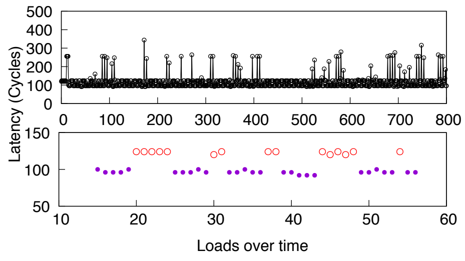
Fig. 6: Bit pattern (100 bits) covertly transmitted by the trojan.

cache hardware (Figure 2). 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 bit values '1' and '0', denoted by C_1 and C_0 respectively. We note that having distinct communication and boundary values remove the need for synchronization on each bit transmission.

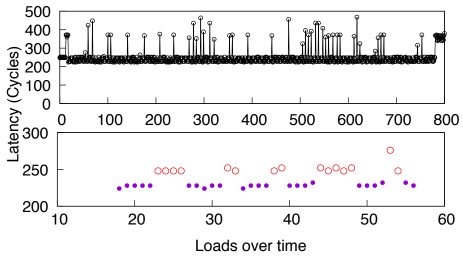
Algorithm 1 describes our implementation for the trojan. The trojan is multi-threaded to *explicitly* control the placement of blocks in S or E state either locally or remotely. For every '1' bit to be transmitted, it puts the cache block in CS_c coherence state for C_1 times, and for every '0' bit transmission, the trojan places the cache block in CS_c for C_0 times. In-between every bit transmission, the trojan places the cache block in CS_b for C_b times to denote bit boundaries.

The spy process is a single-threaded observer that times the cache block accesses using repeated patterns of flushes and reloads on them. Algorithm 2 describes our implementation for the spy. 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'.

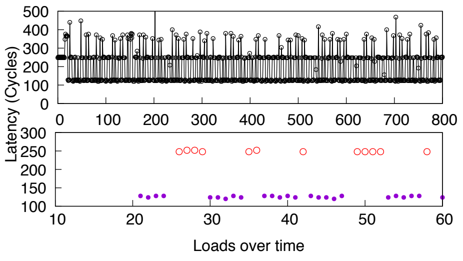
Figure 5 gives a diagrammatic illustration of an *example communication protocol between the trojan and spy*. In this example, the trojan is located in a processor different from that of the spy (Note that the trojan and spy could be in the same processor as well). The trojan modulates the cache access timing for the spy by placing a block 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



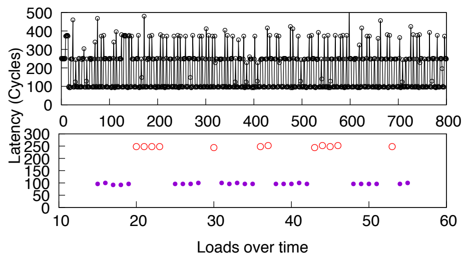
(a) *$LExcl_c$-$LShared_b$*



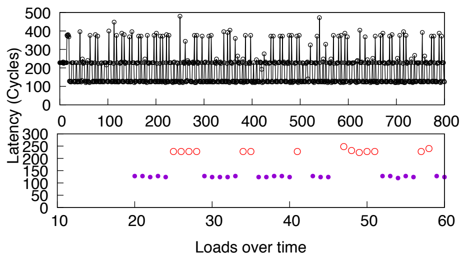
(b) *$RExcl_c$-$RShared_b$*



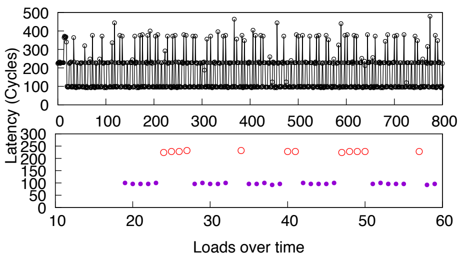
(c) *$RExcl_c$-$LExcl_b$*



(d) *$RExcl_c$-$LShared_b$*



(e) *$RShared_c$-$LExcl_b$*



(f) *$RShared_c$-$LShared_b$*

Fig. 7: Bit Reception by the Spy (corresponding to the bits transmitted in Figure 6) 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.

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 I 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.

VIII. Experimental Results

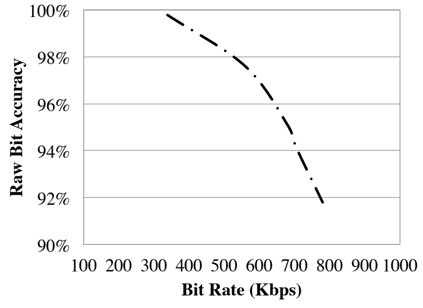
We conduct experiments on a Intel Xeon X5650 2-socket server with a total of 12 cores, the configuration described in Section V. We pin the trojan and spy threads onto specific cores using the *sched_setaffinity* API. All of the reported load latencies were obtained by inserting the *rdtsc* instruction. We implement the 6 attack scenarios listed in Table I and study their bandwidths. Additionally, we implement a covert timing channel with *symbols encoding multi-bits* by leveraging combination pairs of (location, coherence state) and encoding data in *larger-than-binary* representations.

A. Spy's Reception

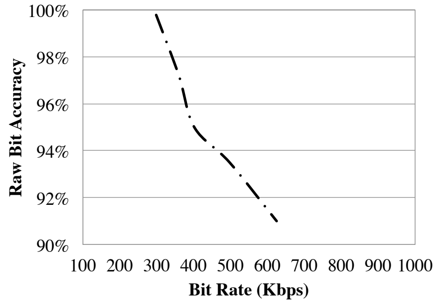
Figure 6 shows the secret (bit) pattern that the trojan intends to covertly communicate with the spy. Figure 7 shows the results of load latencies observed on the spy side. For each combination pair of (location, coherence state), we show two sets of results: the top portion shows the load latencies 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 load latency in the T_c band for one or two consecutive times (See discussion in Section VII-B). 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.

B. Transmission Bandwidth

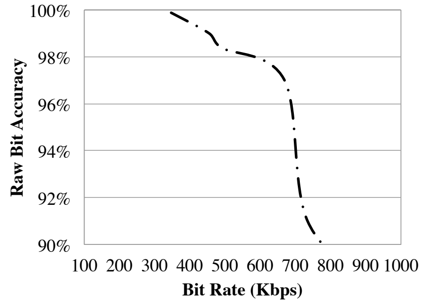
We conduct experiments to study the raw bit accuracy with increasing transmission bit rates between the trojan and spy. We perform this study by tuning two knobs: 1) Reduce the number of consecutive caching operations for shared blocks that communicate bit values and boundaries, i.e., values of C_1, C_0 and C_b. 2) Reduce the interval between shared cache block loads by the spy, i.e., the value of T_s. Refer to Algorithms 1 and 2 for further details on these parameters. Figure 8 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



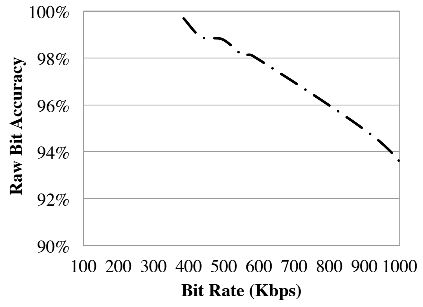
(a) *$LExcl_c$-$LShared_b$*



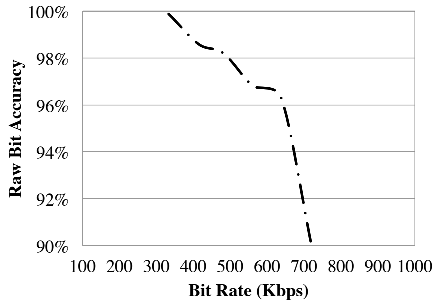
(b) *$RExcl_c$-$RShared_b$*



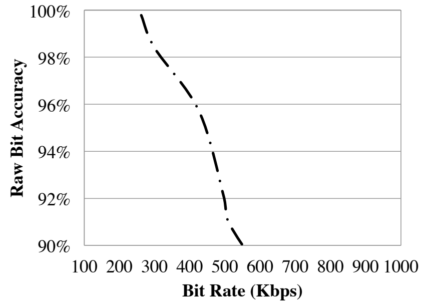
(c) *$RExcl_c$-$LExcl_b$*



(d) *$RExcl_c$-$LShared_b$*



(e) *$RShared_c$-$LExcl_b$*



(f) *$RShared_c$-$LShared_b$*

Fig. 8: Raw bit accuracy as captured by the spy with increase in transmission rates.

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. $RExcl_c - LExcl_b$ begins 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 [13], and is outside the scope of our work.

C. External Noise and Error Correction

To observe noise effects from co-located memory-intensive applications, we run a highly memory-intensive workload, kernel-build [14], that compiles a Linux kernel to benchmark a system or test its stability. This application supports a variety of options including multi-threaded implementation. Note that this experiment simulates an *extreme stress-test* case where a very high memory-intensive multi-threaded workload is co-located with trojan/spy. In this setting, alongside our trojan and spy processes, we spawn a different number of kernel-build threads (1 to 8). Figure 9 shows our experimental results where we observe that, with the increase in number of memory-intensive threads, the bit accuracy levels on the spy side experience a range of degradation.

Specifically, even with six background processes, the spy processes in all of the 6 attack variants are able to achieve fairly high bit accuracy (above 90% on average). However, with 8 external kernel-build processes, we see an observable impact on the trojan-spy communication (11% to 23% increase in raw bit error rate). Meanwhile, we observed subtle differences between different cases. For example, since kernel-build processes saturate the internal bus (L2-LLC) bandwidths, load latency values to E state blocks in remote caches were highly varied while remote LLC accesses (S state blocks) do not suffer from high latency swings when measured by the spy.

To illustrate mechanisms that can improve bit accuracy under noise, we propose and implement a simple error encoding and retransmission protocol. For each packet (64 bytes), 16 parity bits are added to catch any bit flips within 4 Byte chunks. After each packet transmission, the spy checks for parity bits and if errors are detected, it will request for packet resend by covertly transmitting NACK bit. This is achieved by reversing the roles of spy as the transmitter and trojan as the receiver just for transmitting the NACK bit. This process is repeated until successful receipt of the packet. Figure 10 shows the achievable bit rates for trojan-spy transmission without noise and the effective rate with retransmission scheme under medium noise (with 4 kernel-build processes) and high noise (with 8 kernel-build processes) levels. Overall, we can see that the retransmission scheme suffers less than 10% reduction in transmission rate, and incurs 24% worst-case reduction in transmission rate under high noise levels in return for guaranteeing 100% bit recovery. Conceivably, the same NACK mechanism can be used to track non-reception by the spy as well. If the spy was context switched out, the trojan will not

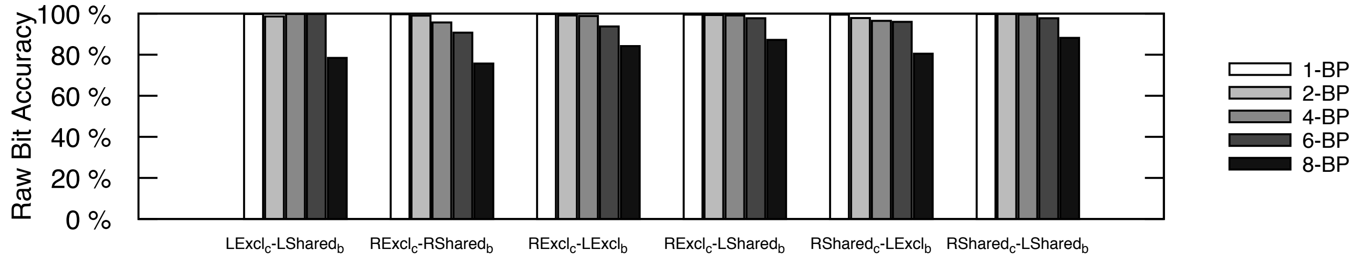


Fig. 9: Raw bit accuracy captured by the spy when co-located with external processes (kernel-build [14]).

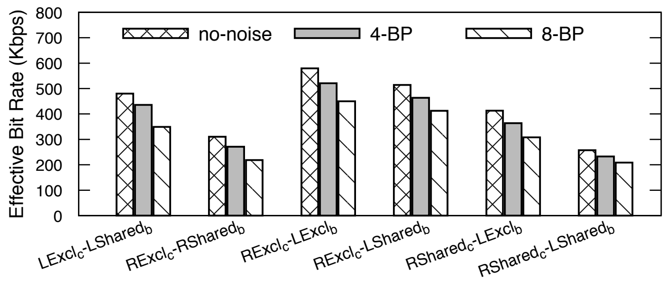


Fig. 10: Effective information bit transmission rate with error correction scheme under medium (4 co-located kernel-build processes) and high (8 co-located kernel-build) noise levels.

receive acknowledgment packet (NACK bit), and hence will retransmit until a successful acknowledgment (NACK=0) is received.

Our experimental results provide a useful insight that the covert timing channels introduced due to coherence states can be robust in terms of bit accuracy and high transmission rates. Also, incorporating even a fairly simple error detection and retransmission scheme can significantly improve bit accuracy with a relatively small impact on peak bit rate.

D. Symbols Encoding Multi-bits

Besides just increasing the transmission speed, the volume of information transmitted by a covert channel can be increased by encoding multiple bits using symbols. Due to the presence of multiple distinct latency bands corresponding to (location, coherence state) combination pairs, we implement a covert timing channel that transmits symbols encoding 2-bits in every transmission. We utilize four combination pairs (*RExcl$_c$*, *LExcl$_c$*, *RShared$_c$*, *LShared$_c$*) to encode one of four distinct symbol values. The spy infers the symbol by issuing load instructions (similar to our algorithm in Section VII-B) and timing the load operation latency corresponding to combination pairs.

Our experiments demonstrate a peak transmission rate of around 1.1 Mbps, which is significantly higher than the 700 Kbps observed when using just one combination pair of (location, coherence state) for encoding binary data for transmission. Figure 11 shows spy's reception of symbols through timed load operations along with a magnified view of the first 9 symbols or 18 bits $(\underbrace{10}\underbrace{01}\underbrace{01}\underbrace{00}\underbrace{01}\underbrace{10}\underbrace{01}\underbrace{10}\underbrace{11})$, in which all four distinct symbols are observed. We note that

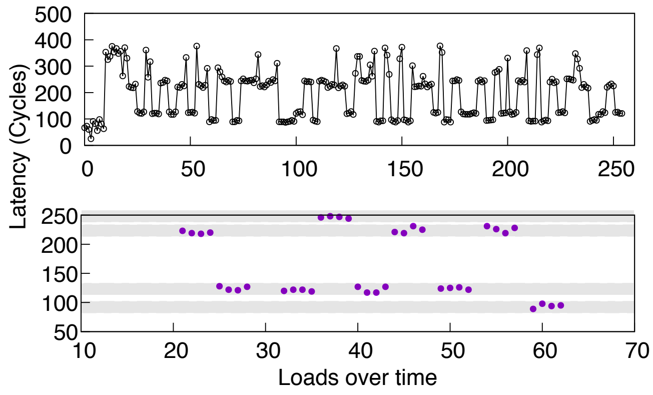


Fig. 11: Multi-bit symbol transmission using 4 combination pairs to encode 2-bit symbols. Magnified view of first 18 bits reception is shown, that captures all 4 possible symbol values.

more sophisticated symbol encoding mechanisms may achieve even higher transmission rates, and our main goal here is simply to demonstrate alternative ways that an adversary can exploit in order to achieve higher bandwidths.

E. Discussion

Applicability to Different Coherence Protocols. Cache coherence protocols in multi-core/multi-processor systems can be categorized into two broad classes: 1. snoop-based protocols, that broadcast coherence messages on a shared bus, and 2. directory-based protocols, that offer higher scalability by maintaining status of cache blocks in directories distributed throughout the system. The Intel Xeon processors evaluated in our experiments deploy a variant of directory-based protocol with LLC's core-valid-bits that direct coherence messages to specific cores with possibly valid blocks. On other directory-based systems, note that additional hops to the *home directory* based on address filters can further create different latency profiles for the adversaries to exploit. For snoop-based protocols, reads on E-state blocks will involve accesses to private caches of other cores since they hold the exclusive ownership of these blocks, while reads on S-state blocks are satisfied by the lower level shared caches that already have a clean copy of the cache block [15]. This is somewhat similar to *LExcl$_c$-LShared$_b$* configuration in Table I. Therefore, our findings extend to different classes of protocols.

Non-inclusive and Exclusive Caches. We study inclusive caches in this work. For blocks present (hits) in non-inclusive LLCs, latency difference between S and E state are still distinct. Given that LLCs are typically large and that S-state blocks are accessed relatively frequently by more than one core, absence of S-state blocks in LLC should be rare. On exclusive caches, both S- and E-state blocks may have similar latency. But still, data accesses in different cache levels and sockets will have distinct latency profiles. Therefore, we note that changing the cache inclusion property alone may not be sufficient to eliminate the timing channels studied in our work.

Potential Mitigation Techniques. To protect computer systems against the covert timing channels, we propose three mitigation techniques: 1) Add targeted noise to shared memory pages by having a monitor thread, that observes accesses to shared memory pages and dynamically issues additional loads. This method disrupts the covert timing channel by changing the coherence states (e.g., convert E to S) and alter spy's timing values; 2) Setup timeouts for KSM to un-merge shared pages with suspicious access pattern so that the trojan and spy communication can be disrupted dynamically. 3) Change hardware implementation, where LLCs are notified of changes from E to M states (that could slightly increase coherence-related traffic). This would enable LLCs to directly respond with E-state blocks, and consequently, latency profiles for E- and S-state blocks will be similar, thus closing the timing channels. Additionally, hardware timing obfuscators can make the latencies of local and remote caches indistinguishable, especially for suspicious applications.

IX. Related Work

Prior studies [16], [17] on Intel and AMD processors have shown that the cache access latencies are usually within a stable band of values that has been observed in our work.

A number of prior studies have demonstrated timing channels on caches [3], [12], [18], [19], [20], [21], microarchitectural units [22], [23], memory bus [24], processor frequency settings [25] and branch predictors [26], [27]. Most of these attacks rely on modulating the access timing behavior of a single hardware resource that may potentially be addressed through *carefully monitoring the unit*, and if possible, isolating or disabling them. In contrast, our work illustrates an attack that leverages the oft-used hardware cache coherence mechanism operating on multiple caches and coherence states.

Recently, Yao et al. [21] demonstrated covert timing channels that leverage non-uniform memory access latencies in multiple sockets. Different from this attack, the proposed covert timing channel exploits combinations of coherence states and cache location to construct timing channels, along with multi-bit symbols that significantly increases the transmission rate. Finally, these prior adversary models [4], [21] rely on user-initiated shared cache-blocks (via shared system libraries). We adopt a broader adversary model where shared physical memory could also be created using KSM.

Evtyushkin et al. [23] have shown a covert channel attack that relies on applications using random number generation, and may potentially be detected by tracking this module call failures.Jiang et al. [28] demonstrated a side channel to recover AES encryption keys using correlation analysis on GPU platforms.

Detection and defenses for microarchitectural timing channel attacks have been studied. Demme et al. [29] introduced a metric to quantify the difficulty level to exploit a system for side channels. Wang et al. [30] proposed secure hardware cache designs with partition-locking and random permutation to thwart cache side channels. Venkataramani et al. [31] proposed techniques to detect contention-based timing channels. Hunger et al. [32] studied contention-based cache covert channels and proposed anomaly-based detection. Liu et al. [33] used Cache Allocation Technology to partition LLC and mitigate information leakage channels. ReplayConfusion [34] utilized record and replay mechanisms to deterministically replay programs with different cache address mapping. Yao et al. [35] proposed techniques to detect Jump-oriented programming based code re-use attacks. Sharp [36] redesigned shared cache line replacement policy to avoid inclusion property that is exploited by the spy to decipher the victim's activity. To defend against memory-based timing channels, Ferraiuolo et al. [37] designed a secure memory scheduling algorithm. Camouflage [38] reshapes the timing of memory requests and responses to a deterministic distribution that eliminate memory access pattern snooping by untrusted parties. Recent works [39], [40] leverage computation logic in emerging memory technology to cryptographically obfuscate memory addresses and memory bus timing to thwart memory bus attacks. Wassel et al. [41] proposed wave scheduling policy to prevent timing channels in NoC architectures. We note that none of these prior defenses are designed to protect system-wide coherence protocols that entail multiple caches.

X. Conclusions

In this paper, we presented a critical vulnerability exposed by an oft-used feature in most modern multi-core and multi-socket processors, namely cache coherence protocol states. We showed how adversaries could exploit cache coherence states and construct covert timing channels in order to illegitimately transmit sensitive secrets to untrusted parties by violating the underlying system security policy. We demonstrated 6 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. We also showed how adversaries could exploit multiple caches and their associated coherence states to encode symbols with multiple bits, and explored potential mitigation strategies. Our experimental results on commercial systems show that the peak transmission bandwidths of these covert timing channels can vary between 700 to 1100 Kbits/sec.

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] 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.

[3] 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.

[4] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Proceedings of Asia Conference on Computer and Communications Security*, pp. 353–364, ACM, 2016.

[5] M. M. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Communications of the ACM*, vol. 55, no. 7, pp. 78–89, 2012.

[6] "Intel QuickPath Architecture," 2012. http://www.intel.com/pressroom/archive/reference/whitepaper_QuickPath.pdf.

[7] 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.

[8] P. Conway and B. Hughes, "The AMD Opteron northbridge architecture," *IEEE Micro*, vol. 27, no. 2, pp. 10–21, 2007.

[9] C. A. Waldspurger, "Memory resource management in VMware ESX server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.

[10] A. Barresi, K. Razavi, M. Payer, and T. R. Gross, "CAIN: silently breaking ASLR in the cloud," in *USENIX Workshop on Offensive Technologies*, 2015.

[11] "Using Intel VTune Amplifier," 2013. https://goo.gl/E9Fp2m.

[12] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proceedings of Symposium on Security and Privacy*, pp. 605–622, IEEE, 2015.

[13] R. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962.

[14] "kcbench." https://linux.die.net/man/1/kcbench.

[15] 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.

[16] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems," in *Proceedings of International Symposium on Microarchitecture*, pp. 413–422, ACM, 2009.

[17] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel, "Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture," in *Proceedings of International Conference on Parallel Processing*, pp. 739–748, IEEE, 2015.

[18] 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.

[19] 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 Workshop on Cloud Computing Security*, pp. 29–40, ACM, 2011.

[20] 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 Conference on Computer and Communications Security*, pp. 199–212, ACM, 2009.

[21] F. Yao, G. Venkataramani, and M. Doroslovacki, "Covert timing channels exploiting non-uniform memory access based architectures," in *Proceedings of Great Lakes Symposium on VLSI*, pp. 155–160, ACM, 2017.

[22] O. Aciicmez and J.-P. Seifert, "Cheap hardware parallelism implies cheap security," in *Proceedings of Workshop on Fault Diagnosis and Tolerance in Cryptography*, pp. 80–91, IEEE, 2007.

[23] D. Evtyushkin and D. Ponomarev, "Covert channels through random number generator: Mechanisms, capacity estimation and mitigations," in *Proceedings of Conference on Computer and Communications Security*, pp. 843–857, ACM, 2016.

[24] 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.

[25] M. Alagappan, J. J. Rajendran, M. Doroslovacki, and G. Venkataramani, "DFS covert channels on multi-core platforms," in *Proceedings of International Conference on Very Large Scale Integration*, IEEE, 2017.

[26] O. Aciiçmez, c. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of Symposium on Information, Computer and Communications Security*, pp. 312–320, ACM, 2007.

[27] 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.

[28] Z. H. Jiang, Y. Fei, and D. Kaeli, "A complete key recovery timing attack on a GPU," in *Proceeding of International Symposium on High Performance Computer Architecture*, pp. 394–405, IEEE, 2016.

[29] 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.

[30] 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.

[31] G. Venkataramani, J. Chen, and M. Doroslovacki, "Detecting hardware covert timing channels," *IEEE Micro*, vol. 36, pp. 17–27, Sept 2016.

[32] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari, "Understanding contention-based channels and using them for defense," in *International Symposium on High Performance Computer Architecture*, pp. 639–650, IEEE, 2015.

[33] 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 *International Symposium on High Performance Computer Architecture*, pp. 406–418, IEEE, 2016.

[34] M. Yan, Y. Shalabi, and J. Torrellas, "ReplayConfusion: Detecting cache-based covert channel attacks using record and replay," in *Proceedings of International Symposium on Microarchitecture*, pp. 1–14, IEEE, 2016.

[35] F. Yao, J. Chen, and G. Venkataramani, "Jop-alarm: Detecting jump-oriented programming-based anomalies in applications," in *Proceedings of International Conference on Computer Design*, pp. 467–470, IEEE, 2013.

[36] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks," in *Proceedings of International Symposium on Computer Architecture*, pp. 347–360, ACM, 2017.

[37] 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 *International Symposium on High Performance Computer Architecture*, pp. 382–393, IEEE, 2016.

[38] 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.

[39] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "Obfusmem: A low-overhead access obfuscation for trusted memories," in *Proceedings of the Annual International Symposium on Computer Architecture*, pp. 107–119, ACM, 2017.

[40] S. Aga and S. Narayanasamy, "Invisimem: Smart memory defenses for memory bus side channel," in *Proceedings of the Annual International Symposium on Computer Architecture*, pp. 94–106, ACM, 2017.

[41] H. M. G. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood, "Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip," in *Proceedings of International Symposium on Computer Architecture*, pp. 583–594, ACM, 2013.