SwiftDir: Secure Cache Coherence without Overprotection

Chenlu Miao
Zhejiang University
clmiao@zju.edu.cn

Kai Bu*
Zhejiang University
kaibu@zju.edu.cn

Mengming Li
Zhejiang University
mmli@zju.edu.cn

Shaowu Mao
Huawei Technologies
maoshaowu@huawei.com

Jianwei Jia
Huawei Technologies
jiajianwei@hisilicon.com

***Abstract*—Cache coherence states have recently been exploited to leak secrets through timing-channel attacks. The root cause lies in the fact that shared data in state Exclusive (E) and state Shared (S) are served from different cache layers. The state-of-the-art countermeasure—S-MESI—serves both E- and S-state shared data from the last-level cache (LLC) by explicitly synchronizing the Modified (M) state across private caches and the LLC. This has to sacrifice the silent upgrade feature that MESI introduces for speedup. Moreover, it enforces protection to not only exploitable shared data but also unshared data. This further slows down performance, especially for write-after-read intensive applications.**

In this paper, we propose SwiftDir to efficiently secure cache coherence against cover-channel attacks without overprotection. SwiftDir fundamentally narrows down the protection scope to write-protected data. Such exploitable shared data can be uniquely identified with the write-protection permission in the memory management unit (MMU) and do not necessarily transit to state M. We validate this idea through tracing system calls of shared libraries on Linux. We then investigate all three commercial cache architectures (i.e., PIPT, VIPT, and VIVT) and find it feasible to hitchhike the address translation process to transmit the write-protection information from the MMU to the coherence controller. Then SwiftDir enforces protection over only write-protected data by serving all requests toward them directly from the LLC with a constant latency. This not only simplifies how MESI handles write-protected data but also avoids how S-MESI overprotects them. Meanwhile, SwiftDir still preserves silent upgrade for efficient handling of unshared data. Extensive experiments demonstrate that our SwiftDir can secure cache coherence while outperforming not only secure S-MESI but also unprotected MESI.

***Keywords*-cache coherence; timing-channel attack; shared data; address translation;**

I. Introduction

Cache coherence states have recently been exploited to leak secrets [65]. Such attacks have a different attacking principle from a plethora of cache timing-channel attacks that exploit the timing difference of cache hits and misses [27], [42], [53], [54], [63], [67]. They exploit a finer-grained timing difference among cache hits over data with different coherence states. Consider the seminal coherence protocol—MESI [46], [51]—for example. Serving a cross-core access request with data in state Exclusive (E) takes about 26 more cycles than with data in state Shared (S) on an Intel Xeon processor [65]. This can be exploited to leak secret information through a timing-channel attack. On 2.67 GHz cores, the secret leakage rate can be as high as 700∼1,100 Kbps.

Shared memory as the prerequisite for coherence attacks, however, is critical for performance speedup and cannot be simply disabled [47]. It can be readily available through shared libraries or memory deduplication [26], [27], [65], [67], [69]. Shared libraries can be dynamically loaded at program runtime instead of having to be jammed into executable files at compile time [1], [6], [37]. This helps to not only control program size but also save memory space when multiple programs with common shared libraries are loaded. Commonly used shared libraries such as `libxul` are larger than 100 MB [25]. Furthermore, shared libraries greatly ease program maintenance. Programs need no recompilation upon possibly frequent updates of shared libraries (e.g., `libc` has evolved through more than 40 versions from 1994 to 2022 [22]). Beyond shared libraries, memory deduplication pushes memory saving to another level [2], [28], [29], [45]. It merges identical memory pages into one and then releases redundant memory allocations. Empirical measurements show that process heaps and shared libraries are among the top sources for redundant pages [5]. The total amount of redundant pages in a system ranges from 11.3% to 85.7% [15], [30]. For individual programs, memory deduplication can save 13.7%∼47.9% of memory space [28]. This promises 1.2×∼1.9× speedup given program cold-starts. Practical speedup effects during program execution should be way much higher given the 100,000× speed gap between memory and disk [31].

The state-of-the-art solution S-MESI [66] protects coherence security at the cost of sacrificing the silent upgrade feature, which is offered by MESI for speedup. Silent upgrade enables a core to locally update E-state data and change the coherence state to Modified (M) without synchronizing this state to the shared last-level cache (LLC). This is also the root cause for coherence timing-channel attacks. When the LLC receives a cross-core request toward an E-state data block, it cannot determine whether the data block has already been modified by its owning core. The LLC (i.e., the coherence controller residing alongside the LLC controller in particular) has to relay the cross-core request to the owner for further handling. In contrast, if a cross-core request hits an S-state data block, the LLC can directly serve it. S-MESI

*Kai Bu is the corresponding author.

thus brings explicit M-state notification back to the game. If the LLC is not notified to upgrade an E-state block to state M, it can make sure that E-state LLC data remain up to date. This makes both E-state and S-state data be directly served from the LLC, successfully mitigating the E/S timing difference.

Albeit S-MESI's effectiveness, we find that its speedup sacrifice essentially arises from its overprotection enforcement. S-MESI pulls back the silent upgrade effect for all data without discrimination. However, as with various other cache timing-channel attacks [26], [38], [67], coherence timing-channel attacks can exploit only shared data through shared memory [65]. It is inessential to overprotect unshared data from coherence timing-channel attacks. We should rethink secure coherence and strive for a solution without overprotection and its caused performance degradation.

In this paper, we present SwiftDir as the first secure yet efficient cache coherence without overprotection. It aims to narrow down the protection scope to only exploitable shared data. To this end, we answer a series of key questions:

- How to accurately identify exploitable shared data?
- How to efficiently notify the coherence controller with data sharing status?
- How to securely modify coherence toward different handling strategies for shared and unshared data?

Through tracing system calls on systems such as Linux, we find that shared data can be accurately identified using their write-protected permission found inside the memory management unit (MMU). We then investigate all three commercial cache architectures (i.e., PIPT, VIPT, and VIVT) to figure out how the MMU interacts with cache controllers. We find it feasible to hitchhike the address translation process to transmit the write-protection information from the MMU to the cache controller. Specifically, while accessing page tables, we extract not only the translated address as usual but also its associated read/write bit. Using this read/write bit as an argument, we introduce only one lightweight modification to cache coherence. That is, we essentially develop a finer-grained `GETS` coherence request, with a newly introduced `GETS_WP` to handle write-protected data and with the original `GETS` to handle the other data.

SwiftDir then closes the E/S timing gap of exploited shared data by essentially removing their E states. It achieves so by directly setting initial loads of write-protected data into state S. The rationale behind such modification is that write-protected data are not supposed to associate with the M state. They do not necessarily require the E state either for ease of silent E→M upgrade. Therefore, our protection scheme not only effectively avoids overprotection in S-MESI but also essentially simplifies how MESI handles write-protected data. In other words, it protects cache coherence through simplification rather than complication. Even if the write-protected permission may be somehow used to regulate unshared data of special interests, SwiftDir can simply yield a higher efficiency out of the enlarged protection scope.

In summary, we make the following major contributions to secure cache coherence even with performance gains.

- We investigate the state-of-the-art S-MESI against coherence timing-channel attacks and identify overprotection as a key barrier to system performance (Section II).
- We present SwiftDir as the first attempt to efficiently secure cache coherence without overprotection (Section III). It narrows down the protection scope to only exploitable shared data.
- We explore implementation strategies that make SwiftDir applicable to practical memory and cache architectures (Section IV). SwiftDir not only effectively avoids overprotection in S-MESI but also essentially simplifies how MESI handles write-protected data.
- We implement SwiftDir using the gem5 simulator [43] and validate security and performance of SwiftDir with extensive single-threaded SPEC CPU 2017 benchmarks, multi-threaded PARSEC 3.0 benchmarks, multi-threaded read-only benchmarks, and write-after-read intensive benchmarks (Section V). Experiment results show that SwiftDir can successfully prevent coherence timing-channel attacks. It outperforms both MESI and S-MESI by completing about 900,000 more instructions per second on a 3 GHz processor. For write-after-read intensive benchmarks, our SwiftDir can reduce the execution time of S-MESI by up to 61.79% and thus yield a 2.62× program speedup.

II. Problem

In this section, we first investigate how existing timing-channel attacks exploit coherence states. We then underline the limitation of the state-of-the-art countermeasure that simply nullifies exploitable yet speedup-oriented states. We will present our SwiftDir solution to secure coherence protocols while retaining speedup benefits in Section III.

A. Cache Coherence

Cache coherence should be preserved on any multicore system such that different cores can access the same value for the same memory location. In a multicore cache hierarchy, all cores share an LLC while each of them owns at least one layer of private cache. Consider, for example, when both core A and core B have a copy of memory location X in private caches. If core A modifies the value of its local copy and core B receives no information about the modification, subsequent accesses to memory location X on core B will incorrectly return its local stale copy. To avoid such incoherence issues, cache coherence protocols associate each cached data block with a coherence state. Coherence states may be updated upon access operations on local or remote cores. They ultimately help to identify whether a data block holds the latest value for its corresponding memory location.

1) Coherence States: At the heart of a cache coherence protocol is a finite-state machine that models how coherence states transit among each other upon which memory access requests. Its logic is implemented via a coherence controller in hardware. Individual coherence controllers are integrated into controllers of different storage components such as cache, memory, and direct memory access (DMA). A coherence protocol not only defines finite-state machines inside each coherence controller but also coordinates the interaction across different coherence controllers. It is memory requests that initially trigger coherence events. Upon receiving a coherence event targeting a certain copy, a coherence controller may take some actions (e.g., invalidating a local copy if another copy with the same memory location is modified on another core) and change the copy's state (i.e., state transition) if necessary. Actions can also trigger further requests or responses. In other words, state transition is not necessarily atomic. A state may need an acknowledge response or a returned data block to transit to another state. States that mark the start and end of a memory access request are referred to as stable states. States in between the transition of two stable states are called transient states [46], [48], [49].

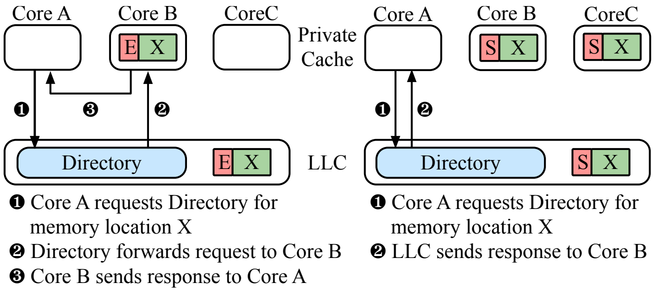
We next review typical coherence protocols focusing on their stable states for brevity. As with most cache coherence designs in the literature, we omit discussing transient states whenever this does not impede understanding.

2) Coherence Protocols: **MSI.** The MSI protocol is seminal and all cache coherence protocols deployed on modern multicore chips inherit its three states—Modified, Shared, and Invalid [32], [46].

- Modified (M). In the M state, a data block is dirty and represents the only copy that holds the latest value for its corresponding memory location across caches and memory. Access requests for this memory location on other cores should be forwarded to the core caching the modified data block.
- Shared (S). In the S state, a data block is clean and shared. It can be used to serve access requests from both local and remote cores.
- Invalid (I). In the I state, a data block is invalid. A request to it on the local core leads to a cache miss.

MESI. The MESI protocol introduces an Exclusive state to make the traditional S state more fine-grained and save coherence traffic. MESI and its slight variants such as MOESI and MESIF [23] prevail in most modern processors including Intel Xeon and AMD Opteron [65].

- Exclusive (E). In the E state, a data block is clean and is exclusively cached on the core where it currently resides. No valid copy for the same memory location resides on other cores.


Core A
Core B
CoreC
Private Cache
E X
S X
Directory
LLC
❶
❷
❸
❶ Core A requests Directory for memory location X
❷ Directory forwards request to Core B
❸ Core B sends response to Core A
❶ Core A requests Directory for memory location X
❷ LLC sends response to Core B

(a) E-State Coherence Request

(b) S-State Coherence Request

Figure 1. Handling of coherence requests for E-state and S-state data in MESI.

The E state helps to accelerate write-after-read accesses [46], [65]. Without the E state, MSI marks a clean data block with the S state and has no means to further determine whether the S-state block is the only copy in the cache hierarchy. To serve a subsequent write request targeting the S-state block, the core cannot simply update the data value or upgrade its S state to the M state. Instead, it needs to first request ownership from the coherence controller; this request triggers further invalidation commands to other cores for avoiding potential stale copies therein [48]. MESI introduces the E state to ensure that a data block is the only cached copy for its corresponding memory location. Once a core initially loads a data block, it sets the block in state E. Then its subsequent write to the E-state block can be completed without triggering coherence messages to other cores.

3) Coherence Architectures: Depending on where to store the coherence states, coherence protocols can be architectured in two ways [60]: snooping and directory-based. Snooping coherence protocols require each core to locally track the coherence states of all its cached blocks. Since a core has no idea of the exact data and coherence states on other cores, cache accesses that might affect copies on other cores trigger broadcast messages. For example, whenever a cache miss occurs, the requesting core broadcasts its requests to all the other cores via a bus or an interconnection network. Every core with the requested data block responds to the requesting core, following broadcast as well. With the simplicity of snooping comes the limited scalability to large-scale multicore chips. Therefore, modern processors turn to directory-based coherence protocols [19], [31], [46]. We focus mainly on directory-based coherence protocols in this paper. They store coherence states of all cached blocks in a centralized or distributed directory. Cores direct their access requests to the directory, which then decides where to forward the requests for fetching the requested data. Different cache layers return the requested data with different access latencies. Such a timing difference has recently been exploited for cover-channel attacks [65].

B. Coherence Attack Exploiting E/S States

Timing difference. We showcase the exploitable timing difference of directory-based coherence protocols in Figure 1

[65]. Specifically, it arises from different access latencies for fetching data blocks in E state and S state upon a cache miss. Figure 1(a) presents the process to access an E-state data block for memory location X in the LLC. Since Core A has no copy of data with memory location X in its private caches, it sends a request for memory location X to the directory (step 1). The directory finds that the X-addressed data block is exclusively cached on core B. Although the LLC also has a copy for memory location X, the E state makes the directory hard to ensure whether the value in the LLC is obsolete. Therefore, the directory forwards the request to the owner—Core B (step 2). Upon receiving the forwarded request, Core B responds Core A with the requested data block for memory location X (step 3). In contrast, when the requested data block is in the S state (Figure 1(b)), the directory ensures that the copy in the LLC holds the same value with that of other copies cached in some core's private caches (i.e., Core B and Core C in this example). The directory thus more quickly returns the requested data block from the LLC instead of another core's private caches.

Measurements on an Intel Xeon processor reveal an about 26-cycle timing difference for accessing E- and S-state data [65]. This is sufficient for building an exploitable timing channel [7], [8], [65].

Threat model. We consider an attacker that exploits the timing difference for accessing shared data in E and S states via a recently discovered timing-channel attack [65]. It employs two colluding processes—sender and receiver on a multicore processor. The sender and receiver first construct shared memory by directly calling a shared library or indirectly leveraging memory deduplication [26], [27], [65], [67], [69]. The sender aims to leak secrets to the receiver without directly transmitting the corresponding data; this helps evade forensics or similar auditing techniques [65]. It modulates the secret through coherence states (i.e., E and S) of shared data. It can also create threads on different cores. As for the receiver, it can neither directly access the secret data nor direct communicate with the sender; but it can decode secrets through the E/S timing channel.

Note that the E/S timing channel abuses only coherence states by read operations on shared data. Read operations crafted by the attacker aim to delicately manipulate coherence states for encoding and decoding secrets. They are different from heavy read operations used in the Rowhammer attack to corrupt victim data [34]. We consider the E/S timing channel orthogonal to another type of attacks that exploits write operations on shared data [11], [47]. Specifically, they leverage the fact that writing a shared page (by memory deduplication) triggers a copy-on-write page fault. A private copy of the shared page need be spawned first; the private copy then serves the write operation. This makes writing a shared page slower by an order of magnitude than writing a non-deduplicated page. An exploitable timing channel is thus formed. A recent software protection called VUsion [47] mitigates exploits of both read and write operations on shared data. Its key idea is enforcing copy-on-access on both shared and non-shared pages. This makes sure that no shared data be brought into caches. However, this may open a door for denial-of-memory and denial-of-cache attacks. Colluding programs can deliberately access a shared library or pre-agreed data to quickly spawn many copies of identical content to fatigue both memory and cache spaces.

In this paper, we follow the direction of the state-of-the-art protection against E/S coherence attacks [66], striving for an efficient hardware solution without inducing additional vulnerabilities. The focus of protection lies in read operations on shared data. For copy-on-write attacks [11], [47], we suggest to consider a copy-on-write page fault as a write miss and integrate efficient handling of write misses. This benefits from the essential difference between read misses and write misses. Read misses serve instructions with inputs. The time for handling a read miss can hardly be taken out of the instruction execution time. In contrast, write misses essentially serve for committing outputs. The quickest way to handle them is to offer some place to write to. For example, a potential efficient solution can be inspired by the non-allocate write policy. Write results may take effect in some dedicated cache/memory space rather sooner before the block/page to be written gets available. We consider this a different generic hardware-design direction and leave it for future work.

As with [65], [66], we also do not consider other orthogonal timing-channel exploits through, for example, cache hit/miss [14], [35], [41], [44], [59], cache replacement [12], [61], port contention [8], TLB hit/miss [17], [24], [57], or branch prediction [4], [16], [18].

Covert-channel attack. Consider a shared memory location X for example. The sender and receiver may agree to transmit bit 1 via E-state X and bit 0 via S-state X. To place X in the E state, the sender can create a thread on one core to initiate a cold-start access to memory location X. To place X in the S state, the sender can create two threads on two cores; both threads then access memory location X. When the receiver accesses memory location X, it measures the access latency and identifies sate E or S given a slow or fast access, respectively. Finally, the receiver retrieves the secret bit by respectively decoding state E or S to bit 1 or 0. This covert-channel attack can leak secrets at a high rate of 700∼1,100 Kbps on 2.67 GHz cores [65].

Side-channel attack. We find that the preceding covert-channel attack can be easily extended to build equivalent side-channel attacks as in [11], [47]. Specifically, the side-channel attack of interest [11], [47] aims to infer whether victim shared-data have been accessed. Two colluding attack-processes can exploit the E/S channel as follows to impose such a side-channel attack on a victim process. First, one attack process accesses victim shared-data, the coherence state of which will be set as E. Within

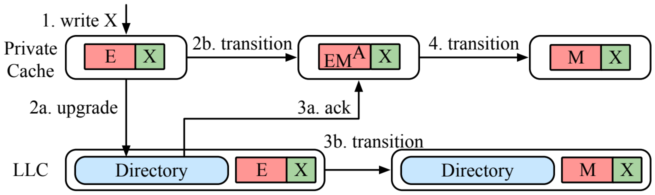


Figure 2. E→M transition in S-MESI.

a predefined interval, if the victim process also accesses the same data, the coherence state transits to S. Otherwise, it remains E. Then, upon the predefined interval times up, the other attack process turns to access this same data. The access latency reveals whether the coherence state is E or S and thus whether the victim process has accessed the victim data. Such side-channel attacks are proven to 1) disclose which websites the victim have accessed and which programs the victim has executed, 2) leak HTTP password hashes, and 3) break address space layout randomization (ASLR) [11], [47].

C. Countermeasure Nullifying E State

The state-of-the-art countermeasure—S-MESI—returns the requested data in both states E and S from the LLC [66]. However, this essentially nullifies the silent upgrade effect from E to M that the E state is supposed to offer. When traditional silent upgrade updates the E-state block in private caches into state M, its corresponding copy in the LLC stays in state E. Directly returning E-state data from the LLC likely results in accessing stale data. Therefore, S-MESI revokes silent upgrade and enforces the M state to be synchronized across both private caches and the LLC.

Figure 2 showcases the modified E→M transition in S-MESI. Upon receiving a write request for a private data block in state E (step 1), the core sends a coherence request to the LLC for privilege upgrade (step 2a) and transits state E to a transient EM^{A} state (step 2b). It then waits for the LLC's acknowledgement (step 3a), which indicates that the LLC permits the upgrade and updates its coherence state from E to M (step 3b). The acknowledgement from the LLC triggers private caches to update the EM^{A} state to the M state. This completes the handling of a write access request. For subsequent access requests toward the M-state block in the LLC, the coherence controller uses the M state to determine that the block has been modified in private caches and it should forward the request to private caches.

Albeit its effectiveness against coherence attacks exploiting E/S states, the state-of-the-art S-MESI [66] essentially degrades MESI to MSI. The E state is necessary for optimizing MSI performance (Section II-A2). Nullifying it from MESI leads to inevitable performance slowdown, especially for programs with heavy writes.

III. Overview

In this section, we present SwiftDir as a fundamentally efficient solution for secure cache coherence. We first identify overprotection as the root cause for inefficiency of S-MESI. It revokes silent upgrade from state E to state M for all types of data. However, only shared data can be exploited by coherence timing-channel attacks. We are motivated to minimize the protection scope. We find that exploitable shared data belong to write-protected data, whose E state is redundant. We can simply remove their E state from coherence to fundamentally throttle the exploited E/S timing difference. This promises a way of protection by simplification rather than complication. SwiftDir can outperform not only the state-of-the-art secure coherence protocol but also unprotected ones.

Table I
Transient states of interest on L1 [21], [46].

State	Description
IS^{D}	I→S/E waiting for Data response
EM^{A}	E→M waiting for LLC's ACK

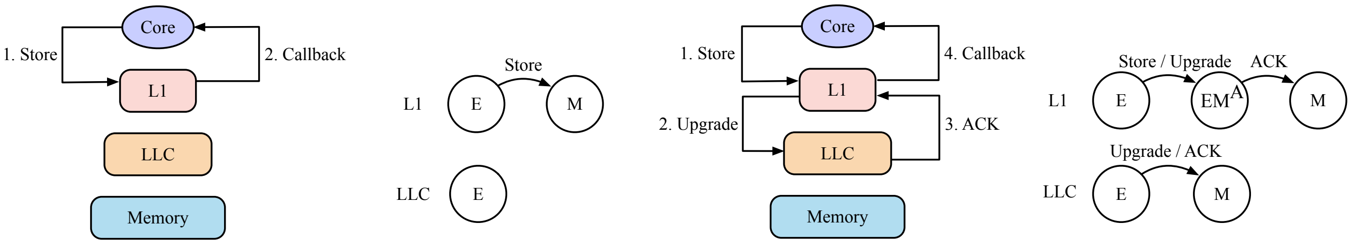
Table II
Transient states of interest on LLC [21], [46].

State	Description
IE^{DB}	I→E waiting for memory's Data and L1's Unblock
IE^{B}	I→E waiting for L1's Unblock
IS^{D}	I→S waiting for Data response
ES^{DB}	E→S waiting for L1's Data and L1's Unblock
ES^{B}	E→S waiting for L1's Unblock

A. Motivation

The state-of-the-art S-MESI traps in an intrinsic dilemma where performance and security are deemed contradictory [66]. The introduction of state E brings better performance at the cost of vulnerability [65]. To regain security, it seems that we have to degrade the E state with performance sacrifice. To showcase this dilemma, we present a thorough analysis of the performance of MESI (Section II-A2) in comparison with that of S-MESI that nullifies the E state [66] (Section II-C).

For ease of understanding the protocol details, we first clarify necessary coherence states and events in Tables I, II, and III. For the L1 cache and LLC, the four stable states—M, E, S, and I—have been defined in Section II-A2. Table I and Table II further define the transient states of our interest. We formulate a transient state as ${S_1S_2}^R$. In most cases, the transient state appears during the transition from stable state S_1 to stable state S_2 and the transition cannot complete until receiving some *R*esponses. For example, transient state IE^{DB} appears during the I→E transition while waiting for a response of the requested *D*ata and then an un*B*lock action of the L1 cache. Only in rare cases does the ending stable state S_2 in ${S_1S_2}^R$ turn to another stable state according to the exact response. For example, IS^{D} ends up in state E if the core that issues the related request is the only core to



(a) MESI

(b) S-MESI

Figure 3. Comparison of E→M transition in MESI and S-MESI.

Table III
COHERENCE EVENTS OF INTEREST [21], [46]. GETS_WP IS THE ONLY COHERENCE REQUEST INTRODUCED BY SWIFTDIR.

Event	From	To	Description
Load	Core	L1	core loads data from cache
Store	Core	L1	core writes data to cache
Data_From_Owner	L1	L1	L1 sends data to remote requestor's L1
GETS	L1	LLC	L1 loads data from LLC
WB_Data_Clean	L1	LLC	write back clean data
Upgrade	L1	LLC	upgrade L1's data for write permission
Unblock	L1	LLC	unblock blocking data
Exclusive_Unblock	L1	LLC	unblock blocking data with exclusiveness
Data	LLC	L1	LLC sends data to L1
Data_Exclusive	LLC	L1	LLC sends data to L1 with exclusiveness
Fwd_GETS	LLC	L1	LLC forwards GETS request to L1
Fetch	LLC	Mem	fetch data from Mem
Mem_Data	LLC	Mem	transmit data from LLC to Mem
	Mem	LLC	transmit data from Mem to LLC
ACK	generic		generic invalidation ack
GETS_WP	L1	LLC	L1 reads write-protected data from LLC

cache the returned data. More coherence events that trigger state transitions are described in Table III.

Figure 3 compares the E→M transition of MESI and that of S-MESI. As shown in Figure 3(a), for traditional MESI, after receiving the core's Store event, the L1 cache line pertained to the memory address to store can immediately transit from state E to state M and then serve the Store request. No coherence events are triggered from the L1 cache to the LLC, where no state transition is involved either; the LLC cache line with the requested memory address remains in the E state. However, after nullifying the E state for security, the E→M transition of S-MESI becomes complicated and incurs overhead. As shown in Figure 3(b), receiving the core's Store request, the L1 cache can no longer follow silent upgrade. First, it should send an Upgrade request to the LLC and transfer to state EM^A, waiting for an ACK response from the LLC. After receiving the Upgrade request, the LLC searches the requested cache line and sees if the cache line is in state E. If so, the LLC sends an ACK response to the L1 cache and then transfers to the M state. The receipt of ACK triggers the L1 cache to transit from state EM^A to state M. Not until then can the actual data modification take place. Unlike in traditional MESI, such a complicated E→M transition in S-MESI slows down the common write-after-read case at a system-wide level.

We observe that the inevitable overhead by nullifying the E state arises from overprotection. Since the coherence attack exploits only shared data, protecting only shared data is sufficient. However, S-MESI nullifies state E and complicates the E→M transition for unshared data. A fundamentally efficient solution should minimize its scope of protection.

In this paper, we are motivated to secure cache coherence against timing-channel attacks without overprotection. To our surprise, once after we narrow down the protection scope, we even do not have to adopt costly solutions like the complicated E→M transition in [66]. Instead, we develop a lightweight modification to secure cache coherence while both simplifying processing of exploitable shared data and remaining traditional efficient processing of unshared data. It breaks down the security-performance dilemma and promises even higher performance than unprotected coherence.

B. Methodology

Toward minimizing the scope of protection to only exploitable shared data, the ultimate goal is to enable a coherence protocol to identify shared data and hide the timing difference from their accesses in E and S states. We highlight three major strategies for our SwiftDir to achieve the goal and elaborate on the associative key ideas afterwards.

Accurate identification of exploitable shared data. This aims to narrow down the source of coherence timing-channel attacks that exploit the timing difference for accessing shared data in E and S states.

Efficient association of data and sharing status in the cache hierarchy. This aims to limit hardware overhead for tracking shared data in the cache hierarchy.

Secure modification of shared-data coherence. This aims to enforce secure coherence on exploitable shared data against timing channels while remaining traditional efficient coherence handling of unshared data.

1) Accurate Identification of Shared Data: There are two ways to produce exploitable shared memory for coherence timing-channel attacks—shared library and memory deduplication [65]. Through investigating memory management on mainstream OSes (Section IV-A), we find that their corresponding access permissions are read-only and copy-on-write, both of which can be categorized into write-protected. We can thus accurately identify exploitable shared data via write-protected data. Furthermore, we find that the MMU already associates a permission bit to a unit of allocated memory to specify whether it is write-protected. SwiftDir directly inherits this bit for identifying write-protected data without introducing any extra complexity.

2) Efficient Association of Cached Data and Sharing Status: A straightforward way to mark write-protected data in the cache hierarchy is to associate each cache line with an one-bit indicator. However, this consumes cache capacity and complicates hardware logic. We observe that an access request needs to traverse the MMU for address translation before the translated address can be used to complete cache access. Therefore, we transmit the write-protected bit along with its corresponding translated address from the MMU to the cache hierarchy. It is used as a real-time argument for access handling procedures without being a necessary hardware add-on to each cache line.

A key question to address hereby is whether the relative position of the MMU with respect to the cache hierarchy makes a difference for MMU-cache interaction in SwiftDir. To answer this question, we thoroughly investigate all common cache architectures including physically indexed physically tagged (PIPT), virtually indexed virtually tagged (VIVT), and virtually indexed physically tagged (VIPT) in Section IV-B. Note that physically indexed virtually tagged (PIVT) is usually not considered in both of academia and industry; it makes no sense to use a virtual address for tagging when its physical address is already known [36]. The investigation reveals that different cache architectures bring no significant difference to our SwiftDir design. Regardless of the exact architecture, SwiftDir can always be satisfied in that prior to accessing the PIPT LLC, the virtual-to-physical address translation has already taken place.

3) Secure Modification of Shared-Data Coherence: Leveraging the identification of shared data from the MMU, we aim to efficiently secure shared-data coherence while not affecting unshared-data coherence. Since the state-of-the-art solution imposes a high overhead on the E→M transition for unshared data (Figure 3), an intuitive efficient solution may specialize an E state for shared data. Consider, for example, using $\mathrm{E_{wp}}$ to denote this newly introduced state, which is a stable state for write-protected data as the write-protected permission can be used to identify shared data (Section III-B1). It regulates both exclusiveness and write-protection. The solution now only needs to make $\mathrm{E_{wp}}$/S timing difference unobservable. Specifically, upon the initial load request for a write-protected data block, the corresponding cache line directly transits to the $\mathrm{E_{wp}}$ state. Then, to serve a subsequent remote load request, the $\mathrm{E_{wp}}$-state block is responded directly from the LLC, the same place where a requested S-state block would be served. Albeit the effectiveness against timing channels, the newly introduced $\mathrm{E_{wp}}$ state does complicate cache coherence maintenance.

In this paper, we manage to secure cache coherence by simplification instead of complication. We do not introduce any extra state to protect coherence states against timing channels. We advocate eliminating the E state for write-protected data and set their initial loads directly with the S state. As aforementioned, the E state aims to accelerate the common write-after-read case. Write-protected data, however, are not supposed to be written. They either do not permit writes at all or trigger copy-on-write and serve the write request on the triggered duplicate. This essentially reveals that write-protected data need no silent upgrade offered by the traditional E→M transition. Once after their first loads to the cache hierarchy, we expect them to remain the latest value and be readily suitable for serving remote loads as S-state data. We thus set initially loaded write-protected data in the S state, eliminating their E state.

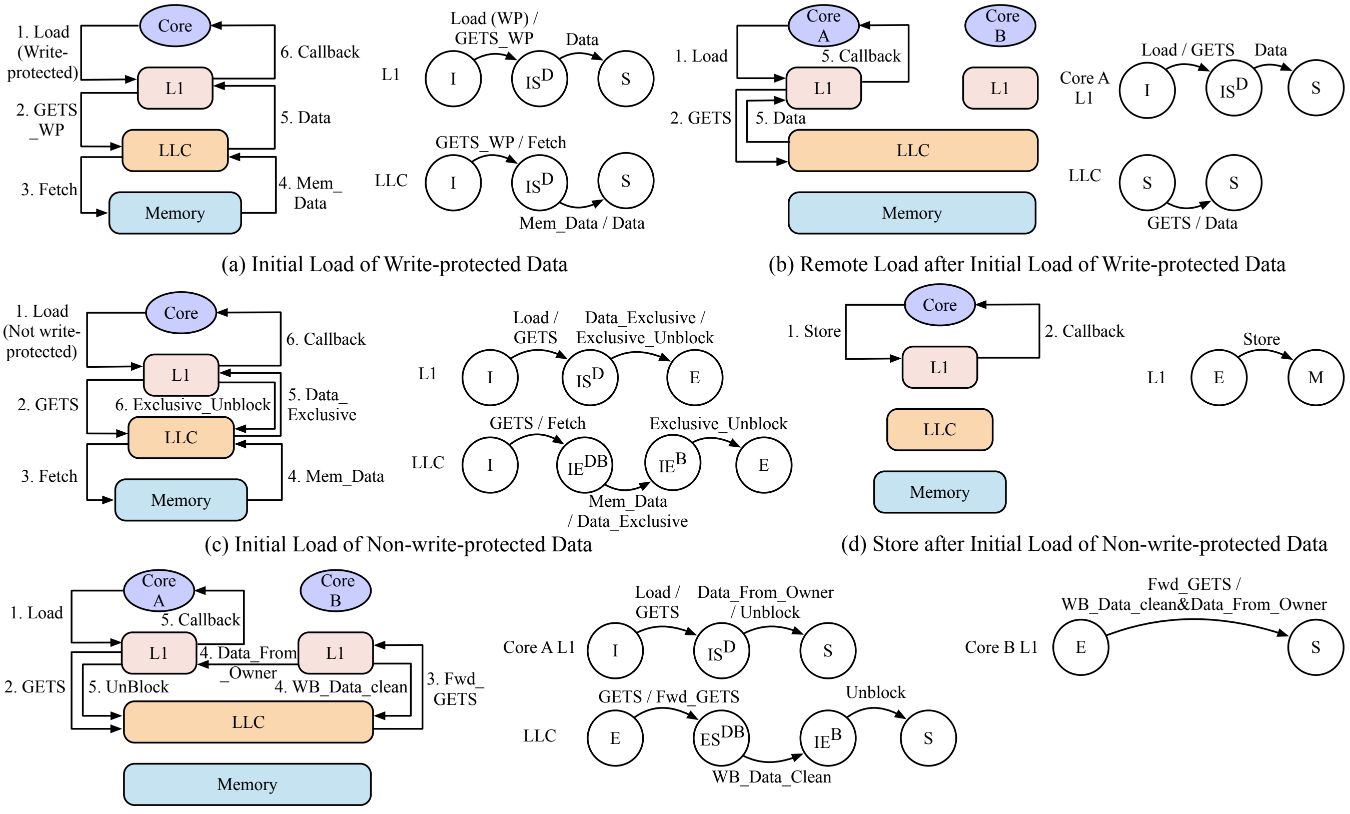
IV. Design

In this section, we detail the SwiftDir design. As sketched in Figure 4, SwiftDir prevents timing channels by simplifying coherence of exploitable shared-data (Figures 4(a)-(b)) and remains traditional efficient coherence for unexploitable unshared-data (Figures 4(c)-(e)). For security, by eliminating the E state for shared-data, the exploited E/S timing difference vanishes. For performance, while imposing no overhead on unshared-data, SwiftDir accelerates the remote load request for initially cached shared-data. This promises a higher performance than both MESI and S-MESI do.

A. Identification of Shared Memory

We leverage the MMU to identify exploitable shared memory and hitchhike the address translation process therein to transmit the sharing status to the cache hierarchy. This section focuses on the former question—how to accurately identify shared memory from the MMU. The key idea is that page table entries (PTEs) of shared memory managed by the MMU set the Read/Write (R/W) field with the write-protected permission.

1) Exploited Shared Memory is Write-protected: We start with investigating that exploitable shared memory by coherence timing-channel attacks should be write-protected. Specifically, the attacker produces shared memory by either shared library or memory deduplication [65].



(a) Initial Load of Write-protected Data

(b) Remote Load after Initial Load of Write-protected Data

(c) Initial Load of Non-write-protected Data

(d) Store after Initial Load of Non-write-protected Data

(e) Remote Load after Initial Load of Non-write-protected Data

Figure 4. SwiftDir coherence.

Shared library. We use the Linux strace tool [56] to trace the system calls of a program using shared libraries. In the tracing process, an essential system call—mmap [39]—is found to establish a mapping between file/devices and memory [39]. Different types of mapping and related write permissions may be created according to the exact arguments for mmap. In particular, the prot argument determines whether the memory page is writable. For writable memory, the flags argument further determines whether writes to the memory page will be made visible to other processes and whether the underlying file will be modified [39]. For a shared library, most of its memory (e.g., code segment and read-only data segment) cannot be written, with the prot argument allowing PROT_READ. Some memory region (e.g., data segment) can be written, with the prot argument and the flags argument allowing PROT_WRITE and MAP_PRIVATE, respectively. MAP_PRIVATE represents a write-protected permission. It first invokes the copy-on-write operation and then writes on the new copy instead of the original memory.

Memory deduplication. Kernel same-page merging (KSM) is a kernel feature for memory deduplication [2]. Using KSM, the OS uses a kernel thread to search for memory pages with the same content. Identical pages are merged to one physical page and redundant pages release the memory allocation. The merging process calls for the write_protect_page function, which handles write-protected memory and permits the merged memory page to be copy-on-write.

2) Write-protected Manifests in PTE R/W Field: Having observed that exploitable shared-memory is write-protected in Section IV-A1, we further show that the write-protected property manifests as the read permission in the R/W field of PTEs. A PTE consists of various fields that specify the address of the memory page and its status such as freshness (i.e., the Dirty bit). All these information helps the MMU to translate virtual addresses to physical addresses and determine whether the physical page can be accessed as requested. In particular, we find that the R/W bit is exactly what we can use to associate with write-protected memory from both shared library and memory deduplication. Without loss of generality, our findings stem from the latest stable release of Linux 5.16.13 as of April 2022 [58].

Shared library. Upon invocation, mmap first allocates only virtual memory pages. The arguments prot and flags passed to mmap determine the protection bits of the virtual memory pages—vm_page_prot. Following demand paging, a physical memory page will be allocated until its corresponding virtual address is accessed and a page fault is triggered. The page fault handling process allocates the physical page and invokes function mk_pte to create its PTE. Function mk_pte takes vm_page_prot of the virtual page as an argument to set the PTE's control bits. When `vm_page_prot` conveys private mapping (i.e., `MAP_PRIVATE` for argument `flags`, corresponding to the writable shared library in Section IV-A1) and unwritable shared mapping (i.e., `MAP_SHARED` for argument `flags` while not `PROT_WRITE` for argument `prot`, corresponding to the unwritable shared library), the PTE's R/W field is cleared to 0 (read).

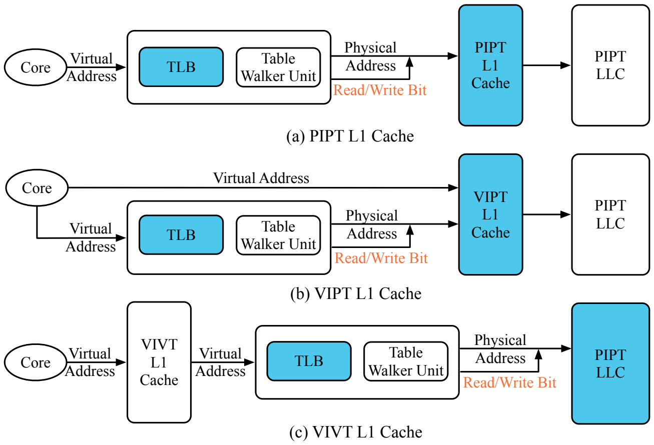
Memory deduplication. In comparison with shared library management, it is relatively easier to link the write-protected property of memory deduplication to PTEs' R/W fields. The `write_protect_page` function for merging memory directly sets the R/W field of the merged page's PTE to 0 (read).

B. Argumentation of Sharing Status

SwiftDir hitchhikes address translation to transmit the write-protected information from the MMU to the cache hierarchy for coherence maintenance. One may wonder that different cache architectures lead to respective complexities to this hitchhiking method. Consider, VIVT, for example. It allows the core to send a virtual address to the L1 cache even prior to address translation. To address such a concern, we thoroughly investigate all the three cache architectures in commercial use—PIPT, VIPT, and VIVT. We find that SwfitDir suffices for the LLC to obtain the write-protected information of the requested data until the request arrives at the LLC. This requirement can be easily satisfied regardless of cache architecture, simply because that the LLC is a physical cache and address translation should have taken place before accessing the LLC.

1) Cache Architecture Basics: Cache architectures can be classified in terms of whether virtual or physical addresses used to search requested data. A cache access handles a requested address in two steps. First, the index bits are used to locate the corresponding cache set. Second, the tag bits are compared with the tag fields of the cache lines within the cache set to determine a cache hit or miss. Accordingly, PIPT and VIVT use physical addresses and virtual addresses in both steps, respectively. VIPT uses virtual addresses for set indexing and physical addresses for tag comparison. Most processors adopt VIPT and PIPT for L1 caches. For example, ARM Cortex-A series use PIPT L1 data caches and use VIPT or PIPT for L1 instruction caches [3]. AMD Zen processors use VIPT for L1 data caches [40]. Intel Skylake uses VIPT L1 caches [52]. VIVT caches are used in relatively earlier processors, such as ARM720T and ARM926EJ-S [3]. For the shared LLC on any known processor, it always follows PIPT.

Figure 5 demonstrates how SwiftDir hitchhikes the address translation process to transmit the write-protected information read from the PTE's R/W field to the cache hierarchy with different architectures. For ease of understanding, we associate each case with a (where, when) property to indicate on which cache layer and at which access step the write-protected information reaches the cache hierarchy.



(a) PIPT L1 Cache

(b) VIPT L1 Cache

(c) VIVT L1 Cache

Figure 5. Transmission of write-protected information from MMU to caches by hitchhiking address translation.

2) PIPT L1 Cache: (L1 Cache, Set Indexing): For a PIPT L1 cache, given that cache access requires physical addresses, address translation takes place before the access request reaches the L1 cache. As shown in Figure 5 (a), the core first requests the MMU for address translation. The MMU finds the corresponding page table entry through the translation lookaside buffer (TLB) or by page table walking, translates the virtual address to a physical address, and checks access permissions. Then it transmits the R/W bit along with the translated physical address to the L1 cache. This makes the write-protected information available as soon as indexing sets on the L1 cache, which will later relay the write-protected information to lower-level caches upon L1 cache misses.

3) VIPT L1 Cache: (L1 Cache, Tag Comparison): Different from PIPT L1 caches, a VIPT L1 cache removes address translation from the critical path such that address translation and cache access takes place concurrently. As illustrated in Figure 5(b), while the MMU is translating the virtual address, the VIPT L1 cache can already use part of the virtual address for set indexing. When the translation process completes, SwiftDir requires that both tag fields and the write-protected information be sent to the VIPT L1 cache. If tag comparison for requested shared data returns a cache miss, a coherence request containing the write-protected information as an argument will be sent to lower-level caches.

4) VIVT L1 Cache: (LLC, Set Indexing): For a VIVT L1 cache, the write-protected information reaches the LLC no sooner than the access request reaches the LLC. The VIVT L1 cache uses virtual addresses for both set indexing and tag comparison. As shown in Figure 5(c), address translation takes place after L1 access and still precedes PIPT LLC access [33]. We can thus guarantee that upon an access request reaches the PIPT LLC, the write protected information can be ready along with the physical address.

C. Modification of Coherence Protocol

SwiftDir imposes minimum modification on existing coherence protocols by narrowing down the protection scope to only write-protected data. Motivated by the observation that write-protected data are not supposed to be written, we consider it less necessary to maintain their E state. SwiftDir thus directly sets an initial load of write-protected data in the S state. This way, write-protected data no longer transit to the E state and are immune from timing-channel attacks that exploit the E/S timing difference. Such a protection simplifies coherence maintenance instead of complicating it, which is deemed as a must by existing protection solutions. We hitchhike the address translation process to pass the identification of write-protected data to the coherence controller. As for non–write-protected data, SwiftDir still manages their coherence using the original efficient coherence protocol such as MESI.

We now detail how SwiftDir maintains coherence. Figure 4 showcases critical coherence transitions for ease of understanding. We follow a two-level cache hierarchy with directory-based MESI as in [65], [66]. Figures 4(a)-(b) and Figures 4(c)-(e) target write-protected data and non–write-protected data, respectively. Necessary coherence states and events for understanding are listed in Tables I-III.

1) Coherence of Exploitable Shared Data: SwiftDir sets initially loaded write-protected data to state S instead of state E as in existing coherence protocols. To this end, we modify the original I→E transition upon an initial load of write-protected data to an I→S transition. This removes the E→S transition for write-protected data and thus throttles E/S timing channels in a fundamental way.

SwiftDir implements the I→S coherence transition for the initial load of write-protected data as shown in Figure 4(a). It removes the unnecessary exclusivity information of write-protected data from the LLC. After the core initiates a load request to a write-protected data block (step 1), the L1 cache receives the write-protection requirement as well as the physical address from the MMU. Since it is the initial load of the requested write-protected data, the L1 cache encounters a local miss. It sends a `GETS_WP` request to notify the LLC that the load request targets write-protected data; L1 then transfers to the IS$^{\mathrm{D}}$ state (step 2). `GETS_WP` is the only introduced coherence request by SwiftDir. It modifies the original `GETS` by integrating the write-protected information as an argument. Besides this request modification, SwfitDir introduces no extra coherence states. Upon receiving `GETS_WP` from L1, the LLC issues a `Fetch` request to memory and transfers to the IS$^{\mathrm{D}}$ state (step 3). The requested data then transmit from memory to the LLC (step 4) and then the L1 cache (step 5) without exclusivity attached. Given no exclusive permission, both the LLC and L1 cache set the received write-protected data in state S (step 5 and step 6).

2) Coherence of Unexploitable Unshared Data: As aforementioned, SwiftDir remains the original coherence maintenance for unshared data that are not vulnerable to the E/S timing channel. Figures 4(c)-(e) illustrate typical coherence transition cases for how SwiftDir manages non-write-protected data using MESI. It leads to no performance overhead. In particular, SwiftDir reserves MESI's efficient handling of common write-after-read operations by silent upgrade from state E to state M in the L1 cache (Figure 4(d)). This helps SwiftDir to outperform existing protection that enforces the E→M transition on both private caches and the LLC (Figure 3(b)).

Table IV
PERFORMANCE COMPARISON IN TERMS OF WHETHER E-STATE SHARED AND UNSHARED DATA ARE EFFICIENTLY HANDLED.

Protocol	**Shared Data** serve E from LLC	**Unshared Data** silent E→M on L1
MESI	✗	✓
S-MESI [66]	✓	✗
SwiftDir	✓	✓

D. Security

SwiftDir secures cache coherence against the E/S-channel timing attack in a fundamental way. Such a coherence attack essentially exploits the timing difference between a remote load of E-state shared data returned from the L1 cache and a remote load of S-state shared data returned from the LLC. The coherence modification by SwiftDir restrains shared data from transiting to the E state. A direct I→S transition is enforced instead (Figure 4(a)). Subsequent remote loads to the shared data in caches thus always hit in S state and get served from the LLC, as shown in Figure 4(b). This instantly closes the exploitable E/S timing channel and prevents the so caused timing-channel attacks.

E. Performance

SwiftDir secures cache coherence while outperforming not only the existing solution but also the unprotected protocol. Since SwiftDir handles shared and unshared data differently, Table IV provides a qualitative performance comparison in terms of whether both types of data can be efficiently handled. We concentrate on E-state related operations because the timing-channel attack under concern exploits the timing gap between accesses over E-state data and S-state data. For shared data, it is relatively more efficient to serve a request hitting on an E-state block on the LLC than to relay this request to the L1 cache for handling. For unshared data, it is relatively more efficient to remain silent upgrade of the original MESI (i.e., silent E→M transition on the L1 cache without notifying the transition to the LLC). We will justify the two observations shortly. Prior to that, we conclude from Table IV that SwiftDir is the only protocol to handle both types of data using respective efficient choices. In contrast, both of the original MESI protocol and state-of-the-art secure coherence protocol (i.e., S-MESI [66]) fail to do so.

Efficient service of E-state shared data from the LLC. SwiftDir essentially replaces state E of shared data with state S. This enables SwiftDir to serve requests for cached shared data directly from the LLC. As shown in Figure 4(b), Core A issues a remote load request to shared data previously loaded by Core B (step 1). The requested data are in the S state in both the L1 cache on Core B and the LLC. Core A's L1 receives the request and encounters a local miss. It then forwards the request to the LLC and transits to the IS^{D} state (step 2). The LLC searches the requested data and results in a local hit. Since the coherence state of the requested data is S, the LLC acts as its owner. The LLC then directly sends the requested data back to Core A's L1 (Step 3), which further forwards the data to Core A and transits to the S state. Throughout the entire process, SwiftDir enforces neither state transition on the LLC and Core B's L1 nor communication between the LLC and Core B's L1.

In contrast, the original MESI sets an initially loaded data block (whether shared or not) on Core B in the E state. Then a subsequent remote load request from Core A will be processed as in Figure 4(e). It induces state transitions on both the LLC and Core B's L1 as well as communication between the LLC and Core B's L1. SwiftDir thus protects exploitable shared data by simplifying their coherence maintenance rather complicating that.

Efficient silent-upgrade of E→M for unshared data on the L1 cache. While the existing secure coherence S-MESI [66] also manages to serve a remote load of shared data from the LLC, it achieves so by sacrificing the efficient silent upgrade feature of MESI. This immediately slows down the E→M upgrade of unshared data (Figure 3(b)). However, SwiftDir remains the original silent E→M upgrade that MESI introduces as a performance booster (Figure 4(d)).

In summary, SwiftDir outperforms the original unprotected MESI and the state-of-the-art secure S-MESI in terms of processing shared data and unshared data, respectively. It thus secures cache coherence against timing-channel attacks while counter-intuitively improving performance.

V. Evaluation

Settings. We implement SwiftDir using gem5 [10], [20], [43], a widely used cycle-level computer system simulator. As summarized in Table V, we run SwiftDir on a processor with 1∼4 cores and a two-level cache design in line with related work [38], [55], [62], [68]. Each core owns private L1 Instruction and L1 Data caches. All cores share the same L2 cache. The baseline cache coherence is the directory-based MESI protocol.

Workloads. We validate security and performance of SwiftDir on both single-core and multicore architectures. For single-core experiments, we run the applications from the SPEC CPU 2017 benchmark package [13]. We use the benchmarks from both SPECrate 2017 Integer and SPECrate 2017 Floating Point suites. For multicore experiments, we run the applications from the PARSEC 3.0 benchmark suite [9]. PARSEC statistics concentrate on each benchmark's Region of Interest (ROI), which includes the parallel code of the benchmark for multicore execution. We then build multi-threaded read-only applications to investigate the true performance overhead if any for protecting different amounts of shared data. Finally, we build typical write-after-read intensive applications to investigate performance slowdown induced by overprotection from the state-of-the-art secure coherence.

Table V
Experiment setup.

Module	Configuration
Processor	1∼4 cores, 3 GHz out-of-order 192-entry ROB 32-entry LQ & 32-entry SQ superscalar width: 8
Private L1 I/D cache	64-Byte block, 4-way, 32 KB RT latency: 1 cycle
Shared L2 cache	64-Byte block, 16-way 2-MB bank per core RT latency: 16 cycles
TLB	fully associative 64-entry ITB & 64-entry DTB
Memory	DDR3_1600_8x8, 1 channel 2 ranks, 8 banks per rank 1 KB row buffers tCAS-tRCD-tRP: 11-11-11

Metrics. We evaluate SwiftDir security using the access latency of data in various coherence states. The statistics should guarantee that exploitable shared data no longer lead to an exploitable timing difference of access latency. For single-core applications, we evaluate SwiftDir performance using the number of instructions per cycle (IPC). For multicore applications, we report the execution time of the ROI because synchronization in these applications affects the actual instruction count [50], [64]. We normalize these metrics of SwiftDir over that of the unsafe baseline MESI, in comparison with the state-of-the-art secure S-MESI. A better performance calls for a higher normalized IPC or a lower normalized execution time.

Results. Extensive experiment results show that SwiftDir successfully throttles the traditionally exploited timing channel of shared-data accesses by unifying handling of all such accesses in the LLC. Access latencies of shared data in emulated SwiftDir are centralized around 17 cycles. Our SwiftDir shows a potential to reduce the execution time of S-MESI by up to 61.79% and thus yield a 2.62× speedup.

A. Security: Timing Channel Prevention

We measure the access latency of traditionally vulnerable shared data to show that SwiftDir successfully prevents the timing channel. The timing channel exploited by existing coherence timing-channel attacks arises from different access latencies over shared data in state E and state S. MESI serves data in the S state directly from the shared L2 cache where resides also the directory controller. Data in the E

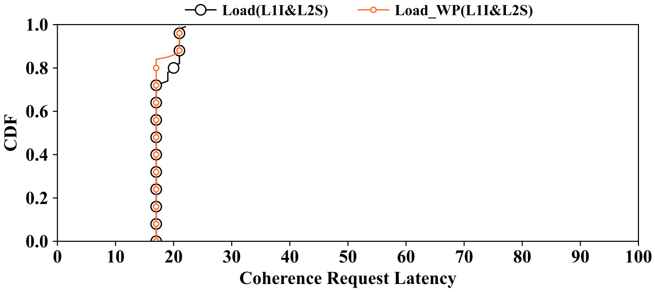


Figure 6. SwiftDir coherence latency.

state are, however, indirectly served from the private L1 cache. To avoid the so caused timing difference, SwiftDir manages to serve all requests to shared data directly from L2. Let Load_WP(L1I&L2S) represent such requests, where L1I&L2S specifies that the requested data block is currently in state Invalidate in the requesting core's L1 and in state Shared in L2.

Figure 6 reports the cumulative distribution function (CDF) of Load_WP(L1I&L2S) by SwiftDir, in comparison with that of Load(L1I&L2S) by MESI. Two observations can be drawn in Figure 6. First, SwiftDir links shared data to write-protected data and unifies the handling of them into Load_WP(L1I&L2S). It thus leaves the attacker with no timing difference to differentiate or exploit. Second, the access latencies of Load_WP(L1I&L2S) are centralized around 17 cycles and highly comparative to that of Load(L1I&L2S). Integration of the write-protection permission into Load_WP(L1I&L2S) thus imposes a negligible overhead on Load(L1I&L2S).

B. *Performance: Single-Threading*

We start performance evaluation with the single-threaded benchmarks in the SPEC CPU 2017 suite. Performance statistics of each benchmark are collected over execution of one billion instructions.

Figure 7 reports the normalized IPC of SwiftDir in comparison with that of S-MESI, with MESI as the baseline. Our SwiftDir outperforms MESI in a relatively constant way. Only for one benchmark—`xalancbmk`—does SwiftDir yield an about 0.003% lower IPC.

In contrast, S-MESI shows a more dynamic result. Nearly half of benchmarks yield lower IPCs using S-MESI than using MESI, especially for `bwaves` (0.12% lower), `wrf` (0.10% lower), `xalancbmk` (0.09% lower), and `xz` (0.07% lower). S-MESI rarely shows a much higher IPC than MESI does for benchmarks such as `blender` (0.28% higher) and `povray` (0.37% higher). This behavior can be explained as follows. S-MESI revokes silent upgrade of MESI. A modification operation in L1 also sets the coherence state of related data in L2 as M. L2 considers this M-state data block as being recently accessed and makes it less susceptible to be replaced by the Least Recently Used (LRU) replacement policy. This further avoids related cache misses and speeds up benchmark execution toward a higher IPC.

In summary, SwiftDir outperforms both MESI and S-MESI in terms of most single-threaded SPEC benchmarks. It yields a 0.03% higher IPC and a 0.01% higher IPC on average than do MESI and S-MESI, respectively. Put in the real context with a 3 GHz processor (Table V), this makes SwiftDir handle 900,000 and 300,000 more instructions per second than MESI and S-MESI do, respectively. Without considering `blender` and `povray` that dominate the average result of S-MESI, it turns to be outperformed by MESI with a 0.005% lower IPC. In this case, SwiftDir yields a 0.03% higher IPC and thus 900,000 more instruction completion per second on average than both MESI and S-MESI do.

C. *Performance: Muti-Threading*

We then evaluate SwiftDir performance using the multi-threaded benchmarks in the PARSEC 3.0 benchmark suite. On a 4-core processor, each benchmark spawns four threads to accelerate ROI execution. Note that the execution time of the ROI instead of IPC is a well accepted metric for evaluating multi-threading performance. We choose the `simmedium` input set for each benchmark. Such an input scale is considered sufficient for measuring performance without bias.

Figure 8 reports the execution time of SwiftDir and S-MESI normalized over that of MESI. SwiftDir outperforms both MESI and S-MESI in terms of a shorter execution time on average. In accordance with its performance behavior for single-threaded benchmarks, S-MESI again shows relatively diverse results. It averagely has a 0.41% longer execution time than MESI does. In contrast, SwiftDir outperforms MESI for most benchmarks. Only for three benchmarks—`dedup`, `freqmine`, and `swaptions`—does SwiftDir become slower than MESI by respectively 3.30%, 0.41%, and 2.65%. We find it challenging to reason about such performance fluctuations. We suspect that they may be related to uneven LLC eviction rates across multiple threads of the same benchmark. Specifically, some threads may have higher data locality than others, which are forced to suffer from relatively more cache misses and thus slower execution. On average, SwiftDir outperforms both MESI and S-MESI with a 2.01% and 2.31% shorter execution time, respectively.

D. *Performance: Amount of Shared Data*

As Table IV summarizes in Section IV-E, SwiftDir promises a better performance with two efficient features—serving E-state shared data from the LLC and preserving silent E→M transition for unshared data on the L1. Section V-B and Section V-C have demonstrated that the two features help SwiftDir outperform MESI and S-MESI on average for both single-threaded and multi-threaded benchmarks with mixed shared and unshared data. We next crystallize how either feature contributes to performance

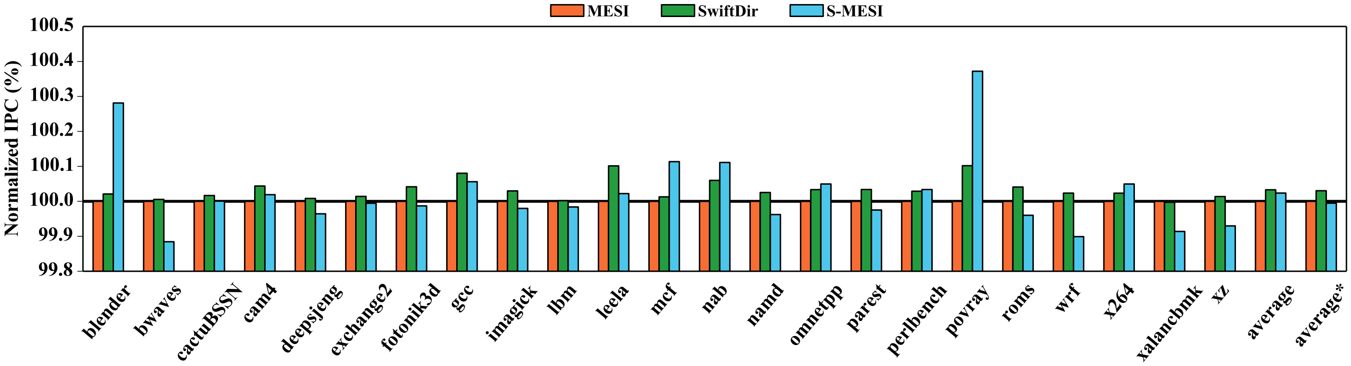


Figure 7. Single-threaded SPEC benchmarks – Normalized IPC of SwiftDir and S-MESI over that of MESI.

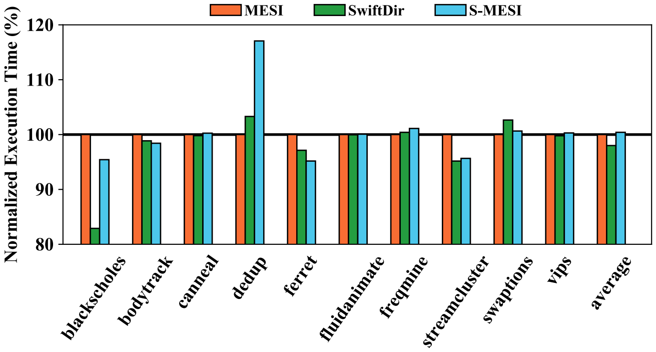


Figure 8. Multi-threaded PARSEC benchmarks – Normalized execution time over MESI.

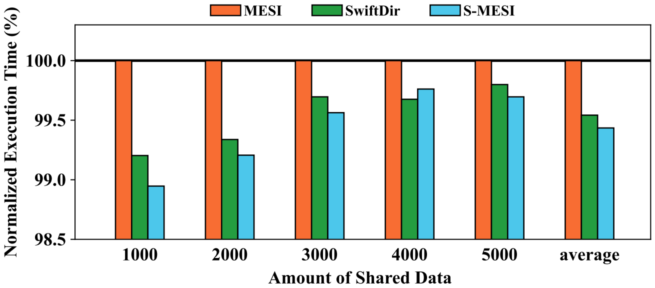


Figure 9. Multi-threaded read-only benchmarks – Normalized execution time over MESI.

speedup. We start with evaluating the effect of serving E-state shared data from the LLC in this section and continue with evaluating the effect of silent E→M transition for unshared data on the L1 in Section V-E.

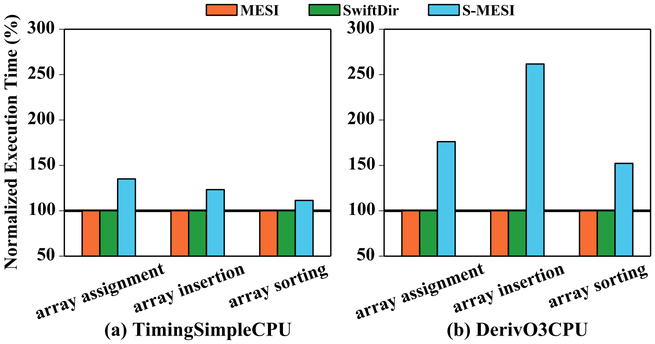
We construct a two-threaded application and pin the threads to respective cores. We first run one thread to access a series of exploitable shared data. Then we run the other cross-core thread to re-access the accessed data through remote loads. Such remote loads lead to E→S transitions for MESI yet more efficient S→S transitions for both S-MESI and SwiftDir. We measure the time for the re-access process. Figure 9 reports the normalized measurement results given different amounts of exploitable shared data ranging from 1,000 to 5,000. SwiftDir shows a comparative performance speedup as S-MESI. SwiftDir and S-MESI reduces the execution time of MESI by 0.46% and 0.57% on average, respectively.

E. Performance: Write-after-read Intensive

Finally, we build write-after-read intensive applications for performance evaluation. We consider these applications with particular interest because S-MESI leads to L1-L2 communication upon every write-after-read operation. Such communication is, however, unnecessary in MESI and SwiftDir where silent upgrade (i.e., E→M transition) takes place in the L1 cache. Furthermore, S-MESI enforces such communication on not only exploitable shared data but also unshared data that are secure against coherence covert-channel attacks. Such an overprotection may further slow down S-MESI. To validate the preceding observations, we compile three typical applications with intensive write-after-read operations—array assignment, array insertion, and array sorting.

Figure 10 reports the execution time of SwiftDir and S-MESI normalized over that of MESI. SwiftDir and MESI show a comparative performance. Both outperform S-MESI in all the three write-after-read intensive applications. In particular, Figure 10(a) and Figure 10(b) report results using `TimingSimpleCPU` and `DerivO3CPU` in gem5 configuration, respectively. `TimingSimpleCPU` specifies a CPU type that does not support out-of-order execution. It helps to scrutinize how coherence overprotection affects write-after-read performance. In contrast, `DerivO3CPU` integrates out-of-order execution into the CPU. We use `DerivO3CPU` to demonstrate how coherence overprotection further hinders performance.

As Figure 10(a) shows, SwiftDir takes a 25.99%, 18.91%, and 10.23% shorter time than S-MESI does to run array assignment, array insertion, and array sorting. This accordingly yields a speedup of 1.35×, 1.23×, and 1.14× even without out-of-order execution. Once augmented with out-of-order execution (Figure 10(b)), SwiftDir demonstrates a higher potential for speedup. It reduces the execution time of S-MESI for array assignment, array insertion, and array sorting by 43.23%, 61.79%, and 34.3%, respectively. In other words, SwiftDir boosts the speedup to 1.71×, 2.62×, and 1.52×.



(a) TimingSimpleCPU

(b) DerivO3CPU

Figure 10. Write-after-read intensive benchmarks – Normalized execution time over MESI.

VI. Conclusion

We have presented and evaluated SwiftDir as the first attempt to secure cache coherence with performance gains. We find that exploitable shared data belong to write-protected data, whose E state is redundant. We can simply remove their E state from coherence to fundamentally throttle the exploited E/S timing difference. This promises a way of protection by simplification rather than complication. SwiftDir explores a series of implementation strategies to benefit practical systems. We validate security and performance of SwiftDir through extensive single-threaded SPEC benchmarks, multi-threaded PARSEC benchmarks, multi-threaded read-only benchmarks, and write-after-read intensive benchmarks. It outperforms not only secure S-MESI but also unprotected MESI.

Acknowledgment

We would like to sincerely thank MICRO 2022 Chairs and Reviewers for your helpful feedback. We would also like to thank authors that published or shared the source code of relate work for helping with our implementation and evaluation. We wish you health and safety during the pandemic.

References

[1] V. Agrawal, A. Dabral, T. Palit, Y. Shen, and M. Ferdman, “Architectural support for dynamic linking,” in *ASPLOS*, 2015, pp. 691–702.

[2] A. Arcangeli, I. Eidus, and C. Wright, “Increasing memory density by using ksm,” in *Proceedings of the Linux Symposium*, 2009, pp. 19–28.

[3] ARM, “Virtual and physical tags and indexes.” [Online]. Available: https://developer.arm.com/documentation/den0013/d/Caches/Cache-architecture/Virtual-and-physical-tags-and-indexes

[4] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, “Branch history injection: On the effectiveness of hardware mitigations against cross-privilege spectre-v2 attacks,” in *USENIX Security Symposium*, 2022.

[5] S. Barker, T. Wood, P. Shenoy, and R. Sitaraman, “An empirical study of memory sharing in virtual machines,” in *ATC*, 2012, pp. 273–284.

[6] S. Bartell, W. Dietz, and V. S. Adve, “Guided linking: dynamic linking without the costs,” in *OOPSLA*, 2020, pp. 1–29.

[7] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison *et al.*, “Speculative interference attacks: Breaking invisible speculation schemes,” in *ASPLOS*, 2021, pp. 1046–1060.

[8] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, “Smotherspectre: exploiting speculative execution through port contention,” in *CCS*, 2019, pp. 785–800.

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: Characterization and architectural implications,” in *PACT*, 2008, pp. 72–81.

[10] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, S. Rathijit, S. Korey, S. Muhammad, V. Nilay, D. H. Mark, and A. W. David, “The gem5 simulator,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[11] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, “Dedup est machina: Memory deduplication as an advanced exploitation vector,” in *S&P*, 2016, pp. 987–1004.

[12] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, “Reload+refresh: Abusing cache replacement policies to perform stealthy cache attacks,” in *USENIX Security Symposium*, 2020, pp. 1967–1984.

[13] J. Bucek, K.-D. Lange, and J. v. Kistowski, “Spec cpu2017: Next-generation compute benchmark,” in *ICPE*, 2018, pp. 41–42.

[14] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, B. Jo Van, and Y. Yuval, “Fallout: Leaking data on meltdown-resistant cpus,” in *CCS*, 2019.

[15] C.-R. Chang, J.-J. Wu, and P. Liu, “An empirical study on memory sharing of virtual machines for server consolidation,” in *ISPA*, 2011, pp. 244–249.

[16] M. H. I. Chowdhuryy, H. Liu, and F. Yao, “Branchspec: Information leakage attacks exploiting speculative branch instruction executions,” in *ICCD*, 2020, pp. 529–536.

[17] S. Deng, W. Xiong, and J. Szefer, “Secure tlbs,” in *ISCA*, 2019, pp. 346–359.

[18] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, “Branchscope: A new side-channel attack on directional branch predictor,” in *ASPLOS*, 2018, pp. 693–707.

[19] A. Franques, A. Kokolis, S. Abadal, V. Fernando, S. Misailovic, and J. Torrellas, “Widir: A wireless-enabled directory cache coherence protocol,” in *HPCA*, 2021, pp. 304–317.

[20] gem5, “gem5.” [Online]. Available: http://www.gem5.org

[21] gem5, “Ruby memory system: Mesi two level.” [Online]. Available: https://www.gem5.org/documentation/general_docs/ruby/MESI_Two_Level/

[22] GNU, “Index of /gnu/libc.” [Online]. Available: https://ftp.gnu.org/gnu/libc/

[23] J. Goodman and H. Hum, “Mesif: A two-hop cache coherency protocol for point-to-point interconnects (2009),” *URL: https://www. cs. auckland. ac. nz/˜goodman/TechnicalReports/MESIF-2009. pdf*, 2004.

[24] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, “Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks,” in *USENIX Security Symposium*, 2018, pp. 955–972.

[25] D. Gruss, E. Kraft, T. Tiwari, M. Schwarz, A. Trachtenberg, J. Hennessey, A. Ionescu, and A. Fogh, “Page cache attacks,” in *CCS*, 2019, pp. 167–180.

[26] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, “Flush+flush: a fast and stealthy cache attack,” in *DIMVA*, 2016, pp. 279–299.

[27] D. Gruss, R. Spreitzer, and S. Mangard, “Cache template attacks: Automating attacks on inclusive last-level caches.” in *USENIX Security Symposium*, 2015, pp. 897–912.

[28] F. Guo, Y. Li, Y. Xu, S. Jiang, and J. C. Lui, “{SmartMD}: A high performance deduplication engine with mixed pages,” in *ATC*, 2017, pp. 733–744.

[29] F. Guo and P. Efstathopoulos, “Building a high-performance deduplication system,” in *ATC*, 2011.

[30] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, “Difference engine: Harnessing memory redundancy in virtual machines,” *Communications of the ACM*, vol. 53, no. 10, pp. 85–93, 2010.

[31] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2017.

[32] A. M. Kaushik, M. Hassan, and H. Patel, “Designing predictable cache coherence protocols for multi-core real-time systems,” *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2098–2111, 2020.

[33] S. Kaxiras and A. Ros, “A new perspective for efficient virtual-cache coherence,” in *ISCA*, 2013, pp. 535–546.

[34] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, “Flipping bits in memory without accessing them: an experimental study of dram disturbance errors,” in *ISCA*, 2014, pp. 361–372.

[35] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, S. Michael, and Y. Yuval, “Spectre attacks: Exploiting speculative execution,” in *S&P*, 2019, pp. 1–19.

[36] J. Lee, S. Hong, and S. Kim, “Tlb index-based tagging for cache energy reduction,” in *ISLPED*, 2011, pp. 85–90.

[37] J. R. Levine, “Linkers and loaders,” *The Morgan Kaufmann Series in Software Engineering and Programming*, 200.

[38] M. Li, C. Miao, Y. Yang, and K. Bu, “unxpec: Breaking undo-based safe speculation,” in *HPCA*, 2022, pp. 98–112.

[39] Linux, “mmap - linux manual page.” [Online]. Available: https://man7.org/linux/man-pages/man2/mmap.2.html

[40] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, “Take a way: Exploring the security implications of amd’s cache way predictors,” in *AsiaCCS*, 2020, pp. 813–825.

[41] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yuval, and R. Mike Hamburg, “Meltdown: Reading kernel memory from user space,” in *USENIX Security Symposium*, 2018, pp. 973–990.

[42] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-level cache side-channel attacks are practical,” in *S&P*, 2015, pp. 605–622.

[43] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, “The gem5 simulator: Version 20.0+,” *arXiv preprint arXiv:2007.03152*, 2020.

[44] G. Maisuradze and C. Rossow, “ret2spec: Speculative execution using return stack buffers,” in *CCS*, 2018, pp. 2109–2122.

[45] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, “{XLH}: More effective memory deduplication scanners through cross-layer hints,” in *ATC*, 2013, pp. 279–290.

[46] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, “A primer on memory consistency and cache coherence,” *Synthesis Lectures on Computer Architecture*, vol. 15, no. 1, pp. 1–294, 2020.

[47] M. Oliverio, H. Bos, K. Razavi, and C. Giuffrida, “Secure page fusion with vusion,” in *SOSP*, 2017, pp. 531–545.

[48] N. Oswald, V. Nagarajan, and D. J. Sorin, “Protogen: Automatically generating directory cache coherence protocols from atomic specifications,” in *ISCA*, 2018, pp. 247–260.

[49] N. Oswald, V. Nagarajan, and D. J. Sorin, “Hieragen: Automated generation of concurrent, hierarchical cache coherence protocols,” in *ISCA*, 2020, pp. 888–899.

[50] B. Panda, “Fooling the sense of cross-core last-level cache eviction based attacker by prefetching common sense,” in *PACT*, 2019, pp. 138–150.

[51] M. S. Papamarcos and J. H. Patel, “A low-overhead coherence solution for multiprocessors with private cache memories,” in *ISCA*, 1984, pp. 348–354.

[52] M. Parasar, A. Bhattacharjee, and T. Krishna, “Seesaw: Using superpages to improve vipt caches,” in *ISCA*, 2018, pp. 193–206.

[53] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, “Systematic analysis of randomization-based protected cache architectures,” in *S&P*, 2021, pp. 987–1002.

[54] G. Saileshwar, C. W. Fletcher, and M. Qureshi, “Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion,” in *ASPLOS*, 2021, pp. 1077–1090.

[55] G. Saileshwar and M. K. Qureshi, “Cleanupspec: An” undo” approach to safe speculation,” in *MICRO*, 2019, pp. 73–86.

[56] strace, “strace: linux syscall tracer.” [Online]. Available: https://strace.io/

[57] A. Tatar, D. Trujillo, C. Giuffrida, and H. Bos, “Tlb; dr: Enhancing tlb-based attacks with tlb desynchronized reverse engineering,” in *USENIX Security Symposium*, 2022.

[58] L. Torvalds, “Linux.” [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/x86?h=v5.16.13

[59] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, “Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution,” in *USENIX Security Symposium*, 2018, pp. 991–1008.

[60] Q. Wang, Y. Lu, E. Xu, J. Li, Y. Chen, and J. Shu, “Concordia: Distributed shared memory with {In-Network} cache coherence,” in *FAST*, 2021, pp. 277–292.

[61] W. Xiong and J. Szefer, “Leaking information through cache lru states,” in *HPCA*, 2020, pp. 139–152.

[62] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, “Invisispec: Making speculative execution invisible in the cache hierarchy,” in *MICRO*, 2018, pp. 428–441.

[63] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, “Attack directories, not caches: Side channel attacks in a non-inclusive world,” in *S&P*, 2019, pp. 888–904.

[64] M. Yan, J.-Y. Wen, C. W. Fletcher, and J. Torrellas, “Secdir: a secure directory to defeat directory side-channel attacks,” in *ISCA*, 2019, pp. 332–345.

[65] F. Yao, M. Doroslovacki, and G. Venkataramani, “Are coherence protocol states vulnerable to information leakage?” in *HPCA*, 2018, pp. 168–179.

[66] F. Yao, M. Doroslovački, and G. Venkataramani, “Covert timing channels exploiting cache coherence hardware: Characterization and defense,” *International Journal of Parallel Programming*, vol. 47, no. 4, pp. 595–620, 2019.

[67] Y. Yarom and K. Falkner, “Flush+reload: A high resolution, low noise, l3 cache side-channel attack,” in *USENIX Security Symposium*, 2014, pp. 719–732.

[68] Z. N. Zhao, H. Ji, A. Morrison, D. Marinov, and J. Torrellas, “Pinned loads: taming speculative loads in secure processors,” in *ASPLOS*, 2022, pp. 314–328.

[69] Z. Zhou, M. K. Reiter, and Y. Zhang, “A software approach to defeating side channels in last-level caches,” in *CCS*, 2016, pp. 871–882.