HARDWARE SIDE-CHANNEL ATTACKS AND SOME SOLUTIONS

- Krishna Kavi, Regents Professor
- University of North Texas
- CSRL.CSE.UNT.EDU/KAVI
- KRISHNA.KAVI@UNT.EDU

WHAT IS A SIDE-CHANNEL ATTACK

A direct attack involves modifying victim’s program or the computing environment in which it runs.

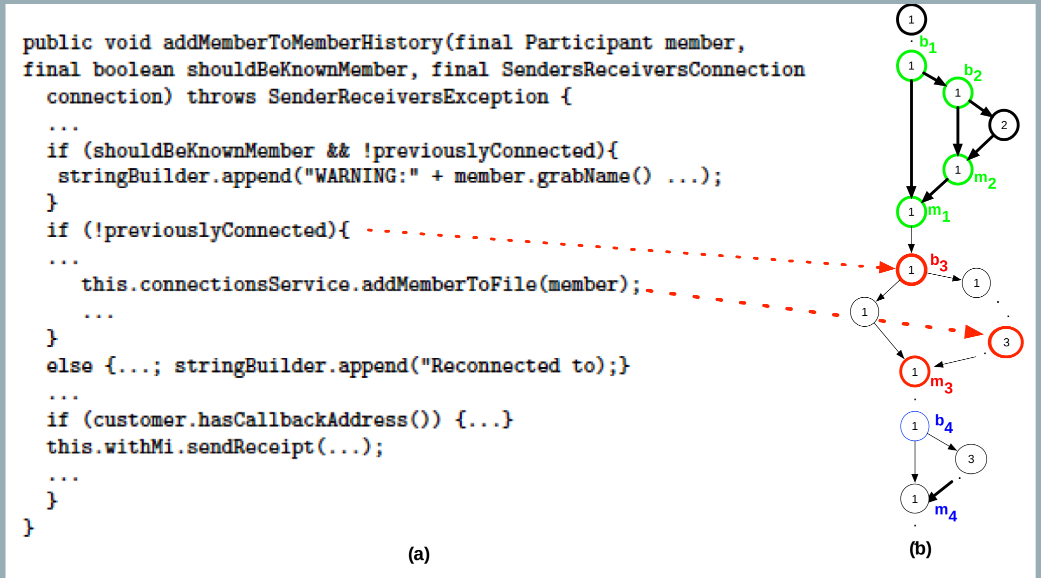
A side-channel attack does not directly modify victim’s application but makes observations about the application and ”derives” information about the victim’s application from the observations.

Some typical types of observations made during side-channel attacks.

Timing Attacks: Measure execution time of user applications

To determine which execution path taken

And possibly obtain values of variables



```
public void addMemberToMemberHistory(final Participant member,
final boolean shouldBeKnownMember, final SendersReceiversConnection
  connection) throws SenderReceiversException {
  ...
  if (shouldBeKnownMember && !previouslyConnected){
   stringBuilder.append("WARNING:" + member.grabName() ...);
  }
  if (!previouslyConnected){
  ...
     this.connectionsService.addMemberToFile(member);
     ...
  }
  else {...; stringBuilder.append("Reconnected to);}
  ...
  if (customer.hasCallbackAddress()) {...}
  this.withMi.sendReceipt(...);
  ...
  }
}
```

(a)

(b)

WHAT IS A SIDE-CHANNEL ATTACK

Timing Attacks: Measure execution time of user applications

Determine cache misses and identify which victim's memory was accessed

Power/Energy Attacks: Measure the amount of power/energy consumed by an application

If the attacker cannot measure timing accurately, it may be possible

to obtain timing information from energy consumption

Electromagnetic waves emitted: To observer what information was displayed on a monitor

Some such attacks on IIoT's in advanced manufacturing systems have been reported

…..

ABC OF CACHE MEMORIES

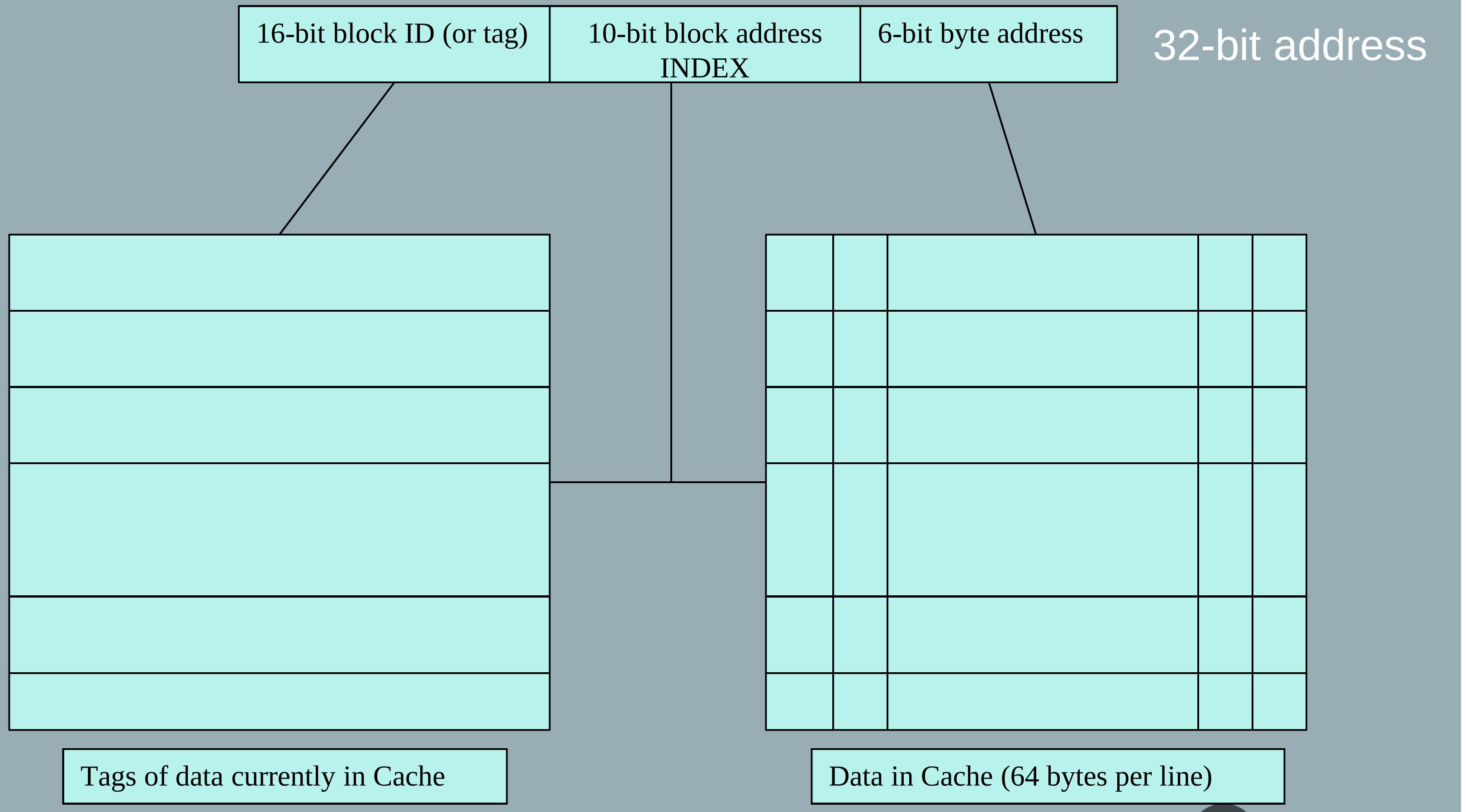
Two architectural features that aid side-channel attacks

Speculative execution

Cache memory accesses

How does a cache work?
Consider a 64KB direct mapped cache with 64byte cache blocks

Notice how the cache is INDEXED



16-bit block ID (or tag)

10-bit block address INDEX

6-bit byte address

32-bit address

Tags of data currently in Cache

Data in Cache (64 bytes per line)

CACHE MEMORIES

If we use set-associative cache, we then refer the Index as the index to a set.

A set contains more than one entry and

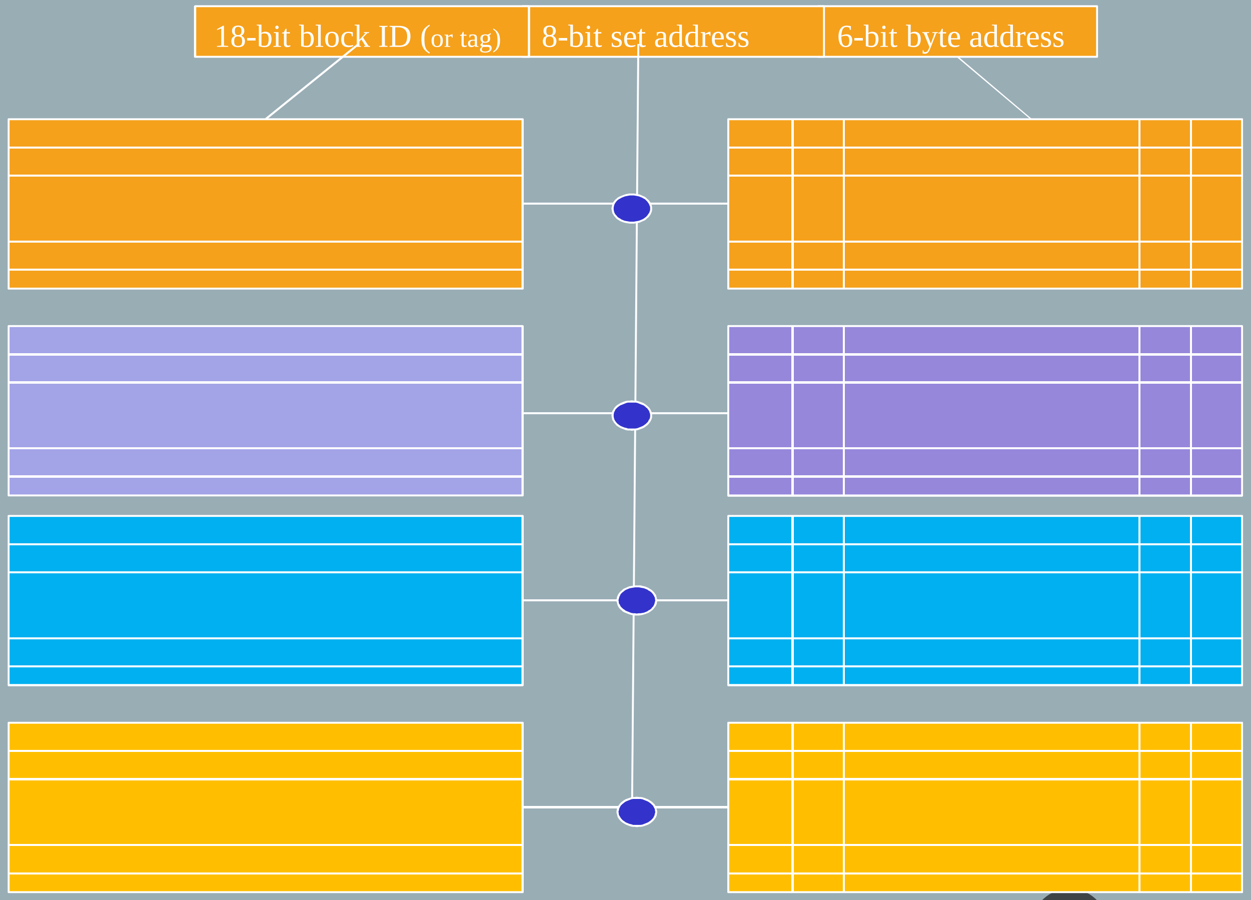
we search all the tags associated to find a match (hit)

Here we have a 4-way set associative cache

I use this depiction for the purpose of understanding.

One can view associativity as splitting cache into equal-sized partitions

A set is comprised of one cache line from each of the 4 portions



CACHE SIDE-CHANNEL ATTACK

The key takeaway is: Multiple addresses map to the same INDEX in cache

since many different address can have the same value in the index field

We need to evict current occupants to make room for new data

We can use LRU to evict one line of a set

Or use other methods including random choice

Some systems “reserve” certain “ways” in a set for secure applications

Most cache based side-channel attacks rely on conflicts in cache

If the attacker’s cache data is evicted, it is evicted by a victim’s data

The index portion of the victim’s data address is the same as that of the attacker

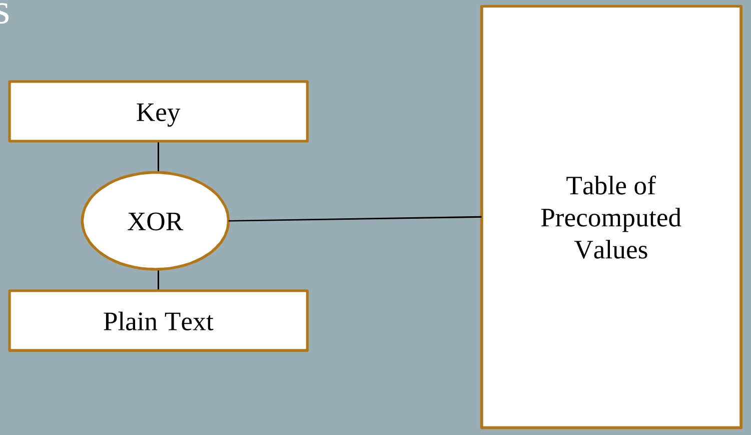
CACHE SIDE-CHANNEL ATTACK

Some examples of well known attacks: Bernstein’s attack on AES

A common implementation uses tables containing some AES computations (SubByte and MixColumns)

The tables are accessed using a portion of the key XORed with plain text

The data in the tables will be stored in Cache



Thus the index to the table forms a partial Cache Address

And we can then predict cache location (or INDEX) from this address

If we know which cache entry is evicted, then we can find the address and thus the table index and a portion of the AES key!

CACHE SIDE-CHANNEL ATTACK

In order to perform any cache attacks, the attacker needs

information regarding cache architecture (associativity, block size and capacity)

And fine-grained measurements on cache misses

Most modern processors have Performance Counters (or registers) that can be programmed

to monitor different types of events, including cache misses

Most Operating Systems have libraries that can be used for this purpose

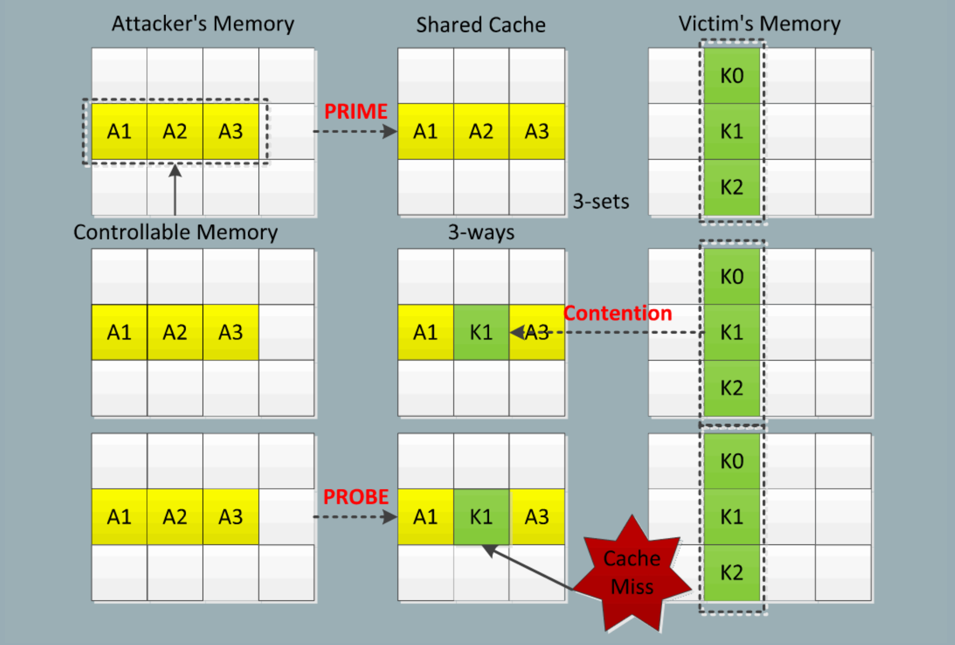
(Android does not provide an easy way to obtain such information)

CACHE SIDE-CHANNEL ATTACK

Some common approaches to cache side channel attacks

Prime and Probe Attack

- Attacker fills entire cache (prime)
- When victim application executes, it will likely evict some of attacker data
- Attacker can detect which cache lines were evicted when his/her code is executed again



CACHE SIDE-CHANNEL ATTACK

Evict and Time

- Attacker evicts a specific cache entry (or entries), accessing a specific address
  - if this causes cache misses when victim program is executed, it reveals victim's address
  - Attacker can repeat this process to evict every cache entry and obtain all addresses accessed by victim

- The information regarding cache misses can be obtained using hardware performance counters
  - Some counters provide fine-grained information → to allow optimizations
  - Some new hardware systems are proposing to NOT provide such detailed information

CACHE SIDE-CHANNEL ATTACK

Flush and Reload (similar to Prime and Probe)

- Can be used to evict "shared" data across multiple cores
  - clflush instruction evicts data from all cache levels including shared Last Level Cache

- Allows attacker to run on a different core and observe victim's execution
- Works to reveal shared pages
  - Shared pages are created to save space for shared information (de-duplication)

In all these attacks, attacker relies on the traditional cache indexing

- Solutions either rely on "partitioning" or "randomizing" cache accesses

CACHE SIDE-CHANNEL ATTACK SOLUTIONS

Partitioning can be achieved either in software or hardware

- Memory can be partitioned such that secure applications are executed in secure partitions
- Some cache designs (ARM with trusted zones) flags data belonging to secure
  (and non-secure) applications
- A non-secure application cannot evict cache lines belonging to secure applications
- However this does not prevent "prime and probe" attacks

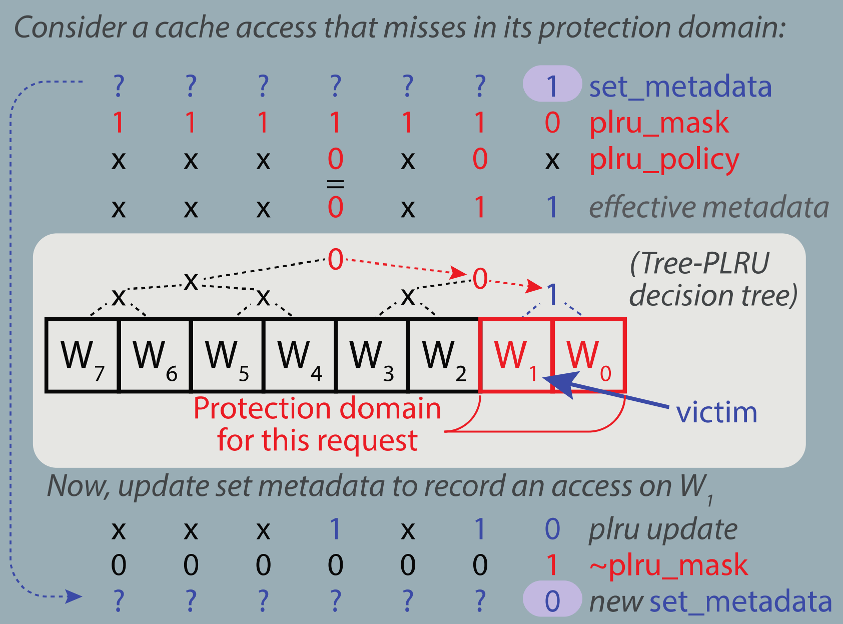
It is also possible to partition cache and reserve some portions for secure applications

A recent proposal reserves "ways" in set-associate cache to secure applications

- For example, in a 8-way associative cache, one can set aside 4-ways for secure computations.
- Non-secure computations cannot use these ways or evict secure ways

However, this limits the available cache and thus cause performance penalties

CACHE SIDE-CHANNEL ATTACK SOLUTIONS



Consider a cache access that misses in its protection domain:

?	?	?	?	?	?	1	set_metadata
1	1	1	1	1	1	0	plru_mask
x	x	x	0	x	0	x	plru_policy
x	x	x	0	x	1	1	*effective metadata*

(Tree-PLRU decision tree)
0
x
0
x
x
x
1
W_7 W_6 W_5 W_4 W_3 W_2 W_1 W_0
Protection domain for this request
victim

Now, update set metadata to record an access on W_1

x	x	x	1	x	1	0	*plru update*
0	0	0	0	0	0	1	~plru_mask
?	?	?	?	?	?	0	*new* set_metadata

Note that we need to protect meta data also (such as LRU and coherency information)

Can cause performance penalties since each application now has smaller cache capacities

Attacker can only evict victim data if they are assigned to the same protection level (hence same ways)
LRU replacement information used for replacement is also per way basis

V. Kiriansky, et. Al. “DAWG: A defense against cache timing attacks in speculative execution processors”, MICRO-2018

CACHE SIDE-CHANNEL ATTACK SOLUTIONS

The second technique is randomizing cache access

Again, this can be achieved in hardware or software

- OS can randomize where user stack/heap data is stored → ASLR
- This makes it difficult for an attacker to launch attacks forcing victims data in
  - specific locations evicted

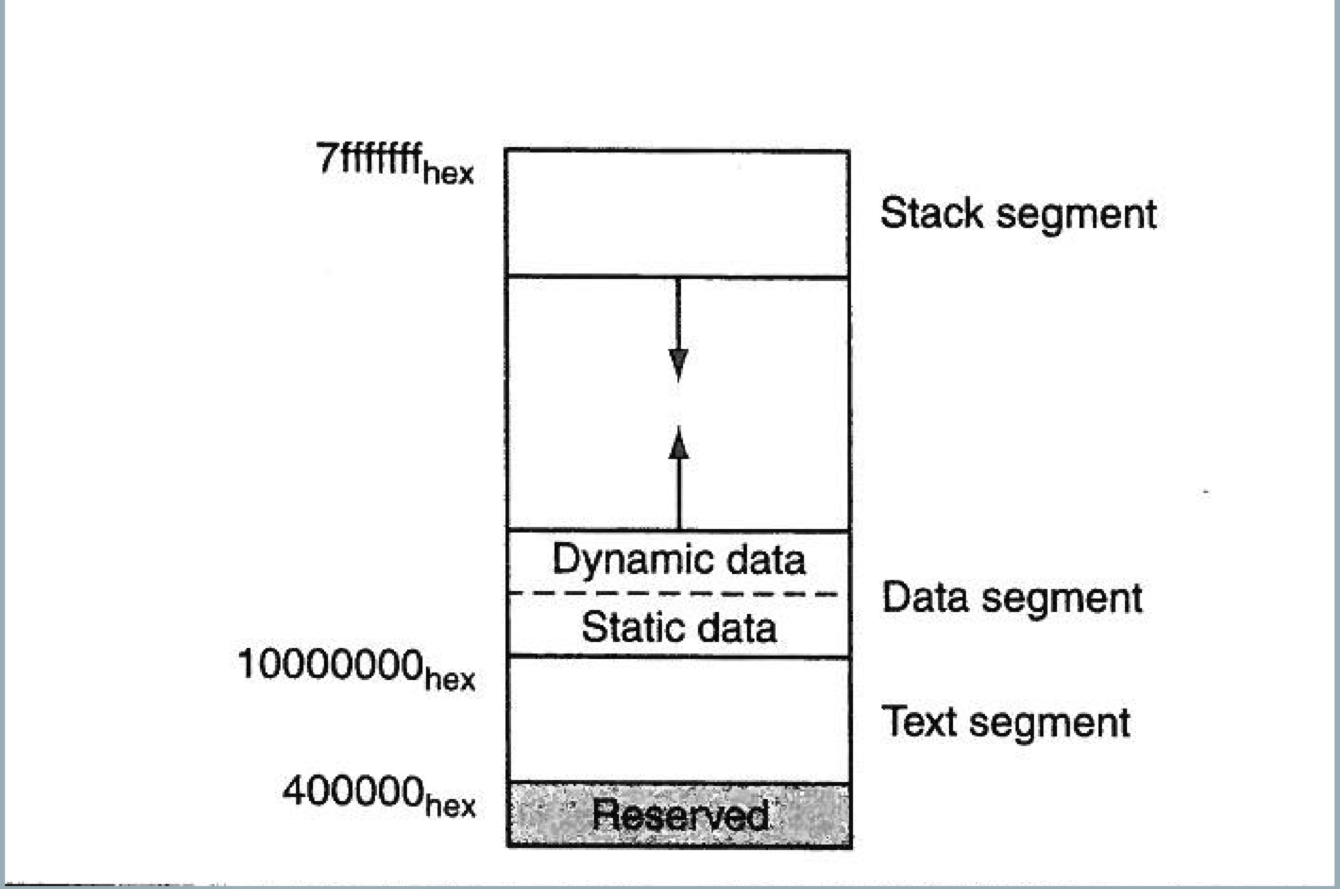
- Applications can also introduce some randomness in the data being accessed
  - create random accesses to unneeded data

All randomization techniques have performance implications

CACHE SIDE-CHANNEL ATTACK SOLUTIONS

Randomizing memory allocation

Some attacks that rely on the knowledge of memory address (such as OS pages) can be mitigated



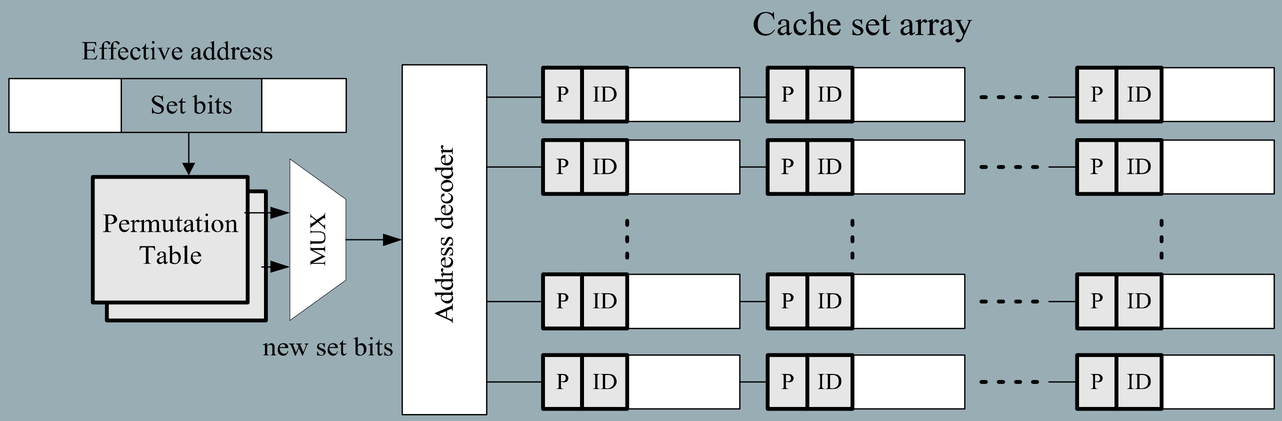
One solutions is randomize where OS libraries are located in the address space (ASLR)

CACHE SIDE-CHANNEL ATTACK SOLUTIONS

Now let us look at some hardware level randomization techniques

Wong and Lee proposed to redirect cache accesses to different locations

- The cache index is used as an index into a “permutation” table
- The permutation table indicates the actual cache location (or index) for the data
- By changing the entries in the permutation table, you can randomize the placement of data



Z. Wang and R. Lee. New cache designs for thwarting software cache side-channel attacks, ISCA-2007

CACHE SIDE-CHANNEL ATTACK SOLUTIONS

A couple of performance problems with this solution

Need to use an additional table lookup to access cache

This delays the “critical path” of cache access, significantly impacting performance if used at L1 level

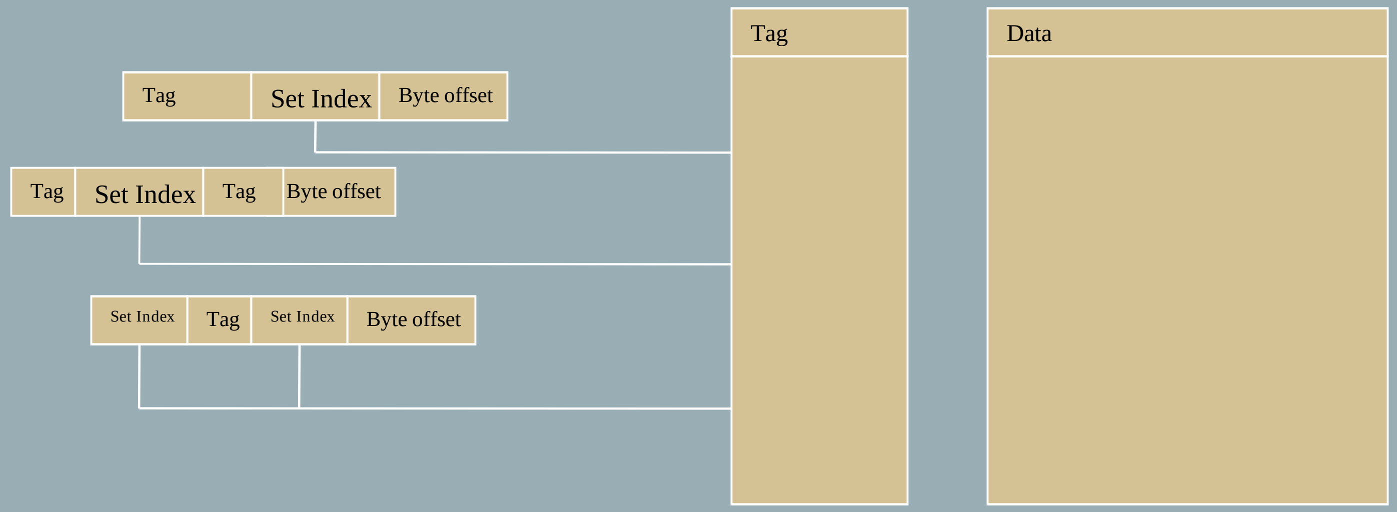
The tables must be reloaded for each application

Otherwise, the same index is redirected to the same cache location

CACHE SIDE-CHANNEL ATTACK SOLUTIONS

Our solution also randomizes cache access, but easier to implement in hardware and does not add to access delays.

The bits comprising index can be programmed for each application



In this figure we show the possibility of having multiple indexes applied “at the same” time
Each index being used for a different application

CACHE SIDE-CHANNEL ATTACK SOLUTIONS

Different indexes can be implemented using "masking"

A configuration register can indicate which Indexing method to use for each process

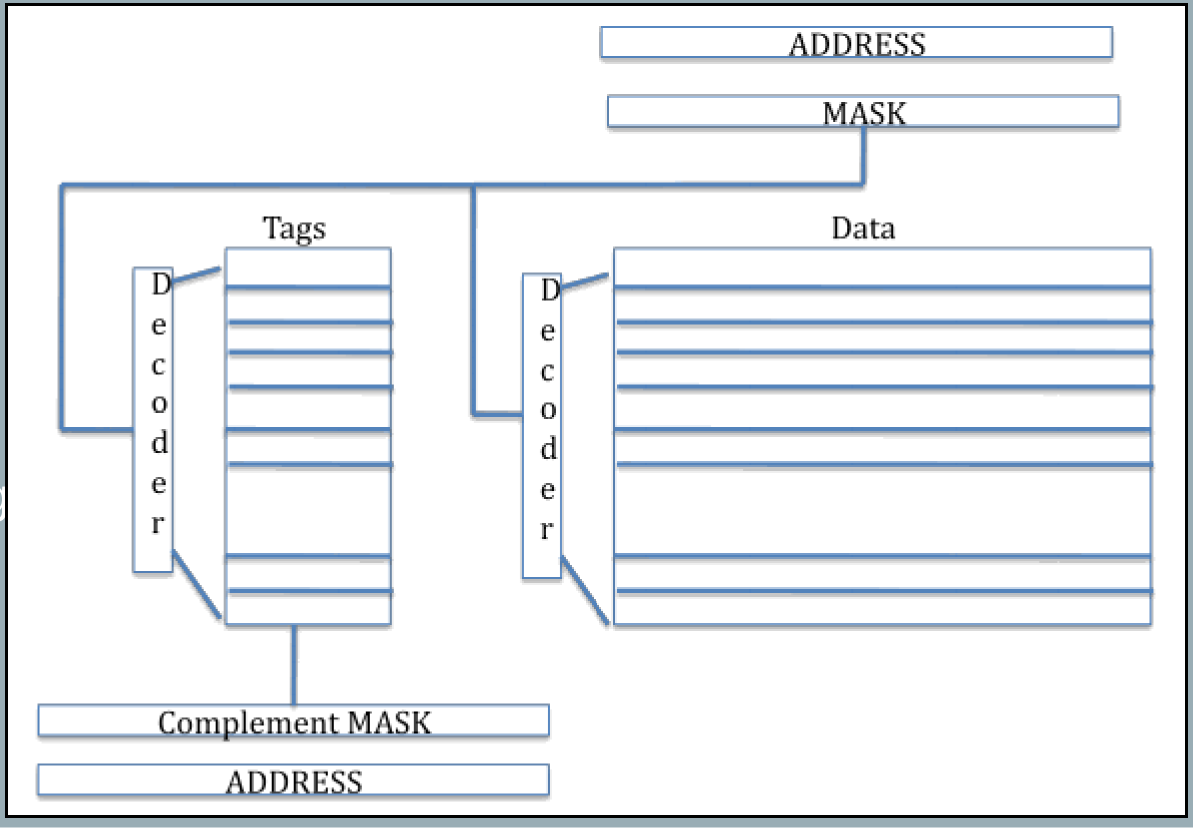


Figure 2. Address decoding using masks

CACHE SIDE-CHANNEL ATTACK SOLUTIONS

Attacker will not be able guess the “index” of the victim data that is evicted (or evicted attacker’s data)

Since the attacker and victim applications use different address bits for index, attacker cannot predict the victim’s memory address

Will this approach increase or decrease cache miss rates (and thus performance)?

Since we may be eliminating some cache conflicts, performance in multicore systems may improve

However, the index choice may reduce cache localities and thus increase cache misses for each application

What are some choices for different index bits?

Shift or rotate index bits
XOR index with a portion of tag bits
AND or OR index with a portion of the tag bits

CACHE SIDE-CHANNEL ATTACK SOLUTIONS

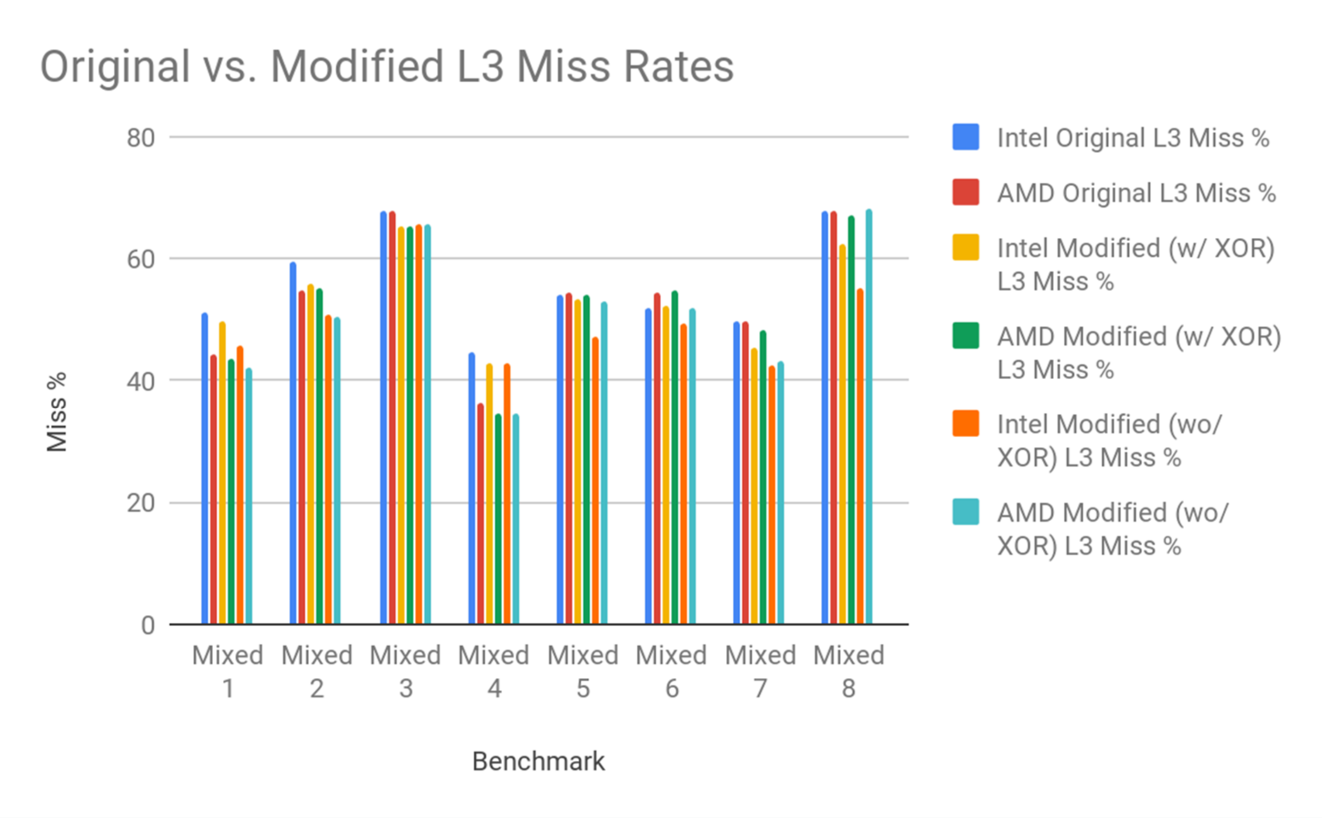
Some initial results

Using XOR index with Tag
and rotating index bits

Each of 4 applications running concurrently use different indexing

This graph shows cache misses

Benchmarks consist of different applications on different cores



CACHE SIDE-CHANNEL ATTACK SOLUTIONS

Some results

Here we show performance loss in terms of (simulated) clock cycles

Note in one case, there is performance gain.

Original Sim Time	Modified Sim Time	Performance Loss
17805945346	17977248299	0.96%
37515648446	38184416139	1.78%
14992571428	15039425602	0.31%
18569982856	18763125843	1.04%
13490766905	13511609002	0.15%
9926563261	9923765935	-0.03%
18249581770	18380349795	0.72%
14997665398	15110962696	0.76%

CACHE SIDE-CHANNEL ATTACK SOLUTIONS

Information Theoretic analysis of cache side channel attacks

How many address bits can an attacker predict based on how addresses are mapped to cache locations?

Consider a simple module mapping (conventional) of addresses to cache lines

→ index depends on some specific bits of an address

16-bit block ID (or tag)	10-bit block address INDEX	6-bit byte address

So, the bit values of the 10 bit indexes can be determined from the set to which an address is mapped

For example, for set 0, all TEN index bits are zero

Entropy idea: if a specific address bit a_i is always 0 (or always 1) for set s_j then entropy = 0

if there is a 50% probability that bit a_i is zero and 50% probability it is one, entropy = 1

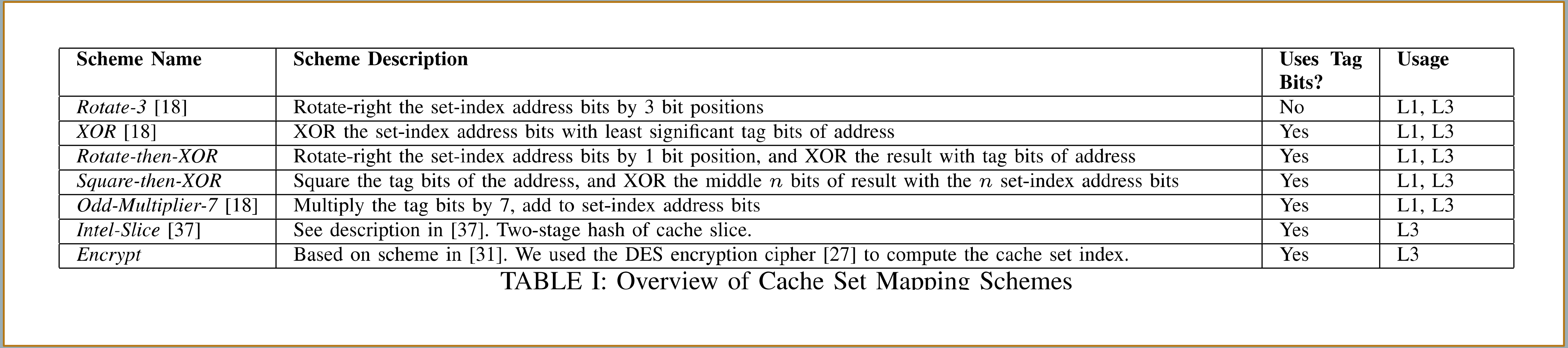
Low entropy → bits are predictable → more information leak

CACHE SIDE-CHANNEL ATTACK SOLUTIONS

We collected information leakage by different address mapping techniques using randomly generated memory accesses and using standard applications (benchmarks)

Tested using several different techniques for randomizing address mapping

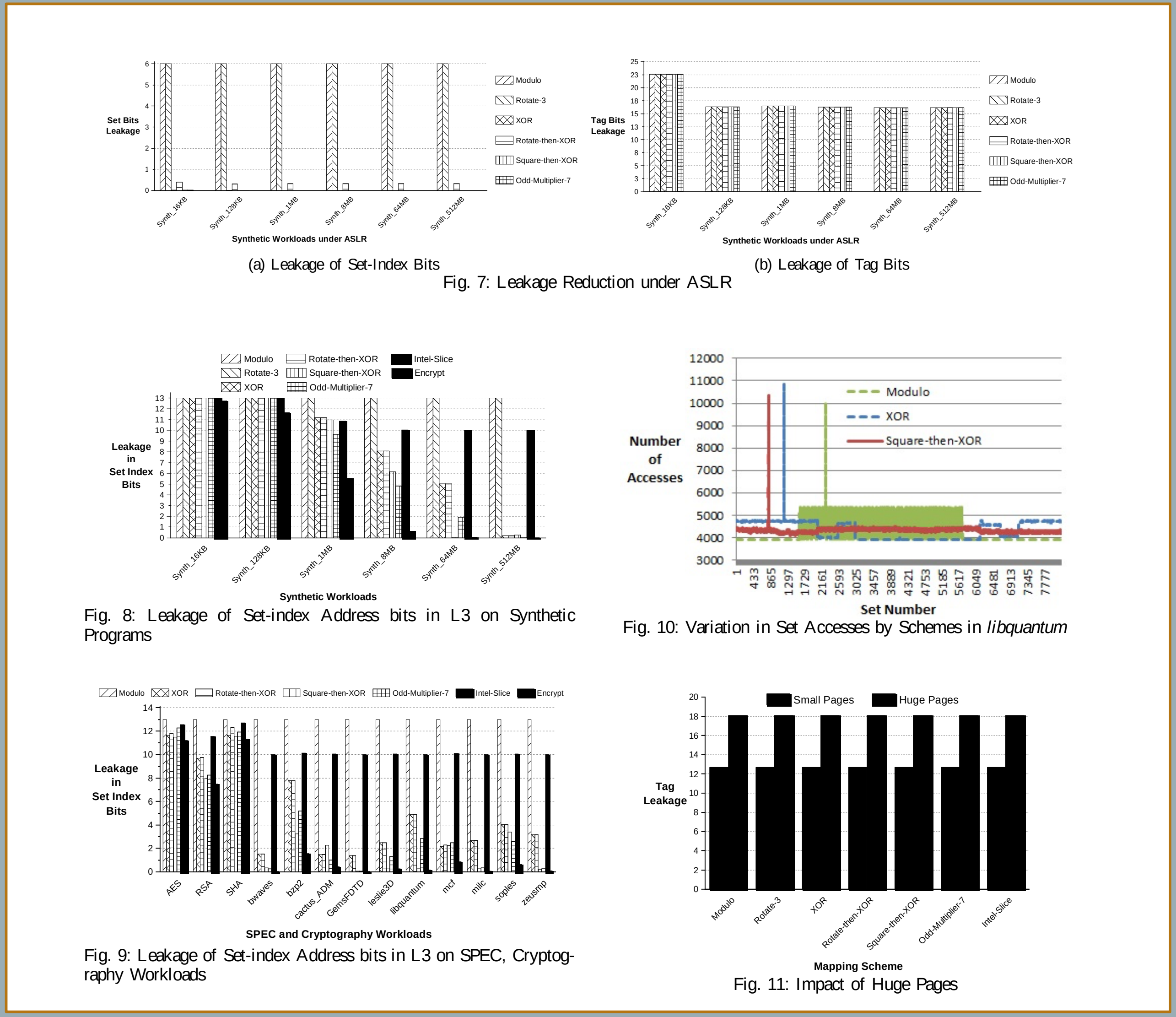
TAG bits	INDEX bits	Byte-Offset



Scheme Name	**Scheme Description**	**Uses Tag Bits?**	**Usage**
Rotate-3 [18]	Rotate-right the set-index address bits by 3 bit positions	No	L1, L3
XOR [18]	XOR the set-index address bits with least significant tag bits of address	Yes	L1, L3
Rotate-then-XOR	Rotate-right the set-index address bits by 1 bit position, and XOR the result with tag bits of address	Yes	L1, L3
Square-then-XOR	Square the tag bits of the address, and XOR the middle n bits of result with the n set-index address bits	Yes	L1, L3
Odd-Multiplier-7 [18]	Multiply the tag bits by 7, add to set-index address bits	Yes	L1, L3
Intel-Slice [37]	See description in [37]. Two-stage hash of cache slice.	Yes	L3
Encrypt	Based on scheme in [31]. We used the DES encryption cipher [27] to compute the cache set index.	Yes	L3

TABLE I: Overview of Cache Set Mapping Schemes

CACHE SIDE-CHANNEL ATTACK SOLUTIONS



(a) Leakage of Set-Index Bits

(b) Leakage of Tag Bits

Fig. 7: Leakage Reduction under ASLR

Fig. 8: Leakage of Set-index Address bits in L3 on Synthetic Programs

Fig. 10: Variation in Set Accesses by Schemes in *libquantum*

Fig. 9: Leakage of Set-index Address bits in L3 on SPEC, Cryptography Workloads

Fig. 11: Impact of Huge Pages

Summary of results

Almost all techniques leak some Index and some Tag bits

- ❖ Small programs leak more
- ❖ Techniques that create more uniform accesses to sets leak LESS

- ❖ Huge pages leak more

Programs inherently have predictable behaviors!

CACHE SIDE-CHANNEL ATTACK SOLUTIONS

Combined Performance vs Security (leakage) metric

May provide higher level of protection, but adversely impact performance

PS metric = (Avg Cache access time)*[(# of bits leaked)/(max bits that can be leaked)

→ Higher is better

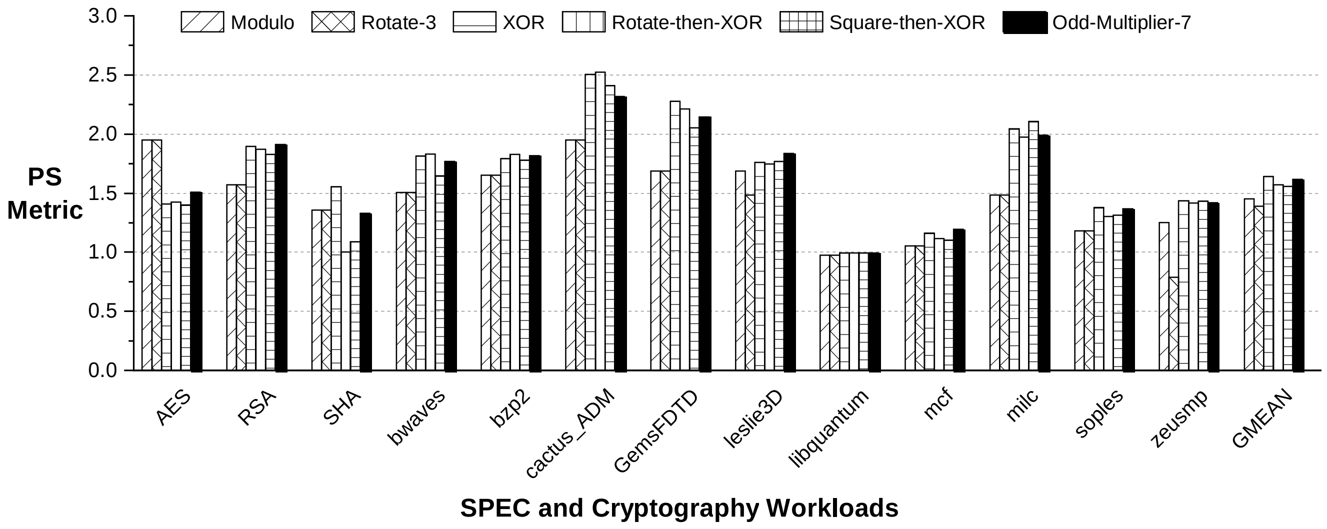


Fig. 12: L1D Performance-Security Product Metric of SPEC and Cryptography Workloads

CACHE SIDE-CHANNEL ATTACK SOLUTIONS

More experimentation is needed

More techniques → periodically changing mapping introduces more randomization (less leakage)

→ but will likely cause performance penalties (more cache misses)

Evaluate PS metric for partitioning techniques

→ partitioning leaks less information

→ but potentially causes higher performance penalties

Need to consider defining “information leakage” for timing attacks

How predictable are the execution times?

ATTACKS BASED SPECULATIVE EXECUTION

Timing Attacks

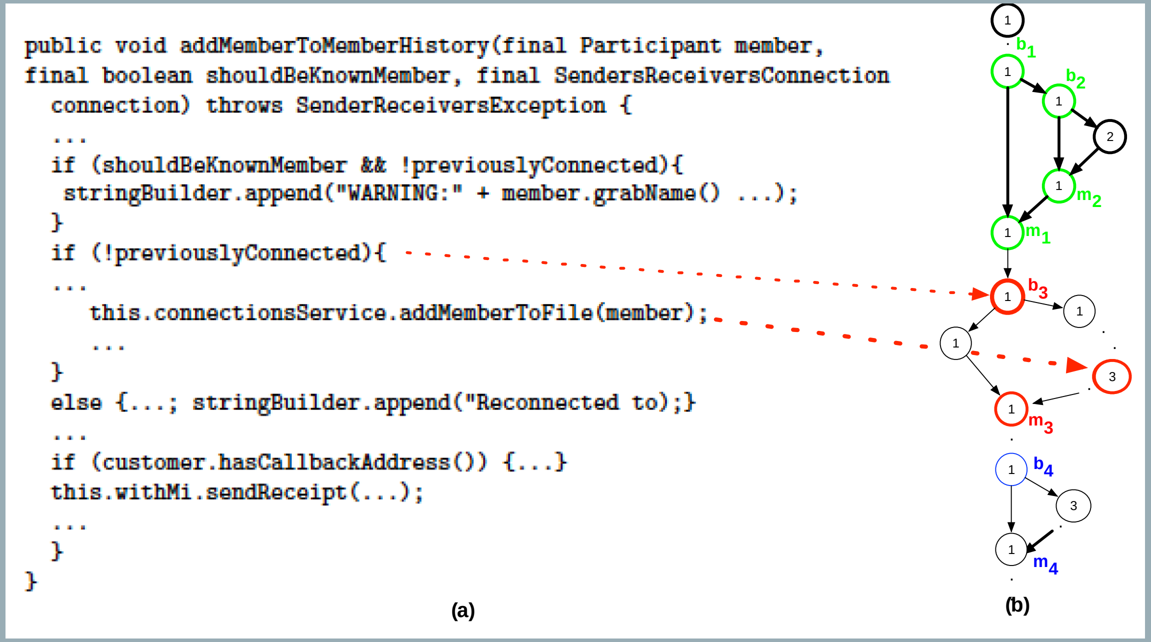
Measure execution times

Use to predict if some cache data is evicted or not

Also to predict which control path is taken

A careful analysis of branches and control paths may predict the input values.

Execution times may depend on the control paths taken during execution

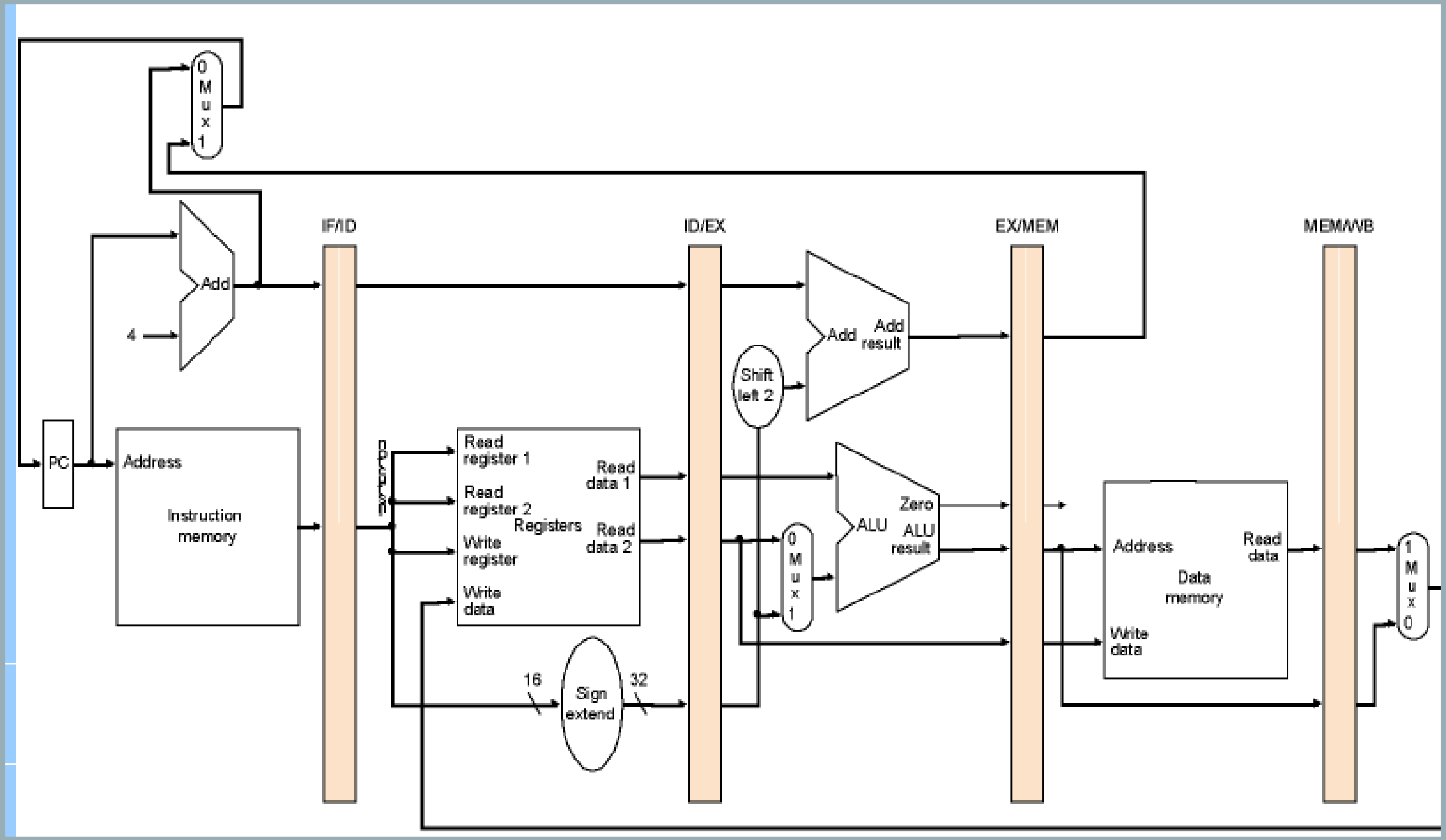


```
public void addMemberToMemberHistory(final Participant member,
final boolean shouldBeKnownMember, final SendersReceiversConnection
  connection) throws SenderReceiversException {
  ...
  if (shouldBeKnownMember && !previouslyConnected){
   stringBuilder.append("WARNING:" + member.grabName() ...);
  }
  if (!previouslyConnected){
  ...
    this.connectionsService.addMemberToFile(member);
    ...
  }
  else {...; stringBuilder.append("Reconnected to);}
  ...
  if (customer.hasCallbackAddress()) {...}
  this.withMi.sendReceipt(...);
  ...
  }
}
```

(a)

(b)

ABC OF SPECULATIVE EXECUTION



Branch decision is known only during Execute Stage

The address of branch target (if branch condition is true) is also known in Execute Stage

We may lose performance due to these delays

And hence the branch prediction and speculative execution

A simplified representation of instruction execution pipeline → in real systems there may as many as 20+ stages → branch prediction can say many cycles

THE BASIS FOR SIDE-CHANNEL ATTACKS

Modern processors use speculative execution and rely on Branch Prediction

Consider this simple example (from my class)

```
                                                      for (i=0; i<n; i++) {a[i] = a[i]+1;}
Loop:   ld      x2, 0(x1)         /* load a[i] into x2
        addi    x2, x2, 1         /* increment a[i]
        sd      x2, 0(x1)         /* store a[i]
        addi    x1, x1, 8         /* compute address of next a[i]
        bne     x1, x3, Loop      /* check if done
```

We can track execution cycles when a instruction starts executing, completes execution and writes results

We can see the impact of speculation by monitoring the execution of this code

SIDE-CHANNEL ATTACKS BASED ON SPECULATION

Iteration Number	Instruction	Fetch	Decode/read inputs	Execute	Memory	Result available	Commit
1	ld X2, 0(X1)	1	2	3	4	5	5
1	addi X2, X2, 1	2	5	6		6	6
1	sd X2, 0(X1)	3	6	7	8		8
1	addi X1, X1, #8	4	5	7		7	9
1	bne X1, X3, looop	5	7	8		8	10
2	ld X2, 0(X1)	6	7	8	9	9	11
2	addi X2, X2, 1	7	9	10		10	12
2	sd X2, 0(X1)	8	10	11	12		13
2	addi X1, X1, #8	9	10	11		11	13
2	bne X1, X3, looop	10	11	12		12	14
3	ld X2, 0(X1)	11	12	13	14	14	15
3	addi X2, X2, 1	12	14	15		15	16
3	sd X2, 0(X1)	13	15	16	17		18
3	addi X1, X1, #8	14	16	17		17	19
3	bne X1, X3, looop	15	17	18		18	20

ld from second iteration should not start executing until bne (branch if not-equal) from first iteration has completed

ld should not start execution until cycle 8.

But by predicting bne will be successful, we will speculatively execute ld from second iteration in cycle 7

Dual Issue Speculative Execution Example

SIDE-CHANNEL ATTACKS BASED ON SPECULATION

Note that the speculation also caused the ld (load) from next iteration to be executed

If the address referenced is not in cache, we will fetch it into cache

Consider the case for a loop like

for (i=0; i<100; i++) {a[i] = b[i]+….;}

Now, look at the case when i=100. Before we complete the test i<100 using bne type instruction

We will assume that bne will succeed and execute next iteration → read b[100] speculatively

When bne for the last iteration is actually tested, we will discard all instruction from iteration i=100

but b[100] is already in cache. Most processors do not remove this data from cache.

SIDE-CHANNEL ATTACKS BASED ON SPECULATION

Speculative execution is the basis for Spectre and Meltdown attacks

An attacker attempts to read secure pages (such as OS pages)

- The system (or runtime checks) will detect such illegal access and abort attacker programs

However, the data is already "read" speculatively and stored in cache

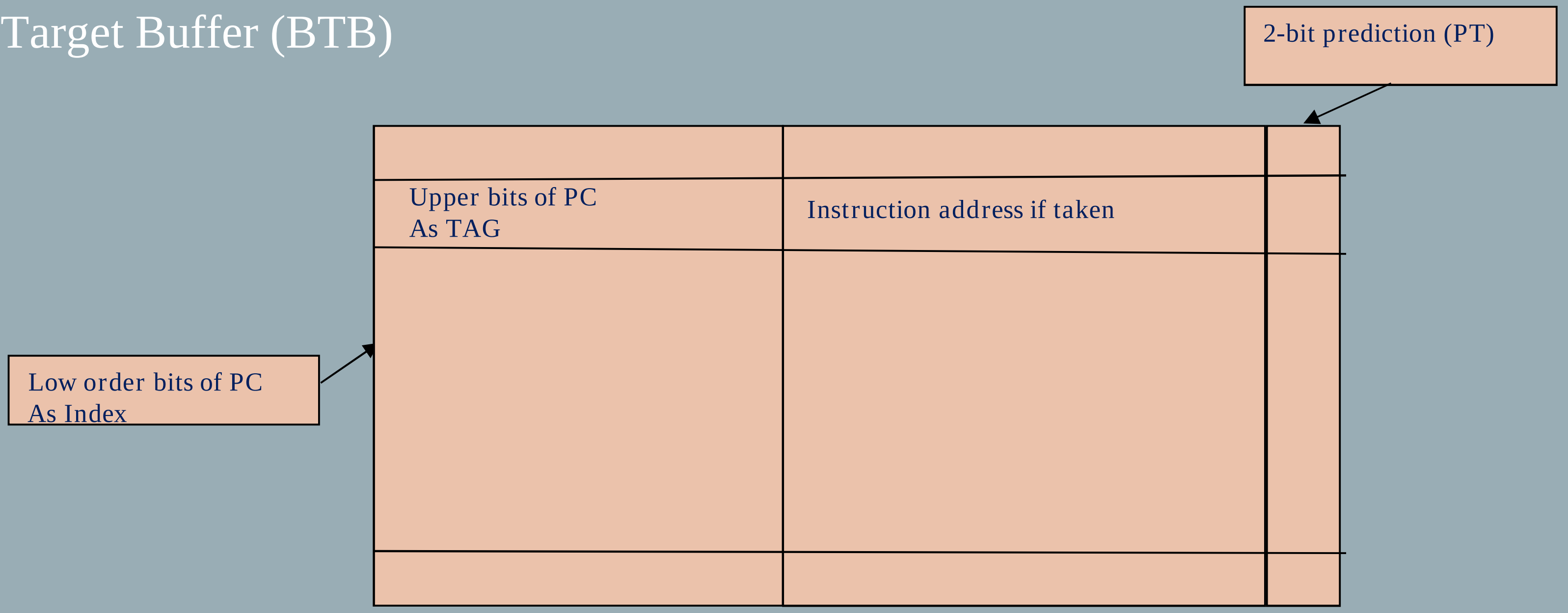
- Systems do not clean up caches for illegal access

Once in the cache, attacker can use cache side-channel attacks to obtain the data in secure pages

Before discussing solutions, let us understand how processors implement speculative execution

THE BASIS FOR SIDE-CHANNEL ATTACKS

Branch Target Buffer (BTB)



A few things to understand here.

The Branch Target Buffer is accessed during instruction fetch

If branch prediction is “taken”, the processor starts fetching instruction from the target address in BTB

Otherwise the PC incremented and next instruction in sequence is fetched

THE BASIS FOR SIDE-CHANNEL ATTACKS

What if we periodically reset the branch prediction table

Assume resetting means that all branches will be predicted as “not Taken”

This is likely to cause more branch mispredictions → causes execution delays

Changes execution times and order compared to when normal speculation is used

Also, cannot be sure if an illegal load will be executed, bringing data to cache

This causes performance loss. But we can control how much lost by changing the frequency of resetting PT

If we reset on every instruction: effectively turning-off speculation completely

If we do not reset, full speculation

How to decide on when to reset PT

a). Based on cache misses in shared cache (depends on what other applications are running)

b), Based on the number of instructions issued → depends on last time PT was reset

DEFEATING SPECULATION

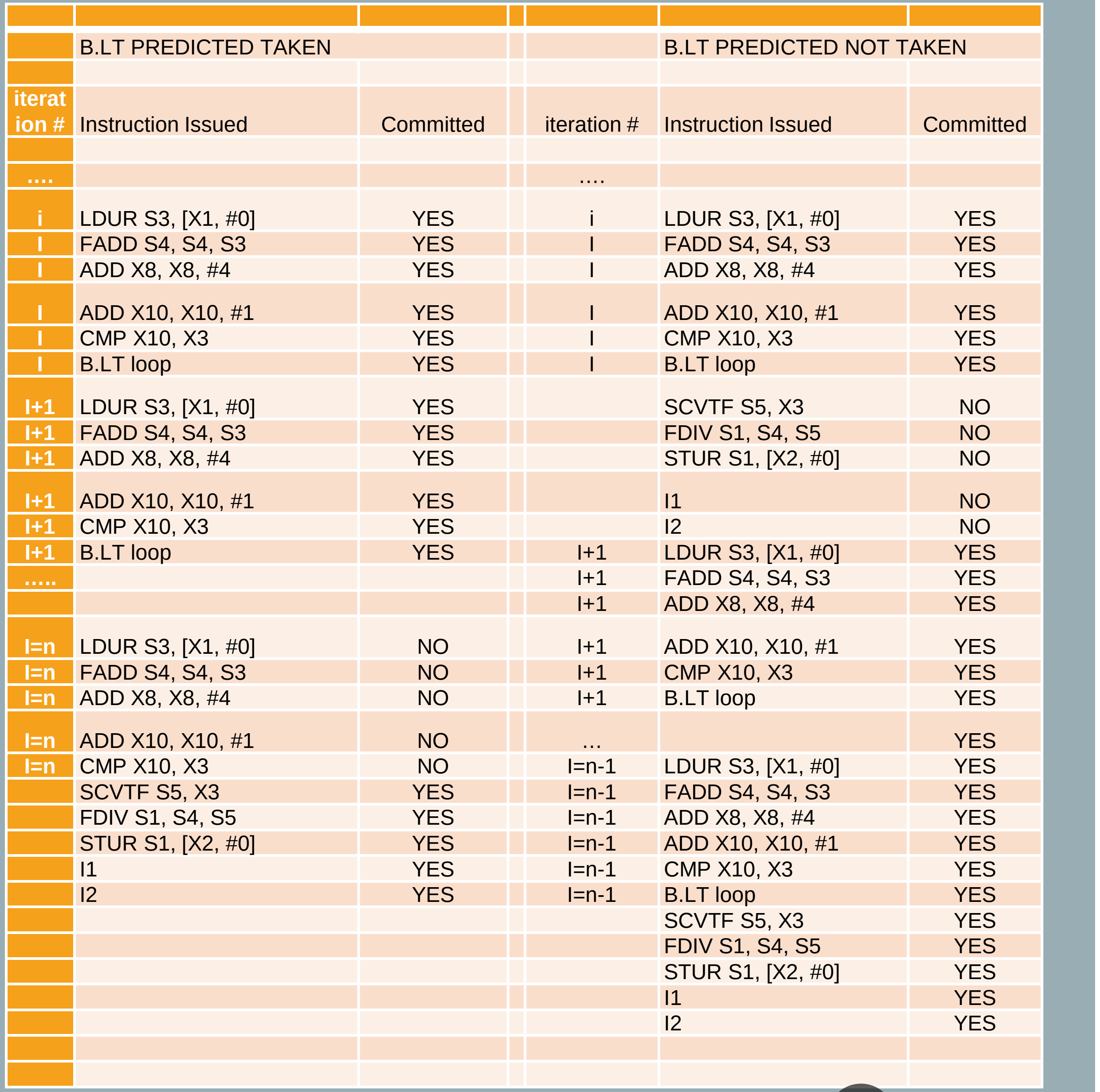
This table shows which instructions will be initiated depending the prediction of BLT loop instruction

Most dynamic predictors predict BLT will be taken (at most after 2 mispredictions)

We proposed to reset this prediction at different times during the execution

This will impact execution times

Also impact if a data item is speculatively accessed or not



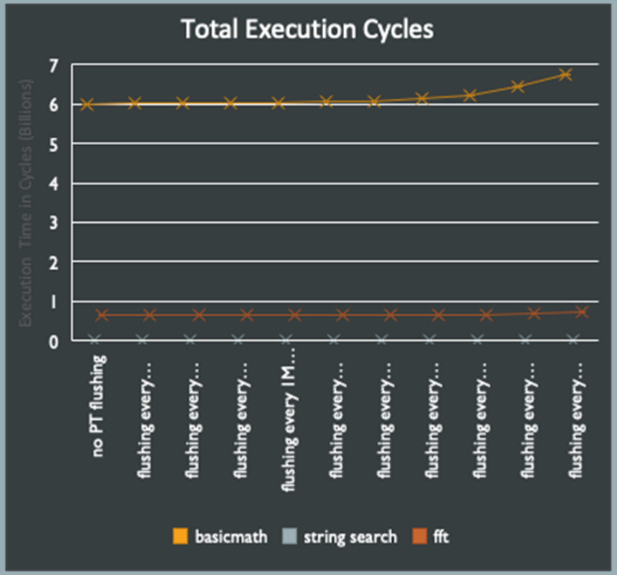
	B.LT PREDICTED TAKEN				B.LT PREDICTED NOT TAKEN	
iteration #	Instruction Issued	Committed		iteration #	Instruction Issued	Committed
….				….		
i	LDUR S3, [X1, #0]	YES		i	LDUR S3, [X1, #0]	YES
I	FADD S4, S4, S3	YES		I	FADD S4, S4, S3	YES
I	ADD X8, X8, #4	YES		I	ADD X8, X8, #4	YES
I	ADD X10, X10, #1	YES		I	ADD X10, X10, #1	YES
I	CMP X10, X3	YES		I	CMP X10, X3	YES
I	B.LT loop	YES		I	B.LT loop	YES
I+1	LDUR S3, [X1, #0]	YES			SCVTF S5, X3	NO
I+1	FADD S4, S4, S3	YES			FDIV S1, S4, S5	NO
I+1	ADD X8, X8, #4	YES			STUR S1, [X2, #0]	NO
I+1	ADD X10, X10, #1	YES			I1	NO
I+1	CMP X10, X3	YES			I2	NO
I+1	B.LT loop	YES		I+1	LDUR S3, [X1, #0]	YES
…..				I+1	FADD S4, S4, S3	YES
				I+1	ADD X8, X8, #4	YES
I=n	LDUR S3, [X1, #0]	NO		I+1	ADD X10, X10, #1	YES
I=n	FADD S4, S4, S3	NO		I+1	CMP X10, X3	YES
I=n	ADD X8, X8, #4	NO		I+1	B.LT loop	YES
I=n	ADD X10, X10, #1	NO		…		YES
I=n	CMP X10, X3	NO		I=n-1	LDUR S3, [X1, #0]	YES
	SCVTF S5, X3	YES		I=n-1	FADD S4, S4, S3	YES
	FDIV S1, S4, S5	YES		I=n-1	ADD X8, X8, #4	YES
	STUR S1, [X2, #0]	YES		I=n-1	ADD X10, X10, #1	YES
	I1	YES		I=n-1	CMP X10, X3	YES
	I2	YES		I=n-1	B.LT loop	YES
					SCVTF S5, X3	YES
					FDIV S1, S4, S5	YES
					STUR S1, [X2, #0]	YES
					I1	YES
					I2	YES

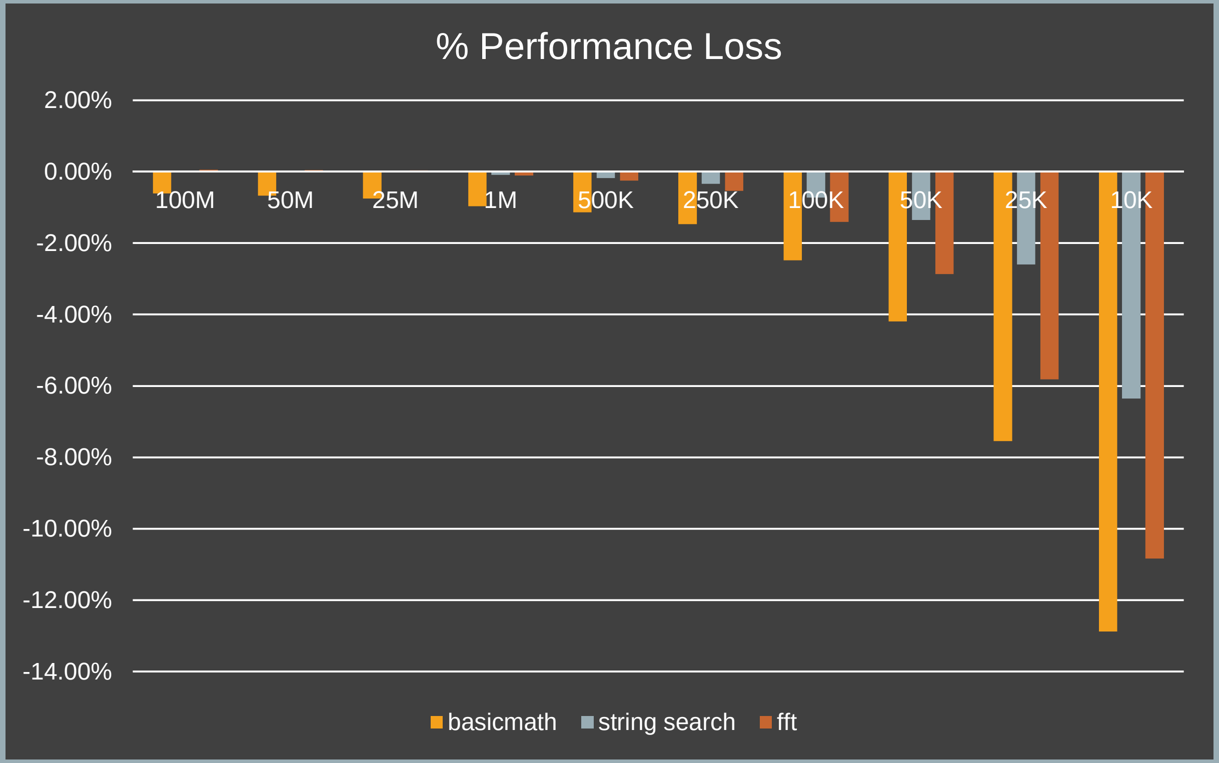
DEFEATING SPECULATION

Some initial results

Resetting PT on certain number of cycles

Single threaded system





More frequent resetting (e.g., every 10K instructions) leads to greater performance loss

DEFEATING SPECULATION

Need lot more analyses

1. Create more realistic architectural simulations

   Use gem5 → a full system simulator

   Can simulate multicore processors and more complex branch prediction techniques
2. Explore ”Random” times for resetting branch prediction

   Number of instructions issued → depends on the prediction

   Number of instructions completed does not depend on prediction

   Number of cache misses → depends on other tasks running at the same time

   → depends on branch prediction
3. Combine Cache Indexing techniques with Defeating speculation
4. Explore combining our techniques with other techniques
5. Most importantly, launch Spectre/Meltdown style attacks and analyze security benefits

SILENT SPECULATIVE LOADS

The main source of attacks is load instructions that are executed speculatively

If the load misses cache, data is brought into cache (and used by speculative instructions)

If the load is a hit in cache, meta data associated with the cache line is updated

LRU information, Coherency state

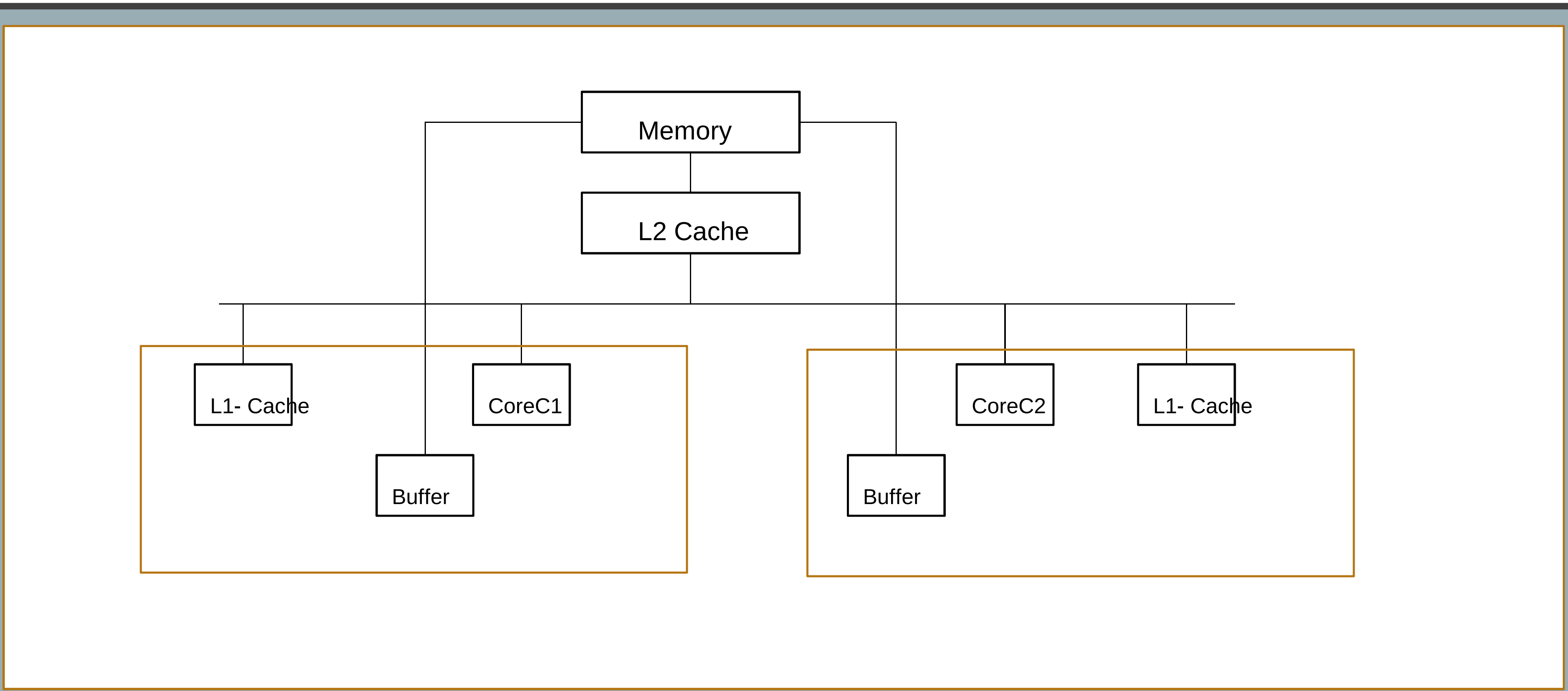
→ these facts reveal information to attacker

A secure architecture should bypass cache hierarchy or make the speculative load "silent"

Some recent proposals

InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy, M. Yan et. al (U of Illinois and Tel Aviv), IEEE MICRO-2018

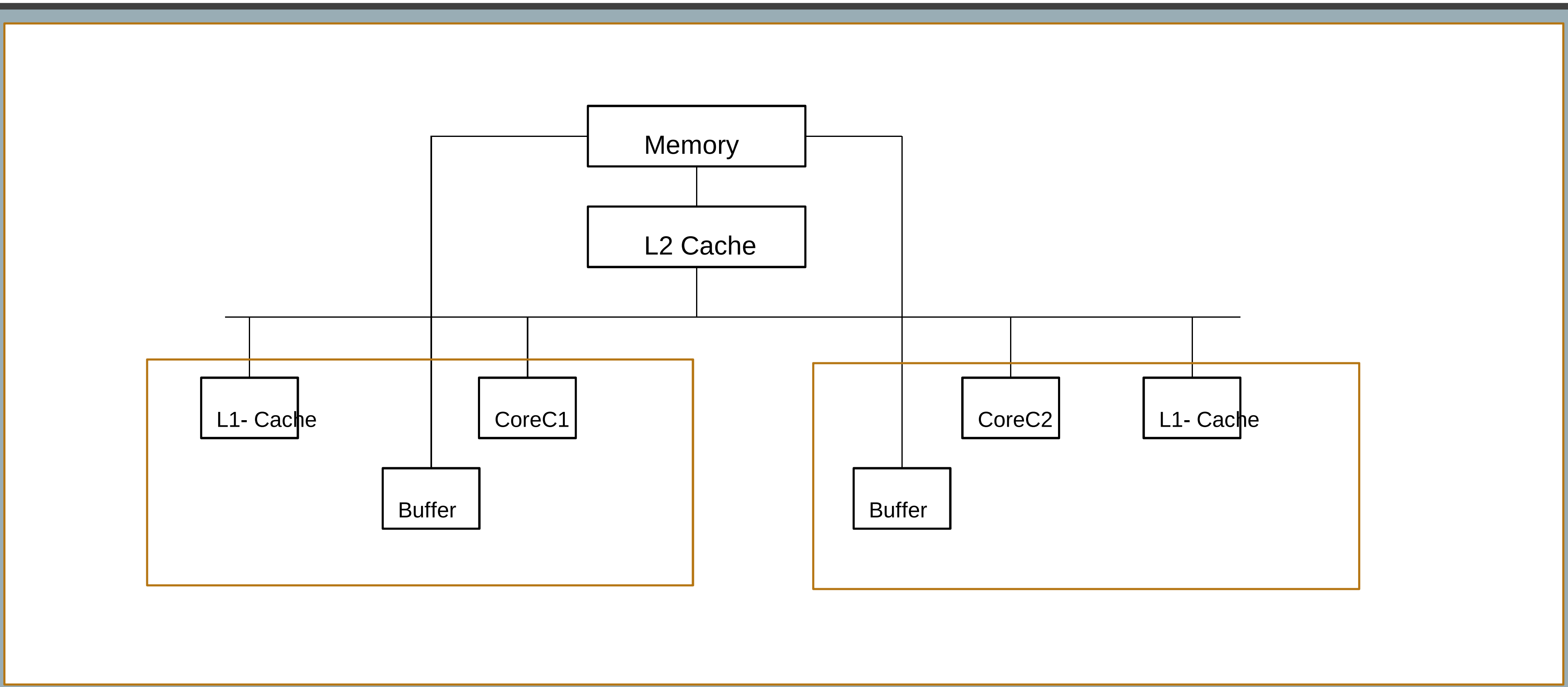
SILENT SPECULATIVE LOADS



Use buffers for speculative load instructions

- If misses in cache, do not bring data into cache, only to buffer
- If hits cache, bring a copy to buffer but do not update meta data

SILENT SPECULATIVE LOADS

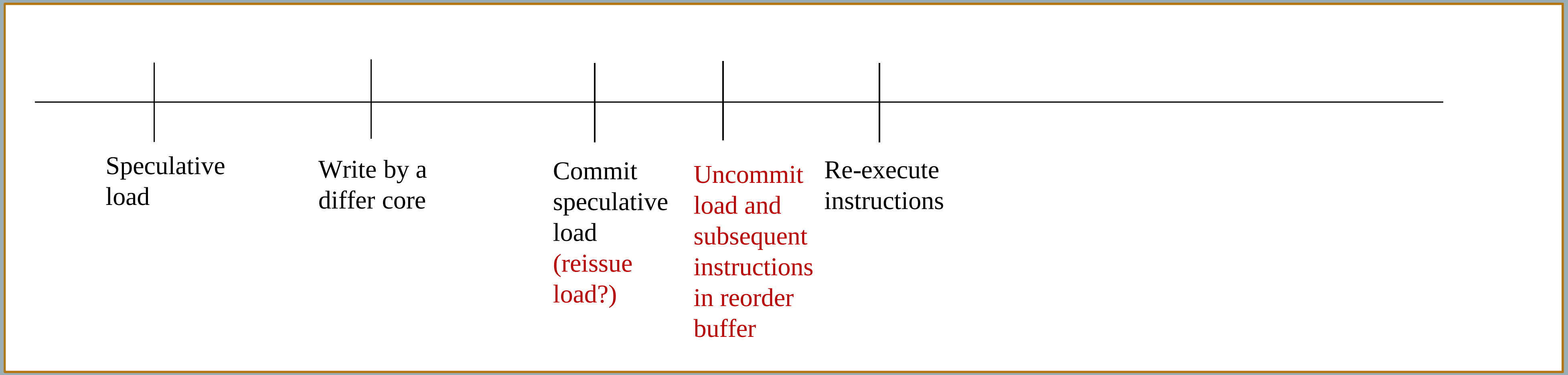


Handling memory ordering (and coherency)

- Speculative load is silent so no other cores are aware of the “read”
- When the speculative load instruction commits, the Core will issue a new non-speculative load
  - If other cores have ”modified” data, the load fails and need to undo all speculative instructions
    - that used speculative load data

SILENT SPECULATIVE LOADS

Note: The memory order is different from current implementations.



Speculative load

Write by a differ core

Commit speculative load (reissue load?)

Uncommit load and subsequent instructions in reorder buffer

Re-execute instructions

Conventional systems assume that the “speculative load” when commits, does not check for writes from other cores

The new implementation reissues a load at commit → the speculative load may have to be invalidated

Performance penalties: The locality of data is not present in buffer
May cause more cache misses since speculative access is hidden from cache
Coherency issues may require “uncommitting” a sequence of instructions
currently not done in architectures

SILENT SPECULATIVE LOADS →TRANSACTIONAL MODELS

We are exploring a variations to using Transactional Memory models → speculation is viewed as a new transaction

TM systems are based on Database Transactions and fulfill 3 of the ACID properties

- A: Atomicity: The entire transaction is treated as a single action, either completes or aborts
- C: Consistency: A transaction guarantees a consistent state (no partial changes)
- I: Isolation: Transactions are executed concurrently, one transaction does not impact others
- D: Durability: Once a transaction completes, and commits, cannot be rolled back

TM architecture are useful when we want concurrency but not pay overhead for locks

```
for (j=0; j<n; j++)
thread_id[j] = spawn_thread(compute_min, j);
for (j=0; j<n; j++)
        join (thread_id[j]);
```

```
void compute_min (j)
 {…local_min = min();
   lock(lock_variable);
          if ( local_min < global_min )
          global_min = local_min;
   unlock(lock_variable);
  exit();  }
```

SILENT SPECULATIVE LOADS →TRANSACTIONAL MODELS

What is needed to Implement Transactional Models in Hardware

1. Need to be able to rollback -- changes must be buffered
2. Need to recognize failures
   - Maintain read and write sets with each transaction
   - Cache coherency extensions
   - Version numbers
3. Need to differentiate between values that are speculated and regular values
   - additional instructions such as speculative-load/store
   - start transaction, end transaction
4. Determine the order of committing/write-back
   - program order
   - one thread at a time (no specified order)

We use branch prediction to start a new transaction/thread (say to execute next iteration)

→ commit the thread if branch prediction is correct, or squash the thread

→ hide all memory accesses of the speculative thread/transaction

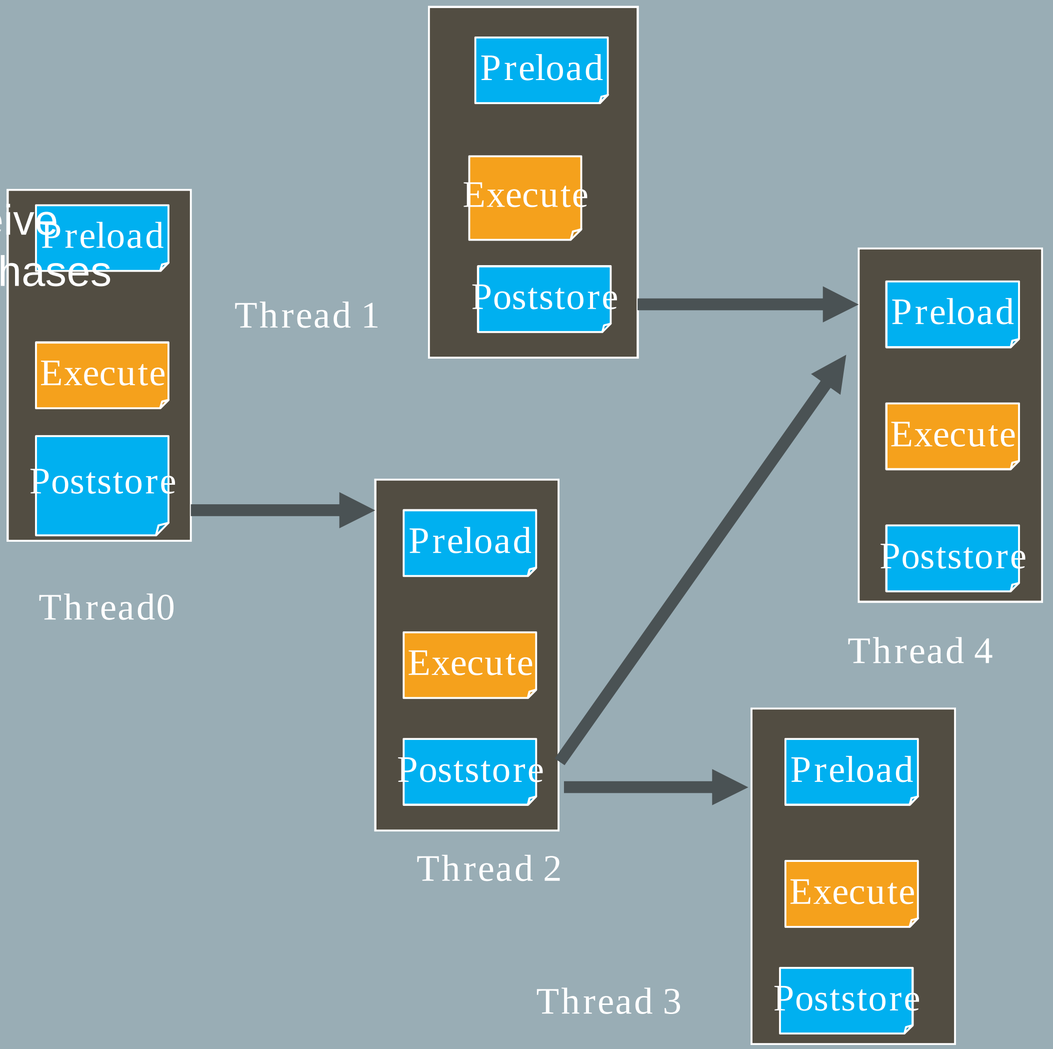
SILENT SPECULATIVE LOADS →TRANSACTIONAL MODELS

Our Own Scheduled Dataflow also permits TM

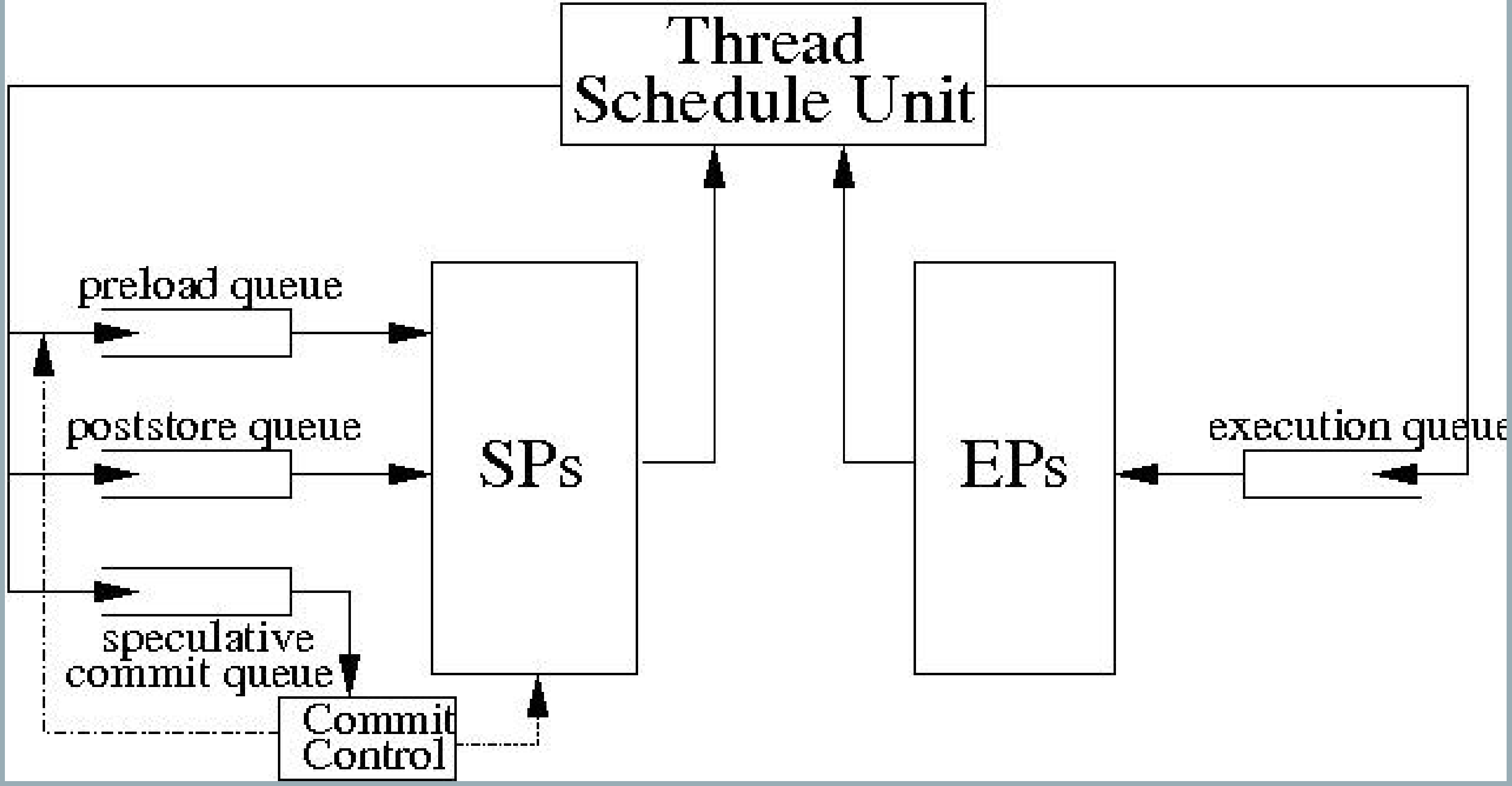
Threads follow Data Driven model: execute when all inputs are receive
Decoupled: each thread consists of Pre-load, Execute, Post-store phases
→ preload brings data into registers
→ execute only works with registers
→ results from registers are stored in memory by post-store

The results f a thread in Registers
→ if committed, post-store proceeds
→ if not, skip post store

We plan to create thread for speculated code
→ basic blocks, loop iterations



SILENT SPECULATIVE LOADS →TRANSACTIONAL MODELS



SILENT SPECULATIVE LOADS →TRANSACTIONAL MODELS

Need more study:

Formalize how a silent speculative load impacts micro-architectural state
Determine what state information can be observed through side channels
Silent loads should assure invisibility

Examples: Caching
LRU (and other meta data)
Coherency
TLB miss
Performance counters
……

Also, explore combined Performance – Security metrics for TM based models

QUESTIONS?

The goal of my talk is to explore collaborations and create groups of faculty and students to explore ideas and solutions.

If interested, contact me

Krishna.kavi@unt.edu
csrl.cse.unt.edu/kavi