# Deja $\mu$: Revisiting microarchitectural optimizations for security

Edwin Mascarenhas
*Computer Science and Engineering*
*UC San Diego*
emascare@ucsd.edu

Brandon Saldanha
*Electrical and Computer Engineering*
*UC San Diego*
bsaldanha@ucsd.edu

Arpan Dutta
*Electrical and Computer Engineering*
*UC San Diego*
adutta@ucsd.edu

Tanmay Patil
*Electrical and Computer Engineering*
*UC San Diego*
tpatil@ucsd.edu

Monil Shah
*Electrical and Computer Engineering*
*UC San Diego*
m3shah@ucsd.edu

***Abstract*—Due to power and area constraints, there is a tight constraint on the available hardware in computer systems. As such, to maximize performance by running multiple processes at the same time or having data from multiple processes in hardware, sharing memory between processes in mandatory. To guarantee security between processes, having different security checks and different privacy domains is a must. As shown time and time again, it is possible to breach this security via security attacks. In our paper, we discuss such architectural optimizations proposed in recent times that do not consider the security aspect while designing the optimization. We propose attacks that can exploit these optimizations and suggest defences that may be used to mitigate these attacks.**

***Index Terms*—cache replacement, prefetcher, microarchitecture, security, speculative vectorisation, Memory accesses, Cache replacement, Instruction prefetcher, Translation Lookaside buffer**

## Introduction

Memory isolation is at the core of today's operating systems. By ensuring that the user programs cannot access another program's memory or the kernel memory, isolation between processes is maintained. This enables hardware to run multiple processes at the same time on personal devices. This can be extended to running multiple processes from different users on the cloud on the same machine.

The Spectre [1] and Meltdown [2] microarchitecture attacks were a huge blow to the computer architecture industry, something that the industry is still trying to recover from. They caused designers to go back and inspect all released designs for possible security flaws that can be exploited. Micro architecture bugs are often a side effect of microarchitectural state modified by the victim process which are observed by an attacker. These observations are used to infer and leak secret information that ideally should remain resident only in the victim's protection domain.

For this study, we investigate three replacement policies: Ship++ [3], Hawkeye [4], Reinforcement Learning based replacement [5]; two instruction prefetchers: DJOLT, and FNLMMA, and 2 other miscellaneous microarchitecture optimizations: Speculative Vectorisation [6] and Early memory access elimination. For each optimization we present a summary of the feature, threat model, attack, mitigation strategy and other related work that could be impact due to the proposed attack.

The proposed attacks break all guarantees of security provided by the memory isolation implemented in computer systems. These attacks target processes on modern desktop machines and laptops as well as cloud servers. By crossing security boundaries, unpriviledged processes can read data from another process or read from priviledged locations/code directly. Priviledged locations include memory of another process, the kernel and in the case of kernel-sharing sandbox solutions (e.g., Docker, LXC) or Xen in paravirtualization mode, the memory of the kernel (or hypervisor), and other co-located instances [2].

## I. Phase I

### A. SHiP++: Enhancing Signature-Based Hit Predictor for Improved Cache Performance

*1) Threat Model:* 1. THREAT MODEL- The goal of the attacker is to infer secret data of the victim by observing metadata updates from a cache's replacement policy. The proposed attack allows to leak victim data with high accuracy, and thus we also present a defense that mitigates current attack and possible future attacks on similar lines. Our assumption is based on following capabilities of the adversary :

- Adversary has the privilege to leak information through cache side channels that rely on time precision measurements. And has the privilege to flush cache lines
- Co-location : We assume that adversary to be co-located on the same core as long as the last level cache is shared. As long as attacker can get some ways allocated in the cache line where the victim is allocated
- User-mode Access : The adversary should have standard access like system calls that allow to run timing precision calls

- Victim source code : The adversary is able to identify gadget in the victim domain.

Apart from using cache side channels, we do not assume that attacker is able to leak data through other channels. Any other channel usage is outside the scope of current security impact

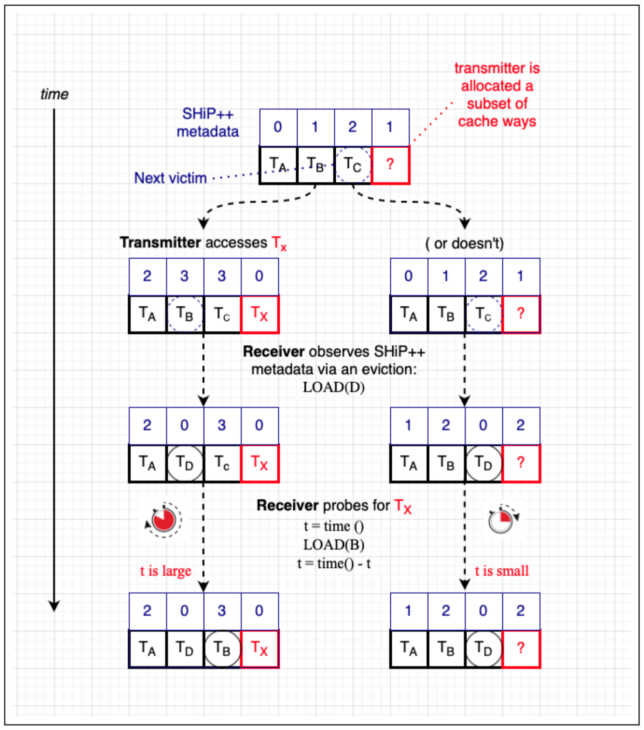

time
SHiP++ metadata
transmitter is allocated a subset of cache ways
Next victim
Transmitter accesses $T_X$
( or doesn't)
Receiver observes SHiP++ metadata via an eviction: LOAD(D)
Receiver probes for $T_X$
t = time ()
LOAD(B)
t = time() - t
t is large
t is small

Fig. 1. Ship Attack

2) *Exploitation:*

*a) Optimization:* This paper is based on an earlier replacement policy called ship. The idea of the paper is to associate reuse characteristics of a cache line based on the signature of a line which can be a characteristic of PC and other meta data. This is done by choosing what RRPV value to be used on a cache hit/miss and associating corresponding confidence counter for signature. The paper tries to improve on the existing SHIP policy by suggesting multiple enhancements. First enhancement is to insert lines with RRPV =0 for high confidence (saturating) reuse of lines. Second is to weigh cache hits and misses similarly(update on first reference) and not have overtraining. Third enhancement associates RRPV = 3 (low priority) to writeback lines, since they are more likely to be evicted from here as well. Fourth is to use different signature for demand and prefetch requests and thus reduce interference. Fifth is the update criteria of prefetch requests if they are followed by prefetch or demand (less priority). Based on these enhancements the paper is able to achieve a 6.2% improvement in IPC over LRU for single core configuration without prefetcher and 4.8% with prefetcher [3].

*b) CAT:* On top of the replacement policy, we assume a CAT style partition to configure each process with a class of service. This way it allocates logical ways to each process. With CAT, any process can hit in any of the way, however a process can cause eviction only from its assigned ways and not from other ways assigned to other process.

*c) Attack:* Ship works by iteratively increasing the RRPV of the ways of a cache line until It finds a cache line to be evicted. For a non-CAT based shared LLC, the adversary can simply prime the cache lines with its own data and wait for Victim to load its data, if one of the adversary's data gets evicted, the victim has likely accessed a line. However things are little different when CAT comes into picture. With CAT, the attack becomes a little complicated. So to make things simpler, the following toy example in Figure [add figure] can be setup with initial state as shown below. Assuming a 4way shared LLC with 1 way associated to Victim and 3 ways associated with adversary, and the cache metadata is reached to this state through priming the cache way. If the victim tries to access data that misses in the cache, then the corresponding metadata update happens for all ways, but the eviction happens only from victim's ways. The adversary then tries to load its own data after some interval under which it thinks that victim would have either accessed the data or not by then, it does a Load D, which replaces some data in Adversary's ways. Now finally it loads B again to see whether transmitter accessed some secret dependent data or not. And by monitoring the time it takes to see if it is a Hit or a Miss for B, the access of Victim can be inferred. Ship is a PC based policy, where sample sets are monitored to examine with what confidence should a line be placed in the Way. This can be overcome by attacker by mistraining the SHCT Table i.e. by loading data at different PCs and making all SHCT entries cache friendly. By doing a periodic access pattern of Hits and Misses, adversary can mistrain the SHCT table to install lines with high confidence. The attack becomes a little more complicated with multicore since SHCT now gets added with the core id making the mistraining a little difficult for multicore systems. There is one small need of reverse engineering here. Ship chooses sample sets based on a random allotment. This might be a little difficult to reverse engineer, but if it can be done the mistraining can be reduced to a whole new level that covers only Sampler Sets.

3) *Mitigation:* Our mitigation for this attack is similar to DAWG [7] with certain additions. The very straight forward defense for all such cache based attacks is to make sure that both the ways and metadata updates are isolated for different processes. The configuration of policy_fillmap and policy_hitmap is software configurable which can be looked up using domain_id that can map to a process. Each memory access must be tagged by this domain_id. The original idea of CAT being able to hit in other domain metadata now is restricted by the programming of policy_hitmap. If a way is not allocated to a domain, it cannot hit in that way. Same for the fill_map, the cache cannot cause fill/eviction in metadata of other domain. This also limits the usage of clflush instruction. The DAWG paper also adds a metadata isolation block. The metadata uses fillmap and hitmap to update metadata of the cache line thereby making it invisible to infer what hits/misses are observed by victim. One of the additional structures that

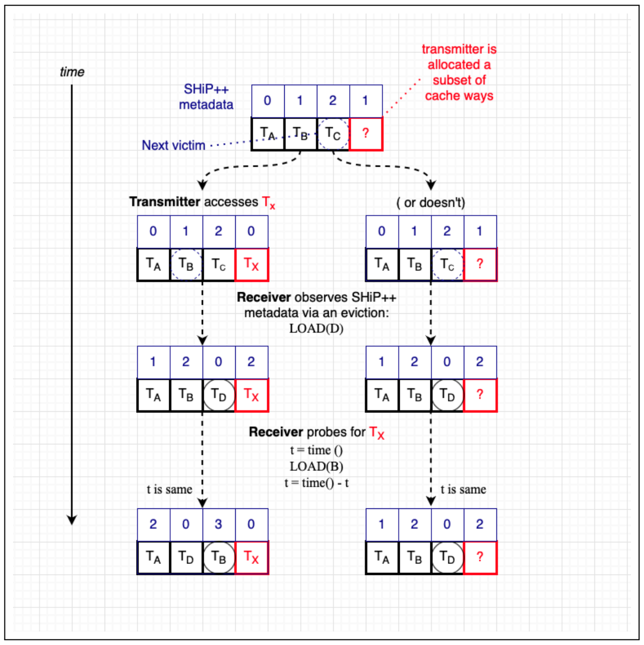


Fig. 2. Ship Defense

needs to be handled is the SHCT table. Since SHCT table is a common structure, this too can cause different side channel to leak data, by observing other data addresses the SHCT based PC touches into, and due to aliasing (because of lower bits usage) the effect can be manifold. So make sure that SHCT table isn't compromised, we should add an additional domain_id Identifier in the SHCT table that can only change when a request corresponding to particular domain is received. The same attack when tried with our defense does not leak data and can be seen in Figure [ship_defense] There are two main overheads for this defense. One is SW overhead in maintaining atomic operations that configure the policy_fillmap and policy_hitmap. The performance overhead of DAWG for this design will be 20% over baseline Ship++. Regarding the timing, there won't be much difference on the critical path as the cache lookups are pipelined in nature and can take an additional gate for checking. Regarding Hardware Overhead, there will be at least domain_id number of bits multiplied by 16K entries overhead for the SHCT structure partitioning.

*4) Related work:* The proposed attack is not just limited to the above replacement policy, and can be generalized to policies that are based on RRPV using a signature. So this attack might be possible for even Hawkeye [4] and Lime [8] which are based on PC and signature.

### *B. Hawkeye: Leveraging Belady's Algorithm for Improved Cache Replacement*

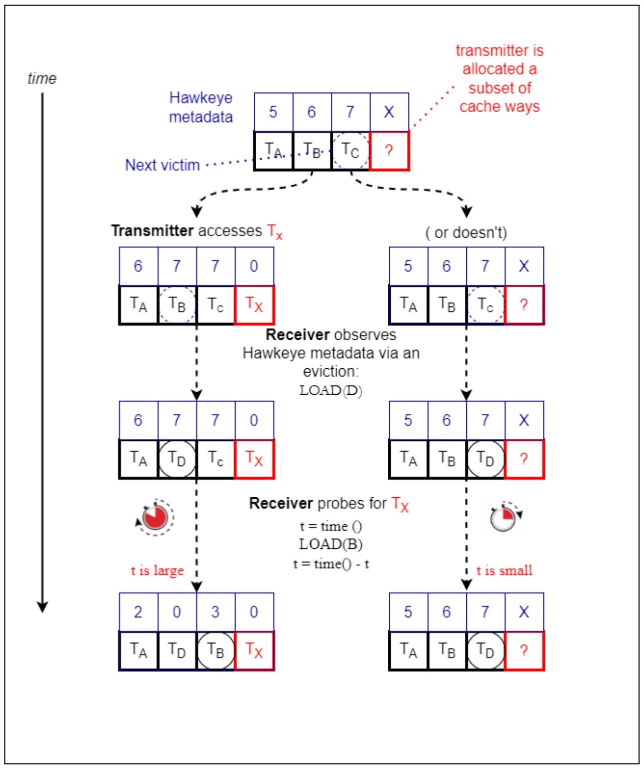
Belady's algorithm for replacement [9] is the optimal replacement policy often used to compare existing replacement policies against. The optimal replacement policy cannot be implemented because it needs knowledge of the future accesses. Hawkeye [4] reconstructs Belady's optimal solution on the past history of accesses using the OptGen algorithm. It then uses a PC-based predictor to learn from OptGen whether the load instructions are cache friendly or cache averse and inserts it with respective priorities.

For replacement Hawkeye uses the RRIP scheme where each entry has a saturating counter associated with it that increments when the sets are accessed. It has a OPT gen vector which is used to keep track of the cache sets and the usability of the cache line that are incoming. It improves over baseline hawkeye by having separate predictors for prefetched and demand accesses. It avoids loading lines that are the same and back to back prefetch and demand access, reducing redundancy. The predictors essentially use the occupancy vector to figure if the line being brought in is cache-friendly or will pollute the cache.

*1) Threat model:* The threat model for Hawkeye assumes a shared LLC. The shared LLC is used as a covert vhannel for the attack to leak data. The attack on these structures don't necessarily need the prefetches to be turned off. This is because there is a separate predictor which tracks the metadata for prefetches. Hence, only the prefetch predictor is updated on a prefetch access and demand accesses are tracked separately by the demand predictor. The algorithm ensure that there are no redundant cache reloads. In other words, if line is stored already by a demand or prefetch access it won't be prefetched again. This mitigates a lot of the noise that could have been introduced due to normal prefetching that would bring in all prefetch data. There still can be noise inserted into the attack results due there being common structures like the sampler cache. So, for most accurate results we wan to ensure that prefetches are disabled. Intel CAT [10] doesn't help this replacement structure. All the Ways are still able to see and change the RRPV counter values of all the tables. This allows attacker processes to be able to change metadata of sets where victim process's data maybe cached. Further, PC attacks are also possible since the SHCT is accessed by PC. It is possible to mistrain the SHCT and cause evictions based on that. The victim accessed must be another user process. Shared address space is not necessary for this attack. However, for PC based attack the shared address space is necessary.

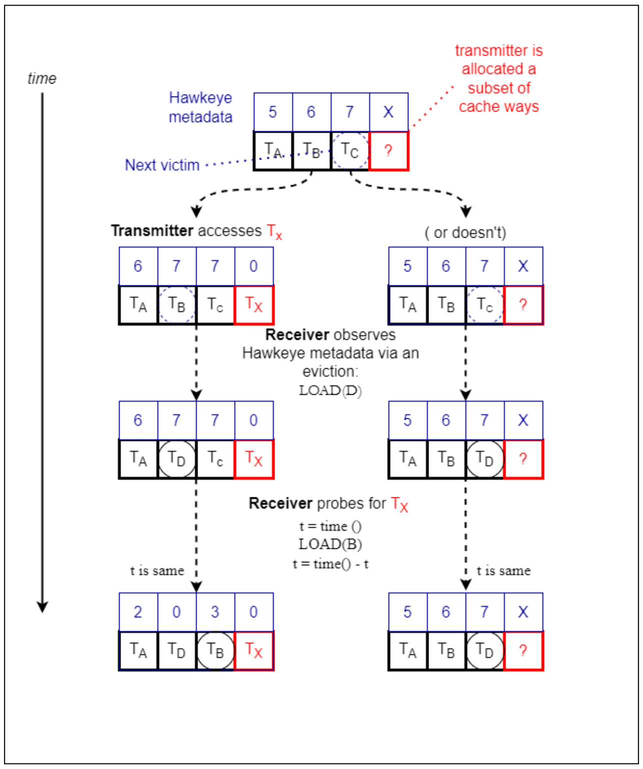
*2) Exploitation:* The attack primarily involves an attacker gadget that is used to update metadata in a way that causes the RRPV values to leak secret data through FLUSH+RELOAD. The following diagram gives an overview of the attack:

For the attack to work we need to mistrain the the replacement policy to create the initial scenario shown in fig.3. In this particular set we have 3 ways allocated to the attacker and the remaining one is allocated to the victim. Intel CAT is enabled here. When the RRPV values of the 3 sets of the attacker's protected domain are 5, 6 and 7 respectively, the attacker can access it's own way and determine if the victim has accessed something in the set or not. It doesn't matter what the RRPV value of the victim's way is. The attacker performs cache accesses to ensure the counters are mistrained to above values. When the victim access $T_X$, is loaded in way 3, its RRPV value is set to 0 since it is recently accessed.



time
transmitter is allocated a subset of cache ways
Hawkeye metadata
Next victim
Transmitter accesses $T_X$
( or doesn't)
Receiver observes Hawkeye metadata via an eviction: LOAD(D)
Receiver probes for $T_X$
t = time ()
LOAD(B)
t = time() - t
t is large
t is small

Fig. 3. Hawkeye Attack



time
transmitter is allocated a subset of cache ways
Hawkeye metadata
Next victim
Transmitter accesses $T_X$
( or doesn't)
Receiver observes Hawkeye metadata via an eviction: LOAD(D)
Receiver probes for $T_X$
t = time ()
LOAD(B)
t = time() - t
t is same
t is same

Fig. 4. Hawkeye Defense

Other RRPVs are incremented by one according to the policy. If $T_X$ is not accessed by the victim, nothing changes. At this point the attacker must probe for a value D that maps to the same set. This will cause an eviction in either way 1 or way 2 based on the highest RRPV value from the left. Therefore, either $T_B$ or $T_C$ will be evicted. By issuing another load that loads $T_B$ we can determine if the victim accessed $T_X$ or not. This will leak the secret value that causes $T_X$ to be loaded or not based on its value.

*3) Mitigation:* For defense we need to ensure that the attacker is unable to read the cache state or metadata. The attacker should not be able to infer the secret from the metadata. The right way to ensure this is to use DAWG style partitioning based on protection domains. Essentially, we need to ensure that the attacker does not have a covert channel. To do that we use DAWG style cache partitioning. The attacker will not have access to the victim ways since attacker can't hit or replace anything in the victim ways. The RRPV is completely separated now.

As seen in fig.4 we can see that the attacker is no longer able to modify victim metadata. This prevents it from attacking directly. However, The PC based attack is still viable since the SHCT can be mistrained by the attacker which is shared by the victim. To avoid this, we suggest partitioning the SHCT as well. There are 2 ways to do this:

- Divide SHCT into multiple ways. Preferably it is better to have same number of ways as LLC. Then use protection domains of LLC to provide access control to the SHCT entries. This way the LLC and the SHCT are in sync but also the attacker is separated from the victim. This is the DAWG style partitioning that can help with the above mentioned attack but not effective against PC based attacks.
- Provide separate SHCTs for separate cores. This involves dividing the SHCt into 4 parts and hard coding them to specific cores. Just like the sampled sets of hawkeye, we would tag the SHCT with core id to make sure that only one core is able to access it. This is needed in addition to the above defense to ensure that PC based attacks are not feasible. Since only one core is accessing it, and cache covert channel is blocked, both PC based and Cache based attacks are blocked.

The main overhead of these defences if that the tables are now partitioned and hence less flexible. This would take a small performance hit. The bigger overhead is the maintenance of policy_fillmap and policy_hitmap like that in DAWG. It would require OS changes and extra hardware registers for implementing this. But it is not as bad as increasing issue width or instruction queue. Hence, the power and area increase should be nominal.

*4) Related works:* The closely related works which are also vulnerable to this attack are policies that use PC based signature tables and use RRPV counters for eviction. One such policy is SHIP++ [3] which uses signature history indexed by PC and RRPV as basic eviction mechanism. This attack will also break that replacement policy. Lime [8] is also vulnerable to this attack.

## II. Phase II

### A. D-JOLT: Distant Jolt Prefetcher

The D-JOLT prefetcher [11] is a hybrid prefetcher consisting of a long distance prefetcher, short distance prefetcher and a fall back stream prefetcher. Prefetch distance is the number of code blocks from where a prefetch is issued to the prefetch target block. Prefetches issued with larger distance have the advantage of issuing early prefetches but can have lower prediction accuracy. To make up for this D-JOLT uses a hybrid approach. D-JOLT is a branch predictor directed prefetcher and keeps track of the target PC as determined by the predictor. Each of the long range and short range predictors compute a hash of the past respective *longrange histlen* and *shortrange histlen* number of program counters and the number of successive returns. This computed signature is used to index into their respective miss tables and issue prefetch requests for the respective code block(s). D-JOLT achieves a 28.9% increase in IPC over a baseline with no instruction prefetching.

*1) Threat model:* For this attack to work we need shared instruction prefetcher between attacker and victim. The main idea is to trigger a victim's secret dependant prefetch that can be observed by the attacker. The first problem is that can the attacker miss-train the prefetcher to trigger a prefetch in the victims code. This can only happen if the prefetcher is shared among the victim and the attacker. This can be ensured by making sure that the victim and the attacker runs on the same physical core, either as hyper threads or as processes scheduled on the same physical core.

- The attacker and the victim are separate processes that has a shared D-JOLT prefetcher.
- The attacker is able to observe the prefetches made by the victim.
- The attacker can expose the state of the internal variables of the victim.
- The attacker knows the binary of the victim before hand.

*2) Exploitation:* The main idea of the attack is to trigger a secret dependant prefetch in the victim code such that the attacker is able to observe the prefetch and determine the value of the secret. For this to work the attacker must first miss-train the prefetcher to fetch something that the attacker is able to observe in his own virtual address space. We know that the shared libraries (like dll in windows) have the same virtual address in every process.

Listing 1. Victim gadget

```
call func_v0
call func_v1
call func_v2
call func_v3
if (secret) {
    call func_vf
}
```

In D-JOlt the prefetchs are triggered on function calls only because these are on only parts of the code that do not have spacial locality. So let's assume that the victim code makes a procedure call based of the secret as shown in the code above. The attacker wants the "call func_vf" instruction to trigger a prefetch which can be observed. Because the attacker know the binary of the victim before hand, they can mistrain the prefetcher.

Listing 2. Attacker gadget

```
call func_a_miss_train_0
call func_a_miss_train_1
call func_a_miss_train_2
call func_a_miss_train_3
call func_af // miss-training prefetches
             //  for this instruction
call func_a_pad_0
call func_a_pad_1
call func_a_pad_2
call func_a_pad_3
call func_shared_dll // target of the
             //prefetch -> shared library
```

The first thing the attacker needs to do is to make sure that the signature generated by victims target call ("call func_vf") and attacks miss-training call is the same. This can be made sure by reverse engineering the signature generation scheme (it is actually already available in the DJolt paper) and placing the attackers procedures "func_a_miss_train_x" in the required locations. D-JOlt has three prefetchers, and we are only targeting the short ranged prefetcher that has a history length of 4 and a distance of 4. Which mean it will use the return address stack of last 4 procedure calls to generate a signature and use the signature to prefetch a procedure that is 4 (distance) calls away from the current procedure call.

So now the signature generated by victim (call func_vf) and attacker ("call func_af") is the same. So the prefetch done by these function calls will be the same. Now the next thing to do is make sure the target of the prefech is some thing the the attacker can reason about. So we choose a shared dll library to be the target. For the short prefetcher the distance is 4, so after 4 procedure calls we add a call to a shared library (dll). We run this attacker code multiple times to train the prefetcher to have high confidence in this preftech (of shared dll).

Now that the setup from the attacker is complete, the attacker will flush the cache to make sure that the shared library (dll) is not in the cache. Now the victim code runs and depending on the secret the shared library will either be prefetched or won't be prefetched.

Listing 3. Attacker measurement

```
cpuid();
t1 = rdtsc();
cpuid();
call func_shared_dll
cpuid();
t2 = rdtsc();
cpuid();
time = t2 - t1;
```

Now the attacker can call the shared dll and measure the execution time. We expect the time difference to is the order of thousands of clock cycles (L1 latency versus RAM latency). If the execution time is lower then the victim must have triggered a prefetch on this shared dll. This reveals the victims secret.

We need to consider other complications in this attack that we have not discussed yet. We want to make sure that the prefetch to the shared library is not triggered by any other function call. It may be triggered by not just short prefetcher but by the long prefetcher or the next line prefetcher. Although this is highly unlikely (in the order of 1 in 2048), but not impossible. To make sure that the attack has as low noise as possible we may want to calculate the signature for all the calls and make it so that the signature of the target and any other call is not the same.

This attack would also work in presence of CAT style partitioning of the cache. This is because the cache hits can be across domains in CAT scheme.

*3) Mitigation:* The security problem arises because of the shared data structures of the prefetcher. For complete isolation among processes we would require that no micro-architectural data-structures are directly shared with malicious processes. For this attack mitigation we would want that the prefetcher data structure is isolated among processes.

- Dynamic signature generation. Use different hash functions (determined at run time) for generation of signatures. This makes it hard to miss-train the prefetcher, but is not completely safe if someone is able to determine/-manipulate the hash function. Almost no performance overhead will be observed because we are only changing the hash function to be dynamic.
- Use CAT style partitioning in the miss table and have separate signature queue, so that the attacker is not able to observe any meta data of the victim. This mitigation will effectively reduce the number of targets a thread is able to remember. We would also need to flush the prefetcher on context switches. This would reduce the performance improvements even further. So the net improvement expected after this mitigation would 5%.
- Use CAT style partitioning in cache along with DAWG style meta data masking to stop the attacker from being able to measure the prefetcher side-effects. This would reduce the baseline performance but would futher increase the prefeomance gains from DJolt prefetcher as the effective cache size available to each thread would be even lower. We expect the performance gains to improve to around 40% over the new baseline and around 25% over the old baseline.

Note the above mentioned mitigation only secures process to process leakage of data. To secure leakage of data among protection domains like user mode and kernel mode, the prefetcher data structure either should not be updated (in kernel mode) or should be even more fragmented (partitioned) to avoid data leakage.

*4) Related work:* The attack that we describe here is not limited to only this particular prefetcher, Any such prefetchers that use a signatue based miss lookahead can be attacked similarly like RDIP [12]

## *B. FNL+MMA Prefetcher*

It is essential to decide which events trigger a prefetch to perform well. The FNL+MMA prefetcher proposes to initiate prefetch requests on I-Shadow cache misses. The I-Shadow cache is s small cache that monitors misses from demands to the cache. Spatial locality tells us that the following line is likely to be used shortly, but fetching every next line leads to over fetching and high bandwidth consumption. It is imperative to determine the likelihood that the following line will be needed soon. The FNL overcomes this difficulty by selecting if the following line will be required and achieves a 16.5% speedup while doing so. On the other hand, the sequence of I-cache misses is partially predictable if no prefetching is used. It can be said confidently that when a particular block B misses, the nth block required after that block is often the same block Bn. This can be said confidently for n as large as 30. The MMA predictor takes advantage of this property, and with a 96KB FNL+MMA block, the machine achieves a 28.7% speed up and decreases the I-cache miss rate by 91.8%. The one downside to this implementation being the 38.3% increase in L2 access.

On an I-Shadow cache miss, FNL prefetches contiguous blocks, but it cannot fetch non-contiguous blocks. The MMA prefetcher targets this limitation of the FNL prefetcher.

Without prefetching, it is possible to predict with high accuracy the next block that should be prefetched when a block miss occurs. This property holds even for the I-Shadow cache that monitors demand accesses on the I-cache. Using this property, when blocks B and B' are missing consecutively in the I-Shadow cache and B' is missing in the I-Cache B' is associated with Block B in a 'next prediction' table. When the same association occurs twice, the entry is considered highly correlated, and on the next occurrence of Block B, B' is prefetched into I-cache.

The only downside to this method is that prefetches are triggered too late. To avoid these late prefetches, the predictor predicts the block that should be prefetched 'n' blocks after a miss on block B.

*1) Threat model:* The attack assumes a common prefetcher between the attacker and victim. Our main agenda is for a secret dependant instruction to trigger a prefetch to a block which resides in shared memory, thus enabling us (the attacker) to observe these prefetches which is turn enable us to decode the value of the secret. The problems are similar to the above D_JOLT prefetcher where the first problem is that can the attacker miss-train the prefetcher to trigger a prefetch in the victims code. This can only happen if the prefetcher is shared among the victim and the attacker. This can be ensured by making sure that the victim and the attacker runs on the same physical core, either as hyper threads or as processes scheduled on the same physical core.

- The attacker and the victim are separate processes that has a shared FNL+MMA prefetcher.
- The attacker is able to observe the prefetches made by the victim in the shared memory by priming the last level cache.
- The attacker can expose the state of the internal variables of the victim.
- The attacker knows the source code of vicitm before hand to identify how to use the hash function to place a shared memory miss.

*2) Exploitation:* The idea of this attack is based on the usage of shared memory call, which when prefetched by a victim would allow the adversary to infer what block has been executed by the victim. The FNL prefetcher is very noisy since it can only prefetch at a distance of 5, whereas MMA prefetcher can prefetch at a distance of 9 blocks, which makes it suitable for us to base our attack. The main structure of the MMA prefetcher is a miss table which associates a miss to some prefetch address block. Two important things required to know about the miss table is that it uses a hashing function to index into the miss table and a partial tag to associate the entry. Reverse engineering the functions is not required since the code of the prefetchers is already available to us which shows the hashing function. Tag bits are simply some upper bits of the address which are partially used in the Block address. We assumed that adversary knows what all addresses are accessed by a victim before a secret dependent call. One of the best things about I-Shadow cache, which associates misses with addresses is that it is similar in structure to I-Cache which makes us not to require any reverse engineering of how entries are populated. Anything that the I-Cache would be populated wit on a miss is what I-Shadow cache should be populated with after a prefetch. Now with this in mind, if our adversary is able to create a sequence of instructions that miss cache and then the final function call at a distance of MMA prefetcher distance is that of a shared library, our prefetcher will learn with high confidence to associate the first miss in the sequence of our attack that eventually caused the shared library to be called as seen in listing 5 of adversary. After mistraining, the adversary will evict the shared library from the LLC. Now when victim is running similar sequence of instructions with same number of misses in the I-shadow cache with the first miss being same as adversary's firs miss, it will fetch the shared library call into the LLC cache which our attacker can probe to find if the victim accessed any secret dependent code as shown in listing 4 of victim . We don't care about other misses that are occurring in the victim code but only about how many misses it generates. Now in order for FNL prefetcher to not create noise and prefetch the same library call, our adversary makes sure to associate the FNL miss that would lead to a shared library call to be different from a miss that would trigger the shared library call from victim, thereby making sure that victim never calls the shared library on any other sequence of misses.

Listing 4. Victim gadget

```

Miss A1 // Miss associated with MMA
Miss A2
Miss A3
...
Miss A5 // Miss not associated with FNL
Miss A6
Miss A7
Miss A8
if (secret) { // 9\superscript{th} Miss
    ...
}
```

Listing 5. Adversary gadget

```

Miss A1 // Miss associated with MMA
Miss A2
Miss A3
...
Miss A51 // Miss associated with FNL
//but not with victim's Miss
Miss A6
Miss A7
Miss A8
Miss A9 { // 9th Miss
    shared_dll_call
}
```

Listing 6. Adversary measurement

```
cpuid();
t1 = rdtsc();
cpuid();
call shared_dll
cpuid();
t2 = rdtsc();
cpuid();
time = t2 - t1;
```

*3) Mitigation:* One of the defenses to prevent this attack is to simply disable Hyperthreading. In some server applications hyperthreading is disabled by default since it does not lead to performance gains due to contention of resources. However, in a personal use scenario, the hyperhreads are useful as they will boost multiple single-thread performance. Hence, we need to come up with a more sophisticated defense like DAWG. The 192 entry I-Shadow cache, 64K entry Touched and WorthPF tables, a 8K-entry miss ahead prediction tables are all shared in a core. These will all need to be partitioned into ways and then allocated based on protection domains. Another option is to store domain ID along with the data and allow update to the data only of the domain ID matches that of the allotted protection domain. The overhead of this would be mainly area. We can still perform these lookups in one cycle but would now require us to have multiple comparators to confirm the domain ID and then issue the load to cache way. But increasing area are also mean increasing power. Just like implementing DAWG, the additional comparison for checking domain ID will also incur only a small overhead. This is because we are adding additional domain ID comparison to the cache TAG comparison. The cache TAG value will be around 44 bits for 128MB LLC. The comparison bits will only be at most equal to the number of ways of the LLC. If we assume 16 ways, it's only 4 bits which an order of magnitude less than 44. Hence, this is a cost effective mitigation strategy. However, since the 8k-entry miss ahead table is not available entirely to the victim process, there will a performance hit. Specifically the table will be divided into LLC way number of partitions. This will cause the victim to only access a part of the table for referencing the miss history. The number of data points available will reduce and the performance hit will be almost equal to the number of partitions. The performance gain is still there however it will be approximately 15x gain. It will not be exactly 16x since there are still some common structures that can be accessed but not used for the attack.

*4) Related work:* The MMA predictor works on the principle of fetching blocks whenever a miss on another block has occurred. When the same blocks are found to be missing in succession with a gap of a particular number of cycles, the latter block is prefetched. All of the prefetchers that use this method of prefetching are vulnerable to the attack described above. An example of this is the Temporal Ancestory prefetcher [13] and MAMA [14] which keeps a track of the prefetches caused due to a previous missing block. When the prefetcher is mistrianed to prefetch a particular block using this concept of distance based prefetch, the secret value can then be leaked.

## C. Speculative Vectorisation with Selective Replay

Vectorisation is a technique to perform an operation on multiple components of a vector, often referred to the SIMD programming model. Today's modern day ISAs like x86, ARM and RISC-V support vector instructions and processors have vector registers and execution units to speed up variety of workloads. Vectorisation compiler passes can convert some loops into vector instructions. However compilers fail to vectorise code in the presence of unknown or infrequent data dependencies thus leaving a lot of performance on the table. Selective Replay Vectorisation (SRV) [6] proposes to optimistically vectorise code at compile time even if the dependencies cannot be determined. At run time, if memory dependency violations are detected in some of the vector lanes it selectively replays those lanes.

When the compiler cannot determine the memory dependencies in a loop but chooses to speculatively vectorise the code it inserts the instructions between a *srv_start* and *srv_end* instructions. This region is called the SRV-region. A predicate register called *SRV-replay register* is used to indicate which vector lanes should be executed. On the first execution all the bits in the register are set. Dependency checks are performed in the Load Store Unit (LSU). If any violations are detected, another predicated register SRV-needs-replay register is set depending on which lanes had the violation and need to be replayed. At the end, if any of the bits in SRV-needs-replay are set, then the value is copied to SRV-replay and the execution repeats. This continues till all the lanes have been executed after which it exits the SRV region.

*1) Threat model:* For this attack we consider personal computers and virtual machines in the cloud that supports Vector instructions. We also assume that the Speculative Vectorisation optimization is enabled. The attacker also has access to a compiler that can generate speculatively vectorised code. We assume that the attacker has unpreviliged access to run programs on this machine. We assume the attacker can perform prime + probe or flush + reload kind of attacks to infer things about the cache microarchitectural state. The Operating system being run on as well as compiler is bug free, i.e. the attacker isn't exploiting a software vulnerability to leak the information. Lastly this attack would work even if the code region has defenses against spectre like inserting an *lfence* instruction to serialize the order of instructions as the exploit detailed in the next section can be applied irrespective of this.

*2) Exploitation:* To construct the exploit we use an important feature of the attack. In SRV, if an exception occurs during execution, the lane number where the exception occurred is identified. Only if the lane is the oldest lane currently executing, i.e. the lane is the first set bit from the LSB in the SRV-replay register the exception is raised and handled. If it is not the oldest lane then the lane is simply marked for re-execution. This is done to guard against exceptions that could occur as a result of using erroneous data after a memory dependence violation. We exploit this by inserting an out of bound access to the location in memory we want to leak in a vector lane that isn't the oldest lane. Consider the following snippet of code:

Listing 7. Speculative vectorisation exploit gadget
```
int *x = read();
for (i=0; i<N; i++) {
    a[i] = b[a[x[i]]* 4096];
}
```

Here the array x contains attacker provided values. For example, $x = \{0, outofboundsvalue, 1, 2\}$. An array of attacker values is provided so that the compiler vectorises the load to a gather instruction. For the toy example we assume that there are 4 vector lanes. When the compiler vectorises this code for the first time all bits are set in the *SRV-replay* predicate register indicating that all lanes should execute. The out of bounds value can be set to *address of secret byte -*

*base address of a*. Lane 0 will execute load for a[0], Lane 1 will execute the load for the address of secret byte and so on. There will be an exception caused in lane 1 due to the invalid access, but since the oldest executing lane is lane 0, the exception will not be raised here but the SRV-needs-replay register will be set to 0b0010 indicating that the second lane has to be replayed. When the execution of the remaining lanes completes, the value of *SRV-replay* is set to 0b0010 and the SRV region is re-executed. At this time again the same access will cause an exception. However the previous load to the address of the secret would have completed and depending on the value of the secret byte, the corresponding cache line for array b will be touched. The attacker can use prime+probe or evict + reload mechanisms to find out what was the value of the secret byte. However since the code is vectorised, several sets in array b will be loaded. To distinguish them, the attacker runs the code first with all legal values in array x that are within bounds for a. After this one of the values is replaced with the out of bounds access. Depending on which set was differently accessed the second time, the value of the secret byte can be determined.

*3) Mitigation:* The key differentiating factor from spectre based attacks is that inserting fences can help mitigating a spectre-like attack [1], but this won't work for this attack since this exploits the exception suppression mechanism of speculative vectorisation. Secondly this doesn't involve mistraining the branch predictors, so any mechanism to harden the branch predictor wouldn't be able to stop this attack.

**Mitigation-1:** A strong possible mitigation strategy like the one proposed in [15] can be modified to protect against this attack. In InvisiSpec, the unsafe speculative loads read data into a new Speculative Buffer (SB) instead of into the caches and thus avoid modifying the cache hierarchy. The data in the SB is completely free of even the cache coherence transactions. When the load is finally safe InvisiSpec makes it visible to the rest of the system. This is done by re-performing the operation. The SB can be used even for the speculative memory accesses done during the SRV region. Once the PC encounters the *srv_start* instruction, the loads following that should be issued to the SB. Only after *srv_end* and when *SRV-needs-replay* register is 0 the loads should be reflected back into the memory subsystem. The estimated slowdown for this approach is similar to that mentioned in the paper i.e. 21% overhead.

**Mitigation-2:** An alternate mitigation strategy could be to undo the loads based on the address corresponding to the set bits in the *SRV-needs-replay* register. If an exception is caused in any of the lanes, a bit is set in the predicate register. An additional table should be maintained to keep track of the load memory addresses corresponding to each lane. The size of this table would be 64bits * number of vector lanes. Depending on whether the *SRV-needs-replay* is set, the corresponding memory address should be flushed from the cache. Alternatively the load could also be squashed. This is a low-overhead mechanism to prevent the attack.

*4) Related works:* There are several other similar work on which the proposed attack could be modified and deployed. For example in the microarchitecture proposed by Pajuelo et. al [16], previous history is maintained to predict whether a group of scalar operations is vectorisable or not. This information is used to execute the instructions in vector mode and incase of misspeculation a recovery mechanism is applied. The SRV attack can be re-purposed here by having a mistraining phase before launching the attack. Similarly the work on speculative vectorisation by Rakesh et.al [17] uses a software emulation layer to keep track of scalar instructions that can be transformed into vector instructions. This is similar to the scenario of the current attack except the vectorisation pass is done dynamically instead of using a compiler. Another similar work by Kumar et. al. [18] that speculatively reorders ambiguous memory references to create vectorisation opportunities could be targeted using a modified version of the proposed SRV attack. Thus this attack can be modified and deployed in various other scenarios.

### *D. Designing a Cost-Effective Cache Replacement Policy using Machine Learning*

As caches are limited in size, it is required to update the cache block constantly with the most optimal data that will be used in the not-too-distant future. Besides this, cache replacement lies on the critical path for a processor execution. Having a versatile cache replacement policy is very important as this gets translated to IPC gains. The paper starts from scratch by considering a huge number of factors that affect IPC based on cache replacement that include the offset, preuse, access type, set number, set accesses, etc. [5]. Based on an extensive reinforcement learning model developed by the authors and tested on multiple SPEC and Cloudsite benchmarks with the entire cache state as input to the model, the authors filter out the features that contribute most to determining the optimal cache state. The paper then implements these features in hardware and moves on to optimize these better to reduce hardware and power consumption. RLR (the policy implemented in the paper), achieves a 3.25% and 4.86% improvement over LRU with a meagre 4.7% increase in hardware budget.

*1) Threat Model:* The attack assumes that the cache lines brought into the cache during the setup phase are brought in by the attacker directly and not by the prefetcher. The RLR policy makes use of the type of cache block (whether brought in by the prefetcher or not) while calculating the priority of lines to evict. Having the prefetcher interfere in our setup may lead to noise in the setup phase which would be reflected in the actual attack and reduces the efficiency of the attack overall. The attack works on a vanilla cache with no protection domains and can also break into an Intel CAT [10] style partitioned cache. The attack takes advantage of the cache replacement policy that inadvertently leaks data to other processes that are occupying the cache set at the same time. We show that even with CAT protection enabled, the one common data structure that remains leaks data in the form of evictions in the other processes ways. The attacker can be an unprivileged process and the victim can be either unprivileged (different process) or privileged (kernel). The attacks consider a reuse distance value of 0 for ease of explanation. The reuse distance is a learnable parameter and the attacker can maliciously train the policy to adopt this value during the setup phase, just before the attack

2) *Exploitation:* The attack proposed by us is a cache metadata based attack; we make use of a common table that is used to keep track of all misses in the sets of the cache and by manipulating values during the setup phase.

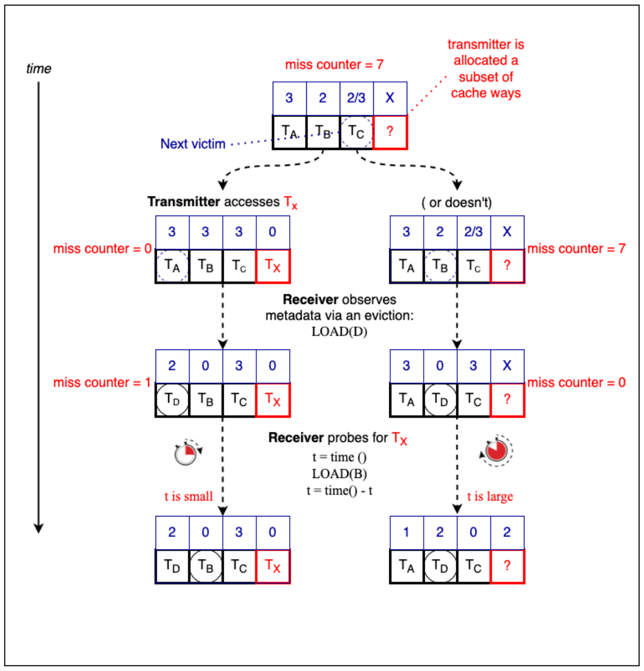


Fig. 5. RLR suggested attack

The metadata used in RLR is a 3 bit miss counter maintained for each set in the cache, a 2 bit age counter used to indicate how many accesses have occurred since the cache line was brought in, a 1 bit hit register that is used to indicate if the cache line was accessed and lastly, a 1 bit type register. The type register is used to indicate if the cache line inconsideration was brought in due to a prefetch or if it was an intentional action from the program. For the multicore design, RLR also stores the core number and uses this to calculate the priority of replacement. As the timing difference is to be observed in the attacker process which will run on the same core, having the victim running on a different core is of no consequence to the attack.

As the RLR plocy does not use the PC for any computations, there is no scope for PC aliased attacks and hence we focus on attacking the metadata used to keep track of the lines directly. In the toy example shown in figure 5, the setup phase sets the metadata for the policy to have the maximum possible set miss value such that any miss in the set will cause the counter to roll over and increment all age counters for lines in sets. We make sure that the hit registers for all line are set by accessing these lines once they are brought in. As the threat model states, we make sure that none of the lines are brought in due to prefetches and this ensures that the prefetch bit for all lines are reset. Depending on the victim's access pattern, the cache state in modified in different ways. On an attacker access after a victim access, the first way in the set is evicted whereas the second way is evicted if the victim has not accessed any data. By probing for the victim access data by checking for data filled into the attacker sets, based on the difference in timing between accesses, we can accurately predict the value of secret data. At the L3 cache (LLC) level, a hit costs 200 cycles whereas a miss costs about 400 cycles. This can easily be timed and used to check which block is missing from the attacker ways to leak data from the victim process. An extension of the attack can also be performed on the L1 cache level, where the victim thread running on the core leaks data to the attacker thread running on the same core via the L1 cache. As our previous attack covers a wider possibility of attacks with processes running on different cores and our suggested defense takes care of all scenarios, we do not focus on this L1 cache attack in detail.

3) *Mitigation:* The attack described for RLR is only possible due to the common miss counter in the set. If this counter is separated for each line, the above cache timing attack is rendered useless. This is shown in figure 6. Separating the age and miss counters was an optimization suggested in the paper to reduce hardware usage while sacrificing performance. Adding a separate miss counter to each line in cache would incur a 47% and a 67% increase in hardware requirement than the optimized vulnerable version. The unoptimized version of the RLR policy is 10% faster than the optimized version. The Hawkeye predictor which has a comparable budget to the unoptimized RLR policy is 20% slower than the latter. Thus, for a specification with a higher budget, the secure RLR policy still performs as good as the competition.

4) *Exploitation:* The attack proposed by us is a cache metadata based attack; we make use of a common table that is used to keep track of all misses in the sets of the cache and by manipulating values during the setup phase.

5) *Related works:* Replacement policies generally have data structures that get updated whenever there is a miss/hit in the cache. This form of attack is common for replacement policies that have a shared data structure between them like LRU, DRRIP [19] and KPC [20]. Replacement policies that use a PC based table to insert lines with specific metadata are out of the scope this defense as the defense focuses on removing the common metadata structure in the cache itself. PC based attacks do not work on RLR because the PC is never used to decide the line that will be replaced.

## *E. A Unified Approach to Eliminate Memory Accesses Early*

The "A Unified Approach to Eliminate Memory Accesses Early" proposes Unified Cache (UC) that can act as a small value cache (narrow-width cache) [21] and memory location to register file mapping. Both these structures unified in a single structure are aimed at reducing accesses to the remainder of the memory system (caches - L1, L2, L3 and RAM). The first

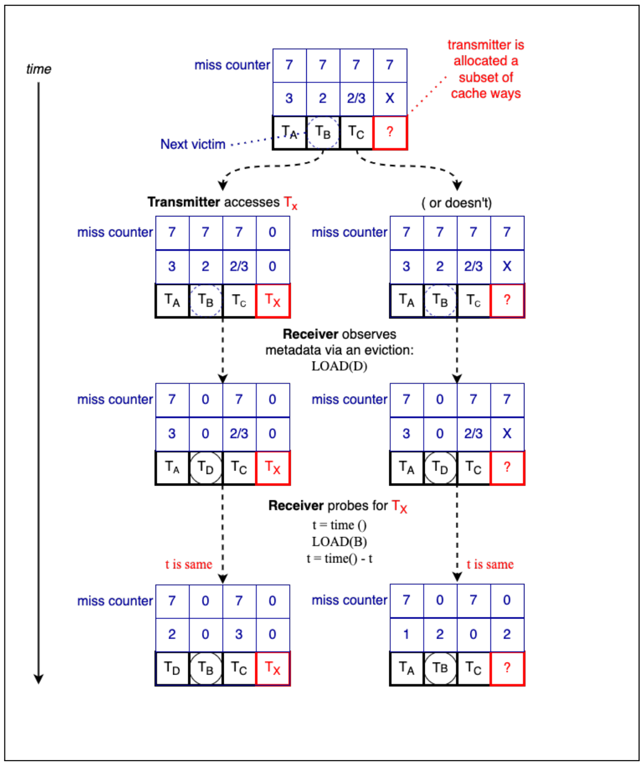


Fig. 6. RLR suggested defense

main idea is that the compiler only has a handful of registers to work with but the actual physical register file has a lot more number of registers. So some value the are being loaded in to the cache were already available in one of the physical register. So instead of getting that data from the caches or RAM, this access can be satisfied from within the CPU. The paper proposes to have a Register File Cache Pointer (RFCP) that is indexed into using the register ID and stores the value in that register along with the memory locations (pointers) that have the same value. The register ID that hold the same value as the memory location is retrieved from the UC and used to index into the RFCP to get the data. The second main idea, Small Value Cache (SVC), is aimed at satisfying loads and silent stores that access small valued integers (8-bits). The memory address is used to index into the UC which stores these small values. The UC is a cache like structure that is index into using memory address. Every entry in the UC can either be small value entry or a memory location to physical register mapping. The UC and RFCP is a structure that is shared among hyper-threads. This shared structure can expose hidden secrets from one thread to another.

*1) UC - Working:* As discussed before the UC is similar to a normal cache. The address or memory location is used to index into the UC. The entry will either store SVC entry or a Reg ID (memory location to physical register mapping). Whenever a store instruction is encountered, We first check if the entry already exists in the UC. If there is a Tag match, the entry already exists in the UC, else a previous entry is replaced using LRU replacement policy. The value of the store is always available in the PRF immediately after the store. So the entry inserted is a Reg ID. The status bits for every entry indicate if the entry is SVC entry or Reg ID or qualifies for both. If the entry qualifies for both then the entry created is used as Reg ID until that physical register is renamed and its contents are updated. If that happens then that entry becomes a SVC entry.

*2) RFCP - Working:* The RFCP tracks all the memory locations associated with the physical registers. The Reg ID is used to index into this structure. Every entry stores P addresses. These are the addresses associated with that physical register. If the contents of a register changes then the physical register no longer has the same contents as the memory locations. So the entries in the UC corresponding to these P addresses are invalidated.

The RFCP is a powerful tool that can be used by the attacker to steal secret information from the victim.

*3) Threat Model:* We assume an attacker can execute unprivileged instructions on the victim system. Our attack requires monitoring the state of the UC and RFCP shared with the victim program. In native execution, this is simply possible by using CPU affinity system calls to achieve core co-residency with the victim process. In cloud environments, previous work shows it is possible to achieve residency on the same machine with a victim virtual machine [22]. Cloud providers may turn hyper-threading on for increased utilization (e.g., on EC2) making it possible to share cores across virtual machines. Once the attacker achieves core co-residency with the victim, she can mount a UCleak attack using the shared unified cache (UC) and Register File Cache Pointer (RPCF). This applies to scenarios where a victim program processing sensitive information, such as cryptographic keys. (The structure of the Threat Model was inspired from TLBleed paper [23].)

- The attacker and the victim are two different processes running on the same core.
- The CPU should have SMT enabled.
- Attacker can expose internal variables of the victim.
- Victim program should be known to the attacker.

*4) Covert Channel - UC:* Same as regular cache side-channel attacks, the UC is also vulnerable to Prime+Probe. Unlike normal cache a new entry is only added to the UC on store instructions. We can use the SVC feature of the UC to populate the entire UC with SVC entries of the attacker. Any loads to these memory locations would then be satisfied by the UC. If the victims kicks out and entry, the load to these memory locations will take longer. The difference in the timing would be small. The access to UC should happen in 1 cycle while the access to L1 would take around 4 clock cycles. We can amplify the measurement by loading the same memory location 100s of times. Since load would not populate the UC, if an entry is missing from the UC it will not be added until there is a store at that location.

*5) Covert Channel - RFCP:* RFCP is the best tool is attackers arsenal. The RFCP associated Reg ID to memory addresses. It does not track who populated the physical register in the first place. If the Victim performs a store and the attacked had a register populated with the same value as the victim's store. Then the RFCP would associate that memory location to attackers physical register, kicking out the attackers entry. So instead of leaking 1-bit at a time, The RFCP can expose the entire value of the victim's secret.

*6) Exploitation - Attack 1:* This attack targets the SVC feature of the UC. The UC is shared between hyper-threads. So any modifications to the state of the UC can be probed by an attacker to reveal secret information.

Listing 8. Victim Gadget - Attack 1

```
if (secret) {
    store small value
}
```

The entries in the UC are only inserted/replaced on store instructions. If we assume that the Victim processes has only one small valued store that is dependant on the the secret, the attacker can simple use Prime+Probe attack on the UC to reveal the secret. Note that this attack will also work with ASLR because we are assuming there is only a single small valued store. It does not matter which attacker entry (set) it kicks out of the UC.

However these conditions are hard to come by. Stores that are not small valued stores will also be inserted in the UC as memory address to register ID mapping. This adds noise to our measurements.

*7) Exploitation - Attack 2:* This attack targets the Reg ID feature of the UC.

Listing 9. Victim Gadget - Attack 2

```
if (secret) {
    store value
}
```

If the victim performs a secret dependant store, it will kick out the attackers entry from the UC. It will also kick out the RFCP entry. The attacker can prime the UC with SVC entries and probe it to detect an eviction from the UC.

The attacker can also prime the UC+RFCP by using multiple independent stores, there by filling up both the UC and the RFCP. The load values to check if there was an eviction.

Again these conditions are hard to come by. If the victim code uses a lot of registers, then their contents would be changed and out primed UC+RFCP entried would be invalidated.

*8) Exploitation - Attack 3 (Real world attack):* This attack targets the UC and RFCP as a whole. We have seen constrained attacks where there are many conditions on the victims gadget. Now let look at a more generic victims gadget.

Listing 10. Victim Gadget - Attack 3

```
if (secret) {
    func1();
}
else {
    func2();
}
```

Assume the Victim performs one of two computations depending on the secret as shown above. The register usage footprint and usage of store instructions in these computations would be vastly different (or lets assume they are different). We can perform a TLBleed like attack on shared UC and RFCP.

The attacker would constantly prime the UC with small value stores and probe with loading these memory locations and measuring the latency. If the loads are satisfied by the UC the latency would be lower else the load would take longer. The cpuid instruction would help us measure the latency of the loads. We expect the latencies to be small values. To avoid interference of the storing these latencies with our Prime+Probe attack on the UC we can add a constant large value to the latencies before storing them.

Listing 11. Attacker Measurement code

```
loop_1:
    store small_value to Ax
    ..
    .. // populate the entire UC

    loop_2:
        cpuid();
        t1 = rdtsc();
        cpuid();
        load Ax // measure latency
        cpuid();
        t2 = rdtsc();
        cpuid();
        latency = t2-t1
```

Because the victim code is already known to us we can train a ML model to identify the secret from the collected signal (latencies). If ASLR is enabled on the victim's system, we can train another ML model that can identify the the offset in the set number created by the ASLR. Because the UC and RFCP are very small structures that can be accessed in 1 clock cycle the time granularity of our measurements would be much higher that TLBleed attack.

*9) Exploitation - Attack 4:* The RFCP associated Reg ID to memory locations. However that register might be populated by the attacker. The cross thread association can reveal secret information.

Listing 12. Victim Gadget - Attack 4

```
store secret
```

If the range of values of the secret is limited then the attacker can reveal the value of this secret. Lets assume that the value of the secret can be one of 8 different values, 0 to 7. The attacker can perform 8 stores with these values and keep the data values intact in the logical registers. This would mean that the physical registers associated with these logical register cannot be added to the free list for register renaming. Also these stores would create a association of these physical registers with the memory addresses of the store the attacker just performed in the RFCP. When the attacker store the secret The RFCP will be updated with the memory location of the victim's store address. This will kick out one of attackers association. Depending on which association was kicked out, the attacker can determine the value of the secret. This attack assumes that there won't be any other stores in the victims code with a value 0 to 7.

Listing 13. Attacker Measurement code

```
store 0 to A0
store 1 to A1
..
store 7 to A7

// Victim code runs

loop_1:
    cpuid();
    t1 = rdtsc();
    cpuid();
    load Ax // measure latency
    cpuid();
    t2 = rdtsc();
    cpuid();
    latency = t2-t1
```

For the dummy code above we assume that the RFCP only holds one memory address corresponding to every Reg ID. The attack would also work if there were P memory addresses linked to every Reg ID. The attacked would have to make multiple store with the same value to Prime all of RFCP.

Similar to our previous discussion if the difference in latency is too low, we can take multiple measurements to amplify the timing difference. For example, if the access time of UC+RFCP is 1 cycle and L1 cache is 4 cycles. loading the same location 100 times would give us a timing of 100 cycles for UC+RFCP and 400 cycles for L1 cache.

*10) Mitigation:* The problem is the shared structures. The following mitigation are possible.

- Separate UC+RFCP structures for hyper-threads. Although this sound unreasonable, The UC and RFCP are very small structures amounting to less than a kilobyte in size. Since modern processors do not support more than 2 hyper-threads per core, replicating these structures is the best solution. The sizes of the UC and RFC can further be reduce if necessary. This mitigation increases the size overhead of the optimization by 100%, but does not reduce the performance of the optimization. In-fact it would result in even better performance due to no aliasing among hyper-threads. This mitigation does not deal with the covert-channel created by having a PRF.
- Partitioning the UC and tagging Physical registers with hyper-threads so that cross thread association is not possible in RFCP. The CAT [10] like partition of UC would have to be accompanied by DAWG [7] like masking scheme for LRU metadata. This scheme would required either static or dynamic partitioning of UC. Dynamic partitioning might also required software support based on the implementation choices. We expect the performance improvement to drop from 8-13% to 4-5% due to reduction in the size of the UC available to each hyper-thread. The size overhead would be in the range of 10s of bytes. Tagging the physical registers would only required 1 bit per physical register.

*11) Related works:* There have been many studies on eliminating memory accesses using extra caches. Any study that uses extra level of cache is venerable to this attack. Although the idea of satisfying some load from with in the cpu by using free registers is an innovative idea, it poses new security threats. The following works are also affected by these attacks.

- SVC like optimizations - [24], [21], [25]
- Reuse of Register File contents - [26]

## III. Conclusion

The most basic assumption when running code on a processor is that data accessed in the process is isolated form other processes ensuring that the data being processed on cannot be accessed by other processes. The safety checks required to implement this are done in software and the hardware designer need not worry about these. As shown time and time again, this is not the case. Previously implemented attack such as flush+reload [27], prime and probe [28] can easily read the state of the cache modified by another process. With attacks such as spectre [1] and meltdown [2] which were a huge blow to the hardware industry, it is imperative to consider security while striving for increased performance and lower power consumption. The attacks described in the paper are but a tiny drop of possible attacks in the humongous ocean of hardware optimizations. Through this paper, we want to raise awareness of security-aware hardware optimizations.

## Acknowledgements

The authors wish to thank Samira Mirbagher Ajorpaz for her continued guidance and effort throughout the quarter along with her suggested timelines to keep the project progress in check. We want to thank Zhenya Ma for supporting us with constructive feedback on our work and for all the infrastructure related support. We wish to thank our apartment mates who helped us get through the quarter and kept us motivated throughout the project.

## References

[1] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[2] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[3] Vinson Young, Chia-Chen Chou, Aamer Jaleel, and Moin Qureshi. Ship++: Enhancing signature-based hit predictor for improved cache performance. In *The 2nd Cache Replacement Championship (CRC-2 Workshop in ISCA 2017)*, 2017.

[4] Akanksha Jain and Calvin Lin. Hawkeye: Leveraging belady's algorithm for improved cache replacement. *The 2nd Cache Replacement Championship*, 2017.

[5] Subhash Sethumurugan, Jieming Yin, and John Sartori. Designing a cost-effective cache replacement policy using machine learning. pages 291–303, 2021.

[6] Peng Sun, Giacomo Gabrielli, and Timothy M Jones. Speculative vectorisation with selective replay. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 223–236. IEEE, 2021.

[7] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987, 2018.

[8] Reena Panda Jiajun Wang, Lu Zhang and Lizy Kurian John. Less is more: Leveraging belady's algorithm with demand-based learning. 2017.

[9] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[10] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family. pages 657–668, 2016.

[11] Tomoki Nakamura, Toru Koizumi, Yuya Degawa, Hidetsugu Irie, Shuichi Sakai, and Ryota Shioya. D-jolt: Distant jolt prefetcher. *The 1st Instruction Prefetching Championship (IPC1)*, 2020.

[12] Aasheesh Kolli, Ali Saidi, and Thomas F. Wenisch. Rdip: Return-address-stack directed instruction prefetching. pages 260–271, 2013.

[13] Mingju Li, Elizabeth Varki, Swapnil Bhatia, and Arif Merchant. TaP: Table-based prefetching for storage caches. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, February 2008. USENIX Association.

[14] Pejman Lotfi-Kamran Ali Ansari, Fatemeh Golshan and Hamid Sarbazi-Azad. Mana: Microarchitecting an instruction prefetcher. 2020.

[15] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441. IEEE, 2018.

[16] Alex Pajuelo, Antonio González, and Mateo Valero. Speculative dynamic vectorization. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 271–280. IEEE, 2002.

[17] Rakesh Kumar, Alejandro Martínez, and Antonio González. Speculative dynamic vectorization for hw/sw codesigned processors. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 459–460. IEEE, 2012.

[18] Rakesh Kumar, Alejandro Martínez, and Antonio González. Assisting static compiler vectorization with a speculative dynamic vectorizer in an hw/sw codesigned environment. *ACM Transactions on Computer Systems (TOCS)*, 33(4):1–33, 2016.

[19] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). page 60–71, 2010.

[20] Jinchun Kim, Elvira Teran, Paul V. Gratz, Daniel A. Jiménez, Seth H. Pugsley, and Chris Wilkerson. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. *SIGPLAN Not.*, 52(4):737–749, apr 2017.

[21] M. d. Islam and P. Stenstrom. Zero-value caches: Cancelling loads that return zero. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 237–245, Los Alamitos, CA, USA, sep 2009. IEEE Computer Society.

[22] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in multi-tenant public clouds. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 913–928, USA, 2015. USENIX Association.

[23] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, Baltimore, MD, August 2018. USENIX Association.

[24] Hui Zeng and Kanad Ghose. Register file caching for energy efficiency. In *Proceedings of the 2006 International Symposium on Low Power Electronics and Design*, ISLPED '06, page 244–249, New York, NY, USA, 2006. Association for Computing Machinery.

[25] Mafijul Md. Islam and Per Stenström. Characterization and exploitation of narrow-width loads: the narrow-width cache approach. In *CASES '10*, 2010.

[26] Soner Önder and Rajiv Gupta. Load and store reuse using register file contents. In Mario Mango Furnari and Efstratios Gallopoulos, editors, *Proceedings of the 15th international conference on Supercomputing, ICS 2001, Sorrento, Napoli, Italy, June 16-21, 2001*, pages 289–302. ACM, 2001.

[27] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. pages 719–732, August 2014.

[28] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming browser-based Side-Channel defenses. pages 2863–2880, August 2021.