Golmaal: Thanks to the Secure TimeCache for a Faster DRAM Covert Channel

Ajaykumar Kushwaha*, Ajay Jain†, Mahendra Patel‡, Biswabandan Panda§
* *Dept. of Computer Science and Engineering, Indian Institute of Technology Bombay, Email: ajaykushwaha@cse.iitb.ac.in*
† *Dept. of Computer Science and Engineering, Indian Institute of Technology Bombay, Email: ajayjain@cse.iitb.ac.in*
‡ *Dept. of Computer Science and Engineering, Indian Institute of Technology Bombay, Email: mahendrapatel@cse.iitb.ac.in*
§ *Dept. of Computer Science and Engineering, Indian Institute of Technology Bombay, Email: biswa@cse.iitb.ac.in*

***Abstract*—Cache-based side-channels can cause information leakage thanks to the latency difference between a cache hit and a miss. One form of cache side channel is the shared memory channel that leading to attacks like Flush+Reload and Evict+Reload. A recent proposal in ISCA 2021 named TimeCache mitigates cache attacks that exploit the reuse of shared data and code between an attacker and the victim. With TimeCache, the first request to any memory address by a process is always a cache miss providing per-process cache line visibility, and it makes sure processes do not benefit from cached data brought in by another process until it pays the price of the corresponding cache miss penalty. Though TimeCache successfully mitigates cross-process shared memory attacks at the caches, it makes the life of a cross-core DRAM covert channel attacker easy.**

We propose Golmaal, a DRAM covert channel that takes the benefit of TimeCache as the sender/receiver need not flush the cache line to reach the DRAM. The first access to a memory address by different processes will always lead to a miss in the cache hierarchy, and notably at the shared last-level cache (LLC). We show that with the TimeCache, for a 16-core system sharing DDR4 DRAM controllers, the bandwidth of a Golmaal covert channel is in the range of 4.53Mbps to 6.82Mbps, whereas without TimeCache, the bandwidth achieved is in the range of 2.73Mbps to 4.61Mbps.

1. Introduction

Side and covert channel attacks through caches pose a threat to the security of the system. Some of the commonly used attack protocols are Prime+Probe [1] and Flush+Reload [2]. Flush-based attacks exploit the reuse of shared data and code. TimeCache [3] is a recent efficient yet lightweight mitigation technique that mitigates shared memory-based attacks like Flush+Reload [2] and Evict+Reload [4]. To mitigate the Flush+Reload cache attack, TimeCache makes the first access to a memory address by any process a compulsory miss at the caches, and most importantly, the last-level cache (LLC). This way, the attacker always gets a longer access latency after the victim's access to a shared LLC line.

A brief introduction to TimeCache. TimeCache eliminates reuse-based cache attacks that use shared data/code cache lines by providing per-process cache line visibility of a shared cache line. With TimeCache, accesses to a shared cache line by different processes are isolated in timing. For example, if an attacker core (core 0) and the victim core (core 1) share an LLC line L, then for a Flush+Reload attack, the following sequence of events will happen with TimeCache: (i) attacker core `flushes` the line L from LLC, (ii) victim core accesses the LLC line L and gets a response from DRAM, and (iii) then when the attacker core reloads the line L, it gets an LLC miss (thanks to per-process visibility cache lines with TimeCache).

TimeCache is implemented as a combination of both hardware and software approaches. Software stores (and restores) the process's caching context along with context switch timestamp at a context switch. The Hardware implements bit-serial comparison logic to allow parallel comparison of timestamps. This timestamp comparison is employed to update the stale caching context. Apart from timestamps, TimeCache uses a per-line per-process *security bit* (s-bit), initially set to 0. When the s-bit is set for a given process, it is not the first time access to a cache line by a process. For a given LLC line, if the s-bit is 0 for a process, the access results in a miss, and the response comes from the DRAM. In this way, TimeCache eliminates cross-core shared memory side and covert channel attacks at the LLC. Overall, TimeCache claims that it only incurs a minor delay of 1.5% due to the first access cache miss.

DRAM covert channel. DRAM covert channel exploits timing differences between a DRAM row-buffer hit and a row-buffer-conflict. First, both the sender and receiver agree on a specific DRAM bank for covert communication (thanks to reverse engineering of DRAM addressing scheme) [5]. For transmitting a "1": Receiver continuously accesses DRAM row i. Sender accesses a DRAM row j. Next, when the receiver tries to access row i, it gets a row-buffer conflict (slow access). Similarly, for transmitting "0", the receiver continuously accesses DRAM row i. The sender does nothing. Next, when the receiver tries to access row i, it gets a row-buffer hit (fast access). For a successful DRAM covert channel, both sender and receiver need to ensure that they get cache misses and access the DRAM

bank. The `clflush` instruction facilitates the same by flushing a cache line from all levels of cache by providing an LLC miss, whenever a sender or a receiver wants to access DRAM. State-of-the-art cross-core covert channel proposed in DRAMA [5] uses the above covert channel protocol and provides cross-core covert-channel bandwidth of 2Mbps.

Opportunity. With TimeCache, it is not possible to mount a cross-core Flush+Reload attack at the shared L3 cache. However, thanks to TimeCache, now it is possible to mount a covert channel with minimal usage of `clflush` as the first reload[1] to an address (e.g., A) by all the processes will come to DRAM. *So, in summary, TimeCache leads to a* Golmaal[2] *as it eliminates shared memory Flush+Reload attack at one shared resource (shared LLC) and in turn facilitates a new shared memory covert channel at another shared resource, the DRAM.*

Our approach. We propose a high-speed DRAM covert channel with multiple senders and receivers running on multiple cores. One DRAM bank is used as a covert channel between *multiple* sender and receiver processes sharing one DRAM bank. All the receivers share, and access one address (say A1) and all the senders share and access one address (say A2). A1 and A2 are mapped to two different DRAM rows of the same DRAM bank. Note that TimeCache provides cache miss for each process for their first access to an address. So, in our covert channel, a pair of senders and receivers communicate iteratively. With our protocol, only the first sender and receiver use clflush. For example, after a `clflush` to A2 by the first sender and A1 by the first receiver, there is no need to use `clflush` by the rest of the senders and receivers as all of them will get LLC misses, thanks to TimeCache. After one round of communication between all the senders and receivers, the sender and receiver need to use `clflush` again. For example, if we use 16 senders and 16 receivers for a covert channel and all the senders want to communicate "1", then with TimeCache, only the first sender and receiver use `clflush` and `LOAD` respective memory addresses. After that the remaining 15 senders and receivers just `LOAD` their respective addresses (A2 and A1) as they get LLC misses even without using `clflush`. The limited usage of `clflush` improves the overall covert channel bandwidth.

Overall, we make the following key contributions:

- We propose Golmaal, a high bandwidth DRAM covert channel. Golmaal exploits the design principle of a recent mitigation technique called TimeCache that mitigates shared memory attacks at the LLC.
- Our covert channel uses multiple senders and receiver processes for covert communication. The effectiveness of our covert channel increases with an increase in the numbers of senders and receivers.
- We quantify the effectiveness of our channel in terms of the raw-bandwidth and error rate. Overall, Golmaal improves the bandwidth of a DRAM covert channel with TimeCache compared to a conventional cache.

1. We use the term load and reload interchangeably, in this paper.
2. Golmaal is a Hindi word that means a deceptive situation with some amount of confusion involved.

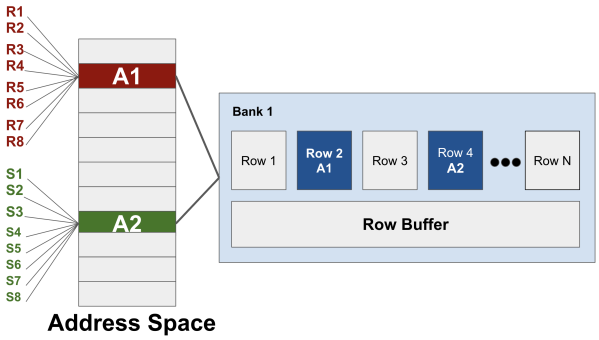

Figure 1. Eight sender (S1 to S8) and receiver (R1 to R8) processes communicating over one DRAM bank. All the senders share and access address A1 and all the receivers share and access address A2. Both A1 and A2 are mapped to two different DRAM rows but one DRAM bank.

2. Golmaal DRAM Covert Channel

Threat model and assumptions. Our covert channel requires shared memory similar to Flush+Reload [2] and Evict+Reload [4] that TimeCache eliminates at the LLC. Note that as TimeCache is a mitigation technique is not implemented on a real machine, we perform our covert channel on a cycle-accurate micro-architectural simulator. Both the sender(s) and the receiver(s) are aware of DRAM addressing (thanks to the effort on reverse-engineering [5]). Sender and receiver agree to communicate through a fixed set of addresses mapped to one DRAM bank. Our covert channel is a cross-core covert channel where the sender and the receiver are part of multiple cores running on the same physical CPU sharing the LLC and DRAM. We measure the memory access latency with `rdtscp` and the memory accesses are performed using volatile pointers similar to [5]. For synchronization between a sender and a receiver, we use the wall clock available per physical CPU.

Shared memory agreement. We implement our covert channel with multiple senders and multiple receiver processes sharing addresses of one DRAM bank of a given rank and channel. Figure 1 provides a high level overview of the agreement between sender and receiver processes. We do not relax the shared memory requirement as our goal is to show the effect of TimeCache at the DRAM when it eliminates shared memory attacks at the LLC.

Latency of `clflush` and `LOAD`. For a successful DRAM covert channel, the sender and receiver need to use `clflush` so that their memory accesses will reach DRAM [5]. As per the Intel manual [6], a `clflush` instruction does the following: it "invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location.

ALGORITHM 1: Receiver process

```
1: while do
2:   for all receivers (receiver-id 1 to N) do
3:     // N receivers receiving N bits of covert
       information in one iteration of for loop
4:     if receiver-id ==1 then
5:       // For the 1st receiver
6:       clflush(A1)
7:       LOAD(A1)
8:       if load(A1) is fast then
9:         databit=0 // Row-buffer hit
10:      else
11:        databit=1 // Row-buffer conflict
12:      end if
13:    end if
14:    if receiver-id >1 then
15:      // For the rest of the receivers
16:      LOAD(A1)
17:      if load(A1) is fast then
18:        databit=0 // LLC miss in TimeCache,
           Row-buffer hit
19:      else
20:        databit=1 // LLC miss in TimeCache,
           Row-buffer conflict
21:      end if
22:    end if
23:    sync-with-sender()
24:  end for
25: end while
```

ALGORITHM 2: Sender process

```
1: while do
2:   for all senders (sender-id 1 to N) do
3:     // N senders sending N bits of covert information
       in one iteration of for loop
4:     if sender-id==1 then
5:       // For the 1st sender
6:       if databit==0 then
7:         Do nothing
8:       end if
9:       if databit==1 then
10:        clflush(A2)
11:        LOAD(A2)
12:      end if
13:    end if
14:    if sender-id >1 then
15:      // For the rest of the senders
16:      if databit==0 then
17:        Do nothing
18:      end if
19:      if databit==1 then
20:        LOAD(A2)
21:      end if
22:    end if
23:    sync-with-receiver()
24:  end for
25: end while
```

The clflush instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and besides, a clflush instruction is allowed to flush a linear address in an execute-only segment". Note that the recent `clflush` implementation flushes the cache line across the entire cache coherence domain, irrespective of inclusive and exclusive nature of cache hierarchy [7]. One can argue about the usage of CLFLUSHOPT for a faster flush based covert channel. However, CLFLUSHOPT is more suitable to flush large buffers (e.g. greater than many KBytes), compared to CLFLUSH [8]. In summary, a `LOAD` to an address after the usage of `clflush` to that address will always lead to a DRAM access.

We perform latency measurements using `rdtscp` for `clflush` and `LOAD` latencies on Intel (Intel IceLake) and AMD (A6-9220 RADEON R4) processors with eight cores. The `clflush` latency varies from 185 cycles to 320 cycles depending on the nature of cache line (clean or dirty) that is flushed. The `reload` latency with DRAM row-buffer conflict is around 375 cycles, whereas with row-buffer hit, it is around 250 cycles, which is easily distinguishable. These latency numbers correlate with the recent Intel IceLake latency numbers for DRAM [9]. These numbers are also similar to the numbers reported in a flushless LLC covert channel [10].

Transmission protocol. Algorithms 1 and 2 provide the pseudocode of receiver and sender processes, for covert communication. Before communicating bits of information, all the senders and receivers need to synchronize themselves, similar to [5]. Also, the sender should run ahead by atleast one DRAM access latency (say 400 cycles) so that receiver can catch up and get accurate bits. The receiver always accesses an address A1 and measures its latency using `rdtscp`. If the latency is fast then it concludes that it has received a bit "0" else "1". Note that with the TimeCache, all the receivers and senders do not use `clflush` to reach the DRAM once the first sender and receiver have completed the usage of `clflush`. This process repeats for every iteration of the `for loop` where `clflush` is used only once by the 1st sender and 1st receiver.

Proof of concept comparison. For the sake of simplicity, let's assume a `clflush` takes 300 cycles to complete and a memory access with row-buffer conflict takes 300 cycles, whereas a row-buffer hit takes around 200 cycles.

Without the TimeCache, a covert channel between eight senders and eight receivers communicating eight bits of covert information through one DRAM bank will take an average of 6800 cycles. A DRAM covert channel takes 500 cycles to communicate a "0", a sender does nothing, and a receiver uses `clflush` (300 cycles on an average) and then reloads an address A1, and observes a row-buffer hit (200 cycles on average). To communicate a "1", sender uses `clflush` (300 cycles) and `reload` (300 cycles) to address A2 and receiver uses `clflush` (300 cycles) and

reload to address A1 (300 cycles), and observe a row buffer conflict. So, in summary, to communicate "1" it takes 1200 cycles. So, on average, without TimeCache, one bit is communicated in 850 cycles (6800 cycles for eight bits).

With TimeCache, to communicate a "0", it takes clflush (300 cycles) and load (200 cycles) from the the first receiver process with a row buffer hit. So in total 500 cycles to communicate a "0" for the first process. For communicating "1", the first sender uses clflush (300 cycles) and reload (300 cycles) to address A2 and the first receiver uses clflush (300 cycles) and reload to address A1 (300 cycles), and observe a row buffer conflict. So, in summary to communicate "1", takes 1200 cycles for the first receiver process. So, on average, it takes 850 cycles to communicate the *first* bit (similar to the conventional cache).

However, for the rest of the seven sender and receiver processes, there is no need for clflush, which saves 300 cycles from the sender side and 300 cycles from the receiver side. Note that, it will take 200 cycles for receiving a bit "0" and 600 cycles for receiving a bit "1". So, on average, it takes 400 cycles per bit to communicate the 2nd bit to 8th bit. In total, with TimeCache, on average, a receiver will get eight bits after 850 + (7 $\times$ 400) cycles = 3650 cycles as compared to 6800 cycles without TimeCache. Please note that for the sake of simplicity, we do not include the synchronization overhead (waiting period of sender and receiver after communicating/receiving one bit of information) between the sender and receiver as the overhead is a constant factor in both TimeCache and the conventional cache. We also do not show the usage of multiple DRAM banks for covert communication as that is just an extension of what we describe above for a single DRAM bank, and multiple senders and receivers can communicate data in a pipelined and interleaved way across multiple DRAM banks.

Timing Diagram. Figure 2 shows the difference in latency with and without TimeCache to communicate a payload of "0s". Figure 2(a) shows the timing sequence without Timecache where a receiver receives a sequence of zeros. Note that to communicate a zero, the sender does nothing. Without TimeCache, the receiver first needs to clflush to make sure it reaches DRAM, then performs a LOAD. Next, the other receiver processes (receiver-2 and so on) perform a sequence of clflush and LOAD operations. This sequence continues for all the processes. However, with the TimeCache, only the first receiver process performs a clflush and the rest of the receiver processes (receiver-2 onward) do not use clflush as the first access to an address made by a process is always an LLC miss. The entire sequence of operations goes on till all the receiver processes finish participating for one iteration (one iteration of the for loop in Algorithm 1) in the covert channel. After one complete iteration, the first receiver process again needs to use clflush. Overall, the less usage of clflush effectively improves the DRAM covert channel bandwidth.

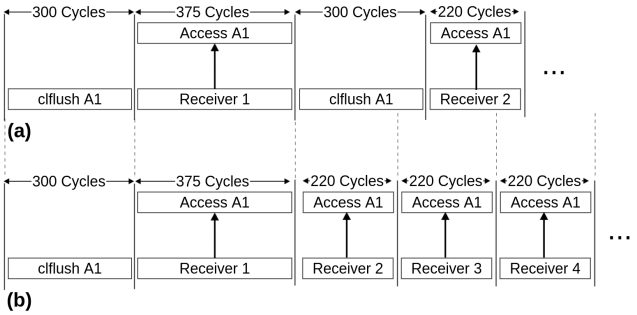

Figure 2. Timing diagram for a payload of "0000...". (a) Covert Channel on a conventional cache and (b) proposed covert channel on a TimeCache.

3. Evaluation

Simulation infrastructure. We simulate our covert channel with an extensively modified ChampSim microarchitectural simulator [11] that faithfully models a detailed front-end and back-end of the processor, and the memory system involving caches, TLBs, DRAM controller, and DRAM. We use the Ramulator [12] DRAM simulator and merge it with ChampSim for a detailed simulation of DRAM timing constraints. ChampSim was used for the recent championships at ISCA 2016, ISCA 2019, ISCA 2020, and ISCA 2021. We simulate a 16-core system with microarchitecture parameters as mentioned in Table 1. We simulate 2-way hyper-threading per core, effectively simulating upto 32 hardware threads (sixteen sender and sixteen receiver processes) on a 16-core system. To synchronize between a pair of sender and receiver processes, we use the wall clock time and use the nanosleep system call for 100ns (the worst case latency of a clflush and LOAD operations, 400 cycles) after every DRAM access by a sender and a receiver. Our simulated parameters are listed in Table 1 and correlates with the recent Intel Sunny Cove. We simulate DDR4 DRAM controllers with a data rate of 3200 MT/sec. We implement this attack through one DRAM bank of a specific DRAM channel. Our DRAM addressing and numbers of ranks, banks, and row size is same as the addressing scheme mentioned in [5] for DDR4.

As TimeCache is not part of commercial multicore systems, we implement our covert channel on a cycle accurate microarchitecture simulator. We agree that the raw bandwidth of our covert channel will decrease based on the real system configuration. However, it will still be faster as shown in the proof of concept example, if TimeCache becomes part of real commercial systems.

Raw bandwidth. Figure 3 shows the raw bandwidth (number of bits communicated per second) of the Golmaal covert channel for a different count of sender and receiver processes. As we can see, the raw bandwidth increases with the increase in senders and receivers, which is intuitive. We observe the maximum bandwidth of **6.82Mbps** when a group of sixteen senders and sixteen receivers communicate a sequence of "0s". With eight senders and eight receivers, we observe a bandwidth of **6.61Mbps**. We see a

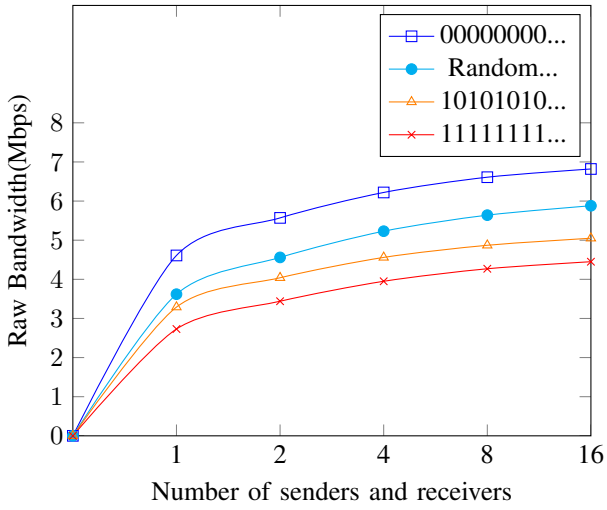

Figure 3. Raw bandwidth for different data payloads with various sender-receiver configurations.

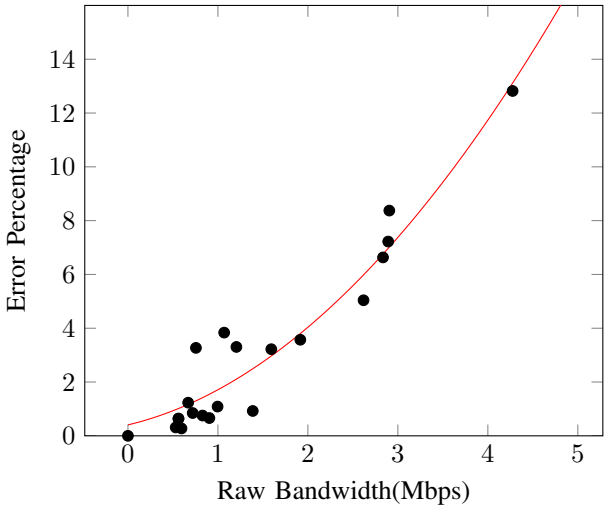

Figure 4. Error percentage and raw-bandwidth(Mbps) for a pair of eight senders and eight receivers communicating a payload of "11111111..1".

Table 1. SIMULATED PARAMETERS.

Core	16 Out-of-order cores, hashed perceptron branch predictor, 4GHz with 6 issue width, 4 retire width, 352 entry ROB
TLBs	64 entry 4-way at L1 DTLB/ITLB (1 cycle), 2048 entries 16-way entry L2 STLB (8 cycles)
MMU Caches	2 entry (PSCL5), 4 entry (PSCL4), 8 entry (PSCL3), 32 entry (PSCL2), searched parallely, one cycle
L1	32KB 8-way L1I (4 cycles), 48KB 12-way L1D (5 cycles)
L2	512KB 8-way associative (10 cycles)
LLC	16 2MB slices, 16-way (average, 80 cycles)
DRAM	1 channel/4-cores, DDR4-3200, 16 banks/rank, 4 bank groups

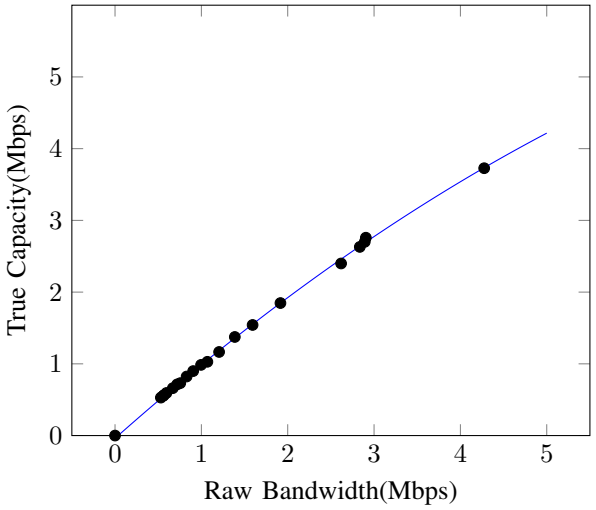

Figure 5. True Capacity(Mbps) and Raw-Bandwidth(Mbps) for eight senders and eight receivers communicating a payload of "1111111...1".

decrease in raw bandwidth when senders and receivers want to communicate a sequence of "1s" with a bandwidth of **4.53Mbps**. While communicating "1s", the senders perform `LOAD` access that decreases the bandwidth as compared to the case where senders communicate a sequence of "0s" where the sender does nothing. While communicating "1s" and "0s" alternatively, the Golmaal channel provides a raw bandwidth of **5.05Mbps**. For the average case, we have generated 10 random sequences, then while calculating the bandwidth, we take the average and get a raw bandwidth of **5.88Mbps**. When compared with the DRAM covert channel without TimeCache, we observe a covert channel bandwidth of **2.73Mbps** to **4.61Mbps** for communication of payloads with all "1s" and all "0s", respectively. Note that, our bandwidth numbers do not consider the effect of system noise as we simulate the channel on a simulator.

In summary, TimeCache improves the speed of a DRAM covert channel as the raw bandwidth without Timecache. Please note that, a DRAM covert channel bandwidth of 5Mbps (more than 600 KBps) is in the range of LLC based covert channels like [13] and [2] that provide a raw bandwidth of 400 KBps. So Timecache, eliminates flush based side/covert channels at the caches and makes it easy to mount a faster covert channel at the DRAM.

Error probability, true capacity, and effect of synchronization period. To understand the effect of errors on the raw bandwidth, we quantify the error rate for a pair of eight senders and eight receivers as eight senders and eight receivers provide the sweet spot in terms of raw bandwidth and error rate. Figure 4 shows an increase in the error rate (bit "1" termed as "0") with an increase in covert channel bandwidth. The covert channel bandwidth is affected by the synchronization delay, the higher the synchronization delay, the lower the error rate and covert channel bandwidth. Figure 5 shows the true capacity in Mbps (which is just below 5Mbps) for the eight senders and receivers communicating a payload of "11111111....1".

Pushing the covert-channel bandwidth. Recent flush-less cache covert channel like Streamline [10] provides very high channel bandwidth. To push the limits of the Golmaal channel bandwidth, multiple DRAM banks (two addresses per bank) can be used concurrently where at a given point

of time, one sender and receiver can communicate through one DRAM bank. The transmission protocol should be a pipelined one and the synchronization among senders and receivers across multiple banks will be the key for high bandwidth. We intend to explore these ideas as part of our future work.

Effect of Memory scheduling policy. Memory scheduling policies can affect the order of LOADs and clflush requests, at the DRAM controller. However, with Golmaal, there is a less scope of reordering thanks to the delay induced because of synchronization between sender and receiver.

Mitigation techniques. The effectiveness of Golmaal DRAM channel can be reduced by implementing TimeCache at the DRAM row buffer. However, this will incur additional design complexity. Another approach of TimeCache can be to project the latency of DRAM on every first-access from a process but not by sending the request to DRAM but by delaying the response from LLC (by adding a constant: 100s of cycles).

4. Conclusion

We proposed a DRAM covert channel named Golmaal on TimeCache that is faster than the DRAM covert channels on a conventional cache. We find that, TimeCache can mitigate the Flush+Reload at the shared cache. However, it improves the bandwidth of a new proposed DRAM covert channel that utilizes multiple processes to fool TimeCache.

References

[1] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[2] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 719–732.

[3] D. Ojha and S. Dwarkadas, "Timecache: Using time to eliminate cache side channels when sharing software," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 375–387.

[4] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive Last-Level caches," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 897–912. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss

[5] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for Cross-CPU attacks," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 565–581. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl

[6] "Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual. Number 253669-033US." 2018.

[7] A. Saxena and B. Panda, "Dabangg: A case for noise resilient flush-based cache attacks," in *16th IEEE Workshop on Offensive Technologies*, ser. WOOT 2022. SAN FRANCISCO, CA: IEEE, 2022, pp. 1–11. [Online]. Available: https://www.cse.iitb.ac.in/~biswa/WOOT22.pdf

[8] "CLFLUSHOPT," http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2014.

[9] "DRAM latency in IceLake," https://www.7-cpu.com/cpu/Ice_Lake.html, 2021.

[10] G. Saileshwar, C. W. Fletcher, and M. Qureshi, "Streamline: A fast, flushless cache covert-channel attack by enabling asynchronous collusion," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1077–1090. [Online]. Available: https://doi.org/10.1145/3445814.3446742

[11] Online. Available: https://github.com/ChampSim/ChampSim, Champsim Simulator.

[12] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[13] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. Berlin, Heidelberg: Springer-Verlag, 2016, p. 279–299. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_14

–

–