Streamline: A Fast, Flushless Cache Covert-Channel Attack by Enabling Asynchronous Collusion

Gururaj Saileshwar
gururaj.s@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Christopher W. Fletcher
cwfletch@illinois.edu
University of Illinois
Urbana-Champaign, IL, USA

Moinuddin Qureshi
moin@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

ABSTRACT

Covert-channel attacks exploit contention on shared hardware resources such as processor caches to transmit information between colluding processes on the same system. In recent years, covert channels leveraging cacheline-flush instructions, such as Flush+Reload and Flush+Flush, have emerged as the fastest cross-core attacks. However, current attacks are limited in their applicability and bit-rate not due to any fundamental hardware limitations, but due to their protocol design requiring flush instructions and tight synchronization between sender and receiver, where both processes synchronize every bit-period to maintain low error-rates.

In this paper, we present *Streamline*, a flush-less covert-channel attack faster than all prior known attacks. The key insight behind the higher channel bandwidth is asynchronous communication. Streamline communicates over a sequence of shared addresses (larger than the cache size), where the sender can move to the next address after transmitting each bit without waiting for the receiver. Furthermore, it ensures that addresses accessed by the sender are preserved in the cache until the receiver has accessed them. Finally, by the time the sender accesses the entire sequence and wraps around, the cache-thrashing property ensures that the previously transmitted addresses are automatically evicted from the cache without any cacheline flushes, which ensures functional correctness while simultaneously improving channel bandwidth. To orchestrate Streamline on a real system, we overcome multiple challenges, such as circumventing hardware optimizations (prefetching and replacement policy), and ensuring that the sender and receiver have similar execution rates. We demonstrate Streamline on an Intel Skylake CPU and show that it achieves a bit-rate of 1801 KB/s, which is 3x to 3.6x faster than the previous fastest Take-a-Way (588 KB/s) and Flush+Flush (496 KB/s) attacks, at comparable error rates. Unlike prior attacks, Streamline only relies on generic properties of caches and is applicable to processors of all ISAs (x86, ARM, etc.) and micro-architectures (Intel, AMD, etc.).

ASPLOS '21, April 19–23, 2021, Virtual, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8317-2/21/04...$15.00
https://doi.org/10.1145/3445814.3446742

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; • **Computer systems organization → Architectures**.

KEYWORDS

Covert-channel Attacks, Shared Caches, Asynchronous Protocols

ACM Reference Format:
Gururaj Saileshwar, Christopher W. Fletcher, and Moinuddin Qureshi. 2021. 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 (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3445814.3446742

1 INTRODUCTION

Covert-channels allow colluding processes to communicate with each other without detection. One of the most commonly exploited channels are cache covert-channels, that emanate from timing differences between accesses to processor caches (tens of ns) and DRAM (~100 ns). As caches are typically shared between processes, VMs, and even multiple processor cores, a sender process can easily influence whether an address shared with a co-running receiver process is in the cache or not, and modulate the latency observed by the receiver for accesses to that address. Consequently, cache covert-channel attacks are one of the fastest and most robust micro-architectural covert channels and have been heavily used to transmit information in several recent transient execution attacks such as Spectre [18], Meltdown [21], ExSpectre [34], etc., in comparison to other micro-architectural covert-channels [2, 8, 11].

To understand the potential for information leakage via caches, it is important to bound the bit-rate achievable for cache covert-channel attacks (the maximum covert-channel bit-rate is typically an upper bound on potential information leakage via a side-channel). Additionally, higher bit-rate covert-channels can allow exfiltration of payloads in shorter times. Consequently, this paper focuses on understanding the limitations in bit-rate for state-of-the-art cache covert-channel attacks and exploring the construction of newer attacks achieving higher bit-rates. Our default focus is on cross-core cache attacks, where a malicious sender and a receiver process execute on two different processor cores and attempt covert communication via accesses to the shared LLC, as such a setting is typical for a virtualized environment with a per-core resource allocation.

The current fastest cross-core covert-channels are flush-based attacks, such as Flush+Reload [40] and Flush+Flush [13]. In these attacks, the sender and the receiver operate synchronously and

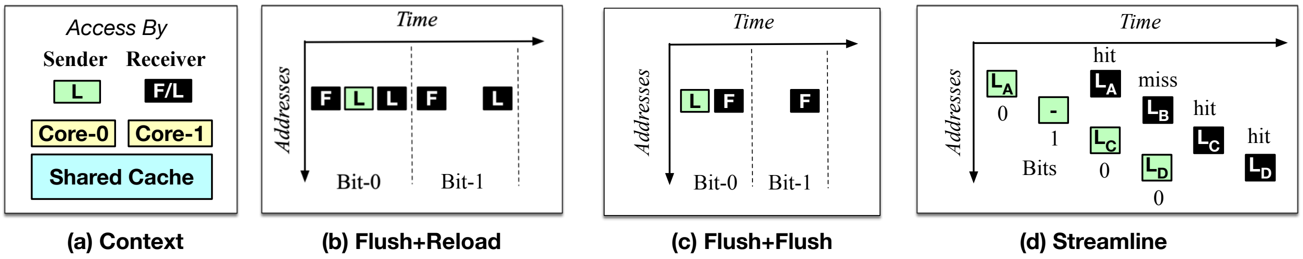


Figure 1: (a) Cache covert-channel attacks allow the colluding sender and receiver processes to transmit information via timing differences on accesses to shared caches. (b,c) Prior covert-channel attacks require synchronized transmission and flushes (F) in addition to loads (L) for each bit sent between the sender and the receiver: both requirements limit the channel bit-rates to 298 KB/s (Flush+Reload) and 498 KB/s (Flush+Flush). (d) In Streamline, the sender and receiver asynchronously communicate on a large sequence of addresses without flushes (each bit transmitted on a new address), achieving a bit-rate of 1801 KB/s.

transmit information within each bit period using the timing difference between LLC hits and misses on a read-only shared address (available in shared-libraries or via deduplication of pages across processes by Linux KSM [3]). In a Flush+Reload attack shown in Figure 1(b), within each bit-period, the receiver uses a cache-line flush instruction (e.g. `clflush` in x86 systems) to evict a shared address from the cache, then waits for the sender to access it, and then reloads the address. If the receiver observes a fast cache-hit (due to the sender accessing it), it decodes a 0, whereas a slow access indicates a 1. The Flush+Flush attack, shown in Figure 1(c), is a faster attack where the receiver measures the latency of the `clflush` on the shared address, which executes faster on an address installed into the cache by the sender; a faster flush indicates the sender sent a 0, whereas a slower flush indicates a 1. Gruss et al. [13] showed these attacks achieve bit-rates of 298 KB/s (Flush+Reload) and 496 KB/s (Flush+Flush) at less than 1% bit-error-rate.

The transmission rate of these channels is limited by two requirements: (1) synchronous communication between the sender and receiver, with each bit being transmitted during a coordinated time window, and (2) having to execute at least one flush and (one or two) load operations within the synchronous window for each bit. With such channels, it is difficult to obtain bandwidths higher than 1 MB/s (i.e. bit-period <125 ns) because the latency of flush is typically 50–70 ns and the latency of memory is approximately 100 ns. A bit period less than 150 ns breaks down the channel due to loss of synchronization. Additionally, attacks requiring cache-line flush instructions are not universally applicable, especially for several ARM processors [12] where the unprivileged use of flush instructions is disabled by default (in ARM v8 ISA) or completely unsupported (in ARM v7 ISA).

Towards investigating the feasibility of a more universal covert-channel attack, we enable *Streamline*, a flush-less attack that is also faster than all known covert-channel attacks. The key idea behind this attack is to enable the sender and receiver to communicate asynchronously over a large number of lines by using the cache to buffer data between the sender and the receiver, and relying on cache thrashing to naturally evict the resident lines (instead of using expensive clflush operations to explicitly evict lines).

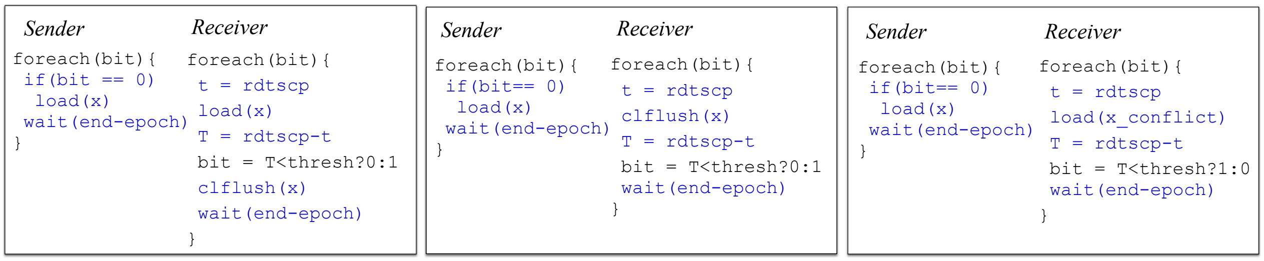
The protocol starts with the sender and receiver sharing a large shared array a few tens of MBs in size (larger than the size of the LLC), rather than a single or a small number of addresses[1], like in prior attacks. The sender and the receiver have a pre-determined sequence of addresses within the array over which they transmit successive bits. The sender transmits on successive addresses without waiting for the receiver as long as the receiver follows behind accessing the same addresses in a streamlined manner, as shown in Figure 1(d). The encoding is similar to prior works, wherein the sender accesses an address to transmit Bit-0 and does not access it to transmit Bit-1, allowing the receiver to infer a 0 or 1 based on whether it observes an LLC-Hit/LLC-Miss respectively. If the shared array is sufficiently larger than the LLC capacity, by the time the sender wraps around to the beginning of the array, the addresses installed during the previous iteration are automatically evicted due to cache-thrashing. Such an attack is significantly faster than previous attacks [13, 40], as it does not require synchronization every bit-period between the sender and receiver and only requires a single operation (one load) per bit. The bit-rate of this attack is only limited by how fast loads can be executed and measured.

To orchestrate Streamline with low error-rates, we face and address two key challenges unique to asynchronous protocols:

Challenge-1. Ensuring the sequence of addresses maps to and occupies a significant fraction of the cache: It is critical that the addresses accessed by the sender map to diverse locations in the cache. Otherwise, successive addresses accessed by the sender may evict previous addresses before the receiver can access them, causing errors. The access sequence also needs to circumvent hardware optimizations, like the prefetcher, designed to predict memory access patterns and preemptively install lines in the cache, and the cache replacement policy, which preemptively evicts lines with low reuse from the cache. We present a general approach to generate an address sequence that occupies a significant fraction of the LLC, fools the prefetcher, and is resilient to the replacement policy, that is applicable to any asynchronous attack.

Challenge-2. Ensuring a bounded gap between the sender and receiver: For Streamline to succeed, it is critical that the sender remain ahead of the receiver, and both execute at similar rates. If

[1]Perceival's [26] pioneering work on covert-channels also used a sequence of array-entries as large as the L1-cache for communication; but it is more than 4x slower than our channel because it still relies on synchronous operation (see Section 5.2)



Sender
```
foreach(bit){
 if(bit == 0)
  load(x)
 wait(end-epoch)
}
```
Receiver
```
foreach(bit){
 t = rdtscp
 load(x)
 T = rdtscp-t
 bit = T<thresh?0:1
 clflush(x)
 wait(end-epoch)
}
```
(a) Flush+Reload Attack

Sender
```
foreach(bit){
 if(bit== 0)
  load(x)
 wait(end-epoch)
}
```
Receiver
```
foreach(bit){
 t = rdtscp
 clflush(x)
 T = rdtscp-t
 bit = T<thresh?0:1
 wait(end-epoch)
}
```
(b) Flush+Flush Attack

Sender
```
foreach(bit){
 if(bit== 0)
  load(x)
 wait(end-epoch)
}
```
Receiver
```
foreach(bit){
 t = rdtscp
 load(x_conflict)
 T = rdtscp-t
 bit = T<thresh?1:0
 wait(end-epoch)
}
```
(c) Take-a-Way Attack

Figure 2: State-of-the-art covert-channel attacks. All existing covert-channels require the sender and receiver to communicate each bit in a synchronized epoch, and wait till the epoch ends before communicating the next bit. Cross-core Flush+Reload attack achieves bit-rate of 298KB/s, cross-core Flush+Flush achieves 496KB/s, same-core Take-a-way attack achieves 588 KB/s.

the receiver is faster, it can overtake the sender and observe a spurious stream of cache misses. Whereas, if the sender goes too far ahead of the receiver, it can evict its own addresses before the receiver can access them. We develop strategies to balance the sender and receiver rates, including a pseudo-random channel encoding to match the rate of DRAM-accesses executed by the sender and the receiver, and matching the number of `rdtscp` executed. While these optimizations reduce the rate-mismatch, minor differences in execution speed between processes are expected on a real system which can cause the gap between the sender and the receiver to grow unbounded over time. As a fail-safe, we enforce coarse-grain synchronization (once every 200,000 bits) between them using a lower bandwidth covert channel, to limit the maximum gap between the sender and the receiver.

Overall, this paper makes the following contributions:

(1) To our knowledge, this is the first paper to propose a high-bandwidth cache covert channel without relying on a synchronized protocol to transmit each bit. Our proposal, *Streamline*, uses the cache to buffer data between the sender and the receiver, and relies on thrashing to naturally evict the data from the cache post transmission.
(2) We discover and overcome obstacles for high-bandwidth attacks, such as circumventing hardware optimizations (fooling the prefetcher and replacement policy) and ensuring a bounded gap between sender and receiver (via rate-matching and coarse-grained synchronization).
(3) We demonstrate Streamline on Intel Xeon Skylake in a cross-core setting and achieve a bit-rate of 1801 KB/s at a bit-error-rate of 0.37%. Our bit-rate is 3.6x higher than Flush+Flush (496 KB/s), the prior-best cross-core attack, and 3x higher than Take-a-Way attack [20] (588 KB/s), the prior-fastest same-core cache attack.
(4) We discover a fundamental limitation of existing load-latency measurement gadgets that prevents latency measurement of multiple loads in parallel and limits the potential bit-rate of Streamline and even other future attacks.

The Streamline attack code is open-sourced at: https://github.com/gururaj-s/streamline.

2 BACKGROUND

We first provide background on recent cache covert-channel attacks and then cover their limitations to motivate our work.

2.1 Attack Model for Cache Covert-Channels

Modern processors typically have a multi-level cache hierarchy with core-private L1 and L2 caches and the L3 or Last-Level-Cache (LLC) shared among multiple cores. Consequently, a covert-channel attack can be *cross-core* [23], where the sender and receiver are in two separate processes on different cores, or *same-core*, where they execute on the same physical core from within two SMT threads [20] or from within different trust-domains in the same process [18, 21]. While our attack is applicable to both attack models, for simplicity, we assume the cross-core attack model that is commonly applicable to a virtualized setting where resource-allocation typically occurs per core, as our default.

2.2 State-of-the-Art Cache Covert-Channels

State-of-the-art attacks operate the sender and receiver in a synchronous manner, where they communicate each bit within a synchronized epoch (corresponding to a bit-period) and wait till the epoch ends before communicating the next bit. Within each bit-period, the sender and receiver execute multiple operations to first encode a bit, then decode a bit, and finally reset the channel to be ready to communicate the next bit.

2.2.1 Cross-Core Flush+Reload Attack [40]. This attack (shown in Figure 2(a)) transmits information using the timing difference between an LLC-hit and an LLC-miss for a load to a shared address. The sender encodes each bit by executing/not-executing a load to the address to convey a bit-0/bit-1. The receiver decodes each bit, by executing a load to the same address and measuring its latency using `rdtscp` before and after the load, decoding bit-0/bit-1 based on whether it observes a cache-hit/cache-miss. Subsequently, to reset the channel, the receiver issues a `clflush` to evict the address from the cache. The sender and receiver both wait till the end of a bit-period to ensure the other has finished its operations, typically synchronizing using `rdtscp` that provides both a shared notion of time, before communicating the next bit using the same address. Gruss et al. [13] showed that this attack can achieve a bit-rate of 298 KB/s at less than 0.05% error-rate.

2.2.2 *Cross-Core Flush+Flush Attack [13].*

This attack is the current fastest cross-core attack and exploits the difference in execution time for a `clflush`, based on whether an address is cached or not. As shown in Figure 2(b), the sender's operations are identical to the Flush+Reload attack, i.e. executing/not-executing a load for encoding bit-0/bit-1. To decode a bit, the receiver executes and measures the latency of a `clflush` to the same address: a faster execution implies that the address was accessed by the sender and cached, and hence a bit-0 transmission, whereas a slower execution implies bit-1. Note that this attack does not require a separate operation to reset the channel, as the `clflush` in the receiver implicitly evicts the address from the cache, allowing the sender to transmit the next bit once the current epoch ends. As it requires one less operation than Flush+Reload, this attack achieves a higher bit-rate of 496 KB/s; but it has a higher error-rate of 0.84%, as `clflush` has a smaller timing difference (~10 cycles) compared to that of LLC-hits and misses (~200 cycles).

2.2.3 *Same-Core Take-a-Way Attack [20].*

This attack is the fastest same-core attack and exploits timing-differences arising from the way-prediction technique used for fast L1-cache accesses in AMD processors. AMD's way-predictors are vulnerable to address conflicts, where accessing two addresses that map to the same way-predictor entry results in evictions of each other from the way-predictor entry and also the L1-cache. Take-a-way proposed a synchronous covert-channel attack exploiting this, as shown in Figure 2(c). Every bit-period, the receiver issues a load to prime a predictor-entry, then allows the sender to execute, and later reloads the same address. To transmit bit-0, the sender executes a load to a conflicting-address that evicts the receiver address, causing the receiver a cache-miss. For bit-1, the sender skips the load, causing the receiver a cache-hit. The sender and receiver then wait for the bit-period to end, before resuming transmission with the same pair of addresses. Take-a-way uses this protocol to launch up to 80 parallel synchronous channels and achieve a bit-rate of 588 KB/s.

2.3 Pitfalls of Existing Attacks

2.3.1 *Synchronous Communication.*

State-of-the-art covert-channel attacks require a synchronous transfer of bits. The operations to reset-bit, encode-bit, decode-bit must be executed in a single synchronous window shared between the sender and the receiver, for each bit before moving on to transmit the next bit. As a result, the size of the synchronous window has to be sufficiently large to accommodate all three operations. Any attempt to decrease the bit-period results in loss of synchronization, and breakdown of channel communication.

2.3.2 *ISA or Micro-architecture Specific Requirements.*

All existing fast covert-channel attacks suffer from limited applicability. For example, Flush-based attacks like Flush+Reload and Flush+Flush, require the usage of a cacheline flush instruction for transmission of each bit. While the x86 ISA supports unprivileged usage of `clflush` instruction, the ARMv8 ISA disables such unprivileged by default. ARMv7 and below do not even support such cacheline flush instructions, making such attacks infeasible on several mobile processors [12]. On the other hand, attacks like Take-a-way exploit features like L1-cache way-prediction only in AMD processors, making such attacks infeasible on other micro-architectures. Finally, although attacks like Prime+Probe [23] that exploit the generic set-associative structure of caches are widely applicable (as they do not require flushes or shared-memory), they are considerably slower. For example, Liu et al. [23] achieve a bit-rate of 75 KB/s, that is 7x slower than the fastest known flush-based attacks.

2.4 Goal: A Fast and Universal Attack

Our goal is to investigate whether bit-rate of cache covert-channels can be significantly improved. To be faster than state-of-the-art, an attack should not require the synchronous operation of the sender and receiver while encoding and decoding bits. At the same time, we investigate if such an attack may be universally applicable to processors of all architectures and micro-architectures; the only requirements for such an attack must then be the existence of shared memory and timing difference between fast shared-cache accesses and slow memory accesses. Such an attack operating without cacheline flushes could also highlight the vulnerability of defenses such as SHARP [37] (that rely on disabling the use of flush instructions) and inform the design of future defenses. To that end, we design Streamline as a fast, flush-less and asynchronous covert-channel.

3 STREAMLINE DESIGN

We first intuitively describe Streamline, then the challenges in enabling it with low error-rates, and how we overcome them.

3.1 High-Level Idea of Streamline

Algorithm: Streamline achieves fast asynchronous communication by transmitting each bit on a different address of a large shared-array. As shown in Figure 3, for each bit transmission, the sender chooses a successive entry (cache line) of the shared array, and installs the entry into the LLC if the bit is 0, else it skips that entry. Then, without waiting for the receiver to access that entry, the sender moves on to the next bit. The receiver follows behind in the same sequence loading each successive entry. If a load is an LLC-Hit, that implies the sender installed that entry into the LLC, and hence the corresponding bit was 0; else if the load is a DRAM-access, that implies the sender skipped the entry, and the bit was 1.

When the sender wraps around to the beginning of the array, the addresses used in the previous iteration must be evicted from the LLC before they can be reused. Unlike prior attacks that explicitly use conflicts [20, 23] or flush instructions [13, 40] to evict addresses, the sequential access pattern of Streamline on the large array (larger than LLC capacity) implicitly induces cache-thrashing, where the LLC automatically evicts previously accessed addresses to accommodate new addresses.

Potential Bit-rate: Streamline does not require any extra operations, such as flushes, to evict previously used addresses because of the cache-thrashing pattern of its accesses.[2] Moreover, for each bit, the sender or receiver does not have to wait for the epoch to complete, but can continue to the next bit. As long as the receiver

[2]Thrash+Reload attack [29] also uses cache-thrashing to evict addresses, but it is a synchronous attack that waits till thrashing evicts an address before transmitting the next bit with the same address. Hence it has a bandwidth of only 4 bits/minute, which is more than a million times slower than Streamline.

Sender

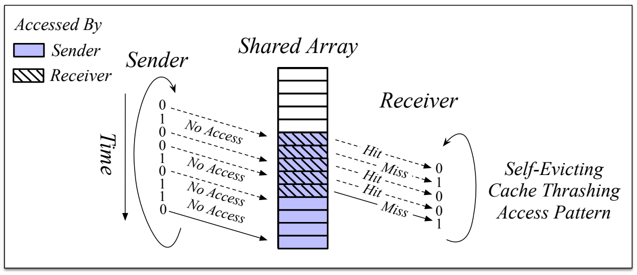
```
uint8_t arr[len]

foreach(bit_i){
 if(bit_i== 0)
  load(&arr[64*(i++)%len])
}
```

Receiver

```
uint8_t arr[len]
sleep(delayed_start)
foreach(bit_i){
 t = rdtscp
 load(&arr[64*(i++)%len])
 T = rdtscp - t
 bit_i = T<thresh?0:1
}
```

(a) Streamline Attack Algorithm



(b) Access Pattern of Sender and Receiver to Shared Array

Figure 3: Overview of the Streamline Attack. The sender and receiver communicate asynchronously via accesses to a shared array `arr` (larger than the LLC). The sender keeps transmitting on sequential entries of the array, without waiting for the receiver to decode. By the time the sequential access wraps around to the start of the array, the entries accessed in the previous iteration are evicted from the LLC due to the cache-thrashing access pattern.

follows behind the sender in a rate-matched manner, the bit-rate is only limited by the receiver's throughput in executing and measuring loads. For example, on an Intel Skylake system where a DRAM access takes ~300 cycles, this channel can potentially exceed a bit-rate of 1.5 MB/s.

Challenges: Ensuring that the asynchronous communication is also error-free is challenging for the following reasons:

(1) **We need to ensure the sender is consistently ahead of the receiver:** If the sender falls behind the receiver at any time, the receiver continuously observes spurious LLC-misses, and erroneously decodes bits as all-1s. We need to ensure the receiver is slower and always follows the sender.

(2) **We need to be able to tolerate slack between the sender and the receiver:** If the portion of the cache over which communication is happening is too small, the sender can self-evict addresses it previously installed in the LLC via cache-thrashing, before the receiver can access them, causing bits to be decoded erroneously. To prevent premature eviction, we need to ensure the addresses installed by the sender spreads over the entire LLC, and are protected from the effects of the replacement policy, and the prefetcher.

(3) **We need to prevent the sender from going too far ahead of the receiver:** Since the cache (that acts as a buffer for our communication) is of a limited size, we need to prevent the sender from wrapping around and lapping the receiver. To that end, we need coarse-grain synchronization (*e.g.*, once in hundreds of thousands of bits) to prevent the sender-receiver gap from growing beyond tolerable limits.

Next, we describe how we address each of these challenges.

3.2 Channel Encoding For Sender-Rate > Receiver-Rate

Figure 4 shows how a naive algorithm for transmission (sender issues load for payload-bit 0, and skips the load for payload-bit 1) can result in burst-errors, due to a payload-dependent rate-mismatch between the sender and receiver. If the payload bits are mostly 0s, the sender can slow down due to slow DRAM accesses and fall behind the receiver. This can cause the receiver to erroneously decode all-1s, as it gets LLC-Misses for addresses not accessed yet by the sender. If the payload is mostly 1s, then the sender can skip several addresses and end up considerably ahead of the receiver. In this scenario, the addresses installed by the sender can get evicted even before the receiver can access them, causing channel breakdown.

Payload	*Sender*	*Receiver*
0	LLC Miss	LLC Hit
1	-	LLC Miss
0	LLC Miss	LLC Hit
0	LLC Miss	LLC Hit
0	LLC Miss	LLC Hit
0	LLC Miss	LLC Hit
0	*Receiver goes ahead of Sender (causing errors)*	LLC Miss
1		LLC Miss
0		LLC Miss

(a) Payload with Many-0s
(Receiver goes beyond Sender)

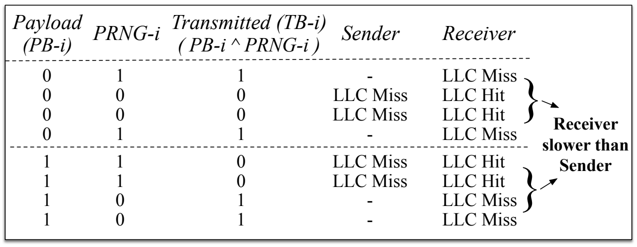
Payload	*Sender*	*Receiver*
1	-	LLC Miss
1	-	LLC Miss
1	-	LLC Miss
Many 1s	*Skipped*	*Misses prematurely evict sender addresses*
1	-	LLC Miss
0	LLC Miss	LLC Miss
0	LLC Miss	LLC Miss

(a) Payload with Many-1s
(Receiver falls far behind Sender)

Figure 4: A naive channel encoding scheme causes a rate-mismatch between the sender and receiver. If the receiver goes ahead of the sender or falls too far behind, it observes erroneous LLC-Misses (in red), leading to errors.

To keep the sender and receiver rates payload-independent, we use a pseudo-random channel encoding. As shown in Figure 5, the sender uses a pseudo-random number generator (PRNG) whose seed is known to both sender and receiver, to modulate payload bits with a sequence of random 0s and 1s. For each payload bit (PB-i), the sender transmits a bit (TB-i) as TB-i = PB-i $\oplus$ PRNG-i. On receiving TB-i, the receiver is able to reconstruct the payload (PB-i) as PB-i = TB-i $\oplus$ PRNG-i, as the PRNG seed is known to it.

The PRNG-based channel encoding equalizes the number of 0s and 1s transmitted (TB-i) in expectation irrespective of the actual payload-bit values (PB-i), as long as the PB-i and PRNG-values are drawn from independent distributions. In this scenario, the sender and the receiver have a comparable number of DRAM accesses, as the sender has a DRAM-access when TB-i is 0, while the receiver has a DRAM-access when TB-i is 1. However, the receiver additionally incurs LLC-Hits when TB-i is 0, which makes the receiver execute at a slower rate than the sender. This ensures that the receiver always follows behind the sender, and the gap between them grows at a deterministic rate.



Payload (PB-i)	PRNG-i	Transmitted (TB-i) (PB-i ^ PRNG-i)	Sender	Receiver
0	1	1	-	LLC Miss
0	0	0	LLC Miss	LLC Hit
0	0	0	LLC Miss	LLC Hit
0	1	1	-	LLC Miss
1	1	0	LLC Miss	LLC Hit
1	1	0	LLC Miss	LLC Hit
1	0	1	-	LLC Miss
1	0	1	-	LLC Miss

Receiver slower than Sender

Figure 5: Modulating payload bits (PB-*i*) with a random-sequence (PRNG-*i*) and then transmitting (TB-*i*) ensures the Receiver is slower than Sender (with equal LLC-misses, but more LLC-Accesses), regardless of payload values.

3.3 Access-Pattern to Tolerate Sender-Receiver Slack

As the sender transmits at a faster rate than the receiver, the gap between the address being accessed by the sender and that being accessed by the receiver, at each moment in time, keeps widening. As the number of addresses that an LLC can store is limited to a finite value, the gap between the sender and the receiver can theoretically increase up to this limit while maintaining a low error-rate. To tolerate a sender-receiver gap close to this theoretical limit, we design the sequence of addresses used by Streamline ensuring:

(1) The sequence maps to a large majority of LLC-sets. Additionally, the sequence should not be predictable by the cache prefetcher, which can disrupt the channel by prefetching addresses into the LLC, irrespective of the payload.

(2) The sequence uses all the ways within a particular LLC-set, and fools the LLC replacement policy. This is essential to ensure the addresses installed by the sender are not prematurely evicted because of replacement decisions.

3.3.1 Achieving High Set Coverage and Fooling Prefetcher. Simple sequences such as accessing sequentially contiguous cachelines have high set-coverage, but are easily predicted by the prefetcher (Intel CPUs have a next-line prefetcher, sequential stream prefetchers, and stride prefetcher [31, 33]) and the channel can be disrupted. Other sequences used in prior works [13], that access one cacheline per 4KB page to fool the prefetcher, have very poor cache set coverage. To identify an optimal access pattern with high set coverage that also fools the prefetcher, we devise the following experiment. We systematically generate sequences of N addresses that access every x-th cacheline in a page and lines from y pages are accessed before the next line from the same page, and measure the latency of each access in the sequence. We repeat this experiment 5 times for $N = 1000$ and $x, y = \{1, 2, 3, 4, 5\}$, and report the miss-rate, i.e. the number of cache-misses observed out of N accesses for each sequence. A higher cache miss rate for a sequence indicates that it is more effective in fooling the prefetcher.

As shown in Table 1, the prefetcher effectively learns access patterns (and lowers miss-rates) if accesses are strided within a single page ($y = 1$) for any stride ($x \geq 1$), or if the accesses are sequential ($x = 1$) irrespective of the number of pages ($y \geq 1$) across which lines are accessed before the same page is re-accessed. However a strided access pattern ($x > 2$) that is spread across more than one page ($y \geq 2$) is highly effective in fooling the prefetchers, with the miss-rate for such a sequence being $> 90\%$. We believe this is because the stride-tracking mechanism operates at page-granularity (as prefetched addresses do not cross page boundaries) and is overwhelmed by the back-to-back accesses across pages.

Table 1: LLC Miss-Rate for a Sequence accessing every xth cacheline within a page, with y pages accessed at a time. (Higher miss-rate implies sequence fools prefetcher better)

x \ y	1	2	3	4	5
1	1.8%	3.7%	2.7 %	2.5%	2.2%
2	6.6%	7.3%	6.7%	6.9%	7.0%
3	11.6%	99.5%	98.9%	90.9%	88.0%
4	15.3%	97.5%	97.8%	95.7%	90.5%
5	17.3%	98.8%	91.8%	91.6%	90.6%

For Streamline, we pick the access pattern that best fools the prefetchers, *i.e.* a stride of 3 spread over 2 pages ($x = 3, y = 2$). This pattern covers 1/3rd of the LLC sets as it accesses every 3rd line within a page, which is significantly better than accessing one cacheline per 4KB page (as in prior-work [13]). We also empirically observe that sequences that start from the middle of a 4KB page are better at fooling the prefetcher stride-tracking (e.g. 14th cacheline). The exact equation for calculating the index of the shared byte-array for each bit (i) of the payload is given by Equations 1–3.

$$Pg\text{-}num = 2 * int(3 * i/128) \quad + \quad i\%2 \tag{1}$$

$$Cl\text{-}num = (14 + 3 * int(i/2)) \ \% \ 64 \tag{2}$$

$$array\text{-}index = (Pg\text{-}num * 4096 + Cl\text{-}num * 64) \ \% \ arr\text{-}sz \tag{3}$$

3.3.2 Covering LLC-Ways by Fooling Replacement Policy. To ensure the addresses accessed in Streamline occupy a majority of the LLC-ways, and are protected from premature eviction from a set, we need to fool the LLC replacement policy. Prior work [4] reverse engineered the LLC replacement policy in Intel CPUs, showing it to maintain 2-bit age values per cacheline for tracking re-use (similar to re-use bits in RRIP Replacement [14]). A new line is assigned an age-value of 2 or 3 (based on CPU generation), and subsequent LLC-hits to such lines decrements their ages, till they saturate at 0. A line with age-3 within a set is evicted, when a new line is to be installed to the set; if no such line exists, all the ages in the set are incremented till an age-3 line is found.

To fool the replacement policy and avoid premature eviction of addresses installed by Streamline, we need to prevent them from being the oldest line in the set. To that end, we engineer extra LLC hits to the addresses installed by the sender into the LLC, to decrement their age and protect them from preemptive eviction. We achieve this by making the sender re-access previously installed addresses, after a lag of 5000 bits, to ensure this trailing access results in an LLC hit (the addresses are likely to be evicted from the L1 and L2 cache within 5000 bits).

Figure 6 shows the bit-error-rate for Streamline as the sender-receiver gap (in number of bits) increases, for different access patterns: a naive sequence accessing one cacheline per page, a sequence with high LLC set-coverage from Section 3.3.1, and a sequence with high coverage of LLC sets and ways from Section 3.3.2. The naive sequence shows an increase in error-rate beyond a sender-receiver gap of 1000 bits, the sequence with high LLC set-coverage beyond 4000 bits, and the sequence with high coverage of LLC sets and ways retains low error-rates till a gap of 40,000 bits between sender and receiver. With this, Streamline tolerates a sender-receiver gap of 1/3rd the LLC capacity (128,000 addresses).

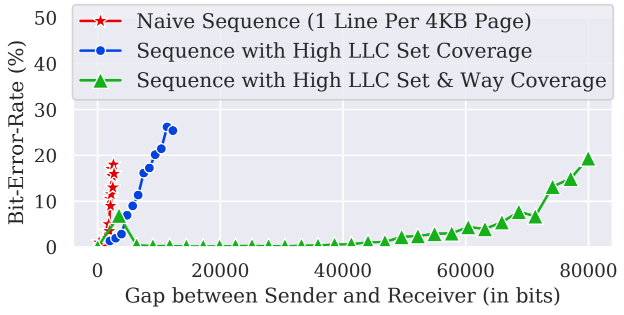


Figure 6: Error-rate versus Sender-Receiver Gap. With a sequence of addresses that covers a majority of LLC sets and ways, Streamline builds in considerable tolerance to slack between the sender and receiver.

3.4 Techniques to Bound Sender-Receiver Slack

In an ideal scenario where the sender and receiver are perfectly rate-matched, the gap between the sender and receiver would remain constant, and never go beyond the tolerable slack. To that end, we attempt to rate-match the sender and receiver as best as possible and then explicitly bound the maximum slack between the sender and receiver, to ensure the sender never wraps-around the array and laps the receiver.

3.4.1 *Matching Sender and Receiver Load Execution-Rates.*

One of the key reasons for the rate mismatch between the sender and the receiver is that the receiver measures the latency of the loads it executes while the sender does not. To measure the load latency, the receiver uses `rdtscp` instructions in the sequence `rdtscp; load; rdtscp;`. Such usage of `rdtscp` serializes the execution of loads of different bits for the receiver, whereas the sender (without such serializing instructions) can issue loads of multiple bits in parallel. As a result, despite issuing two loads per bit (one for transmission and one for fooling the replacement policy), the sender can execute at a faster rate than the receiver. Hence, to throttle the sender, we add a load-serializing `rdtscp` per bit in the sender that limits its load execution-rate and reduces its mismatch with the receiver.

3.4.2 *Periodic Coarse-Grain Synchronization.*

In a realistic setting, the sender and the receiver will always have a non-zero drift. To prevent the sender-receiver gap from exceeding a tolerable limit due to this, we synchronize the sender and receiver at a coarse granularity (e.g. every 200,000 bits) using a separate low-bandwidth covert channel. When the sender reaches the end of an epoch of 200,000 bits, it stops transmitting and waits. Once the receiver completes 195,000 bits of the epoch, it communicates a bit on the synchronization-channel to the sender, to permit the sender to resume. As synchronization is extremely infrequent (e.g. once in 200,000 bits), any low-bandwidth covert-channel can be used without any bandwidth loss (we use a Flush+Reload channel for synchronization).

To justify our choice of synchronizing every 200,000 bits, Figure 7 shows the sender-receiver gap (in bits) as the bits transmitted increases. We compare three access patterns: (a) using the tailored access pattern from the previous section that covers a majority of the cache, (b) the tailored access pattern with the addition of a rate-limiting `rdtscp` for the sender, and (c) the further addition of halting the sender periodically based on synchronization every 200,000 bits. With the first access pattern, the sender-receiver gap crosses a threshold of 40,000 bits (beyond which error-rate goes above 1%) within a transmission of 100,000 bits. With the second access pattern that rate-limits the sender, the sender-receiver gap remains within the threshold till 400,000 bits. To ensure that the sender-receiver gap is below the threshold (*i.e.,* the channel operates within the threshold of 1% error-rate) indefinitely while also keeping some head-room, we add synchronization between the sender and receiver every 200,000 bits transmitted (*i.e.* the third access pattern). With this, Streamline can maintain the channel error rate below 1% for billions of bits transmitted.

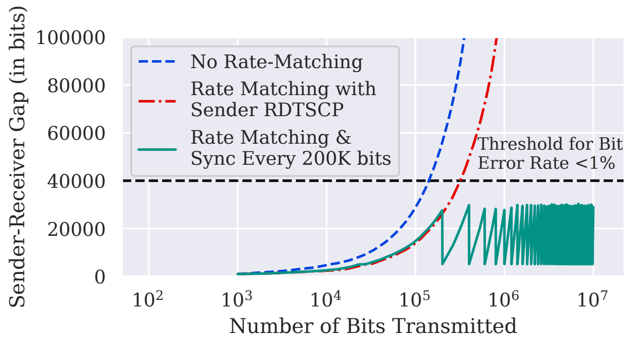


Figure 7: Gap between Sender and Receiver vs Number of bits transmitted. Rate-limiting the sender to match its rate with the receiver and using coarse-grain synchronization to halt the sender every 200,000 bits, ensures the gap is maintained below 40,000 bits (within this threshold, the error-rate stays below 1%).

3.5 Overall Algorithm for Streamline

Figure 8 shows the algorithm for Streamline incorporating the error-mitigating techniques from Sections 3.3 and 3.4. The modulation of the payload with the PRNG-sequence (Section 3.2) happens off the critical path and hence is not shown.

Sender

```
foreach(bit_tx[i]){

 //to rate-limit
 rdtscp

 //to transmit bit
 if(bit_tx[i] == 0)
  load(&arr[index(i)])

 //to beat real-policy
 if(bit_tx[i-5000] == 0)
  load(&arr[index(i-5000)])

 //synchronize every 200K
 if(i%200000 == 199999)
   FR_Sync_With_Receiver()
}
```

Receiver

```
sleep(delayed_start)

foreach(bit_rx[i]){

 //to receive bit
 t = rdtscp
 load(&arr[index(i)])
 T = rdtscp - t
 bit_rx[i]=T<thresh?0:1

 //synchronize every 200K
 if(i%200000 == 195000)
  FR_Sync_With_Sender()
}
```

Figure 8: Algorithm for Sender and Receiver in Streamline to achieve fast and asynchronous communication, incorporating techniques to ensure low error-rates.

4 RESULTS

In this section, we evaluate the covert-channel transmission bit-rate and bit-error-rate that Streamline achieves.

4.1 Methodology

We run our experiments on a 4-core Intel Skylake CPU (Intel Xeon E3-1270), with an 8MB LLC, running at a frequency of 3.9 GHz (the results were also successfully reproduced on Intel Core i7-8700K Kaby Lake and Core i5-9400 Coffee Lake CPUs). For our system, we measure the average LLC-Hit latency to be 95 cycles, and LLC-Miss latency to be 285 cycles. So, we use a threshold of 180 cycles for the receiver to determine if a load is an LLC-Hit or Miss. We use large pages for the sender and receiver, to minimize any effects due to TLB misses. We pin the sender and receiver processes to two different cores, to ensure all communication is through the LLC. We assume the sender and receiver share an array (default size of 64 MB) with read-only permissions that Streamline uses for communication; we analyze other array sizes in Section 4.4.

4.2 Streamline Channel Bit Rate and Error Rate

Figure 9 shows the bit-rate in KB/s and bit-error-rate for Streamline, plotted against the size of the payload that is transmitted (in bits). As a high-speed covert-channel typically has more utility at large payload sizes, we evaluate Streamline for payload sizes of 200,000 bits to 1 billion bits. We report the bit-rate by measuring time from receiver-start to end, divided by the number of bits transmitted, averaged over 5 runs. Streamline achieves a steady-state bit-rate of 1801 KB/s at a bit-error-rate of 0.37%. This corresponds to a steady-state bit-period of 265 CPU cycles, that is in between the latency for an LLC Hit and a DRAM access on our system.

Streamline's bandwidth is limited by how fast the receiver executes and measures a load for each bit, and is not limited by synchronization as in prior works. Although DDR4-DRAM has a bandwidth of 150 Million accesses/second, Streamline is still limited to a bandwidth of <2 MB/s, due to serialization of loads by the `rdtscp` used to measure load-latency at the receiver. As the receiver cannot execute multiple loads in parallel, it is forced to wait till each load completes (incurring raw LLC-Hit/ DRAM-access latency per bit-period), before issuing the next load. We discuss how this bandwidth limitation is fundamental to all future attacks in Section 4.6.

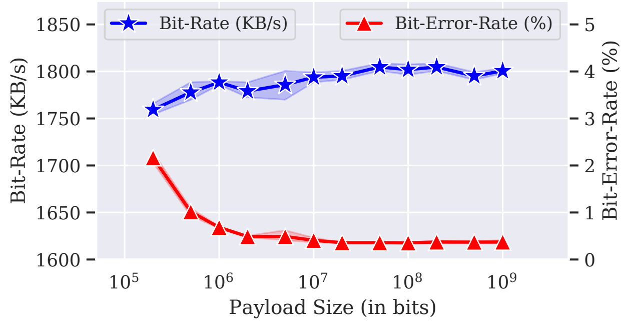


Figure 9: Covert-channel Bit-rate and Bit-error-rate vs Payload Size (shaded regions represent 95% CI, *i.e.* confidence intervals). Streamline has bit-rate of 1801 KB/s at error-rate of 0.37% (note the non-zero start of the Y-Axis for bit-rate).

In Figure 9, the bit-error-rate for a payload size of 200,000 bits is ~2% and relatively higher than the stable bit-error-rate of 0.37% for larger payload sizes. This is because of high error-rates incurred during the first 5000 bits, when the trailing accesses to fool the replacement policy have not started (during the first 5000 bits, the error-rate goes up to 20% as shown in Figure 6). However, this transient increase in error-rate is amortized as payload size increases, becoming imperceptible beyond 5 million bits, and the error-rate stabilizes at 0.3%.

4.3 Analysis of Errors and Error-Correction

Table 2 shows the breakdown of error-rates by type – 0 to 1 bit errors and 1 to 0 bit errors, for different payload-sizes. As payload size increases, the 1 to 0 errors (which form a significant fraction of the total errors for smaller payloads) drop considerably. At the same time, 0 to 1 errors stay the same, and become a dominant fraction for larger payloads.

0 to 1 bit errors typically manifest when an address accessed by the sender gets evicted from the LLC before the receiver can access it, because the gap between receiver and sender grew too large or because of cache usage by other system processes. In either scenario, we observe these errors appear in bursts that are hard to correct without re-transmission. On the other hand, we observe that 1 to 0 errors occur when a DRAM access is faster than our LLC-hit threshold, resulting in a false declaration of an LLC-Hit. We expect these errors to be randomly distributed as these accesses form the tail of the DRAM latency distribution, with a high chance of them being single-bit errors. Hence, we develop an error-correction scheme for Streamline that corrects single-bit errors.

To add error-correction, we break our payload into 8-byte packets and append each packet with a (72,64) Hamming Code before transmission, that can correct 1-bit errors and detect 2-bit errors occurring during transmission. We pick this specific design with a

Table 2: Breakup of Error-Rates for Different Payload Sizes

Payload Size (in bits)	2×10^5	10^6	10^7	10^8	10^9
Total Bit-Error-Rate	2.16%	0.68%	0.41%	0.35%	0.37%
Rate of 1 to 0 Errors	1.95%	0.44%	0.12%	0.09%	0.11%
Rate of 0 to 1 Errors	0.22%	0.25%	0.29%	0.26%	0.27%

12.5% overhead (1-byte code per 8-byte data) to maintain equivalence with the error-mitigation framework developed by Gruss et al. [13] (a more robust mechanism including re-transmission) that incurs a similar 12% overhead for the Flush+Flush attack.

Table 3 shows the bit-rate and bit-error-rate with and without our error-correction scheme for transmitting 1 billion bits. With error-correction, the effective data bit-rate goes down by approximately 10% to 1598 KB/s, while the bit-error-rate goes down to 0.12%. The remaining bit-errors are due to uncorrected or miscorrected bits, because of the presence of multi-bit errors within a packet. We observe a similar 0.2% drop in error-rate for all payload sizes.

Table 3: Streamline with and without Error-Correction (parenthesis includes margin-of-error for CI=95%)

Configuration	Bit-rate	Bit-Error-Rate
Without Error-Correction	1801 KB/s (± 3)	0.37% (± 0.04%)
With (72,64) Hamming Code	1598 KB/s (± 2)	0.12% (± 0.01%)

4.4 Sensitivity to Shared Array Size

Table 4 shows the error-rate of Streamline with a payload of 100 million bits, as the size of the shared array used for covert communication varies. As our system has an 8MB LLC, we evaluate shared array sizes of 8MB to 64MB (1x to 8x the LLC size). As the size decreases from the default value of 64MB to 32MB, the error-rate stays close to 0.3%. However, it increases considerably below 32MB, reaching 3.2% for 16MB and 28% for 8MB. Having a sufficiently large shared array is critical for the robustness of the channel, as Streamline relies on the cache-thrashing behavior of its accesses to evict addresses previously used for communication. Without evictions via cache-thrashing, Streamline cannot effectively reuse array addresses on a wrap-around, resulting in high error rates. Streamline needs an array that is at least 3x the size of the LLC for inducing effective cache-thrashing, as its access pattern only loads every 3rd cache line of the array to fool the prefetcher. Hence, Streamline incurs higher error rates with array-sizes that are 2x (16MB) and 1x (8MB) the LLC-size.

Table 4: Streamline with Different Shared Array Sizes (parenthesis includes margin-of-error for CI=95%)

Shared Array Size	Bit-Error-Rate
64MB (default)	0.35% (± 0.02%)
32MB	0.33% ± 0.01%
16MB	3.2% (± 0.1%)
8MB	27.5% (± 0.1%)

4.5 Sensitivity to Synchronization Period

Table 5 shows channel characteristics for a payload of 100 million bits, as the synchronization period is varied. Across all periods, the bit-rate remains above 1780 KB/s because the synchronization overhead is negligible. However, the bit-error-rate increases to 0.7% if the synchronization-frequency decreases once every 500,000 bits, as the sender-receiver gap exceeds tolerable limits (going beyond 500,000 bits leads to channel breakdown). On the other hand, increasing synchronization frequency to once every 25,000 results in minor differences in error-rates (increase by 0.1% versus our default of once every 200,000 bits), but we observe these tend to be single-bit errors that are easily correctable.

Table 5: Streamline with Different Synchronization Periods (parenthesis includes margin-of-error for CI=95%)

Synchronization Period	Bit-rate	Bit-Error-Rate
Every 500,000 bits	1818 KB/s (± 5)	0.65% (± 0.05%)
Every 200,000 bits (default)	1802 KB/s (± 7)	0.35% (± 0.02%)
Every 100,000 bits	1797 KB/s (± 6)	0.37% (± 0.03%)
Every 50,000 bits	1783 KB/s (± 10)	0.40% (± 0.02%)
Every 25,000 bits	1791 KB/s (± 5)	0.46% (± 0.01%)

4.6 Limiter for Covert-Channel Bit-rate

Streamline mitigates two key bottlenecks faced by prior covert-channels: (a) the transmission bottleneck (the requirement of loading and resetting an address with separate operations every bit), and (b) the synchronization bottleneck (sender waits until the receiver has decoded a bit before transmitting the next bit). Thus, we improve the covert-channel bit-rate by >3x compared to state-of-the-art. Our work, however, exposes a new bottleneck for covert-channels – *the measurement bottleneck.*

To see the problem, consider that the bit-rate in Streamline is determined by the rate at which the receiver can execute and measure loads. All existing methods to measure load latency on x86 systems result in load serialization for accurate latency measurement. For example, we use the sequence `rdtscp; load; rdtscp;` (other works use `rdtsc;lfence;` instead of `rdtscp`). Fundamentally, these sequences need to ensure that the second timer instruction samples the time after the load returns (which they do with either an explicit `lfence` or fence-like semantics as in `rdtscp`). This fencing implies that the next load cannot execute until the previous one returns. This loss of parallelism means that each bit-period is limited by the load latency to access DRAM or LLC (between 100 – 300 cycles). Thus, Streamline is limited to a bit-period of 267 cycles, as is any future attack requiring timing loads.

Note that using gadgets like sibling counting-threads [19, 30], for measuring time without `rdtscp`, is not viable for measuring the latency of multiple loads executing in parallel in x86. Executing and measuring loads in parallel with such timer-variables leads to re-ordering of timer-loads and potential TSO violations on Intel CPUs [28] – these cause speculative timer-loads to be squashed, resulting in incorrect latency measurements. We leave the study of new methods to measure load-latency of multiple loads in parallel, to reach higher bit-rates, for future work.

4.7 Resilience to System Noise

In Streamline, the sender buffers bits for the receiver in LLC locations to enable asynchronous communication. However, considerable cache-activity from co-running processes on the system can cause lines installed by the sender to get evicted before the receiver accesses them, adding noise to the channel. However, Streamline can achieve noise-resilience by limiting the time-window where cache lines installed by the sender are vulnerable to eviction (*i.e.* the time window for which the line is installed by the sender but not yet accessed by the receiver). This can be achieved by reducing the maximum sender-receiver gap, by using a smaller synchronization period for the sender and receiver.

Our default implementation uses a synchronization period of 200,000 bits that limits the sender-receiver gap to a maximum of 40,000 bits. Reducing the synchronization-period to once every 50,000 bits limits the maximum sender-receiver gap to 8,000 bits (as shown in Fig 7) without affecting the bit-rate (as shown in Table 5). In this case, as the buffer size used at any given time is no more than 8000 bits, i.e. 6% of our 8MB LLC (131,000 lines), the potential for interference from co-running processes is significantly diminished.

To evaluate the attack fidelity under noise, we evaluate Streamline while running applications stressing the CPU caches simultaneously on a different core. We use `stress-ng` [16] (configurable kernels that stress system resources) and run applications from `"-class cpu-cache"` that stress the CPU caches.

Figure 10 shows the error-rate of Streamline transmitting a payload of 100 million bits (averaged over 5 runs) using synchronization periods of 200,000 and 50,000 bits, while each application from `stress-ng` is co-running (pinned to an adjacent core). While the error-rate of the channel reaches a worst-case of 15% with the sync-period of 200,000 bits, it is limited to a worst-case of 0.8% when the sync-period is reduced to 50,000 bits and relatively noise-resilient (close to the error-rate of 0.3% under noise-free setting). On the other hand, Streamline's bit-rate with co-running `stress-ng` applications is slightly diminished and varies from 1500-1800 KB/s due to increased memory latency and queuing delays.

We also tested Streamline while simultaneously running the Chromium-60 web-browser streaming Youtube videos and found error-rate to be 0.5-0.6% (no impact on bit-rate). Note that infrequent spurious noise evicting line at such low rates can be mitigated using error correction codes (discussed in Section 4.3).

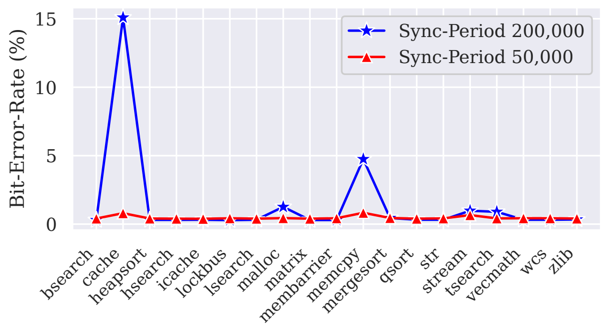


Figure 10: Error-Rate of Streamline under co-running `stress-ng` workloads, for sender-receiver synchronization periods of 200,000 and 50,000 bits.

5 COMPARISON WITH PRIOR WORK

In this section, we compare Streamline with an implementation of Flush+Reload [40] covert-channel on our system and also distinguish our work from other previously proposed covert-channels.

5.1 Comparing Flush+Reload and Streamline

Figure 11 shows the bit error rate versus bit rate for the Flush + Reload [40] covert channel (averaged over 5 runs), generated on our system using the code from the Arch-Sec tutorial at ISCA-2019 [9] (note that this is not a fully optimized implementation, so the trend is more representative than the actual values). To obtain the error-rates at different bit-rates for Flush+Reload, we reduce the transmission window per bit (time for which the sender and receiver perform accesses to communicate a bit) from 32,768 cycles to 256 cycles, while artificially ensuring that the synchronization related errors remain negligible, so that the errors are only due to transmission (note that the error-rate obtained is a lower-bound). We observe the error-rate stays low (below 1%) until 200 KB/s (bit-period of 2000 cycles), but beyond 200 KB/s the error-rate considerably increases beyond 10%. This is because Flush+Reload requires multiple operations (loads for transmission, flush for reset) to be executed within progressively shorter bit-periods. In comparison, Streamline achieves an error-rate of 0.3% with a bit-period of 265 cycles as it only requires a single operation per bit for transmission and also does not require synchronization for every bit.

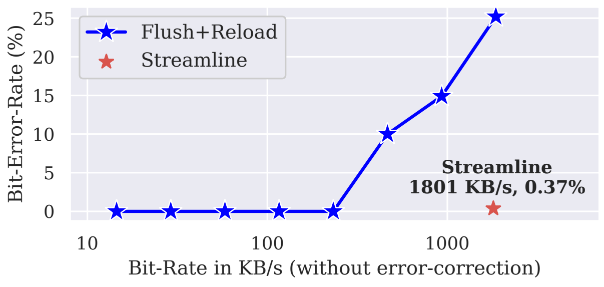


Figure 11: Bit-rate and bit-error-rate (without error-correction) of Flush+Reload attack versus Streamline.

5.2 Comparison with other Covert-Channels

Table 6 shows bit-rates and error-rates of cache covert-channel attacks in recent works, compared with Streamline. As state-of-the-art bit-rates for these attacks are difficult to achieve without hand-tuned assembly implementations (not publicly available to our knowledge), we present results reported in the respective papers.

Flush+Flush and **Flush+Reload** implementations by Gruss et al. [13] were the prior fastest cross-core covert channels, with bit-rates of 298 KB/s and 496 KB/s. Streamline achieves 3.6x and 6x higher bit-rates at comparable error-rate as it (a) is asynchronous (the sender does not wait for the receiver each bit), and (b) only requires a load per bit (no flushes). Note that the hardware used in Streamline (Intel Xeon Skylake) only has a ~15% speedup compared to that used in these prior attacks (Intel i7 Haswell) [7] and Streamline's bit-rate increase of 3.6x far outweighs any potential hardware speedup. In fact, Streamline is likely to have similar bit-rates across recent processor generations as its bit-period is bottlenecked by

Table 6: Prior Cache Covert Channels (Bit-Rate>50 KB/s)

Attack	Attack Model	Bit-Rate	Bit Error Rate
Take-a-way [20]	Same-Core	588 KB/s	1–3%
Flush+Flush [13]	Cross-Core	496 KB/s	0.84%
Prime+Probe (L1) [26]	Same-Core	400 KB/s	–
Flush+Reload [13]	Cross-Core	298 KB/s	0%
Prime+Probe (LLC) [23]	Cross-Core	75 KB/s	1%
Xiaong and Szefer [36]	Same-Core	72 KB/s	<2%
Streamline (this work)	**Cross-Core**	**1801 KB/s**	**0.37%**

LLC/DRAM latencies which have largely stayed similar. Moreover, Streamline is more universally applicable as it does not rely on cacheline flush instructions (required by flush-based attacks), that are unavailable for unprivileged use on ARM CPUs by default.

Prime+Probe attacks exploiting set-conflicts were pioneered by Perceival [26] on L1-Caches. This work used an array as large as the L1-Cache for communication, transmitting a bit per array-entry (similar to our proposal). However, it used a synchronous protocol, where the sender accesses L1-cache sets to transmit while the receiver waits, and only once it completes the receiver accesses all L1-cache lines and checks for conflicts while sender waits, that limits its bit-rate to 400KB/s [26]. Subsequent works [23, 24] demonstrated a synchronous cross-core Prime+Probe attack on slower LLCs, achieving a bit-rate up to 75KB/s [23]. Streamline is much faster than these attacks, primarily due its asynchronous operation, where the sender and receiver do not have to wait on the other frequently (although Streamline requires shared-memory; Prime+Probe does not). The strategies in Streamline may also be used to enable a faster asynchronous Prime+Probe attack in future works, where the bit-rate is not be limited by synchronization.

Other attacks exploit the replacement policy metadata in L1 caches [36] or in LLCs [4], or even coherence protocols [32, 39] to transmit information more stealthily. However, they suffer a synchronization bottleneck like Prime+Probe or Flush+Reload that limits their bit-rate. Streamline uses insights from prior replacement policy attacks to enable a considerably faster asynchronous attack.

Thrash+Reload [29], a variant of Flush+Reload attack, uses cache-thrashing instead of flush to evict an address each bit. Unfortunately, it is quite slow due to its synchronous nature: the sender waits for the receiver to thrash the cache by accessing more addresses than LLC capacity before it transmits the next bit (bit-rate of 4 bits/min over the network [29] and up to 1000 bits/s natively). In Streamline, the transmission itself induces cache-thrashing, and eviction of previous addresses is automatic (without needing the sender or receiver to wait), helping it achieve 14000x higher bit-rate.

Take-a-way [20] on AMD machines has the highest known bit-rate (588 KB/s) for a same-core covert channel attack. It achieves this by communicating over 80 concurrent synchronous channels, leveraging different L1 cache sets. On the other hand, Streamline transmits over a large number of addresses in a single asynchronous channel and achieves a 3x higher bit-rate. Streamline is also more general as it only relies on generic LLC properties like sharing among cores and thrashing, unlike Take-a-way that exploits way-prediction features only known to be exploitable in AMD CPUs.

6 DISCUSSION

Real-World Applicability: Streamline is applicable to the classic covert-channel setting [5, 34, 39] between trojan and spy processes (both controlled by an adversary), where a trojan malware has infiltrated a sand-box and gained access to secrets, and needs to communicate with a spy program with access to network ports (capable of exfiltrating data or communicating with a command-and-control server). Streamline can enable transmission of high bandwidth payloads (such as video-streams) in such a setting that is not possible with existing covert-channels. While Streamline is also applicable as a covert-channel for transient-execution attacks [18, 21], its benefits may not be as apparent, as the bit-rate bottlenecks in such attacks are typically not the covert channel, but other factors related to transient execution in the victim.

Shared-Memory Requirement: Streamline requires a shared-array that is 2-4x the size of the LLC. This can be easily achieved by the colluding processes with an `mmap` of shared libraries, either using a single large shared library (*e.g.*, libQtWebKit.so is ~32MB) or by chaining a sequence of smaller libraries (*e.g.*, libc.so, libcrypto.so are ~2MB each). Memory may also be shared between processes via OS-based deduplication (*e.g.*, Linux KSM [3]), as in prior attacks [39]. Future work could also explore asynchronous communication without shared-memory (*e.g.*, using conflicts in shared sets for bit-transmission) to avoid this requirement.

Applicability to Hyper-Threads: While we evaluate Streamline between processes running on different cores, it is also applicable to hyper-threads simultaneously running on the same core sharing the L1/L2-cache. An advantage of such a setting is that a smaller shared-array is required for thrashing L1/L2 caches, but the smaller difference in hit-vs-miss latencies for these caches is a challenge. For these reasons, in a cross-thread setting, the L2 cache is a more suitable target for Streamline than the L1 cache.

7 MITIGATION STRATEGY

Defenses that restrict the unprivileged usage of cacheline flush instructions to mitigate flush-based attacks, such as SHARP [37] or ARM ISA, are incapable of mitigating Streamline. There are three main mitigation strategies that might be used to restrict Streamline: detection, noise-injection, or isolation. We describe these below.

Detection based approaches attempt to identify attacks either by profiling them using performance-counters available in commodity hardware [1, 6, 25] or by using specialized hardware [5, 38] to detect contention-patterns prevalent in such attacks. Performance-counter based detection is unlikely to specifically detect Streamline as its cache-access rates and cache-miss rates are quite similar to generic memory-intensive applications (e.g. those processing streams of data). Detectors using specialized hardware, that have the capability to infer detailed cache re-use and contention patterns, have a higher chance of detecting Streamline. For example, record-replay techniques [38] can profile the distribution of cache-hits and misses and potentially infer the cache-access pattern used in Streamline. However, such detection-tools can also be easily fooled by an adaptive version of Streamline that shapes its distribution of cache-hits/misses (using extra LLC-accesses) to match a benign workload and avoid detection. For these reasons, a detection-based approach is not a fool-proof mitigation strategy.

Noise-injection based strategies can attempt to reduce the fidelity of any cache covert channel by dislodging the cachelines used by the sender and receiver for communication via cache-accesses from a co-running application. While most prior attacks only use a single or a small group of addresses for communication and are highly vulnerable to disruption if noise-injection is targeted at these addresses, Streamline utilizes a sequence of addresses (that can be made unpredictable) that makes targeted noise injection more difficult. Moreover, for the noise-injection to be successful, addresses installed by a sender need to be dislodged before the receiver accesses them. Reducing the number of bits buffered in the LLC at a time and the time-window for which each address is buffered, by reducing the synchronization-period (as discussed in Section 4.7) limits the exposure of the attack to noise-injection.

Hardware designs like randomized prefetching [10] or randomized cache fills [22] may naturally hinder Streamline operation by introducing noise, although such designs also cause slowdown for benign applications. On the other hand, random-replacement can add noise to the channel, but is unlikely to fully prevent Streamline. If shorter synchronization periods are in use where less than 10% of the LLC space is actively used as a buffer at any given time (Section 4.7), even random replacements due to co-running process activity are unlikely to dislodge a significant portion of the lines between the sender and receiver accesses to disrupt the channel. Streamline can tolerate infrequent interference due to random replacement using ECC (as discussed in Section 4.3).

Isolation based approaches prevent processes in different trust-domains from sharing cache locations and are highly effective at mitigating shared-memory based covert-channel attacks. Such cache isolation can be achieved by disabling shared-memory or deduplication (e.g. disabling Linux KSM [3]), or by using cache-partitioning techniques that eliminate cross-domain hits required for transmission in Streamline: by allocating disjoint groups of LLC sets to processes in different trust-domains [15, 27], or by duplicating shared cachelines across trust-domains [17, 35]. Such techniques eliminate Streamline and all cache attacks exploiting hits on shared cachelines. However, all such solutions either have performance costs or face scalability challenges (cache-partitioning requires allocation at the limited granularity of ways or sets) or require support from system-software to classify processes into trust-domains (needed for cache-partitioning) that could limit their practical applicability.

8 CONCLUSION

This paper advances the state-of-the-art in cache covert-channel attacks by systematically analyzing the bit-rate bottlenecks for existing attacks, including synchronization and transmission bottlenecks, and proposing a faster *Streamline* attack that overcomes these bottlenecks. With its asynchronous operation, Streamline achieves a bit-rate of 1801 KB/s, which is 3–3.6x faster than prior-fastest attacks. Streamline is also flush-less and only exploits generic cache properties and hence is applicable to CPUs of all ISAs and micro-architecture unlike prior attacks. Finally, this work also highlights a new *measurement* bottleneck (inability to measure the latency of simultaneously executing loads) that limits the bit-rate for all existing covert-channels and also this new class of asynchronous covert-channels. Overcoming this can enable even faster attacks.

ACKNOWLEDGMENTS

We thank Stephan Diestelhorst and Mohammad Behnia for informative discussions on load-latency measurement techniques. We also thank our shepherd Baris Kasikci and anonymous reviewers of ASPLOS'21 for their helpful feedback and suggestions. We also thank Swamit Tannu, Sanjay Kariyappa, and Poulami Das for their feedback. This work was supported in part by the following sources: SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory (CRISP), NSF Grants 1954521 and 1942888, an Intel ISRA center, and a gift from Intel. Gururaj Saileshwar is partially supported by an IISP Cybersecurity PhD Fellowship.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact presents the code and methodology to run our Streamline cache covert-channel attack. We provide the C++ code for the sender and receiver processes engaged in covert communication. Although the attack itself is not specific to an OS, ISA, or micro-architecture, the code is written with the assumption of an x86 Linux system and an Intel CPU that is a Skylake or a newer generation model. The code may be compiled with a standard compiler and run natively to execute the covert-communication. We also provide scripts to run the attack in several configurations demonstrated in Section-IV of our paper (with and without ECC, varying the shared array size and the synchronization period) and provide a Jupyter notebook to visualize the results.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** Implementation of Streamline attack in C++.
- **Compilation:** Tested with gcc v6.4.1, but the code should compile with most standard compilers.
- **Run-time environment:** Requires Linux (Tested on Fedora 25 and Ubuntu). *Sudo* privilege needed for stable bit-rate measurement (to pin processes to cores and disable DVFS).
- **Hardware:** Requires Intel CPU of Skylake or a newer generation. Tested on Intel Xeon E3-1270 v5 (Skylake), Core i7-8700K (Kaby Lake), and Core i5-9400 (Coffee Lake) CPUs.
- **Metrics:** Transmission Bit-Rate and Bit-Error-Rate.
- **Output:** Streamline Results from Sections 4.2, 4.3, 4.4, 4.5 (Tables 2,3,4, and Figure 9) can be reproduced.
- **Experiments:** Instructions to run experiments and generate tables and figures are available in the README file.
- **Time needed to complete experiments:** 3-4 hours
- **Publicly available?:** Yes.

A.3 Description

A.3.1 Link: The code is available at https://github.com/gururaj-s/streamline and https://doi.org/10.5281/zenodo.4322033.

A.3.2 Hardware Dependencies: The attack requires an Intel CPU from Skylake or a newer generation, as the Streamline logic to fool the LLC replacement policy and prefetcher is currently tuned for these CPUs. Additionally, system-parameters like LLC Size, LLC and DRAM access-latency need to be set before running the attack (we describe how these can be discovered in the README).

A.3.3 *Software Dependencies:*

- GCC compiler (tested with version 6.4.1)
- Command line utilities: cpupower (for setting frequency).
- Python3 and Jupyter Notebook (for plotting)
- Python3 Packages: pandas, matplotlib, seaborn.

A.4 Installation

The makefile compiles the sender and receiver executables with the command `make all`.

A.5 Experiment Workflow

The following steps are needed to run the experiments:

- Enable Support for Transparent Huge Pages
- Fix CPU Frequency to a stable value using `performance` frequency governor for correct bit-rate calculation.
- Set system-specific parameters in `src/utils.hh` mentioned in Section A.7.
- Compile the sender and receiver processes using `make all`.
- Run all the experiments using `./run_exp.sh`.

A.6 Evaluation and Expected Result

The artifact provides `run.sh` script to run following experiments:

- Figure-9 : Streamline bit-rate and error-rate as the payload size varies from 200,000 bits to 1 billion bits
- Table-2 : Breakup of error-rates for different payload sizes.
- Table-3 : Streamline Bit-Error-Rate with error-correction.
- Table-4 : Streamline Bit-Rate for different array sizes.
- Table-5 : Streamline Bit-Rates and Error-Rates for different synchronization periods.

The generated results can be visualized using the provided Jupyter notebook `visualize_results.ipynb`.

A.7 Experiment Customization

The following system-specific parameters need to be updated in the `src/utils.hh` (as per the instructions in README):

- LLC Size in bytes.
- LLC Miss Threshold in cycles.
- CPU Frequency in MHz
- Path to the shared file containing the shared array

A.8 Notes

Note that the bit-rate for Streamline on a new system will vary based on the DRAM and LLC access latency of the system. The error-rates are expected to be low (1-5%) for a successful orchestration of the attack. If significantly higher error-rates are observed, some potential reasons could be:

- Improper configuration of LLC-Miss-Threshold. This may need to be manually tuned for your system.
- Sender and receiver processes not properly pinned to separate cores or being context-switched leading to loss of synchronization. Running the sender and receiver tests specified in README with the Linux `perf` tool could help to check if this is the case.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] Manaar Alam, Sarani Bhattacharya, Debdeep Mukhopadhyay, and Sourangshu Bhattacharya. 2017. Performance Counters to Rescue: A Machine Learning based safeguard against Micro-architectural Side-Channel-Attacks. *IACR Cryptology ePrint Archive* 2017 (2017), 564.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 870–887.

[3] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium*.

[4] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. 2020. RELOAD+ REFRESH: Abusing cache replacement policies to perform stealthy cache attacks. In *29th USENIX Security Symposium (USENIX Security 2020)*.

[5] Jie Chen and Guru Venkataramani. 2014. CC-Hunter: Uncovering covert timing channels on shared processor hardware. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 216–228.

[6] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. 2016. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing* 49 (2016), 1162–1174.

[7] CPU-Monkey. (accessed January 15, 2021). Comparing Intel Xeon E3-1270 v5 vs Intel Core i7-4790. https://www.cpu-monkey.com/en/compare_cpu-intel_xeon_e3_1270_v5-601-vs-intel_core_i7_4790-355.

[8] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices* 53, 2 (2018), 693–707.

[9] Christopher Fletcher, Mohit Tiwari, Mengjia Yan, Mohamad El Hajj, Shijia Wei, and Yasser Shalabi. 2019. Covert Channel Attack Tutorial at ISCA 2019. https://github.com/yshalabi/covert-channel-tutorial.

[10] Adi Fuchs and Ruby B Lee. 2015. Disruptive prefetching: impact on side-channel attacks and cache designs. In *Proceedings of the 8th ACM International Systems and Storage Conference*. 1–12.

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

[12] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. 2017. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In *26th USENIX Security Symposium (USENIX Security 17)*. 1075–1091.

[13] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 279–299.

[14] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH Computer Architecture News* 38, 3 (2010).

[15] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *21st USENIX Security Symposium (USENIX Security 12)*. 189–204.

[16] Colin King. (accessed January 15, 2021). Ubuntu Wiki: Stress-NG. https://wiki.ubuntu.com/Kernel/Reference/stress-ng.

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

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

[19] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*. 549–564.

[20] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *Proceedings of the 2020 ACM Asia Conference on Computer and Communications Security*.

[21] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*. 973–990.

[22] Fangfei Liu and Ruby B Lee. 2014. Random fill cache architecture. In *47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 203–215.

[23] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 605–622.

[24] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. C5: cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 46–64.

[25] Mathias Payer. 2016. HexPADS: a platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems*. Springer, 138–154.

[26] Colin Percival. 2005. Cache missing for fun and profit. In *Proceedings of BSDCan*.

[27] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. 2009. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*. 77–84.

[28] Alberto Ros, Trevor E Carlson, Mehdi Alipour, and Stefanos Kaxiras. 2017. Non-speculative load-load reordering in TSO. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 187–200.

[29] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*. 279–299.

[30] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 3–24.

[31] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. 2018. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 131–145.

[32] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE.

[33] Krishnaswamy Viswanathan. (accessed August 1, 2020). Disclosure of Hardware Prefetcher Control on Some Intel Processors. https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html.

[34] Jack Wampler, Ian Martiny, and Eric Wustrow. 2019. ExSpectre: Hiding Malware in Speculative Execution.. In *NDSS*.

[35] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium (USENIX Security 19)*. 675–692.

[36] Wenjie Xiong and Jakub Szefer. 2020. Leaking Information Through Cache LRU States. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 139–152.

[37] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. 2017. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 347–360.

[38] Mengjia Yan, Yasser Shalabi, and Josep Torrellas. 2016. ReplayConfusion: detecting cache-based covert channel attacks using record and replay. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[39] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. 2018. Are coherence protocol states vulnerable to information leakage?. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 168–179.

[40] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. 719–732.