# Reverse Engineering the Stream Prefetcher for Profit

Aditya Rohan
*MSE, IIT Kanpur*
*raditya@iitk.ac.in*

Biswabandan Panda
*CSE, IIT Kanpur*
*biswap@cse.iitk.ac.in*

Prakhar Agarwal
*CSE, IIT Kanpur/Tower Research*
*prakhar.agrwl98@gmail.com*

***Abstract*—Micro-architectural attacks exploit timing channels at different micro-architecture units. Some of the micro-architecture units like cache automatically provide the timing difference (the difference between a hit and a miss). However, there are other units that are not documented, and their influence on the timing difference is not fully understood. One such micro-architecture unit is an L2 hardware prefetcher named Streamer. In this paper, we reverse-engineer the Stream prefetcher, which is commercially available in the Intel machines. We perform a set of experiments and provide our observations and insights. Further, we use these observations to construct a cross-thread covert channel using the Stream prefetcher, with an accuracy of 91.3% and a bandwidth of 54.44 KBps.**

## 1. Introduction

Timing channels through different micro-architecture units like branch predictors and caches are practical and are a threat to many systems, including handheld devices, clients/server systems, and clouds. Timing channels observe the fundamental property of "latency differences between cache hits and misses" to infer about the cache blocks that are accessed by the victim (cryptographic) application. Attacks such as Flush+Reload [1] and Prime+Probe [2] [3] use the timing difference of accessing cache lines in order to communicate bits. However, the recent wave of disclosing microarchitectural attacks has exploited most of the well-known units. Hence, there is a need to *reverse-engineer* similar lesser-known micro-architecture units that might be used to mount a new side/covert-channel attack.

To attack a micro-architecture unit, we need first to understand it properly. One such micro-architecture unit is a hardware prefetcher, which is a popular off-chip memory latency hiding technique employed in all the commercial machines. Different kinds of prefetchers are employed at different levels of cache catering to different kinds of access patterns. One of the hardware prefetchers named Streamer is employed in Intel machines.

**The problem:** Intel manual [4] does not provide many details regarding the functioning of the stream prefetcher. To that effect, we reverse-engineer the Intel Stream prefetcher to discover the unknown properties and construct a covert channel.

**Our goal** is to find out answers to the following questions related to the Stream prefetcher: (i) Is it shared between two hyper-threads. (ii) How many entries are there in a stream table? (iii) What is the range of prefetch degree and distance that a Streamer uses? (iv) Is it possible to create a cross-thread covert channel using the Streamer?

Our contributions are as follows: (i) We reverse-engineer the Stream prefetcher answering the questions mentioned above (Section 3). (ii) We propose a novel inter-thread covert channel through the stream prefetcher. Our covert channel provides a communication bandwidth of about 54.44 Kbps with an accuracy of 91.3% (Section 4). To the best of our knowledge, this is the first paper that attempts to reverse-engineer the Stream prefetcher.

## 2. Background

### 2.1. Hardware Prefetchers

Modern CPUs use data prefetching to reduce the costly off-chip DRAM accesses and hide the miss latency at various levels of cache. Some well known prefetching techniques are the next-line prefetching, stream prefetching [5], and stride prefetching [6]. In next-line prefetching, if a request for a cache line X is received, then the line X+1 is prefetched. The stream prefetcher prefetches a stream of lines at a varying prefetch distance from the current line X. Stream prefetcher assumes a contiguous access pattern in a particular direction and exploits the spatial locality of such a pattern. The stride prefetcher observes a stride in the access pattern and prefetches lines at the observed stride. The stream prefetcher is the prefetcher of interest for this paper.

### 2.2. Stream prefetcher

The stream prefetcher prefetches a stream of lines. We can explain the functioning of the streamer in the following three steps: (i) the first miss, say to cache line X, initiates a stream, (ii) the second miss to cache line X+Y defines the direction of the stream in this case, and (iii) the third miss, at X+Z (where Z>Y), confirms the direction. Prefetching begins at the next miss, X+D. The streamer maintains a stream table, which is filled upon confirmation of the direction of the stream. This table stores multiple entries per OS page and stores prefetching-metadata for several pages at once. A miss can only trigger prefetching if such an entry exists for the same OS page.

The three-step process to decide and confirm the direction of the stream helps improving prefetch accuracy. Higher accuracy is essential because prefetched lines evict other lines already present in the cache, and if the prefetched lines are never used, then it only pollutes the

cache. The degree to which an incorrect prefetch affects the cache also depends on how many lines the prefetcher prefetches. The “prefetch degree” is the number of lines the streamer prefetches at a given point of time, and the “prefetch distance” is how far ahead the prefetched cache-line is from the trigger line.

### 2.3. Covert Channel

A covert channel is any usual information channel that is used to share information in an unintended way, violating the system’s security and privacy policies. All shared resources can be possible covert channels. Covert communication at the micro-architecture level uses one of the following protocols:

**Prime+Probe: [2], [3]:** In a Prime+Probe covert channel, the receiver first fills all lines of a cache set with its data. The sender process, if now accesses any of the lines in that cache set, then it evicts the receiver data. On subsequent access to the same cache set, the receiver observes a cache miss. For a pre-decided line in the cache set, the sender can send a 1 (receiver observes a hit) or 0 (receiver observes a miss).

**Flush+Reload [1]:** Flush+Reload covert channel relies on sender and receiver physically sharing memory pages. The sender flushes a pre-decided cache line using instructions like `clflush` and waits for the receiver to access the same cache line. The received bit is 1 if the receiver measures a low access latency, otherwise it’s 0. This information can be used to communicate between the sender and the receiver process.

Building on the basics of the stream prefetcher and the covert-channel, we now discuss the reverse engineering experiments and observations.

## 3. Reverse Engineering the Stream Prefetcher

In this section, we experiment with the Stream prefetcher to study its behavior. Intel manual contains only a brief explanation on the behavior of hardware prefetchers, but recently [7] Intel revealed a way to control the hardware prefetchers. A user can now control the prefetchers using bits 0 to 3 of the Model Specific Register (MSR) present on every core at the address `0x1a4`. We use these bits to disable all prefetchers except the stream prefetcher at the L2 cache. The test system for all experiments described in the following sections, comprises of a dual-core Intel Kabylake Core i5-7200U CPU, with a 64KB L1 cache, 256KB L2 cache, and an LLC of 1.5MB/core, with each core supporting simultaneous multi-threading (SMT). All the experiments use two threads (T1 and T2) that belong to the same physical core. We ensure the experiments are free from any system noise by isolating the physical core. This prevents any applications from being scheduled on the experiment core. We perform all experiments about 100 times, to ascertain the results. We performed the experiments on Intel Skylake i7 processors as well for testing. However, throughout the paper, we use the Core i5 machine to discuss our results and observations.

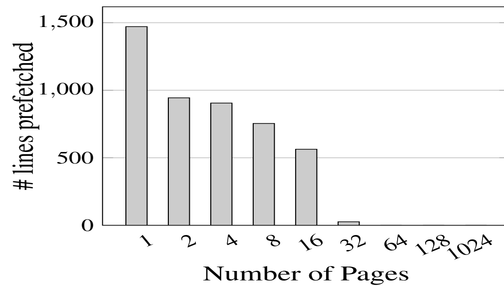


Figure 1: Plot indicating that the size of Stream table is 16 entries. Refer 3.1 for the details.

### 3.1. Stream Table Size

In this experiment, we determine the size of the stream table. We launch a program that accesses the first three lines of an OS page. The number of pages increases in every iteration by order of two i.e., 1, 2, 4, 8, and so on. By accessing the first three lines of all the pages, we create a single stream table entry for each page, such that future accesses to the same page would trigger the prefetcher. After every iteration, we access the fourth line of the first page in the sequence, which remains same for every iteration. If a stream table entry exists for the first page, then we expect to see some prefetch activity (prefetched cache lines).

If the number of pages accessed in an iteration is more than the number of entries in the stream table, then the entry made by the first page will be evicted from the table, assuming the least recently used policy for eviction of the entries of the stream table. If the entry for the first page is evicted, further accesses to that page will not trigger the prefetcher, resulting in the absence of any prefetch activity. From Figure 1, we can see the total number of lines prefetched reduces abruptly after 16 pages. This indicates that on accessing more than 16 pages, the stream table entry for the first page is evicted from the table.

**Observation:** *The L2-stream prefetcher maintains a table of 16 entries to track the prefetch behavior of various pages.*

### 3.2. Stream Prefetcher: Is it Shared?

We experiment to find out whether two SMT threads of a single physical core share the stream prefetcher at L2 cache. We run a program that spawns two threads, each bound to one of the SMT threads of the physical core 0 of the test system. Each thread accesses a large array of size 4MB, large enough to cover all three levels of caches. Each thread accesses every cache line within an OS page, either in the +ve or -ve direction. For example, for an OS page X, accesses to cache lines X+1, X+2, and X+5 constitute a +ve direction stream and accesses to cache lines X+7, X+3, X+1 constitute a -ve direction stream. We run the following four experiments: *(i) both threads accessing in the +ve direction, (ii) both threads accessing in the -ve direction, (iii) one thread accessing in the +ve direction and other one accessing in the -ve direction, and (iv) one thread accessing in the -ve direction and other one accessing in the +ve direction.*

If the stream prefetcher is shared among threads, then threads running on the same physical core can use the

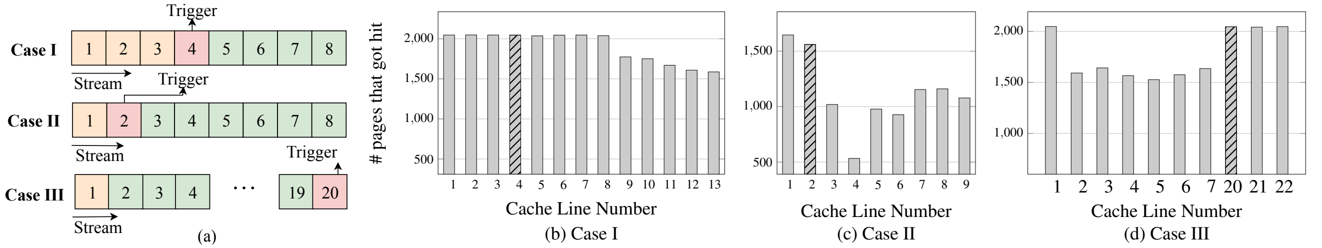


Figure 2: (a) Three cases of prefetch trigger. (b),(c), and (d) show the number of OS pages that got hits for the given cache line numbers, for case-I, case-II, and case-III, respectively. Cross-hatched lines represent trigger lines.

TABLE 1: Effect of stream direction on execution time (in terms of average cycles over 100K runs).

| Stream direction | | Avg. completion cycles | |
|---|---|---|---|
| Thread-1 | Thread-2 | Thread-1 | Thread-2 |
| +ve | -ve | 44045 | 42700 |
| +ve | +ve | 37922 | 34697 |
| -ve | +ve | 46194 | 45142 |
| -ve | -ve | 43280 | 39971 |

same stream table entry. Therefore, if a thread makes an entry in the stream table, then the next thread can start prefetching from the first miss, and the execution time of the second thread will be lower than the first one. The second thread will start prefetching after the first access (skipping the need to train the prefetcher). However, if the threads do not share the stream table entry, then the execution time for either thread will not improve.

**Observation:** *Table 1 shows the result of this experiment and supports our hypothesis. Threads with same direction access pattern, do use the same stream table entry, which we can see in the reduced number of cycles required for completion for two pairs: (i) +ve/+ve and (ii) -ve/-ve as compared to +ve/-ve and -ve/+ve pairs.*

At this point, we know that SMT threads on the same physical core share the stream prefetcher at L2 cache.

### 3.3. Unraveling the Prefetch Aggressiveness

Now that we know that the stream prefetcher is shared between logical cores on the same physical core, we can proceed to unravel some other features regarding the distance and degree (aggressiveness) of the prefetcher.

**Case I:** In this experiment, thread-1 (T1) accesses the first three cache lines of a page. Next, thread-2 (T2) accesses one line from that page. In this case, T2 accesses the 4th line within that page. This case exhibits our common understanding of the stream prefetcher's behavior. The expectation is that since three accesses can define a stream direction, T1's accesses make an entry in stream table for +ve direction prefetching. T2's access then acts as a trigger and prefetches a set of lines according to the streamer's distance and degree. Next, T1 reloads all the 64 cache lines (numbered 1 to 64) within that page in 64 different iterations. Note that a 4KB OS page consists of 64 cache lines, each of 64 bytes. Case I of Figure 2 (a) shows that T1 accesses the first three lines in the +ve direction and T2 accesses the 4th line as a trigger.

To avoid triggering the prefetcher unintentionally, we reload only one line per page in one iteration. For a given page, T1 iterates 64 times, monitoring the effect of the experiment on each line. We repeat this for all 2048 pages, and show (Figure 2 (b)) the count of the pages in which we get L2 cache hits for a specific cache line. For example, if we want to monitor the $n$th line for L2 hits/misses, we repeat T1's access-pattern for every page and observe whether this causes the $n$th line to get a hit or a miss. In case, the $n$th line is a hit in all the pages, then the count will be 2048. We repeat this experiment for each line within the monitored page. In Figure 2 (b), the first four lines show almost perfect hit count out of 2048.

Subsequent lines indicate a significant number of hits as well. However, the hit count varies depending on the distance and the degree. Note that for line nos. 14 to 64, the count is zero, which shows that the prefetcher does not prefetch any lines beyond line no 13. We can use this experiment to estimate distance and degree. We also perform a similar experiment in the -ve direction. In the reverse experiment, T1 accesses lines 64, 63 and 62, to create a stream table entry for -ve direction. Accessing line 61 triggers the prefetcher, and prefetches some lines in the -ve direction. We see results similar to the +ve direction stream table entry here as well.

**Observation:** *The distance and degree in Case I is dynamic, ranging from one to four and four to eight lines, respectively.*

### 3.4. Understanding the Stream Trigger

**Case II:** We now know that three consecutive cache misses in a particular direction create an entry in the stream table, and the fourth miss triggers the prefetching. We were also able to verify this for both +ve and -ve directions. The first two misses decide the direction, and the third miss ascertains it. We find anomalies to this rule. We run an experiment similar to Case I, using only one line access before T2's access. Case II of Figure 2 (a) shows this experiment. We expect to see a higher number of hits for the first two lines and near-zero hits for the rest of the lines. Our experiments show that this is not the case. The prefetcher was getting triggered even after only two accesses in the same direction. Figure 2 (c) shows the count of pages that got the hits. We can see a significant number of hits for lines up to line number nine.

**Observation:** *One inference that we can draw from this experiment is that the trigger line accessed by T2 helps in deciding and confirming the direction and works as a trigger to prefetch more lines.*

**Case III:** In this experiment, T1 accesses the first line, and T2 accesses the 20th line (it can be any line number). From the previous experiment, we only expect to see some lines prefetched after line number 20. However, we see that the prefetcher prefetches six lines after the first line, as well. Figure 2 (d) shows the results for the same. This

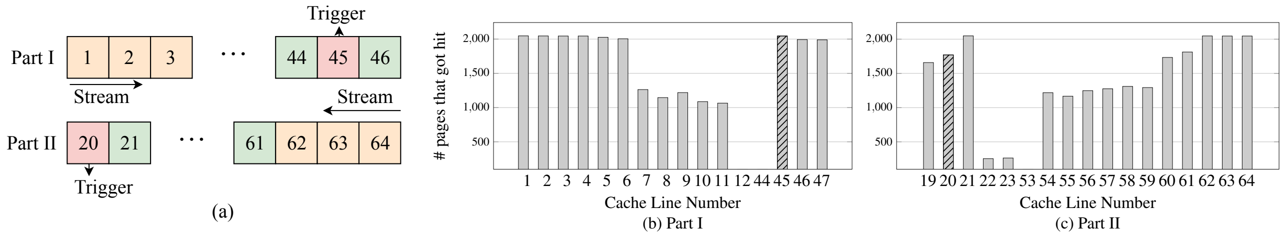


Figure 3: (a) Case IV: Effect of stream directions in two parts. (b) and (c) shows the number of OS pages that got hits for the given cache line numbers, and for the two parts. Cross-hatched lines represent trigger lines.

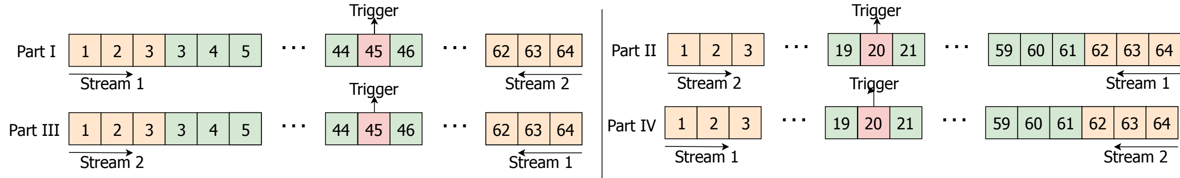


Figure 4: All parts of Case V with the stream access direction and the trigger line number.

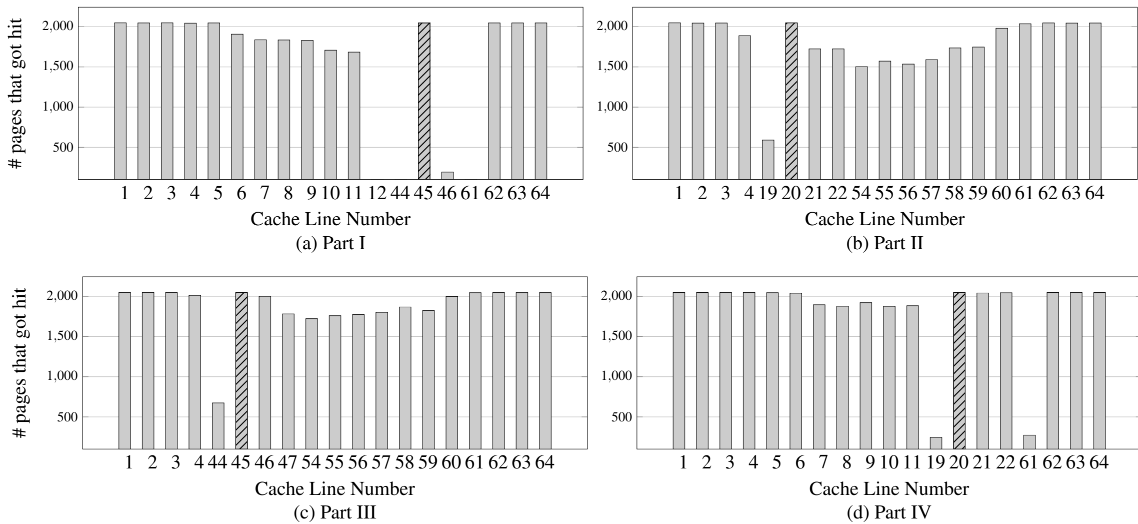


Figure 5: The plots show the number of pages that got hits for Case V. Cross-hatched lines represent trigger lines.

case is also shown in Case III of Figure 2 (a).

**Observation:** *This raises some questions about our understanding of the stream prefetcher. According to this experiment, on a trigger access, prefetching happens at the trigger line as well as the previous cache miss. Case II and III suggest the possibility that there is another undocumented hardware unit that behaves as a prefetcher. However, we cannot control it with the MSR bits.*

### 3.5. Dominant Stream Direction

**Case IV:** In the first three cases, we see some unexpected behavior with the degree and trigger of the stream prefetcher. The direction of the stream table entry can be yet another point of variance in the prefetcher behavior, and we do the following experiment to establish the same.

As we can see from Figure 3 (a), T1 does two sets of accesses, one in +ve direction ... one in -ve direction, and accesses the 20th line from both the directions. Part I of the Figure 3 (a) shows stream access from the +ve direction and access to line number 45 (the 20th line from the reverse direction) as the trigger. In Part II, T1 accesses the stream in the -ve direction and accesses line number 20 as a trigger.

**Observation:** *This makes the two experiments equivalent in all respects except for the access direction. The expected behavior would be laterally inverted plots in terms of page count that got hits. Figure 3 (b) shows that the stream prefetcher prefetches lines only in the +ve direction from the trigger line 45. However, in Part II, lines are prefetched in both the directions from the trigger line number 20 (Figure 3 (c)). This experiment suggests that the prefetchers in +ve and -ve direction do not behave the same.* Next, we explore this difference in functionality.

**Case V:** In this experiment, we test the effect of stream table entries that are accessed in both the directions, on the prefetch behavior. Part I of Figure 4 shows the first, where T1 accesses the first three lines of each page, creating a stream table entry in the +ve direction. Then T1 accesses the last three lines of the same page in reverse order to create a stream table entry for the -ve direction. T2 then comes in and accesses the trigger line, line number 45 for Part I. Figure 4 marks the first access sequence as stream one and the other access sequence as stream two.

The first and the second parts of the experiment are lateral inversions of each other. In Part I of Figure 4, stream one is the +ve stream, and stream two is the -ve stream and the trigger line is the 20th line from stream

two, i.e., line number 45. Whereas in Part II, stream one is the -ve stream, and stream two is the +ve stream. The trigger for part two is the 20th line from the +ve direction.

For this experiment, we expect inverted plots for Part I and Part II. However, as we can see from Figure 5 (a) and (b), this is not the case. Figure 5 (a) shows that lines around the trigger line, the 45th line is this case, show a negligible number of hits. Whereas Figure 5 (b) shows significant hits around the trigger line (cache line number 20) in either direction. This experiment suggests the +ve stream-table entry is dominant compared to the -ve entry.

The third and fourth parts also present similar experiments (Part III and Part IV of Figure 4). In Part III, stream one is the -ve stream and stream two the +ve stream. The trigger line is the 20th line from the first stream, i.e., line number 45. In Part IV, stream one is the +ve stream, and stream two is the -ve stream, and the trigger line is the 20th line from the first stream, line 20. Plots in Figure 5 (c) and (d), show that the streamer prefetches more lines in either direction from the trigger line when the first stream is a -ve stream, as in Part III.

**Observation:** *(i) If the first stream is in the +ve direction, then the second stream's prefetch degree gets suppressed to a higher degree compared to when the first stream is in the -ve direction. In Figure 4, Part I and Part IV, we can see that Stream 2 only accesses the last three lines, which show a significant number of hits in the individual plots. In contrast Part II and Part III, the first four lines are getting a considerable number of hits (Figure 5 (b) and (c)), but Stream 2 accesses only the first three lines. We can also say that the distance of the trigger line from the first stream affects the degree of prefetch at the trigger line. When the trigger is closer to Stream 1 as in Part III and Part IV, the prefetch degree is higher in both directions when compared to Part I and Part II.*

### 3.6. Limitations of the Streamer

From the previous experiments, we can see that two threads running on the same physical core can share the stream table entries for the same shared pages. Sharing of physical pages between the hyper-threads defeats the point of a possible side-channel, as the threads can easily share information via the memory. We perform an experiment to study the effect of huge pages on the stream prefetcher to find a way to overcome the aforementioned shortcoming.

We allocate a huge amount of memory in five huge-pages of size 2MB each. We then create a stream table entry by accessing the first three lines of the first huge-page. We then access a trigger line, which keeps incrementing over iterations. For every trigger line, we monitor the prefetch activity due to the same stream table entry created by accessing the first three lines. To monitor the prefetch activity, we use the approach from Case I, Figure 2. We reload ten lines in front of the trigger, one in each iteration. We keep increasing the trigger line until there is no observable prefetch activity. Our experiments show that no activity can be observed after the trigger line crosses 64, which is the page boundary of a 4KB OS page. This experiment, after multiple runs, shows the same results, indicating that each stream table entry is only valid for a 4KB OS page, even when the huge-pages are enabled.

Interestingly, when huge-pages are not enabled, a stream table entry created for a 4KB OS page can prefetch lines into the next 4KB page if the trigger line is one among the first two lines of the next page. For any subsequent lines, no prefetch activity can be observed if a stream table entry does not exist for that page. This resembles the next page prefetcher (NPP) [8], which demonstrates the ability to prefetch across page boundaries, but to a limited degree. On enabling huge-pages, this cross-page prefetching effect cannot be seen at all.

**Observation:** *The stream table entries are only valid for the 4KB OS pages that they were created for, even when huge-pages are enabled. For normal 4KB OS pages, cross-page prefetching can be seen for the first two lines of the page.*

*Next, we discuss the construction of a covert channel as a proof of concept, using the findings of this section. To the best of our knowledge, this is the first paper to show a stream trigger activity-based covert channel.*

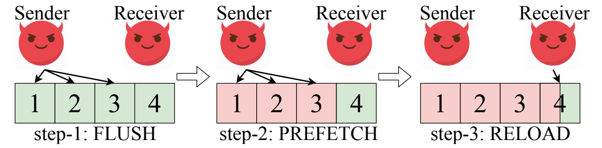


Figure 6: **Step-1:** Sender flushes all lines of the shared page.
**Step-2:** Sender then accesses first three lines, in either +ve (for **1**) or -ve (for **0**) direction, triggering the prefetch.
**Step-3:** Receiver reloads line four, hit means **1**, miss means **0**.

## 4. Covert Channel Construction

**Threat Model:** For the covert channel, we assume that the attacker is able to trick a third-party to run the malicious code by poisoning a third-party library. The attacker can also pin the Sender and Receiver processes to the same physical core. The two programs, now running on the two hyper-threads, have access to a shared stream prefetcher. The sender and receiver are trying to communicate covertly using the stream prefetcher, which cannot be accessed using any specialized instructions. Our threat model is similar to the one proposed in [9].

**Flush+prefetch+reload:**We propose a way to use the stream prefetcher as a covert channel with a sender-receiver model. The three-step communication process is also shown in Figure 6. The sender is pinned to thread-0 of physical core-0 and always accesses the first three lines of a pre-decided page in the shared memory. For sending a bit value 1, it triggers prefetching of lines in the +ve direction following line number three. For sending a bit value 0, the sender always accesses the first three lines in the -ve direction, prefetching no lines in front of line number three. The receiver, pinned to thread-1 of physical core-0, always accesses line number four and measures the access latency. If the sender accesses lines in the +ve direction then line number four will be present in the cache due to prefetching. When the receiver reloads the fourth line, it will observe a hit. On the other hand, if the sender accesses the first three lines in the -ve direction, then the prefetcher will be trained for the -ve direction prefetching. However, the prefetcher can only prefetch within the page

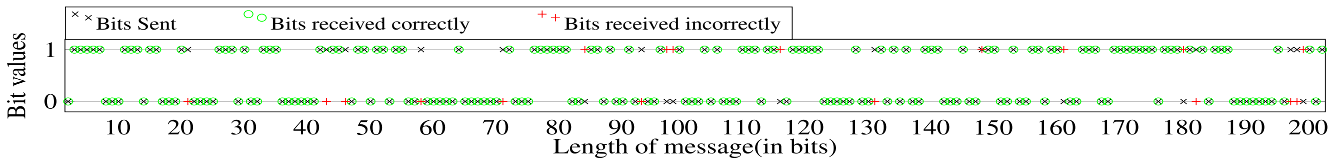


Figure 7: A snapshot of 200 random bits sent over the stream prefetcher as a covert channel.

boundary, and the fourth line will not be prefetched. Now when the receiver reloads the line, it will see a miss.

A cache hit on accessing the fourth line indicates bit value 1, and a miss indicates bit value 0. We use a calibration tool [10] to determine the L2 cache miss threshold. After the receiver receives a bit, the sender flushes all the cache lines of the same page to reduce noise while sending the next bit.

**Synchronization:** We performed this experiment by sending random bit strings, as shown in Figure 7, on an isolated physical core of Intel Skylake i5 processor.

To synchronize the bit communication, we implement a simple while loop lock using two shared variables (S & R) initialized to -1. The sender updates the first variable (S), and the receiver updates the second variable (R). For receiving the *ith* bit, the receiver waits till the sender sets the value of S to *i*. The sender sets the value of S to *i* only after training the stream table entry for the *ith* bit. After that, the sender waits until the receiver updates the value of R to *i*. The receiver only updates the value of R after receiving the *ith* bit. The sender then proceeds to send the *(i+1)th* bit, and the receiver waits until the sender updates the value of S again.

**Accuracy, bandwidth, and error-rate:** Through this covert channel, we achieve an accuracy of up to 91.3% at a bandwidth of 54.44 KBps, and an error rate 4.73 KBps (represents the number of bits lost due to erroneous transfer of bits). The error rate, though small but existent, can be explained with the help Case V from Section 3.5. Flushing all lines of the page from the cache after sending each bit does not flush the stream table entries for the given page. Thus after sending a '1' and a '0' bit, the stream table has entries for both +ve and -ve directions. As shown in Case V (Figure 4), if both +ve and -ve stream table entries exist for a page prefetching can happen in either direction from the trigger line. To ascertain this hypothesis, we run the same covert channel by sending 1000 '1' and '0' bits, to achieve an accuracy on 99.9%.

## 5. Related Work

From the past few years, there have been a couple of papers exploring possible side and covert channel attacks using hardware prefetchers. Bhattacharya et al. [11] study the effect of next-line prefetching in the context of information leakage. Cronin et al. [9] use the stream table size to communicate covertly between processes and achieve a 41.6 KBps transmission speed. A process 'A' creates a few stream table entries while executing, then yields the CPU, process 'B' then takes over and trains a few stream table entries of its own. If the total number of entries created by 'A' and 'B' exceeds the size of the stream table, then by an LRU replacement policy some of 'A''s entries will be evicted. When it's time for 'A' to run again, it'll have to retrain the prefetcher, thus the increased time of execution. This time difference can be used to communicate bits covertly. However, this covert channel is different from the one proposed in this paper since we observe the prefetch activity on a page to communicate bits. Our covert channel also provides higher communication bandwidth. Shin et al. [12] use another hardware prefetcher, the IP-based stride prefetcher, to perform a side-channel attack that is specific to the OpenSSL crypto-library [13].

## 6. Conclusion

In this paper, we proposed a set of experiments to reverse-engineer the stream prefetcher at the L2 cache. These experiments provide insights into the previously unknown subtle issues related to the stream prefetcher. We also propose a novel way to exploit the stream direction, trigger, and prefetch degree of the prefetcher to construct a covert channel. Future work involves mounting a side-channel using the stream prefetcher. The experiments and insights of this paper can be used to present more potent side-channel attacks in the future.

## 7. Availability

https://github.com/car3s/Whispering-Streamers.

## 8. Acknowledgement

We would like to thank all the anonymous reviewers for their helpful comments and suggestions. This work is supported by the SRC grant SRC-2853.00.

## References

[1] Yarom *et al.*, "Flush+ reload: a high resolution, low noise, l3 cache side-channel attack," in *Usenix Security 14)*, pp. 719–732, 2014.

[2] Osvik *et al.*, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*, pp. 1–20, 2006.

[3] Liu *et al.*, "Last-level cache side-channel attacks are practical," in *2015 IEEE S&P*, pp. 605–622, 2015.

[4] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*. No. 253669-033US, March 2018.

[5] Tendler *et al.*, "Power4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–25, 2002.

[6] Baer *et al.*, "An effective on-chip preloading scheme to reduce data access penalty," in *Supercomputing'91*, 1991.

[7] V. Viswanathan, "Disclosure of h/w prefetcher control on some intel processors," in *Intel SW Developer Zone*, 2014.

[8] "Inconsistency in tlb miss counter." Intel Developer Zone, 2015.

[9] Cronin *et al.*, "A fetching tale: Covert communication with the hardware prefetcher," in *2019 HOST*, pp. 101–110, 2019.

[10] Gruss *et al.*, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Usenix Security 15)*, pp. 897–912, 2015.

[11] Bhattacharya *et al.*, "A formal security analysis of even-odd sequential prefetching in profiled cache-timing attacks," in *HASP@ISCA 2016*, pp. 6:1–6:8, 2016.

[12] Shin *et al.*, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *ACM CCS*, pp. 131–145, 2018.

[13] OpenSSL, "Cryptography and ssl/tls toolkit," 2018.