Hindawi
Security and Communication Networks
Volume 2023, Article ID 8031859, 12 pages
https://doi.org/10.1155/2023/8031859

Research Article

Cross-Core and Robust Covert Channel Based on Macro-Op Fusion

Han Wang and Ming Tang

Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan, China

Correspondence should be addressed to Ming Tang; m.tang@126.com

Received 16 September 2022; Revised 11 January 2023; Accepted 24 February 2023; Published 13 April 2023

Academic Editor: Jinwei Wang

Covert channels based on the CPU front-end typically utilize time or power differences caused by contention. This requires that the sender and receiver are located on the same and quiet physical core. We propose one channel exploiting the macro-op fusion in the front-end, called MFCC, which has cross-core and robust properties. Macro-op fusion is one of the optimization strategies in x86 microarchitecture, which aims to fuse two macro-ops and decode them into a single micro-op. We reverse the constraints of macro-op fusion and find that decoders stop decoding after two macro-op fusions in one cycle. Thus time differences appear in two identical loops with different virtual addresses, represent-ing one and zero, respectively. We build two types of MFCC: the sender and receiver running in the same thread but operating at different privilege levels, and the two running in two different processes. The accuracy within a single thread is almost independent of the CPU load, while the accuracy of interprocess transmission maintains more than 0.8 even when the CPU load is 100%. Finally, we propose three possible protection strategies: eliminating the macro-op fusion mechanism, adding noise to the hardware counters, and making the events delivered at a defined point in time. This paper demonstrates that even minor optimizations in the front-end can lead to covert transmission.

1. Introduction

Covert channels use nonstandard methods to transmit information, violating system security policies. Recently, there has been renewed interest in side-channel and covert channel attacks against microarchitecture due to the rediscovery of optimization in it. The CPU microarchitecture is the underlying implementation of the instruction set architecture (ISA), providing details like pipelinization, instruction parallelism, out-of-order execution, and branch prediction. It can be divided into three parts: front-end, backend, and memory. The front-end is responsible for fetching, decoding, and delivering instructions to the backend. DSB and IDQ/LSD can store decoded instructions and deliver them to the backend directly when hitting to improve performance. The goal of the backend is to execute instructions and write the results back to memory if necessary. We will cover the front-end decoding path in Section 2.

Since the first caches attack on the AES algorithm [1], it has been attracting a lot of interest [2–5]. Spectre [6] and Meltdown [7] proposed in 2018 broke memory isolation, showing that briefly accessed secret information can be leaked to the attacker. The two attacks triggered a huge amount of innovative scientific inquiry into the backend components because the front-end will not become a bottleneck in most cases.

Early works in the CPU backend tended to target specific execution units. Wang and Lee demonstrate that processes running on the same core interfere with each other when using multiplication units, which allows an attacker to identify the victims multiplication instructions [8]. Aciicmez and Seifert go further to complete the distinction between multiplication and squaring operations and propose a side-channel attack against the RSA implementation in the OpenSSL [9]. The floating-point units can also be exploited similarly, but unfortunately these instructions are less applied in cryptographic algorithms [10]. Due to the contention of floating-point units, new speculative execution instructions may delay logically older nonspeculative execution instructions. Some researchers have thus constructed a high-bandwidth low-noise covert channel [11].

Contention between two threads is noticed after hyperthreading appears, which allows two threads to share specific components on the same physical core. When contention happens, the execution time increases. Aldaya et al. [12] and Bhattacharyya et al. [13] show how to implement side-channel attacks using the contention of the backend execution engine.

Recent trends in microarchitecture security have led to a proliferation of studies focused on the front-end decoding components, with no backend stalling guaranteed. Ren et al. [14] reverse the organization of DSB and exploit the idea that when the attacker and victim run on the same physical core; the attackers' instructions, evicted from the DSB, will differ concerning the virtual address of the victims instructions, which causes different execution times of the attacker. Deng et al. [15] demonstrate a causal relationship between decoding paths in the front-end and the programs execution time. Puddu et al. [16] show significant differences in execution time of balanced branches, blocks with the same instructions but different virtual addresses, through repeated interrupts in SGX, but it has not been able to provide a convincing explanation for the timings exploited by the attack.

However, since most of the front-end components can not be shared across cores, previous studies based on contention have been limited to transfer information on the same core and suffered from CPU noise. Noise means that another normal program may corrupt the attackers' carefully designed contention conditions. Therefore, we have designed a robust cross-core covert channel based on the front-end macro-op fusion, called MFCC.

Macro-op fusion combines two macro-op into a single one before or during decoding, which are later decoded into one fused micro-op, which also remains fused in the backend and gets executed on port 0 or 6. This technique is used in many modern microarchitectures and will be discussed more in Section 3.

Because the decoder will stop decoding after two macro-op fusions in one cycle, the sender can use the time variation to create a covert channel, and the receiver measures the time variation to receive the secret message. Slight time differences may occur in two identical loops with different virtual addresses due to different positions where they stop decoding. Our covert channel uses two identical code blocks to convey secret information to the receiver, which improves the resistance to the code audit. We propose two kinds of attacks. In the first attack within the same process where the sender and receiver operate at different privilege levels, we demonstrate our channels feasibility. In the multiprocess attack, the sender and receiver are in different processes, closer to reality.

In summary, we make the following contributions:

(1) We validate the disclosed mechanism of macro-op fusion and reverse the unknown part of macro-op fusion in the Intel Skylake microarchitecture. Our conclusions can be applied to other modern x86 microarchitectures.

(2) We design the first cross-core robust covert channel in the front-end. Based on macro-op fusion, our channel achieves more than 0.95 accuracy within a thread and 0.8 between processes when the CPU load is 100%. It can be applied to all Intel microarchitectures with macro-op fusion, from the Core 2 to the Alder Lake.

Cross-core means that the sender and receiver do not need to locate on the same core, which is required by previous works [14–16]. Robust means that the attacker does not have to carefully set up the caches status or create contention, which could get disturbed by a normal program at any time. Our channel can work when the CPU is 100% loaded.

The remaining part of the paper proceeds as follows. Section 2 includes the necessary background knowledge. We reverse restrictions on macro-op fusion in Section 3. Based on that, we propose a covert channel that works robustly and across cores in Section 4. Section 5 analyses the results of our work and discusses the possible defenses.

2. Background

The x86 decoding pipeline, i.e., the front-end, is primarily responsible for fetching x86 instructions, decoding them, and delivering the decoded micro-ops to the backend. In this section, we will introduce three decoding paths, MITE (microinstruction translation engine), DSB, and LSD, as shown in Figure 1.

2.1. MITE. Instructions are first prefetched from the L2 cache to the L1 cache. Then, up to 16 bytes of instructions can be fetched from the L1 cache per cycle to the predecode buffer, in which their boundaries and prefixes get detected and marked. After that, predecoded instructions are sent to the Instruction Queue, waiting for being decoded into micro-ops in decoders. There are one complex decoder and four simple decoders in Intel Skylake. The complex one is capable of emitting from one to four micro-ops per cycle while the simple one can only emit one micro-op. Agner argues that it has been impossible to get a throughput of more than four micro-ops per clock cycle when decoding [17]. Our hypothesis is that for every micro-ops issued by the complex decoder, one less simple decoder is activated. The micro-ops are stored in the IDQ (Instruction Decode Queue), waiting for backend execution.

2.2. DSB. DSB (decoded stream buffer) was formally introduced in the Sandy Bridge microarchitecture [1]. It can store the decoded instructions, making it possible to transfer

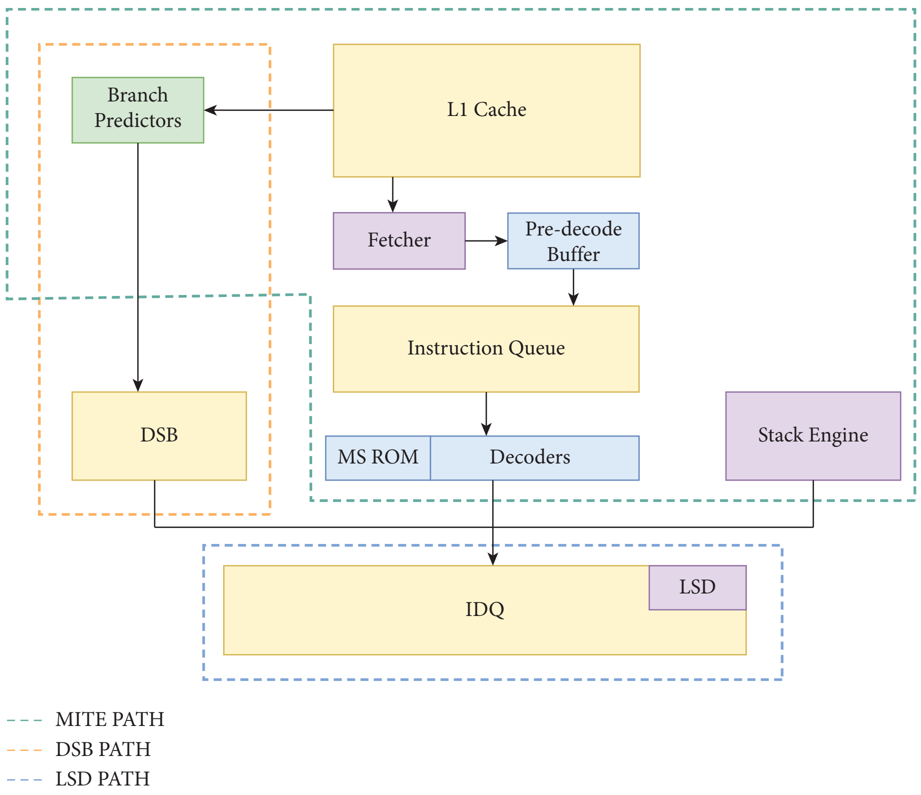


Figure 1: Structures of the CPU front-end.

them directly to IDQ when hitting. DSB can provide an average hit rate of 80%, higher in some specific cases, which reduces the impact of the complex x86 decoding [18].

2.3. LSD and IDQ. LSD (loop stream detector) detects eligible loops in the IDQ and lock them down. With the MITE and DSB turned off, these micro-ops will be streamed to the backend directly until a branch is incorrectly predicted. In the erratum SKL 150, Intel disabled the LSD on client processors through a microcode update [19].

3. Analysis of Macro-Op Fusion

In this section, we will describe how macro-op fusion causes time differences. Firstly, we verify and reverse the conditions and restrictions for performing macro-op fusion. Based on that, we complete our front-end cross-core robust covert channel. The experiments are conducted on a Skylake CPU, Intel i7-10700, with Ubuntu 20.04.

3.1. Macro-Op Fusion Causes Time Differences. Macro-Op fusion is a hardware optimization used in many modern microarchitectures. In Skylake, the decoder fuses an arithmetic or logical instruction and a subsequent jump instruction into a single compute-and-jump micro-op, which remains fused in the backend and is executed on port 0 or 6.

In the Skylake microarchitecture, macro-op fusion can be performed up to twice per cycle, after which decoders will stop decoding. Time differences may occur in two identical loops with different virtual addresses due to different positions where they stop decoding, which can be amplified by multiple iterations. As shown in Figure 2, when decoders have performed two macro-op fusions, the remaining instructions will get decoded in the next clock cycle. However, if the virtual address differs, two macro-op fusions will happen in the last two decoders and thus instructions from two iterations can get decoded in the same cycle.

We verify and reverse the mechanisms involved to confirm that this observable time difference is indeed caused by macro-op fusion.

3.2. Restrictions on Macro-Op Fusion. Only specific instruction pairs can be macrofused as Agner describes in Table 1 [17]. In addition, six conditions and restrictions affect the performance of macro-op fusion. The first one is officially given by Intel, and the third one is described in [17], which are verified in the Appendix. We reverse the remaining as follows:

(1) Macro-op fusion can be performed at most twice per cycle
(3) Decoders stop decoding after two macro-op fusions
(4) Instructions crossing the 16 byte boundary can also be macrofused
(5) Macro-op fusion may lead to a performance decrease
(6) Virtual address also affects execution time outside the SGX
(7) Macro-op fusion is performed in the decoders

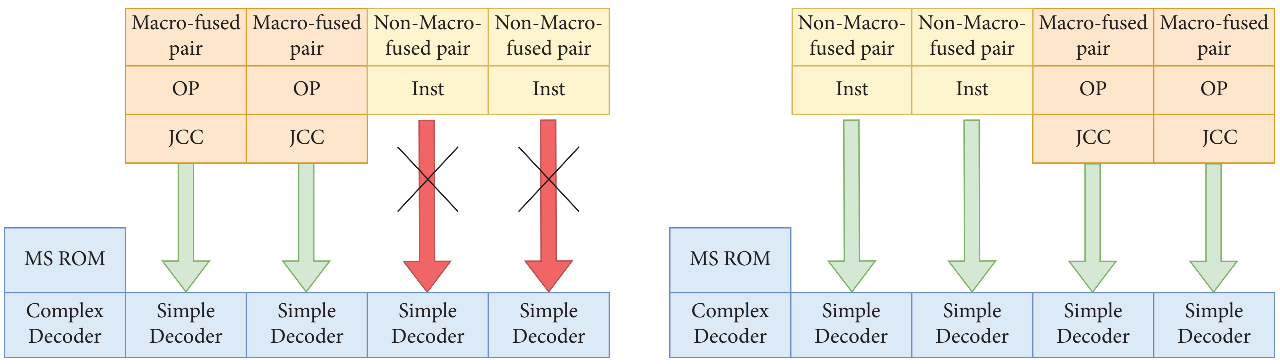


Figure 2: How macro-op fusion causes time differences. The location where the two macro-op fusions get decoded causes time differences.

Table 1: Pairs of instructions that can be macrofused. It Includes the opposite jump instructions.

The first instruction	The following instruction
cmp	*jz, jc, jb, ja, jl, jg*
add, sub	*jz, jc, jb, ja, jl, jg*
adc, sbb	None
inc, dec	*jz, jl, jg*
test	All jump instructions
and	All jump instructions
or, xor, not, neg	None
shift, rotate	None

3.2.1. Decoders Stop Decoding after Two Macro-Op Fusions. In this section, we will test the number of macro-op fusions before decoders stop working, for our covert channel needs to confirm it. It is supposed that decoders stop working after two macro-op fusions.

The test code is shown in Figure 3. In case 1, it is impossible to perform two macro-op fusions consecutively. Because of that, even if two macro-op fusions are decoded in the same cycle, at least one shift instruction is decoded at the same time. Case 2, on the other hand, performs two macrofusions consecutively, resulting in the two shift instructions will be decoded in the next cycle. Since the total length of the six instructions is less than 16 bytes, there are certain cycles in which the two shift instructions of the second test case can not be decoded together with the compare and jump instructions in the next cycle. Each case is repeated 4000 times and looped 10^6 times for a total of 2.4×10^{10} instructions per test case.

The experimental results are shown in Table 2. It can be seen that the number of cycles where micro-ops are delivered to IDQ from MITE (r1002479) is significantly lower in the first test case with similar numbers of micro-ops delivered to the IDQ from the MITE (r479). This proves that the decoder stops decoding after performing two macro-op fusions. The similar wall-clock cycle number is because the IDQ absorbs the bubbles introduced by decoders. The reason why the first case can not issue 4 micro-ops per cycle to the backend will be explained later in this section.

3.2.2. Single Macro-Op Fusion May Lead to Decoding Speed Reduction. Macro-op fusion does not necessarily mean a speed improvement. A single macro-op fusion may also

```
.test1:
%rep 4000
    test ecx, ecx
    jz .exit_loop
    shl eax, 1
    test ecx, ecx
    jz .exit_loop
    shl ebx, 1
%endrep
```

```
.test2:
%rep 4000
    test ecx, ecx
    jz .exit_loop
    test ecx, ecx
    jz .exit_loop
    shl eax, 1
    shl ebx, 1
%endrep
```

Figure 3: Test code that decoders stop decoding after two macro-op fusions.

Table 2: Number of active decoder cycles (r1002479) and number of transmitted instructions (r479).

Test cases	Cycles	r1002479	r479
#1	8, 014, 691, 739	5, 354, 685, 262	16, 002, 858, 134
#2	8, 014, 342, 810	6, 013, 800, 242	16, 002, 095, 349

lead to a throughput decrease. That is why the first test case in Figure 3 can not issue 4 micro-ops per cycle.

There are three test cases in Figure 4, where test and *jz* instructions are macrofused, and any jump instructions are not macrofused. In the 2 MF case, there are two pairs of *test* and *jz* instructions, which means macro-op fusions happen twice. The 1 MF case includes one combination of test and *jz* instructions, and one combination of or and *jz* instructions. With two or-*jz* pairs, macro-op fusion will not happen in the 0 MF case. Each case is repeated 40 times and looped 10^6 times.

The number of decoded instructions per cycle (DIPC) can be calculated from the number of cycles where micro-ops are delivered to IDQ from MITE (r1002479) and the number of micro-ops delivered to IDQ from MITE (r479).

$$\mathrm{DIPC} = \frac{r479}{r1002479}. \quad (1)$$

Table 3 presents the experimental results on decoders. There is a clear trend of increasing DIPC as the number of macro-op fusions decreases, resulting in a rise in cycles. The possible reasons for the lower decoding will be discussed later in this section.

.2MF:	.1MF:	.0MF:
%assign i 0	%assign i 0	%assign i 0
%rep 40	%rep 40	%rep 40
.test%+i:	.test%+i:	.test%+i:
add eax, 0×123	add eax, 0×123	add eax, 0×123
add ebx, 0×23	add ebx, 0×23	add ebx, 0×23
test ecx, ecx	test ecx, ecx	or ecx, ecx
jz .test %+ i	jz .test %+ i	jz .test %+ i
test ecx, ecx	or ecx, ecx	or ecx, ecx
jz .test %+i	jz .test %+ i	jz .test %+ i
%assign i i+1	%assign i i+1	%assign i i+1
times 32 nop	times 32 nop	times 32 nop
%endrep	%endrep	%endrep

FIGURE 4: Test code that single macro-op fusion may lead to decoding speed reduction.

TABLE 3: DIPC of groups with different macro-op fusion numbers.

Test cases (MF)	DIPC
2	3.59322436
1	3.69357949
1	3.79700302

3.2.3. Impact of Virtual Addresses on Execution Time. The work [14] explores the influence of virtual addresses on the number of fetched instructions and further on the execution time. However, they have failed to observe it outside the SGX, maybe suffering from DSB hits. In our experiments, the effect of virtual addresses on the execution time is also observed outside of SGX, and macro-op fusion amplifies it.

We use nop instructions to adjust the virtual addresses of test codes in Figure 4. Every test case is repeated 40 times and looped 10^6 times. With the change of virtual addresses, the cycles represent a trend of periodic change outside of SGX, as shown in Figure 5(a). The number of macro-op fusions does not change the overall trend of cycles but amplifies the effect of the virtual address on cycles. The 2 MF test case has lower cycles when macro-op fusions can be performed normally, and higher cycles when decoding is stopped. It is important to note that even at the virtual addresses where the three curves overlap, we can still get distinguishable the time differences.

As shown in Figure 5(b), the increase of virtual addresses also leads to periodic fluctuations in DIPC. The three test cases at the overlapping point in Figure 5(a) have different execution times because of the different DIPCs. This is one of the advantages of macro-op fusion to MFCC that it amplifies the effect of the virtual address on the execution time. The greater time difference allows us to better distinguish between 0 s and 1 s.

3.2.4. Location where Macro-Op Fusion is Performed. The component where macro-op fusion is performed is not yet determined. It should not be earlier than the predecode buffer, where the CPU knows the exact instruction length, or later than the decoders, whose output is already micro-ops.

One view holds that macro-op fusion takes place in the IQ (instruction queue), where two macro-ops are fused into one complex macro-ops [20], as shown in Figure 6. There is also a view that the macrofusion is carried out in the decoders [17].

Based on the conclusion earlier in this section that macro-op fusion may also lead to a throughput decrease, we propose a possible deduction that if one instruction can be macrofused into the last decoder; its decoding is delayed and enters the first decoder in the next cycle to check whether the next one is a jump instruction that can be macrofused. This is consistent with the conclusion that DIPC rises as the number of macro-op fusions decreases.

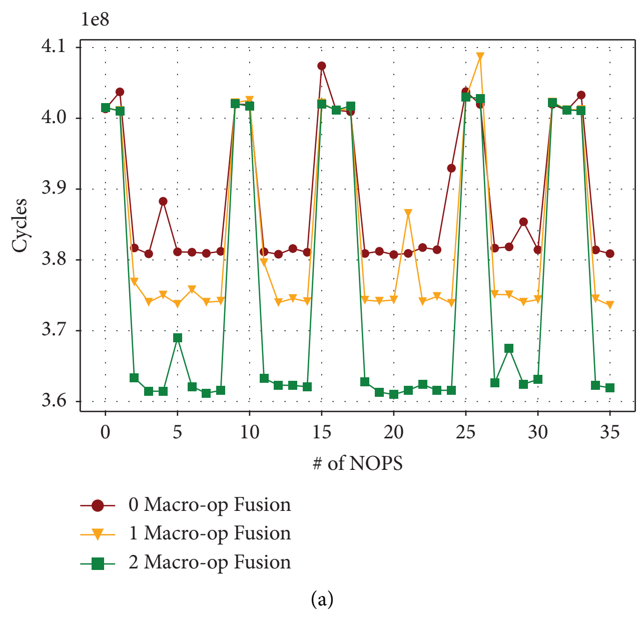
Based on the conclusion that instructions crossing the 16 byte boundary can also be macrofused without any throughput loss, we infer that if any instruction that can be macrofused is stored in the IQ while the next one is still in the predecode buffer, then the instruction should wait and enter decoders in the next cycle, which conflicts with the conclusion that the throughput will not decrease.

Thus, we believe that the macro-op fusion is performed in the decoders.

4. Covert Channel Based on Macro-Op Fusion

4.1. Design of the Covert Channel. Since most of the front-end components can not be shared across cores, previous studies [14–16] based on contention have been limited to transfer information on the same core and suffered from CPU noise. Noise means that another normal program may corrupt the attackers' carefully designed contention conditions. The feature of stopping decoding after two macro-op fusions enables us to construct a covert channel, MFCC (the macro-op fusion covert channel), using two identical loops with different virtual addresses, which has overcome such shortcomings.

For the threat model, we assume that on the same machine, the sender and receiver are both malicious. The sender holds secrets but has no access to the Internet and the receiver tries to extract the secret information by measuring timing changes and later transmit secrets to the attacker somehow. Our work focuses on the covert channel where the sender and receiver communicate. The sender and receiver need to get a timestamp and start at a specific time point. We



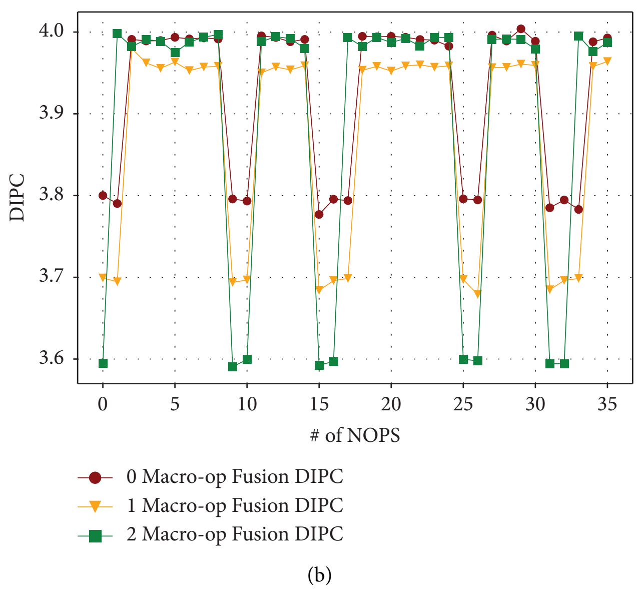


FIGURE 5: Impact of virtual addresses on execution time. (a) Execution cycles under different virtual address. (b) DIPC under different virtual address.

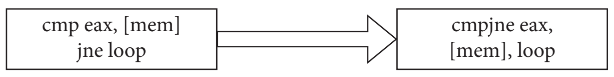


FIGURE 6: Two macro-ops are fused into one complex macro-ops.

demonstrate vulnerabilities in both single-thread settings where the two run in different privileges, and cross-core settings. We assume that coresidency on the same server can be achieved by using methods proposed by [21]. Thus, MFCC can be effective in the cloud environment as many previous attacks [3, 4] do.

Figure 7 explains the structure of MFCC, where the sender has a secret. The number of nop instructions is used to adjust the virtual address of the code block, according to message 0 or 1. The two add instructions and two cmp-and-jmp pairs are 16 bytes in length, equal to the length of the Fetcher. A prefix like 66H may influence the decoding throughput. Thus, we do not use registers like AX and BX. 32 nop instructions are used to make the MITE throughput the only bottleneck because they need to get decoded but do not execute in the backend ports. Code block A and Code block B send 1 and 0 respectively.

Each block is executed 10^6 times, collecting a total of 1000 points. The distribution of 1000 points is shown in Figures 8(a) and 8(b). In milliseconds, the two blocks can be roughly distinguished; while in finer granularity, the number of cycles, the two blocks can be completely distinguished.

In MFCC, we use execution time to distinguish them, avoiding root privileges. We use a shared timestamp to synchronize the sender and receiver. In a more practical scenario, our channel can use techniques proposed by [22–25] independent of the common clock. Every child thread of the sender transmits only 1 bit. Thus, the receiver measures the runtime of each thread to distinguish the 0 s and 1 s.

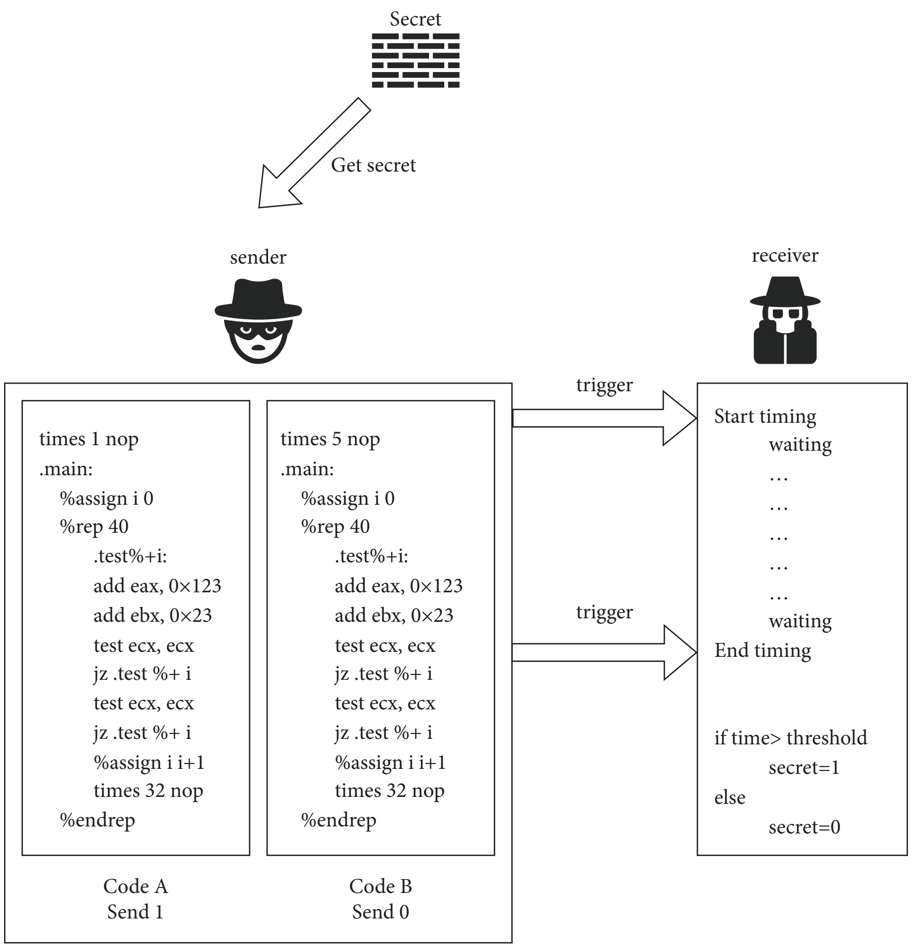
4.2. Parameter Analysis. We build two types of MFCC: the sender and receiver running in the same thread but operating at different privilege levels, and the sender and receiver running in two different processes. In experiments, the message is a 10000 bits string.

4.2.1. Number of loops. Each loop creates a difference in execution time between Code block A and Code block B. The higher number of loops, the less difficult it is to distinguish between the two. Results are shown in Table 4. As the number of loops increases, the accuracy tends to increase while the speed drops.

The number of loops represents a trade-off between accuracy and speed. Comparing the results of two code blocks, the speed and accuracy are impressive when transmitting within the same thread, because there is no overhead of cloning processes and less noise interference.

4.2.2. CPU Load. Most studies in the field of the microarchitecture covert channel have focused on exploiting the competition between the sender and receiver for some shared resources. This leads to problems that channels can not transmit beyond the resource hierarchy and hardly resist noise in the real world. The 1 bit secret is sent on the eviction of the DSB in [16], which means the sender and receiver have to locate at the same core. When another process switches to the core where they are located, the well-laid-out states of DSB will become disorganized. However, CPU load has a much smaller impact on MFCC since our channel does not depend on the competition and has a low requirement for threshold.

We use look-busy [26] to impose CPU load on MFCC, which takes various approaches to authorize the CPU to



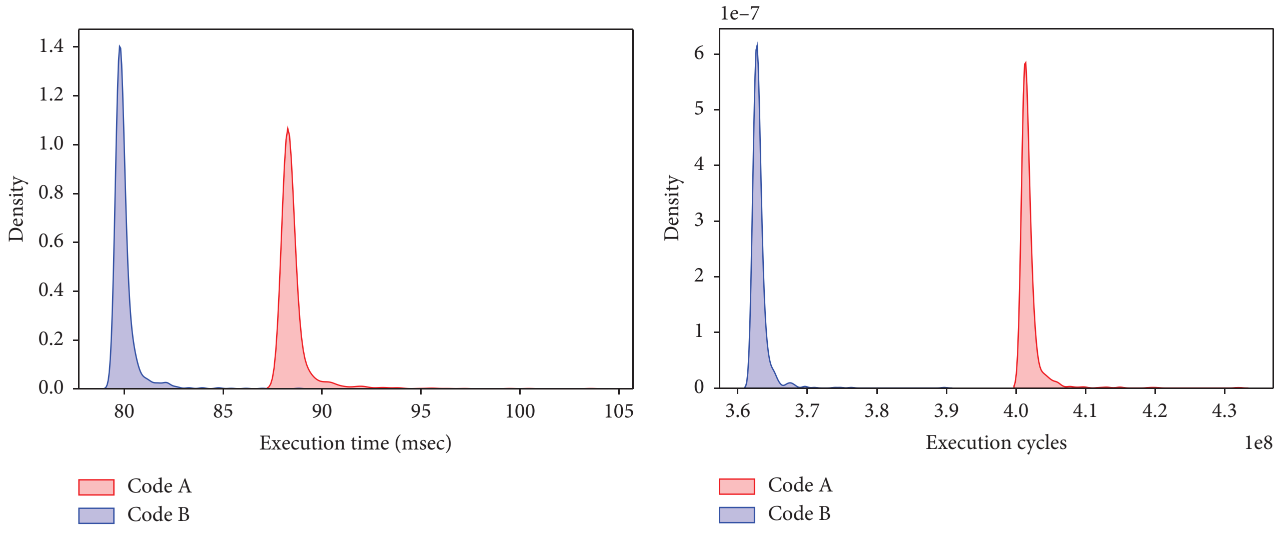
```
times 1 nop
.main:
  %assign i 0
  %rep 40
      .test%+i:
      add eax, 0×123
      add ebx, 0×23
      test ecx, ecx
      jz .test %+ i
      test ecx, ecx
      jz .test %+ i
      %assign i i+1
      times 32 nop
  %endrep
```

```
times 5 nop
.main:
  %assign i 0
  %rep 40
      .test%+i:
      add eax, 0×123
      add ebx, 0×23
      test ecx, ecx
      jz .test %+ i
      test ecx, ecx
      jz .test %+ i
      %assign i i+1
      times 32 nop
  %endrep
```

```
Start timing
      waiting
      ...
      ...
      ...
      ...
      ...
      waiting
End timing


if time> threshold
      secret=1
else
      secret=0
```

Figure 7: Schematic diagram of MFCC.



(a) (b)

Figure 8: Distinguishability of code block A and B.

operate at a fixed CPU load. As shown in Figure 9, the accuracy within a single thread is almost independent of CPU load, while in interprocess transmission, the accuracy gradually decreases with the increase of CPU load. However, the accuracy of interprocess transmission maintains more than 0.8 even when the CPU load is 100%.

Table 4: Speed and accuracy under different numbers of loops.

Loop count	#1		#2	
	Speed (kbps)	Accuracy	Speed (kbps)	Accuracy
100	21.645	0.7034	112.36	0.9979
500	12.315	0.9339	23.585	0.9927
1000	8.104	0.9125	11.834	0.9588
10000	1.097	0.9778	1.191	0.996
100000	0.118	0.9939	0.119	0.9996

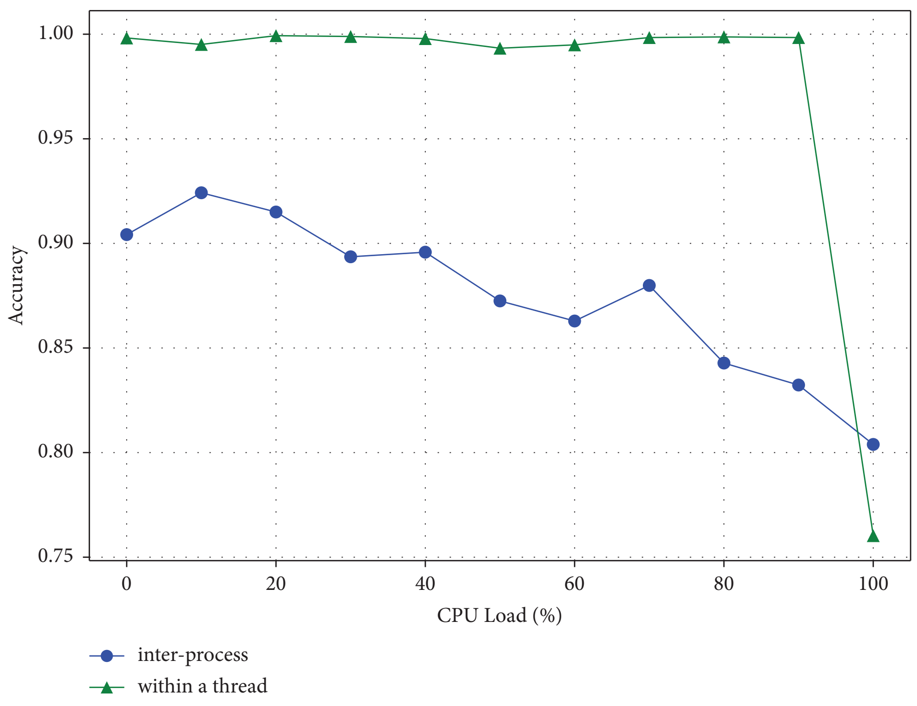


Figure 9: Impact of CPU load on accuracy.

When a CPU is heavily loaded, there must be many processes colocated with our sender and receiver. However, since the decoders are not shared between threads or processes, another process will not affect how the decoders work for our sender and receiver. It only makes the receiver run longer. In such a scenario, the runtime of the sender will increase as a whole whether it sends 0 s or not, but it is still distinguishable as stated in 4.2.

The difference in accuracy between the two is mainly due to the fact that inter-process communication requires cloning processes, which consumes some time and results in a smaller time difference between 1 s and 0 s.

4.2.3. Threshold for Distinguishing 1 s and 0 s.

The choice of threshold is theoretically dependent on noise and the number of loops, which can bring a significant effect on the accuracy.

The experimental results are shown in Figure 10. When the loop count is 1000, the threshold is incremented from 200000 to 300000 in steps of 1000. It can be observed that the accuracy remains around 0.5 at the beginning, which means that the threshold cannot distinguish between 1 s and 0 s at all and starts to increase rapidly around 17.8, reaching the maximum accuracy of 17.95 and then decreases rapidly. It can also be found that we have a wide range of threshold selection, which is advantageous in dealing with high CPU load. Different numbers of loops will have different ranges of the threshold. The noise will only have a direct impact on the accuracy, and it is not necessary to modify the threshold under a specific CPU load.

5. Discussions

5.1. Results Discussions.

In this section, we will evaluate the experimental results, including a comparison with other similar works, discussions about shortcomings, and possible improvements.

5.1.1. Speed Comparison.

MFCC can achieve a speed of 857.46 kbps with an error rate of about 5.85% for transmission within a thread across privilege levels, and 12.315 kbps with a BER of 6.61% for interprocess transmission.

Our work does not have a significant advantage in speed compared to other front-end covert channels. The speed of a single thread is roughly comparable, while the interprocess communication is slower, as shown in Table 5.

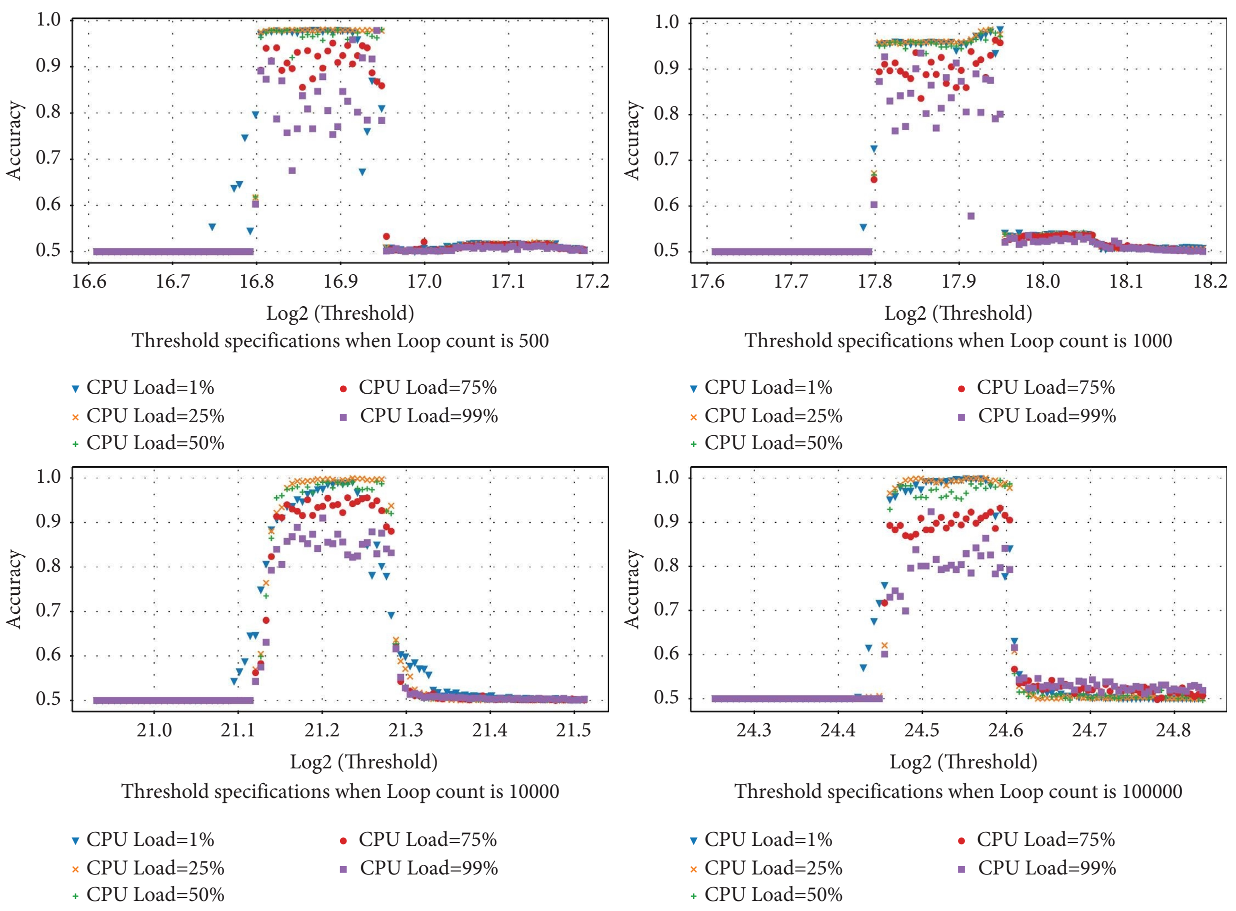


Figure 10: Impact of CPU load and loop count on threshold and accuracy.

Table 5: Transmit speed comparison. MT is a multithreaded transmit and MP is a cross-process transmit.

Work	Speed (kbps)	Error (%)	Cpu model
Leaky frontends (MT) [15]	200.37	4.62	Intel E-2286G
Leaky frontends (MT) [15]	1410.84	0.00	Intel E-2286G
Dead uops (MT) [14]	250	5.59	Intel i7-8700T
Dead uops [14]	965.59	0.22	Intel i7-8700T
MFCC (MP)	12.315	6.61	Intel i7-10700
MFCC	857.46	5.85	Intel i7-10700

Although the bandwidth is in the middle level among attacks reported in the literature [27], our covert channel can transmit cross cores and under high CPU load, which enables us to work in the real world, as discussed in 4.2.

5.1.2. Shortcomings and Improvements. We do not find a convenient handshake in the communication between processes to guarantee synchronization between the receiver and sender. Thus we take the most primitive way in that the receiver starts timing when the cloning is carried out and ends timing when the sender is released. This is very straightforward and efficient at the cost of a speed drop. In Table 4, the speed difference between intrathread and interprocess communication reaches about 40 times, mostly caused by cloning. The speed of MFCC can be further improved if we can find a more suitable way to handshake.

In Section 4.2, we discussed the change in accuracy caused by the threshold. But the threshold to distinguish 1 s and 0 s may be different among different devices. Thus we propose the concept of dynamic adaptive thresholds. Before transmitting secrets, the sender and receiver use a piece of known information to determine the best threshold. When the accuracy reaches 0.9, then secrets start to transmit. This allows for better adaptation to different CPU microarchitectures.

Moreover, since our MFCC does not depend on any shared resources, concurrent transmission of secrets is feasible. By having multiple senders transmit a part of secrets, the speed can increase significantly between processes. If requiring high-level accuracy, secrets can be sent by multiple senders.

```
%assign i 0
%rep 10
  %rep 8
    .test%+i:
      test ecx, ecx
       jz .test%+ i
       %assign i i+1
     %endrep
    times 24 nop
  %endrep
```

Figure 11: Test code that at most two macro-op fusions happen in one cycle.

Table 6: Macro-op fusion can be performed at most twice per cycle.

Performance monitor unit	Result
Cycles	10, 181, 821, 638
Instructions	40, 198, 387, 276
r10e	32, 093, 761, 570
r1b1	8, 100, 347, 112
r1002479	10, 100, 951, 100
r479	32, 108, 844, 511

```
Align 32 .test:
    %assign i 0
    times 14 nop
    %rep 4000
        .test%+i:
        times NOPS1 nop
        ecx, ecx
        jz .test%+i
        times NOPS2 nop
        %assign i i+1

%endrep
```

Figure 12: Test code that crosses the 16 byte boundary of the fetcher can also be macro-fused.

Table 7: Instructions across the 16 byte boundary of the fetcher can be macrofused. r1002479 is the number of micro-ops are delivered to IDQ from MITE. r479 is the number of micro-ops delivered to the IDQ from the MITE.

Test case	Cycles	Instructions	r1002479	r479
NOPS1 = 0, NOPS2 = 12	13, 043, 078, 943	56, 016, 000, 820	13, 013, 118, 488	52, 019, 903, 856
NOPS1 = 2, NOPS2 = 10	13, 052, 102, 429	56, 016, 000, 821	13, 010, 725, 499	52, 016, 153, 156

The disputed bits are voted or retransmitted through majority voting, which can improve accuracy without speed loss.

5.2. Possible Mitigations. We find that existing defenses or ideas can protect against MFCC at the cost of performance degradation.

The most intuitive mitigation would be to remove the macro-op fusion. There was a similar situation before that in the erratum SKL 150, Intel disabled the LSD on client processors through a microcode update [28]. This will bring about a 10% performance degradation as the difference between 2 macro-op fusions and 0 macro-op fusion in Figure 5(a) indicates.

One possible approach is to add noise to the hardware counters. Our channel relies on the performance counter returning an accurate timestamp, which is used to calculate the execution time and thus distinguish between 1 s and 0 s. However, in addition to hardware counters, it is possible for networked programs to obtain reliable timestamps from the network. Hu [29] proposed to modify the IO buffers to support delayed virtual interrupts for the purpose of adding noise on all available clock sources.

Another possible approach is to make all events delivered at deterministic time points [30, 31]. This poses a greater challenge to the hardware implementation and there is a performance loss. In a virtualized environment, the difficulty is reduced because of the hypervisor.

Overall, it is important to achieve a delicate balance between security and performance as performance decreases in exchange for security improvements.

6. Conclusion

This study set out to develop a covert channel that transmits robustly and across cores. We verify and reverse the restrictions on macro-op fusion and construct one based on that. To resist code audit, our channel uses two identical loops to transmit secrets. MFCC not only works across cores without sacrificing the speed of transmitting secrets but also shows good accuracy under high CPU load, which has not been accomplished by any previous front-end covert channels. It can achieve a transmission speed of 857.46 kbps with an error rate of about 5.85% within a thread, and 12.315 kbps with an error rate of 6.61% in interprocess transmission. We propose three possible protection strategies: eliminating the macro-op fusion mechanism, adding noise to the hardware counters, and making the events delivered at a defined point in time. Moreover, it is possible to exploit micro-op fusion in the similar way.

Appendix

In this section, we verify one condition and one limitation of macro-op fusion, which are already disclosed.

Macro-op fusion can be performed at most twice per cycle. We will test the maximum number of macro-op fusions that can be performed in one cycle in the Skylake microarchitecture. The code used to perform the test is shown in Figure 11, where the test and *jz* instructions are macrofused. Nop instructions are executed quickly on the backend and do not reach the execution ports, which makes the front-end decoding the only bottleneck and reduces the DSB usage. The whole case is looped 10^9 times in total.

Results are shown in Table 6. The DIPC of this code block is about 3.179, which is below the upper limit of 4 micro-ops per cycle. According to the assumption that at most two macro-ops fusions can be performed per cycle, a code block with 8 combinations of test and *jz* needs 4 cycles to complete decoding and 24 nop instructions need 6 cycles to complete decoding. Thus, 32 microinstructions (one test-*jz* pair produces one micro-op) need 10 cycles in total to complete decoding. The theoretical DIPC is 3.2, which is close to the experimental results.

The r10e is the number of micro-ops sent by the RAT to the RS, which decreases by 8 × 10^9. It proves that macro-op fusions have been performed in the front-end. The r1b1 measures the number of micro-ops actually executed. It proves that nop instructions are not executed in the backend ports. The number of micro-ops executed per cycle, calculated by r1b1 and cycles, is about 0.8, which means no stalling in the backend. This ensures the validity of the conclusion.

Instructions that cross the 16 byte boundary of the fetcher can also be macrofused. The purpose of this section is to verify the conclusion that instructions across the 16 byte boundary of the fetcher can be macrofused, with no decrease in speed. The test case is shown in Figure 12. Results are shown in Table 7. The virtual addresses are aligned at the beginning of the test cases using ALIGN 32. In the first one, NOPS1 = 0 and NOPS2 = 12; in the second one, NOPS1 = 2 and NOPS2 = 14. Test and *jz* instructions are both 2 bytes long, so the total length of the test cases is 16 bytes, which is just the length of the fetcher. The only difference is that the test and *jz* instructions in the first case will cross the 16 byte boundary while the second one will not. Each case is repeated 4000 times and looped 10^6 times, for a total of 5.6×10^{10} instructions per case. In both cases, the difference between the total number of instructions and micro-ops transmitted from the decoder to IDQ is about 4×10^9, which proves that the two instructions crossing the 16 byte boundary of the fetcher can be macro fused. In addition, there is no significant difference in cycles between the two, which proves that there is no impact on the decoding speed.

Data Availability

We believe that our original data can be reproduced using the code listings provided by the paper. The basic and necessary data are listed in the paper. The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the Science and Technology Project of State Grid Corporation of China, Research on Key Technologies of High Security for Embedded CPU Core (Grant no. 5700-202041394A-0-0-00).

References

[1] Intel, "Intel 64 and IA-32 architectures software developers manual," *System Programming Guide*, Vol. 3B, Intel, Santa Clara, CA, USA, 2010.

[2] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: defeating cache side-channel protections with TLB attacks," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, USA, August 2018.

[3] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ Flush: a fast and stealthy cache attack," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Berlin, Germany, June 2016.

[4] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel at- tacks are practical," in *Proceedings of the 2015 IEEE symposium on security and privacy*, San Jose, CA, USA, May 2015.

[5] Y. Yarom and K. Falkner, "FLUSH+ reload: a high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX security symposium (USENIX security 14)*, San Diego, CA, USA, August 2014.

[6] P. Kocher, J. Horn, A. Fogh et al., "Spectre attacks: exploiting speculative execution," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, July 2019.

[7] M. Lipp, M. Schwarz, D. Gruss et al., "Meltdown: reading kernel memory from user space," in *Proceedings of the 27th*

USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, August 2018.

[8] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Proceedings of the 2006 22nd Annual Computer Security Applications Conference (ACSAC06)*, Miami Beach, FL, USA, December 2006.

[9] O. Aciicmez and J.-P. Seifert, "Cheap hardware parallelism implies cheap security," in *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*, Vienna, Austria, September 2007.

[10] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May 2015.

[11] J. Fustos, M. Bechtel, and H. Yun, "SpectreRewind: leaking secrets to past instruc- tions," in *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*, New York, NY, USA, November 2020.

[12] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*, Francisco, CA, USA, May 2019.

[13] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner et al., "Smotherspectre: exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, London, UK, November 2019.

[14] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead uops: leaking secrets via Intel/AMD micro-op caches," in *Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architec- ture (ISCA)*, Valencia, Spain, June 2021.

[15] S. Deng, B. Huang, and J. Szefer, "Leaky frontends: security vulnerabilities in pro- cessor frontends," 2021, https://arxiv.org/abs/2105.12224.

[16] I. Puddu, M. Schneider, M. Haller, and S. apkun, "Frontal attack: leaking control- flow in SGX via the CPU frontend," in *Proceedings of the 30th USENIX Security Sym- posium (USENIX Security 21)*, Berkeley, CA, USA, August 2021.

[17] F. Agner, "The microarchitecture of intel, amd, and via cpus: an optimization guide for assembly programmers and compiler makers," 2021, https://www.agner.org/optimize/microarchitecture.pdf.

[18] Intel, "Intel 64 and IA-32 architectures software developers manual," *System Programming Guide*, Vol. 3B, Intel, Santa Clara, CA, USA, 2021.

[19] "Mitigations for jump conditional code erratum, ," , 2019.

[20] Intel, "Skylake (Client) - Microarchitectures - Intel - WikiChip," 2021, https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client.

[21] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. Swift, "A Placement Vulnerability Study in Multi-Tenant Public Clouds," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C, USA, August 2015.

[22] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Ti- wari, "Understanding contention-based channels and using them for defense," in *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, February 2015.

[23] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: high-speed covert chan- nel attacks in the cloud," in *Proceedings of the 21st USENIX Security Symposium (USENIX Security 12)*, Bellevue, WA, USA, August 2012.

[24] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *Proceedings of the 25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, USA, January 2016.

[25] C. Maurice, M. Weber, M. Schwarz et al., "Hello from the other side: SSH over ro- bust cache covert channels in the cloud," in *Proceedings of the 2017 Network and Distributed System Security Symposium. Internet Society*, San Diego, CA, USA, January 2017.

[26] Devin, "Lookbusy – a synthetic load generator," 2022, http://www.devin.com/lookbusy/.

[27] J. Szefer, *Survey Of Microarchitectural Side And Covert Channels, Attacks, And Defenses*, Yale University, New Haven, CT, USA, 2016.

[28] Intel, "Intel coorporation: 6th generation Intel processor family - specification update,".

[29] W. M. Hu, "Reducing timing channels with fuzzy time," *Journal of Computer Security*, vol. 1, no. 3-4, pp. 233–254, 1992.

[30] A. Aviram, S. Hu, B. Ford, and R. Gummadi, "Determinating timing channels in compute clouds," in *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, Chicago, IL, USA, October 2010.

[31] W. Wu and B. Ford, "Deterministically deterring timing attacks in deterland," 2015, https://arxiv.org/abs/1504.07070.