Differential Analysis and Fingerprinting of ZombieLoads on Block Ciphers

Till Schlüter[1] and Kerstin Lemke-Rust[1]

Bonn-Rhein-Sieg University of Applied Sciences, Sankt Augustin, Germany
till.schlueter@smail.inf.h-brs.de, kerstin.lemke-rust@h-brs.de

Abstract. Microarchitectural Data Sampling (MDS) [16, 18] enables to observe in-flight data that has recently been loaded or stored in shared short-time buffers on a physical CPU core. In-flight data sampled from line-fill buffers (LFBs) are also known as "ZombieLoads" [16]. We present a new method that links the analysis of ZombieLoads to Differential Power Analysis (DPA) techniques and provides an alternative way to derive the secret key of block ciphers. This method compares observed ZombieLoads with predicted intermediate values that occur during cryptographic computations depending on a key hypothesis and known data. We validate this approach using an Advanced Encryption Standard (AES) software implementation. Further, we provide a novel technique of *cache line fingerprinting* that reduces the superposition of ZombieLoads from different cache lines in the data sets resulting from an MDS attack. Thereby, this technique is helpful to reveal static secret data such as AES round keys which is shown in practice on an AES implementation.

Keywords: ZombieLoad · side channel attack · microarchitectural data sampling (MDS) · differential analysis · DPA · block cipher · AES · cache line fingerprinting.

1 Introduction

Many attacks targeting optimization mechanisms of modern CPUs have been published in recent years. A subset of those, Microarchitectural Data Sampling (MDS) attacks, are applicable to the majority of Intel CPUs sold in the last decade [5], affecting a significant number of personal computers and servers worldwide. ZombieLoad [16] is one representative from this class. It allows an attacker to observe data from memory lines that have recently been loaded or stored at the time of the attack on the same physical core. An application of this instrument to cryptographic implementations is the *domino attack* that reconstructs the secret key from an OpenSSL implementation of the AES [16].

In this paper, we investigate further methods to extract secret keys from block cipher implementations using ZombieLoad primitives. First, we present a new differential technique by considering that ZombieLoads can also stem from intermediate computational results of a block cipher. We recover the key

by linking ZombieLoads with analysis techniques originating from Differential Power Analysis (DPA): we predict intermediate results of the cryptographic implementation and use a statistical analysis to determine the key. Because of this, protecting the secret key, for example by transferring it as rarely as possible or in obfuscated form, might not be sufficient to protect cryptographic implementations from MDS leakage. Second, we provide cache line fingerprinting as a useful tool to reliably associate ZombieLoads to their original static byte sequence within a memory line.

This paper is structured as follows: We give an introduction into transient execution and cache leakage, MDS attacks with focus on ZombieLoad, and DPA in section 2. Next, we make the following contributions: In section 3, we transfer the idea of analyzing intermediate results of a cryptographic algorithm from DPA into the domain of MDS post-processing. We present the general analysis procedure as well as a case study to an AES implementation. In section 4, we further introduce an independent tool called *cache line fingerprinting* that makes it easier to assign leaked bytes to their originating cache lines. We show that this tool is suitable to leak AES round keys in an exemplary implementation.

2 Preliminaries

2.1 Transient Execution and Cache Leakage

The publication of Spectre and Meltdown [8, 9] introduced a new class of side-channel attacks that use effects of microarchitectural optimization techniques of modern CPUs to leak data across privilege boundaries. These attacks make use of specially crafted *transient instructions*, i.e., instructions that are erroneously executed due to false predictions or out-of-order processing. Following this observation, many further transient execution attacks targeting different microarchitectural structures have been published in recent years [3].

Transient execution attacks often influence microarchitectural structures like caches in a targeted manner to encode information into them: An attacker process allocates a so-called *probe array* in memory and ensures that it is not cached. The size of the probe array is often set to $256 \cdot p$ bytes, where p is the page size in bytes, e.g., $p = 4096$. This array is suitable to encode the value v of a single byte into the cache by transiently accessing index $v \cdot p$. When an element of the probe array is accessed by a transient instruction, the memory line containing this array element is loaded into the cache and remains cached even when the CPU detects the erroneous execution and discards all speculative results [9].

Depending on the microarchitectural implementation, some security checks are not yet performed during transient execution. Thus, an attacker can use transient instructions to perform unauthorized load requests to otherwise inaccessible memory regions and extract them using the cache side-channel [9].

2.2 Microarchitectural Data Sampling (MDS) and ZombieLoad

Microarchitectural data sampling attacks extract in-flight data of concurrent processes, e.g., values that are loaded from or stored to memory in close temporal

proximity to the attack. MDS attacks published to date target different microarchitectural structures to extract data from. They focus on store buffers [2], fill buffers [16, 18], or structures concerned with bus snooping [4]. Further improvements include targeted attacks on attacker-specified memory regions [17] and attacks that overcome the common precodition that victim and attacker share the same physical CPU core [14].

In this paper, we mostly build upon the findings of the ZombieLoad publication [16]. ZombieLoad allows to observe values loaded from (or stored in) memory at the time of the attack across logical cores. It presumably leaks values that are present in line-fill buffers (LFBs), an intermediate stage between the L1 cache and higher-level caches. LFBs are shared among all processes executed on the same physical CPU core, including processes that are concurrently executed on the same physical but different logical core. On vulnerable CPUs, memory load operations are not canceled immediately when a fault (e.g., a page fault) occurs. Instead, the load may speculatively be answered with a value from an LFB. Until the fault is handled and the result of the load request is discarded, the CPU may execute additional transient instructions that operate on stale values from an LFB and leak them through a cache-based side-channel. Leaking data may stem from concurrent user-space applications, the kernel, SGX enclaves, hypervisors or virtual machines running on sibling cores [16].

Attack variants. There are multiple ZombieLoad variants that differ with regard to how a faulting load instruction is created. In this paper, we use variants 1 and 2. Varaint 1 is closely related to the Meltdown attack, abusing transient instructions after a page fault. Variant 2, also knwon as TSX Asynchronous Abort (TAA), makes use of insufficient transient error handling in the transactional memory implementation when a memory conflict occurs. Both variants are well suited for attacks among user space processes on Linux systems [16].

Further ZombieLoad variants use properties of SGX, uncacheable memory regions or page-table walks [16]. A similar approach to leak data from LFBs is Rogue In-Flight Data Load (RIDL) [18] which uses page faults that occur due to demand paging.

Domino Attack. The domino attack is a ZombieLoad case study to leak the key from an AES implementation [16]. To leak a 16-byte (round) key, the key has to be observed multiple times and extracted byte-by-byte. First, the attacker samples bytes from varying positions in arbitrary LFB entries. To connect these single bytes to chains that frequently appear together (and therefore are key candidates), so-called *domino bytes* are leaked in addition. A domino byte connects two neighboring bytes to each other: it consists of four bits from the first byte and four bits from the second byte. Domino bytes are leaked through the same cache side-channel as previous data bytes, but in a separate ZombieLoad iteration. Finally, the attacker searches the sampled data for chains of 16 consecutive bytes that are backed by both bytes from LFB entries and domino bytes to find AES key candidates [16].

2.3 Differential Power Analysis and its Application to White-Box Implementations

DPA [7,10] exploits the fact that processing an algorithm on a hardware platform causes (noisy) physical side-channel leakage such as data-dependent power consumption over time. If a cryptographic algorithm is considered, the side-channel leakage can include information about secret cryptographic keys.

A DPA attacker samples the power consumption of a circuit over time and thereby obtains a high number of noisy sampled data per execution of an algorithm, i.e., a DPA trace. As DPA is a statistical attack, many repetitions of algorithm execution are carried out using varying known input (or output) data to the cryptographic implementation. For analysis, the attacker predicts the intermediate state of the cryptographic implementation and thereby derives the expected power consumption in a hypothetical model dependent on known input (or output) data and a key hypothesis using a divide-and-conquer approach. If the model is suited and leaks exist, the predicted values show a high degree of correlation with the measured samples for the correct key. The model is typically applied to internal intermediate state values that occur at the beginning (or the end) of the cryptographic computation.

Adaptations of this attack on a white-box software implemenation used Dynamic Binary Instrumentation (DBI) tools to record traces of memory transactions that are comparable to noise-free DPA traces and can be analyzed in a similar way [1].

3 Differential Analysis of ZombieLoads

3.1 System and Attacker Model

System Model. A victim process runs a block cipher that repeatedly encrypts or decrypts a given input with a fixed key. The input varies periodically. We assume that the block cipher computations are performed in software on a CPU that is vulnerable to an MDS attack.

Attacker Model. An attacker program runs concurrently to the victim process on the sibling core and constantly collects samples using an MDS attack, e.g., ZombieLoad. We assume that the attacker knows the inputs or outputs (plaintexts or ciphertexts) as well as the used block cipher. The attacker has the capability to freely choose the byte index within the LFB entry from which the next sample is leaked. Assuming that the LFB consists of N_b bytes, the attacker maintains N_b sampling pools for each known plaintext (or ciphertext). Let N be the number of different plaintexts (or ciphertexts), each sampling pool is indexed with the pair (i, j) with $0 \leq i < N_b$ and $0 \leq j < N$.

3.2 Attack Procedure

The differential attack is divided into two phases, sampling and analysis. These can optionally be preceded by a profiling phase.

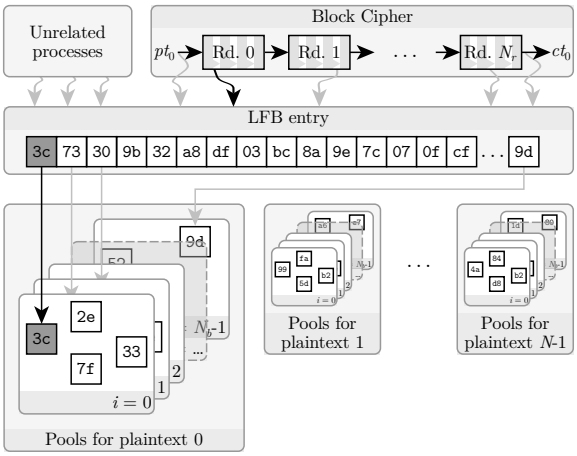

Fig. 1. Sampling process for differential analysis

Sampling. Figure 1 illustrates the data flow during sampling. While the victim process executes the block cipher, the attacker collects samples from varying byte indices i and sorts each of them into the according sampling pool indexed with the pair (i, j) of LFB byte index (i) and a reference to the plaintext that was encrypted during sampling (j). In practice, the observed samples may stem from observable intermediate results of block cipher operations in one of N_r rounds, from the processed plaintext or ciphertext or from unrelated processes which contribute additional noise. The sampling process has three distinctive properties: First, a very low sampling rate, i.e., the sampling frequency is small compared to the frequency of block ciphering operations, second, asynchronous sampling, i.e., a synchronization with the ciphering process is hard for the attacker's process, and third, random sampling, i.e., the origin and outcome of the next sampling process is unknown.

Analysis. In the analysis phase, a divide-and-conquer key hypothesis test similar to DPA [7,10] is executed. The attacker targets each key byte individually. For each key byte, the attacker computes the intermediate values that are expected to leak for each of the 256 possible values. For the correct key byte hypotheses, the expected intermediate values appear in the corresponding sampling pools.

Profiling (optional). The exact analysis algorithm clearly depends on the leakage pattern of the victim application and the used block cipher. The attacker can either guess which intermediate values leak during computation or use an identical implementation to the victim process to characterize the probability distribution of relevant observables in the sampling pools. We refer to the latter option as *profiling*. To profile an implementation, we set it up to process a fixed known input with a fixed known key repeatedly while ZombieLoad samples are

collected. All intermediate values that potentially occur can easily be computed off-line or extracted from a running process using a debugger. We search the extracted samples for the computed intermediate values. Those values that occur in the correct sampling pools are candidates for attack vectors in the analysis phase.

3.3 Case Study: Practical Application to an AES Implementation

To show that our attack is indeed practical, we picked a byte-oriented AES implementation called *aes-min*[1] as a target to tailor an exemplary differential attack. All AES functions are implemented closely to the specification [13] in C and without optimizations in terms of side-channel resistance. *aes-min* is meant to be used on embedded devices with limited resources [11] but can also be compiled for general-purpose CPUs.

Table 1. CPUs under investigation.

CPU	Microarchitecture	Microcode	Environment	TSX	AES-NI
i3-2120	Sandy Bridge	`0x28`	Lab	✗	✗
i7-2620M	Sandy Bridge	`0x1a`	Lab	✗	✓
i5-4300M	Haswell	`0x1c`	Lab	✗	✓
Xeon E3-1270v6	Kaby Lake	`0xb4`	Cloud	✓	✓
i7-8650U	Kaby Lake R	`0x96`	Lab	✓	✓

Testing Environment. We used the CPUs from Table 1 running Debian 9 and gcc 6.3.0-18 for our experiments. We distinguish between lab CPUs running an uniform operating system image with LXDE user interface, and Cloud CPUs, which were rented as dedicated machines from a hosting provider running a pre-installed Debian 9 image without graphical user interface. Relevant mitigations including Kernel Address Space Layout Randomization (KASLR), Kernel Page-Table Isolation (KPTI), or clearing vulnerable CPU buffers were disabled on all systems[2]. We used a signal handler to handle the fault triggered by ZombieLoad attack variant 1 architecturally, as TSX is not supported by some of our systems.

Profiling *aes-min*. We executed the profiling step on the i7-2620M CPU using ZombieLoad variant 1. We computed the AES state after each operation off-line and used these values to assign the recorded samples to their originating state with high probability. Figure 2 shows that we observed leakage from the AES state after every operation in every round, as well as plaintext (0/input) and ciphertext (10/output). About 59 % of the samples could not be assigned to any of the AES states and are therefore considered noise. There are two states that

[1] https://github.com/cmcqueen/aes-min/tree/728e156091/

[2] Kernel parameters: `nokaslr nopti mds=off tsx_async_abort=off`.

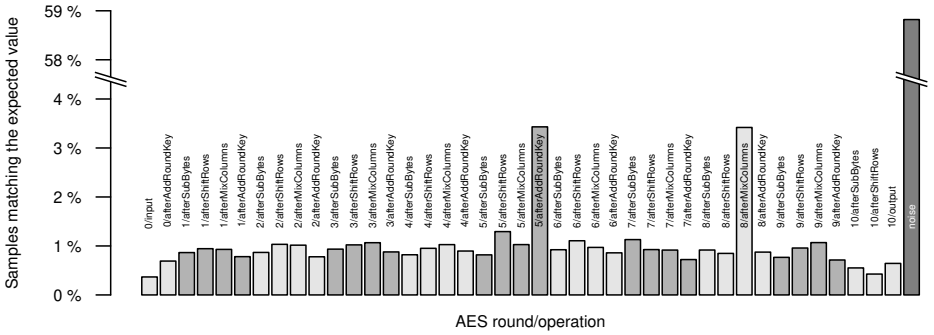

Fig. 2. Empirical assignment of samples to AES operations for *aes-min* on i7-2620M.

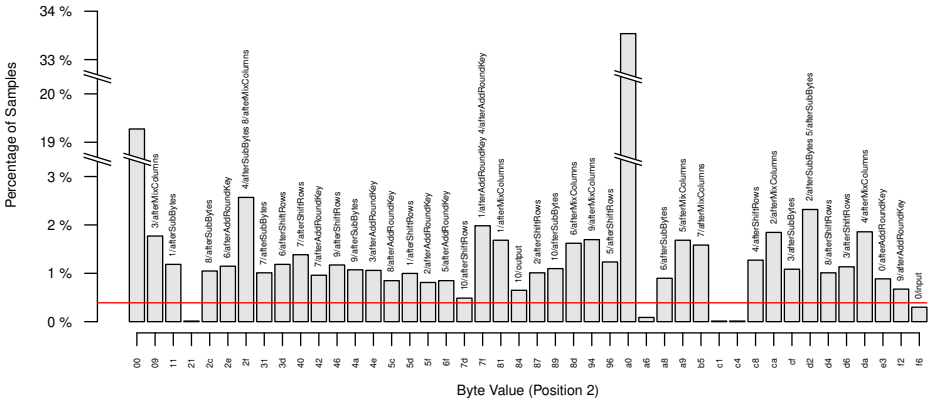

Fig. 3. Empirical distribution of samples at byte position 2 for *aes-min* on i7-2620M.

appear to leak more samples than the other. These states contain the byte value 00, which is generally overrepresented because it is the fallback value for failed ZombieLoad attempts.

Figure 3 shows all the byte values that occurred during an *aes-min* encryption for a specific byte position. We observed that noise is limited to seven values on average for most byte positions. Many of the 256 possible values do not appear at all. The red line in Figure 3 shows the expectancy value for the assumption that all recorded samples distributed equally over the 256 possible values. Most noise values stay below the red line while bytes that occur in intermediate states stay above. This value could therefore be used as a threshold to filter out noise.

Analysis Algorithm for *aes-min*. Our analysis algorithm in Figure 4 conducts the key hypothesis test: the attack targets each byte of the key on its own (line 2) and also considers the samples recorded for each plaintext independently (line 3). All further steps are conducted only with those samples that match these properties, i.e., which were sorted into the same sampling pool.

Require: p: Array containing the plaintexts used during the attack.
SP: Matrix of sampling pools after measurement.
Ensure: k^*: Array containing the best key hypothesis for each index.
1: $\mathbf{C} = 0$
2: **for** $idx = 0$ **to** 15 **do**
3: **for** $pt = 0$ **to** length$(p) - 1$ **do**
4: **for** $hyp = 0$ **to** 255 **do**
5: $afterARK = p[pt][idx] \oplus hyp$
6: $afterSB =$ `SubBytes`$(afterARK)$
7: **if** $afterARK \in SP[idx][pt]$ **and** $afterSB \in SP[idx][pt]$ **then**
8: $c_{idx,hyp}$ += 1
9: **end if**
10: **end for**
11: **end for**
12: $k^*_{idx} = \underset{hyp}{\text{argmax}}\ c_{idx,hyp}$
13: **end for**

Fig. 4. Differential analysis algorithm for *aes-min* (pseudo code)

We compute the first steps of an AES encryption for a known plaintext byte. Remember that the AES encryption starts with `AddRoundKey` followed by a `SubBytes` operation [13]. Since the correct key byte is unknown, we simply perform these calculations for all 256 possible values (line 4-6). For the correct key byte hypothesis, we expect that both intermediate results appear at least once in the curent sampling pool (line 7). If this is the case, we record the observation in a result matrix $\mathbf{C} = (c_{idx,hyp})$, $0 \leq idx < 16, 0 \leq hyp < 256$. $\mathbf{C}$ is initialized to zero (line 1). Each element $c_{idx,hyp}$ represents a counter that is increased when all expected intermediate values for key hypothesis hyp at index idx occur in the current sampling pool (line 8). The underlying idea for key recovery is that the counters for the correct key hypotheses increase faster than for any other hypotheses the more different plaintexts are analyzed. If leakage exists, the most probable 16-byte key hypothesis can finally be extracted from $\mathbf{C}$: the column index hyp of the maximum value in each row idx specifies the most probable key byte at index idx (line 12).

Success Rates for *aes-min*. Figure 2 shows that we identified 41 intermediate AES observables that appear in ZombieLoads during *aes-min* encryptions. We simplify that each observable AES intermediate result appears in the samples and additional noise is neglected. Considerng the algorithm of Figure 4, false positives for wrong key hypotheses can be produced by `AddRoundKey` and `SubBytes` operations in the subsequent nine AES rounds, any pair of the 21 remaining observable AES operations (plaintext byte, ten `ShiftRows`, nine `MixColumns` and one `AddRoundKey` which yields the ciphertext byte), or combinations thereof. They are sieved out by considering multiple plaintexts.

We performed 200,000 noise-free simulations of this leakage pattern for a single byte and received approximately 13.01 competing key hypotheses on av-

erage after the first plaintext, including the correct one. We further simulated 200,000 noise-free attacks on the full AES key to determine how many plaintexts are required to find it. We consider the key found as soon as the most probable key byte hypothesis matches the correct key byte for all 16 positions. Using the outputs of `AddRoundKey` *and* `SubBytes` as shown in Figure 4, line 7, two plaintexts were enough to leak the full key in very rare cases (0.02 %). After three plaintexts, 69 % of the attacks were successful, 98 % succeeded after four plaintexts and eight plaintexts were required at most. If we only used the value after `AddRoundKey` *or* `SubBytes` in line 7, three plaintexts would be sufficient for very rare cases (0.03 %), 25 % of the attacks succeeded after four plaintexts, 83 % after five, 97 % after six, and all after considering eleven plaintexts.

Table 2. Experimental results for the differential attack on different CPUs ($n = 10$ repetitions).

	CPU	Variant	No. of samples	Samples/ plaintext	Avg. duration (s)	Avg. key bytes	Full key recoveries
(1)	i3-2120	1	30,000	500	3.4	14.7	1/10 (10%)
			100,000	1,000	10.0	16.0	10/10 (100%)
	i7-2620M	1	60,000	8,000	6.2	14.9	1/10 (10%)
			200,000	4,000	53.7	16.0	10/10 (100%)
	i5-4300M	1	20,000	1,000	8.0	13.2	1/10 (10%)
			200,000	4,000	65.4	16.0	10/10 (100%)
(2)	E3-1270v6	1	3,000	500	732.7	0	0/10 (0%)
	i7-8650U	1	3,000	500	1,033.4	0	0/10 (0%)
(3)	E3-1270v6	2	800,000	300	405.1	11.7	0/10 (0%)
	i7-8650U	2	600,000	1,000	122.3	14.8	4/10 (40%)
			800,000	300	197.2	15.8	8/10 (80%)

Experimental Results for *aes-min*. We implemented[3] a wrapper program around *aes-min* that takes a 16 byte plaintext, which is repeatedly encrypted with a hard coded key until the process is shut down. It performs approx. 298,000 encryptions per second on the 9-year-old i3-2120 and approx. 528,000 on the most recent of our CPUs (i7-8650U). This victim process is started by the attacker who also chooses the plaintext. ZombieLoad samples are collected while the process is running and stored in the corresponding sampling pools. The plaintext is changed in predefined intervals by stopping the victim process and starting it again with a different plaintext. While this approach technically implements a chosen-plaintext scenario, it should be noted that it is not strictly necessary for an attacker to choose the plaintext freely. The attack is also applicable in a known-plaintext scenario where each plaintext is repeatedly encrypted during a time frame known to the attacker. After collecting a sufficient number of samples, we stop the victim program and execute the analysis algorithm from Figure 4.

[3] The source code of all implementations used in this paper can be found at https://github.com/tillschlueter/zombieload-on-block-ciphers.

We executed the attack based on ZombieLoad variant 1 first. We set the number of samples to collect to 3,000, 10,000, 20,000, 30,000, 60,000, 100,000, or 200,000, while changing the plaintext every 500, 1,000, 4,000, or 8,000 samples. For each parameter combination, we repeated the attack 10 times. A subset of the results is listed in Table 2, section (1). For each CPU, we list results for two parameter sets: first, for the smallest number of samples and plaintexts required to recover the full AES key in 1 out of 10 tries in our experiments, and second, for the smallest parameters required to recover the key in 10 out of 10 tries. The average attack duration for the first parameter set is between 3.4 to 8.0 seconds, while it varies between 10.0 and 65.4 seconds for the second parameter set. The number of plaintexts used ranges from 8 to 100.

The attack was unsuccessful for the Kaby Lake CPUs in our test field, as shown in section (2) of Table 2: We observed many unrelated samples with value `0x00` at those positions where we expected AES intermediates. Filtering out these values led to very low average sampling rates (< 10 B/s) in this attack scenario, and the resulting samples were still unrelated to the targeted intermediate values. Because both affected CPUs support TSX, we tried using ZombieLoad variant 2 instead. To overcome low sampling rates with variant 2, we allocated the probe array on a single 2 MB page. This allows us to load the address translation into the Translation Lookaside Buffer (TLB) in advance, ensuring that no costly page table walk has to be performed during transient execution. With variant 2, we received samples containing the targeted intermediate values, as well as many unrelated samples, leading to many false positives in the analysis phase. To counter this effect, we collected more samples (200,000, 400,000, 600,000, or 800,000) and changed plaintexts more frequently (after 300, 500, or 1,000 samples). In this way we could find up to 11.7 key bytes on average on the Xeon E3-1270v6 CPU, while full key recovery was possible on the i7-8650U (see Table 2 (3)). For the latter CPU, we list the parameter set with the smallest number of samples and plaintexts required for full key recovery and the parameter set that led to the most successful attacks in our experimental setup (8/10 tries).

We conclude that type and quantity of leakage strongly depends on the specific environment and the microarchitecture, and that implementations that leak fewer values in one setting may behave differently in others.

4 Cache Line Fingerprinting

In this section, we propose a new fingerprinting technique to extract constant chains of consecutive bytes (like cryptographic keys) from frequently observed cache lines. As Wampler et al. [19] demonstrated in a Spectre scenario, it is sometimes possible to access the probe array multiple times from within a single transient execution window. This approach also works in the context of MDS attacks [12,15]. Instead of using the additional capacity to extract multiple bytes from the same LFB entry, we propose to transfer an additional byte that identifies the LFB entry from which the data value was sampled with high probability. We call this byte a *cache line fingerprint*.

4.1 Attack Procedure

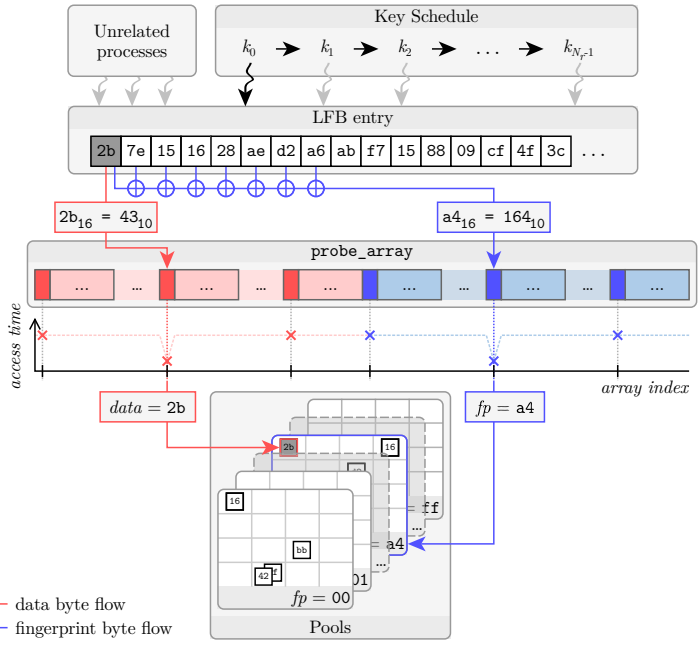

Fig. 5. Storage of leaked bytes in pools. For each byte, the appropriate pool is selected based on the fingerprint of the originating LFB entry.

An overview of the cache line fingerprinting method is given in Figure 5. During transient execution, the probe array is accessed twice. To prevent collisions of these accesses, the length of the probe array is doubled to $4096 \cdot 512$ bytes. As in previous attacks, the first access is used to encode a data byte from the LFB into the first half of the probe array. Then we make use of the fact that the whole LFB entry is accessible during transient execution and generate a fingerprint using a function that maps from a subset of the LFB entry content to a single byte. We achieved good results with a fingerprint function that computes the logical XOR operation on the first eight bytes. This function covers a sufficiently large subset of the LFB entry to identify it and is yet fast to compute. The fingerprint byte is encoded into the second half of the probe array. Finally, the attacker program iterates over the extended probe array and recovers a pair $(data, fp)$ containing a data byte and a fingerprint byte.

Again, we store the recovered bytes in multiple pools as shown in Figure 5. We set up one pool for each possible fingerprint value, i.e., 256 pools for an 8-bit fingerprint. Each data byte *data* is stored in the pool that is indexed with the fingerprint value *fp*. Within the pool, data bytes are further separated by their position in the LFB entry.

When the same cache line is observed again in a later ZombieLoad iteration, the generated fingerprint will be identical. This property allows to assign bytes from several ZombieLoad iterations to the same fingerprint pool and therefore to the same cache line – apart from collisions, i.e., when two or more cache lines are mapped to the same fingerprint. If necessary, collisions could be further reduced by changing the fingerprint function or determining expected probability distributions for the samples in a fingerprint pool.

4.2 Practical Application to an AES Implementation

We mount a cache line fingerprinting attack on the calculation of the AES key schedule in OpenSSL 1.1.0l to extract two round keys and compute the initial key. This method enables us to automatically identify AES (round) keys in the leaked sample set and even works if the initial key is subject to increased noise while some round keys are less affected.

A victim process uses the `EVP_*` functions in OpenSSL's *libcrypto* to repeatedly compute the AES key schedule and encrypt a message while an attacker program concurrently performs ZombieLoads with cache line fingerprinting on a sibling core. For AES128, 11 round keys (k_0, k_1, ..., k_{10}) are computed. The first round key k_0 is equal to the initial key.

The attack is divided into sample collection and analysis phase. During sample collection, the attacker program collects samples continuously and sorts them into pools based on their fingerprints. After a predefined number of samples was collected, the analysis phase begins. The attacker checks for each pool whether it contains a continuous chain of 16 bytes starting at a 16 byte offset. Each of these chains can be seen as an AES round key candidate. If multiple colliding byte values were recorded for any index within a pool, the attacker may choose the value based on the frequency of occurrence.

If the attack was successful, at least one round key is among the detected chains. For each chain, the attacker should first check whether the chain itself is the AES key k_0. Otherwise, the inverted key schedule yields 10 key candidates per chain, including k_0 if the observed chain is a valid round key. If two or more round keys leaked, the attacker can also find the AES key k_0 without knowing any plaintext-ciphertext pair for verification. Let k_{r_1} and k_{r_2} be two different AES round keys among the extracted chains. We pick any of the recorded chains and assume it is round key k_{r_1}. We calculate the set of potential previous round keys using the inverted AES key schedule. If k_{r_2} is among the calculated round keys, we know we probably picked a correct round key and can calculate k_0.

We executed the attack on the CPUs in our testing environment that support AES-NI (see Table 1). We noticed that the transient execution window of ZombieLoad variant 2 is too small to compute even the simple XOR-based fingerprint, so we confined to variant 1 in all cases.

The attack was successful on both CPUs listed in section (1) of Table 3. We tried sample sizes from 60,000 to 120,000 samples and observed the highest success probability after recording 100,000 samples on both devices. The attack is still susceptible to noise, which explains the relatively low rate of successful

Table 3. Experimental results for the cache line fingerprinting attack on different CPUs ($n = 10$ repetitions).

	CPU	No. of samples	Avg. duration (s)	Full key recoveries
(1)	i7-2620M	100,000	12.1	9/10 (90%)
	i5-4300M	100,000	11.3	3/10 (30%)
(2)	E3-1270v6	100,000	10.9	0/10 (0%)
	i7-8650U	100,000	12.1	0/10 (0%)

full key recoveries: it fails as soon as one byte of a round key is misdetected. Again, we noticed strong differences in leakage patterns and quantity depending on the microarchitecture: On both CPUs listed in section (2) of Table 3, the attack was infeasible due to too much noise from parallel activity.

4.3 Discussion

Comparison to the Domino Attack. In the domino attack [16], values from all observed cache lines overlap in the entire sample set; the only distinctive property of recorded samples is the index inside the LFB entry. The domino attack deals with a set of frequency distributions where each distribution reflects all values observed at a given index. All samples in our improved attack have two distinctive properties: index and fingerprint. This leads to 256 sets of frequency distributions, one set per fingerprint pool. In each set, data from different cache lines only superimpose where the fingerprint collides. Another advantage of cache line fingerprinting is the direct association of the fingerprint with the data value, while the domino attack samples data bytes and domino bytes at different points in time and with no inherent connection between them. It is noted that cache line fingerprinting shares the limitation with the domino attack that only constant values that appear repeatedly can be extracted from cache lines. Clearly, if the first eight bytes of a cache line change, the fingerprint likely changes as well.

Comparison to RIDL's Mask-Sub-Rotate. Van Schaik et al. [18] propose Mask-Sub-Rotate (MSR), a technique that may look similar to ours at first sight. Both approaches access the full cache line during transient execution. While MSR leaks sequences identified by some sub-sequence, our technique extracts all frequent constant sequences, without knowing any identifying marker sequence. Furthermore, we leak two bytes per iteration (value and fingerprint), while MSR leaks either a single byte value (if the cache line begins with the identifying sequence) or nothing (otherwise).

5 Mitigations

Both the differential attack as well as cache line fingerprinting require that ZombieLoads can be collected. For general countermeasures to prevent ZombieLoad

leakage we refer to [16]. Considering that disabling both HyperThreading and TSX is necessary to fully mitigate ZombieLoad on vulnerable CPUs [6], we assume that mitigations are still incomplete on many real-world systems.

As a software mitigation, the presented differential attack can be prevented similarly to the secret sharing approach in [16] by masking the entire cryptographic implementation [10]. Masking flattens the statistical distributions in the sampling pools towards a uniform distribution which thereby prevents the described leakage of the differential attack. Second-order attacks are assessed to be difficult as the leakage frequency of ZombieLoads is much smaller than the execution frequency of AES, however, it should be ensured that mask and masked data are never stored in the same cache line to make it harder for an attacker to target the mask. A limited countermeasure to cache line fingerprinting could be to only mask the key storage. As cache line fingerprinting depends on static cache data, a regularly refreshed masking of the key storage, e.g., after a certain number of AES executions, counteracts this attack assumption.

6 Conclusion

In this paper, we showed that sampled intermediate results of cryptographic implementations can be used for key recovery in an MDS scenario. This differential attack can succeed with very few known plaintexts or ciphertexts and does not require the key bytes to be directly observable via MDS. Exploitable leakage was observed with the implementation *aes-min* and the AES key was successfully recovered in less than four seconds on an i3-2120 CPU. While the actual leakage pattern strongly depends on the specific implementation, environment, and CPU, we want to raise awareness that intermediate values can be used to recover secret data in MDS contexts as well and should be considered as a potential threat. This is especially true for algorithms other than AES without special hardware-support which rely even more on software-based computations that are potentially susceptible to leakage. As second main contribution, we proposed the use of cache line fingerprinting in order to pin ZombieLoad samples to a cache line with high probability. This allows to extract constant byte sequences more efficiently for key recovery.

Acknowledgments. We would like to thank our anonymous reviewers and our shepherd Daniel Gruss for their valuable feedback. We also thank Michael Schwarz for sharing his knowledge on the subtleties of implementing microarchitectural attacks.

References

1. Bos, J.W., Hubain, C., Michiels, W., Teuwen, P.: Differential Computation Analysis: Hiding Your White-Box Designs is Not Enough. In: Gierlichs, B., Poschmann, A.Y. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2016. vol. 9813, pp. 215–236. Springer, Berlin, Heidelberg (2016)

2. Canella, C., et al.: Fallout: Leaking Data on Meltdown-Resistant CPUs. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 769–784. CCS '19, ACM, New York (2019)
3. Canella, C., et al.: A Systematic Evaluation of Transient Execution Attacks and Defenses (May 2019), https://arxiv.org/pdf/1811.05441v3
4. Intel Corp.: Deep Dive: Snoop-assisted L1 Data Sampling (2020), https://software.intel.com/security-software-guidance/deep-dives/deep-dive-snoop-assisted-l1-data-sampling
5. Intel Corp.: Processors Affected: Microarchitectural Data Sampling (2020), https://software.intel.com/security-software-guidance/insights/processors-affected-microarchitectural-data-sampling
6. Kernel Development Community: Hardware vulnerabilities – The Linux Kernel documentation (2020), https://www.kernel.org/doc/html/v5.8/admin-guide/hw-vuln/index.html, see MDS and TAA subsections.
7. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) Advances in Cryptology — CRYPTO' 99. vol. 1666, pp. 388–397. Springer, Berlin, Heidelberg (1999)
8. Kocher, P., et al.: Spectre Attacks: Exploiting Speculative Execution. In: 40th IEEE Symposium on Security and Privacy (S&P'19). San Francisco (2019)
9. Lipp, M., et al.: Meltdown: Reading Kernel Memory from User Space. In: 27th USENIX Security Symposium (USENIX Security 18). USENIX Association, Baltimore (Aug 2018)
10. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks. Springer US, Boston (2007)
11. McQueen, C.: aes-min. https://github.com/cmcqueen/aes-min/blob/728e156091/README.md (Dec 2018)
12. Moghimi, D., Lipp, M., Sunar, B., Schwarz, M.: Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In: 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Boston (Aug 2020)
13. National Institute of Standards and Technology: Specification for the Advanced Encryption Standard (AES) (Nov 2001), https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf, FIPS 197
14. Ragab, H., Milburn, A., Razavi, K., Bos, H., Giuffrida, C.: CrossTalk: Speculative Data Leaks Across Cores Are Real (2020), https://download.vusec.net/papers/crosstalk_sp21.pdf
15. Schlüter, T.: ZombieLoad-Angriff auf den Advanced Encryption Standard. Master's thesis, Bonn-Rhein-Sieg University of Applied Sciences (Jan 2020)
16. Schwarz, M., et al.: ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 753–768. CCS '19, ACM, New York (2019)
17. van Schaik, S., Minkin, M., Kwong, A., Genkin, D., Yarom, Y.: CacheOut: Leaking data on Intel CPUs via Cache Evictions (2020), https://arxiv.org/pdf/2006.13353v1.pdf
18. van Schaik, S., et al.: RIDL: Rogue In-Flight Data Load. In: 40th IEEE Symposium on Security and Privacy (S&P'19). San Francisco (2019)
19. Wampler, J., Martiny, I., Wustrow, E.: ExSpectre: Hiding Malware in Speculative Execution. In: Proceedings 2019 Network and Distributed System Security Symposium. Internet Society, San Diego (2019)