DramaQueen: Revisiting Side Channels in DRAM

Victor van der Veen
Qualcomm Technologies Inc.
vvdveen@qualcomm.com

Ben Gras
Intel Corporation
ben.gras@intel.com

***Abstract*—The only way to process sensitive information securely is to ensure that every aspect of the processing is constant-time: No aspect of the execution, be it algorithm, implementation, or (micro)architecture, may depend on the sensitive information, be it in timing, code accesses, or data accesses. In practice, however, seemingly constant-time algorithms may leak information through unexpected data-dependent behavior of the compiler, or through the hardware that executes them. It is attractive to make non-constant-time implementations secure using software layers. Therefore, there is value in frameworks that add security to current implementations, be it as a defense-in-depth measure, or as a measure to isolate known insecure behavior from observation.**

In this paper, we analyze what might be considered a straw-man framework to implement such isolation: What if, in an effort to make existing implementations increasingly resistant against microarchitectural side-channel attacks — which mostly seem to exploit CPU caching behavior — we not only disable Synchronous Multithreading, but also force all sensitive accesses to use non-cacheable memory? This would surely cut off a great deal of known side channels.

We show that an algorithm that exhibits secret-dependent accesses, protected by such a framework, nevertheless leaves open one remaining exploitable hardware resource: the DRAM bank conflict timing channel. By observing DRAM accesses using the bank conflict side channel, we deduce secret-dependent access patterns. We revisit, reproduce, and extend the state-of-the-art and find that due to aggressive interleaving applied by modern processors, the granularity of the side channel is closer to a cacheline than to the DRAM row size. Using weak attacker assumptions, we show full key recovery results when applied to a cryptographic algorithm that is not implemented using constant-time principles, using just a single trace capture. Out of 82 of 100 trials, the median brute force key recovery effort after side channel processing was 2^{30} or less, reducing the time required to leak cryptographic keys to minutes.

I. Introduction

Side-channel research is an active research area [1]–[18], as side channels may reveal sensitive secrets in practice. Recognizing the security risks in having multiple security domains share micro-architectural resources, which allow for these side channel attacks, a number of generic mitigation approaches have been proposed. We discuss some categories here.

Generally, accurate sources of timing are needed, and one suggestion has been to limit these [19]–[21]. Other approaches include a generic cache side channel protection framework [22]–[27], oblivious RAM [28]–[32], or hardware-assisted methods [33]–[35], typically repurposing hardware features not intended for security but having usable security properties as a side effect [36]–[38].

Recognizing that cross-thread micro-architectural attacks are difficult to prevent reliably, OpenBSD has disabled Simultaneous Multithreading (SMT) by default [39]. We assume an unprivileged attacker that can not monitor power meters, and therefore power side channels [40]–[42] will not work [43]. We discuss the threat model in detail in Section III-A.

A. Straw-man design of ultimate protection

In this work we aim to analyze the security properties of a *straw man cache side channel protection framework*. Taking the OpenBSD approach of minimizing side channel risks by minimizing shared microarchitectural resources to its ultimate conclusion, we propose what might appear to be a system of increased protection against certain concurrent microarchitectural side channels. We only consider side channels that operate concurrently to the victim, i.e., on their own core or CPU thread, as time-sliced approaches can be blocked by the OS [44]. Our straw-man design proposes the following countermeasures:

- Disabling SMT, which precludes side-channel attacks relying on concurrent on-core resource sharing, such as on-core caches [45], TLBs [3], branch prediction buffers [46], line fill buffers [18], blocking a large number of attacks [1], [12]–[17].
- Not allowing security-sensitive code to cache its code or data, which precludes all side-channel attacks relying on off-core caches [4], [47] or CPU interconnects [48], [49]. (Monitoring the interconnect fabric will be able to observe the volume of DRAM accesses, but previous work depends on observing LLC traffic to make the attack work.)

At first sight, this might appear to be nearly perfect protection. After all, if there are no micro-architectural shared resources, how can there be side channels? As we shall see, resource sharing remains in the form of DRAM, and we shall explore how harmless or harmful it may be as a side-channel avenue when all other paths for side channels appear to be blocked. For the purposes of our attack, we assume an open page policy in the memory controller, a topic we will revisit when discussing the threat model, in Section III-A.

This work builds upon the insights in previous research that reverse engineers DRAM geometry [50]–[52], and analyzes the impact on software behavior and information leakage [53], [54]. Earlier work [53], [55] has reported on the power of observing DRAM accesses, but in this paper we take a different perspective: How strong is this signal if all accesses

are to DRAM, instead of just the LLC cache misses? We build the system that the straw-man design proposes, and quantify the answer to this question by letting a spy observe the DRAM contention signal with fine time granularity of a victim known to do secret-dependent memory accesses.

B. Contributions

This work makes the following contributions.

1) We propose and build the straw-man design of a seemingly ideal side channel attack protection system by marking all code pages that perform secret-dependant operations as *non-cacheable* (or *uncached*).
2) We show that, while a system may theoretically be not vulnerable to cache attacks, an attacker can still monitor the DRAM bank conflict signal with high time granularity. We reliably extract a cryptographic key from a victim by observing the DRAM bank conflict signal with just a single trace capture.
3) In the process, we partially reproduce and confirm the validity of prior art on DRAM-based side-channel attacks [53], but also expand it by weakening assumptions. While [53] demonstrates a covert channel where same-row co-residency is not required, we show that it is also not strictly required for a spy to have access to the same row in a bank as the victim when performing a side-channel attack.

In summary, rather than demonstrating a new attack of great novelty, we wish to make the novel demonstration that the straw-man side channel defense of using only uncached accesses is at best an incomplete proposal.

II. Background

In this section, we discuss DRAM geometry, DRAM interaction with a Host (CPU/SoC), and possible timing side channels that an attacker gets for free with modern DRAM devices.

A. DRAM Geometry

DRAM stores data as an electrical charge in capacitors. Several capacitors form a column and multiple columns are further grouped into a row. Multiple rows form a bank. Since DDR4 and Low Power (LP)DDR5, multiple banks together could optionally form a bank group. Up to 16 DRAM banks or 4 bank groups comprise a rank, and a single DRAM device may include multiple ranks.

A Memory Controller (MC) acts as a Host and communicates with DRAM through an asynchronous interface. To increase both storage capacity and throughput, a processing element may be equipped with multiple DRAM devices, each connected to a dedicated channel with its own MC. The physical address space is then often interleaved, for example, every 256 Bytes or 512 Bytes of physically contiguous memory could map to a different memory channel (and thus MC) [53].

B. DRAM Commands

At a hardware level, the Host uses dedicated command pins to issue commands over the command bus to DRAM, while the data pins are used to transfer actual bits to and from the DRAM over a data bus. Then, only a few commands are required to implement support for read and write transactions:

1) Activate (`ACT`) takes as parameter a bank and a row. When issued, DRAM activates the given row in the bank, ensuring that data from the capacitors are moved towards an in-DRAM structure typically referred to as *row buffer*. After issuing `ACT`, MC must wait for some time before issuing the next command. This time is referred to as `tRCD` (RAS to CAS delay — RAS/CAS stands for Row/Column Address Strobe).
2) Read (`RD`) or Write (`WR`). This commands takes as parameter the column in the row that MC wants to access. After issuing `RD` or `WR`, MC must wait for `tCL` (CAS latency) before data can be transferred.
3) Data transfer. This is when data moves from the Host to the Guest (DRAM) or vice versa.
4) Pre-charge (`PRE`). This command takes as parameter the bank that should be closed and instructs DRAM to move the contents from the row buffer back into the actual capacitors. MC must always pre-charged a bank before the next `ACT`. MC must wait for `tRP` before issuing such next `ACT`.

C. Timing Side Channels in DRAM

Above commands allow for great flexibility and optimization possibilities, depending on where the requested memory is stored and how it is accessed. Figure 1, shows how accessing only two addresses already introduces various optimizations and thus timing side channels. We assume an attacking CPU that accesses two uncached memory locations in a loop, with sufficient time before looping back to ensure that read data propagated back to the CPU. This ensures that the read operations are completed by MC, and that MC cannot reschedule them. This is obtained through a memory fence (`mfence` on x86) or instruction barrier (`isb` on ARM) and can be found in many Rowhammer [56]–[58] Proof-of-Concepts.

In Figure 1A, our addresses fall in **two different channels** (CH0 and CH1) and can thus get processed by two different MCs concurrently. The figure shows that at timestamp 3, both CH0 and CH1 will issue a `RD` instruction. Both read requests will complete by timestamp 8.

In Figure 1B, our addresses fall in the **same channel, but in different banks**. Because each bank has its own row buffer, banks can be accessed and commanded at (almost) the same time (bank-level parallelism). Naturally, command and data pins are still shared (both banks are accessed by a single MC). This means that the `ACT` command to row `R` in bank `B` must be issued before MC can issue `ACT B', R'`. However, MC does not have to wait for the DRAM to complete the expensive in-DRAM row activation operation. This makes that the scenario in Figure 1B is only marginally slower than when two addresses fall in different channels.

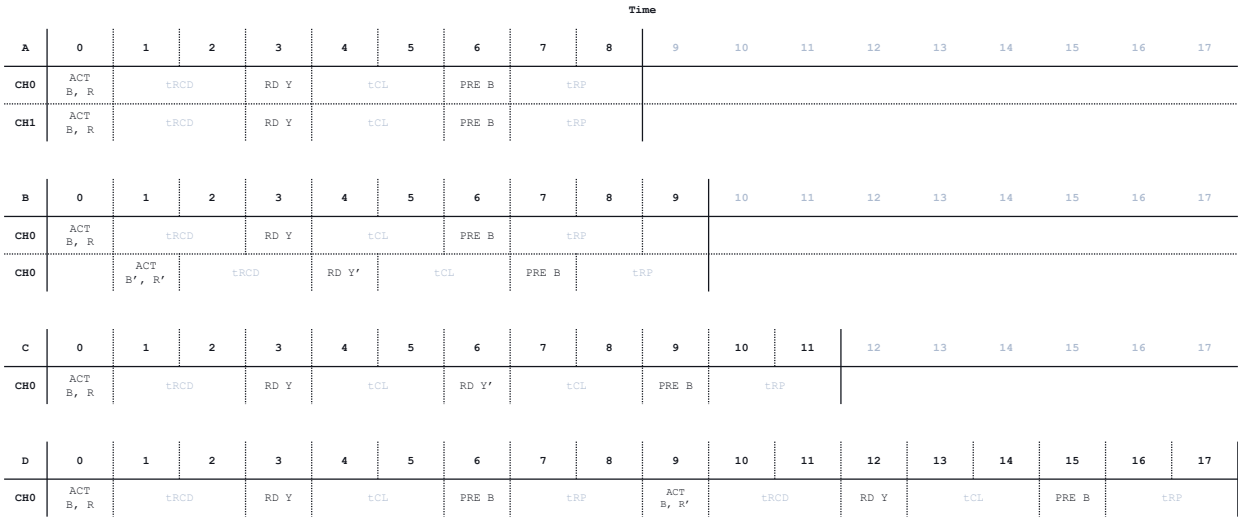

Figure 1: Depending on where two addresses are physically located, accessing them results in different command sequences and timings. Note that this is a simplified diagram, with each timeout, e.g., `tRP`, taking defined as two "time slots"

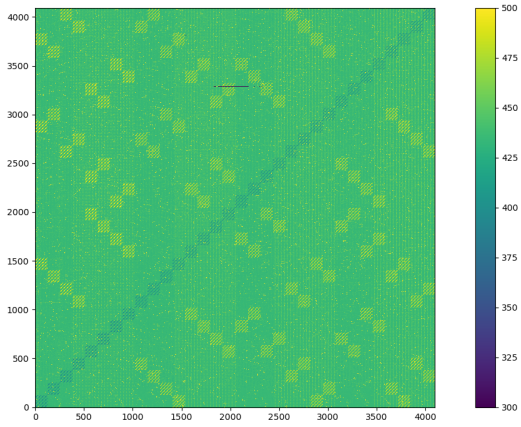

Figure 2: DRAM bank conflict signal (high latencies) and shared row-buffer signal (low latencies).

In Figure 1C, our addresses fall in the **same bank and in the same row**. Here, we depict a common optimization: When data is requested in a row that is already open, i.e., the row is already activated, there is no need to pre-charge the bank and activate the row again. This is a so-called *page hit* and after issuing the `RD` to column `Y`, MC will issue a second `RD` to column `Y'`. Regardless this optimization, this scenario is still slower than the previous ones because DRAM must now wait for one full transfer to complete before it can start the second.

In Figure 1D, we see a worst-case scenario in which our addresses are in the **same bank, but in different rows**. In this scenario, MC must issue and wait for an additional `PRE` and `ACT` to complete before it can read from the second address. This is a *page miss*. We also refer to this as a bank conflict.

D. Timing Side Channels in Practice

In Section IV, we will demonstrate a side channel attack on a Intel Coffee Lake platform. Showing the platform-independence of the DRAM bank conflict phenomenon, Figure 2 shows access latencies for all possible combinations of selecting two memory locations in a 256KB physically contiguous uncached memory region on a Pixel 4XL device, using a stride of 64B. The experiment is similar to Figure 3 of [57] and shows how accessing different address pairs in a tight loop results in different access latencies. From the figure, we can observe side channels as outlined earlier in Figure 1. We step 64B between accesses.

We observe a **recurring pattern of small squares with high latency**, e.g., at $X = 1500$ and $Y = 0$, that indicate a bank conflict / page miss. These squares of bank conflicts are harder to predict than those in Figure 3 of [57], but nevertheless reveal a clear signal exists, allowing one to observe whether two addresses fall in the same bank (but different rows).

We observe a **diagonal of fast accesses** from the bottom left corner to the top right corner. These addresses are close to each other in the physical address space and likely fall in the same bank and sometimes also in the same channel: Zooming in on the figure, we see that only one in four accesses is fast, meaning that in this case, same channel, same bank, same row accesses are faster than any other access pair. We speculate that this is caused by an MC optimization that converts Figure 1 to a scenario in which our instruction barrier does not result in MC pre-charging the bank. We expect that MC can avoid the additional `ACT` and `PRE` in several successive iterations, greatly speeding up the access time. Note that there is a limit to how long a row can remain open, meaning that eventually MC must issue the pre-charge and additional `ACT`.

Zooming on the diagonal fast access blocks, but also on the bank conflicts blocks, we observe that only **one in four successive access pairs triggers a conflict**. This suggests that the device uses 4 channels — which is indeed according its specification — and that each channel is 64B interleaved, i.e., each 64B falls in another channel.

We observe that the **majority of accesses has roughly the**

same access time. This is expected, as most pairs will fall in different banks, either in the same or a different channel.

The **sharp black horizontal line** near X = 2000 and Y = 3300 is a small series of measurements failures where our timer reported an excessively large or small (negative) time delta. These outliers are filtered and automatically replaced with a dummy value of 0.

We observe an additional **signal encoded as faint vertical lines** of slightly slower accesses, repeating every 4 accesses (channel conflict), e.g., from X = 1500 to X = 2000, but with gaps between groups of such vertical lines, e.g., from X = 2000 to X = 2500. We do not fully understand the origin of this signal.

In the remainder of this paper, we focus on exploiting the bank conflict / page miss side channel: A spy can continuously time individual accesses to different banks to determine whether a victim accessed that bank as well — leaking information about the victim's memory access pattern.

III. Threat Model and Attack

A. Threat Model

We assume a victim running sensitive code on a dedicated processing core — a CPU in our case study. As the attacker doesn't share a core (either by a 2nd hardware thread or by time-slicing), this precludes cross-thread side channels from working. An attacker with access to the same machine, but not the same core, can run arbitrary unprivileged code. We assume no software vulnerabilities and correctly functioning process isolation. To protect itself from cross-core side channel attacks, the victim leverages a kernel module [23], [59] to mark memory pages containing sensitive data and code as uncacheable. The attacker may run on any processing element that shares a memory controller with the victim, assuming it is fast enough to trigger and detect bank conflicts with the victim: our threat model does not require the attacker and victim to share the same operating system running on the same subsystem, only that the attacker can obtain physical memory that can generate a bank conflict with the victim. We also assume, for practical reasons, that the attacker can trigger the victim at will, and that the attacker knows the code that the victim is executing. We assume an unprivileged attacker that does not have access to reading power meters, and so power side channels [40]–[42] will not work [43].

The attacker may trigger the victim. This means the attacker may cause the victim to start executing security-sensitive code, such as a cryptographic signing operation using a private key, an unlimited number of times. The victim will use a different, random key each time.

While we do not show this in our case study, the victim may run on a digital signal processing core while the attacker process runs on the CPU, assuming the attacker can get access to memory that can induce bank conflicts with the victim. Due to commonly applied aggressive interleaving we believe this is a weak assumption. For the purposes of our attack, we assume an open page policy in the memory controller, likely causing a relatively strong signal to the attacker compared to a closed-page policy.

The page policy is a configuration choice that is usually controlled by system firmware. A system administrator may reason that selecting a closed-page policy eliminates the side channel. We hypothesize, however, that even a closed-page policy can leak information about whether two addresses share a bank: Always closing the page after an access would transpose Figure 1C to match the command sequence in Figure 1D, but it would not remove same-bank contention. This means that it should still be possible to distinguish same-bank accesses (Figure 1C and D), from different-bank accesses (1A and B). We leave quantifying the power of this side channel under a adaptive or closed page policy to future work.

B. Attack

In our case study, both attacker (spy) and victim run on the application core subsystem as a typical Linux process. Both are unprivileged. The victim is a classic cryptographic library, performing operations using bits of a private key. The attacker and victim are running on two different physical cores. The attacker seeks to find a single cacheline that is in the same DRAM bank as (co-resident with) any memory that the victim accesses while executing the sensitive code. When successful in finding such a cacheline, the attacker can observe the interference caused by the victim to the attacker in the form of occasional latency increases. We assume this latency increase is caused by row changes in the DRAM bank due to an open row policy. This is because, if the victim accesses a particular row, when the attacker tries to access a different row, a precharge + activation must happen for the attacker access. This causes a latency difference compared to the victim not doing an access in the same bank between accesses, and this is the basis of the side channel.

To find such a cacheline that, when accessed in DRAM, sees a significant victim-induced disturbance, the attacker proceeds as follows. The attacker allocates a range of memory. The attacker then guesses a cacheline from this range, and accesses this selected attacker cacheline in a loop. The attacker measures the latency of each access. The mean of these latencies gives the attacker a baseline latency. After establishing a latency baseline, the attacker triggers the victim execution, and monitors the same cacheline again. If the attacker-observed mean latency is now higher and above a threshold, this cacheline is a candidate for being co-resident with victim memory. This makes this attacker cacheline a candidate for the side-channel attack. We then repeat the process of capturing a trace of latencies, for many different addresses (i.e., cachelines), while the victim is executing, and save these traces for later evaluation.

The final step of the attack is key recovery. The assumption is that the observed interference pattern will be different for different parts of the keybit-dependent execution of the victim. As an example of the attacker-observed latencies, see Figure 3. If the attacker can recognize these phases reliably, they can recover the key. To do this, the attacker processes the traces

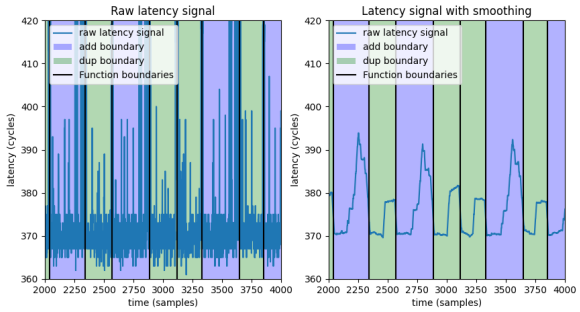

Figure 3: Visualization of attacker-observed latency, before and after smoothing.

Figure 4: Sketch of the insecure version point-scalar multiplication in libgcrypt 1.6.3, showing secret-dependent accesses.

```
void
mul_point_scalar (point result,
   scalar scalar, point point)
{
  for (j=nbits-1; j >= 0; j--) {
   ec_duplicate_point (result);
   if (bit_on (scalar, j))
    ec_add_point(result,point);
  }
}
```

using simple post-processing and a machine learning classifier, combined with ground truth information, in order to recover the secret key. Details of that phase follow in Section IV.

C. Victim

The software target ('victim') is a cryptographic algorithm implemented in `libgcrypt 1.6.3`, specifically the known-insecure version of the `_gcry_mpi_ec_mul_point` point-scalar multiplication. This routine multiplies a scalar by a point on an elliptic curve, and is used in secret key operations such as message signing. The secret is the scalar. The multiplication is implemented by processing the string of secret key bits one by one, and at each step, unconditionally doubling the result, and conditionally adding the elliptic curve point value to the current result. The point is only added to the result if the secret key bit is a 1. Clearly, this makes the code execution path secret-dependent. A conceptual sketch of the code is shown in Figure 4, where the secret-dependent access can be seen: the function of adding a point is only executed when a secret key bit is 1.

IV. Case Study

A. Design

In our case study, we show results of a key recovery experiment. Our testbed is a 3.1GHz Intel i9-9900 Coffee Lake with 2×32GB DDR4 DRAM, 1.2V, configured at 2133 MT/s. It is running Ubuntu Linux. We were not able to definitively determine the configuration of the memory controller w.r.t. page policy, but we believe the latencies seen by the attacker justify the assumption that we are seeing an open-page policy.

We attack a victim executing cryptographic code as described in Section III-C, on which we aim to apply the straw-man protection design as discussed in Sections I and III, as follows:

1) Disabling SMT, precluding cross-thread side channels.
2) Marking all code that is invoked in a secret-dependent way as uncacheable. To do this we re-use ConTExT [23], at [59]. We do not use the code for its intended purpose, which is defending against Spectre, so we do not intend to imply we break the defense. For the purpose of implementing our straw-man protection it works well, however.

As described in Section III, the attacker collects many traces generated from observing 100 different cachelines, each of which showing apparent victim-sensitive latencies. Each of these attacker cachelines are a candidate for our side channel.

To assess how many of these candidates can be used to mount a successful attack, for each attacker cacheline we train a SVM classifier on 4 of the captured traces using available ground truth, and evaluate the performance of the classifier and key recovery process using 16 other captured traces of the same cacheline.

In order to train a classifier on a captured trace, we first normalize the signal by subtracting the mean, dividing by the standard deviation, and smoothing the result (a moving average of 100 samples). The goal of the classifier is to identify executions of the 'duplicate' and 'add' phases of the secret key processing (see Figure 4), each revealing direct information about the secret key bits. While each of these phases may generate hundreds of samples, we wish to know only when they start to execute, so that we count only a single instance each time. In other words, we wish to know the function boundaries.

To do this, we cut a single execution trace into many overlapping sub-traces of 300 samples each, and train the classifier using labels depending on where the sub-trace starts. If the sub-trace starts at the start of an 'add' phase, we give the sub-trace a label of 1. If it starts at the start of a 'duplicate' phase, we give it a label of -1. The majority of sub-traces start somewhere in between these two, and are given a label of 0. If the classifier were to be operating perfectly, when asked to classify a signal, we expect to see a short spike of label 1 whenever an 'add' starts executing, a short spike of label -1 whenever a 'duplicate' starts executing, and 0 at all other moments.

In all cases, each trace is processing a different, randomly generated key. We repeat this training and evaluation process for each of the 100 selected cachelines, each giving a set of traces.

B. Results

In our case study, the attacker tries all cachelines from a 200kB buffer (3200 cachelines), and find that 987 (31%) give some sort of signal when the victim is executing vs. when it is not. As an example of a trace, see Figure 3, where we observe the latency signal while a victim is executing, both the raw

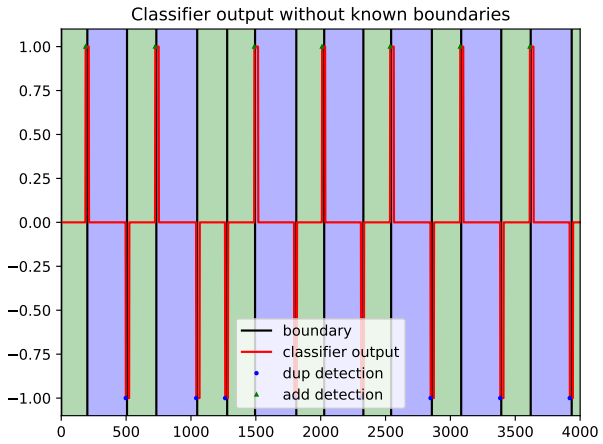

Figure 5: Visualization of key recovery.

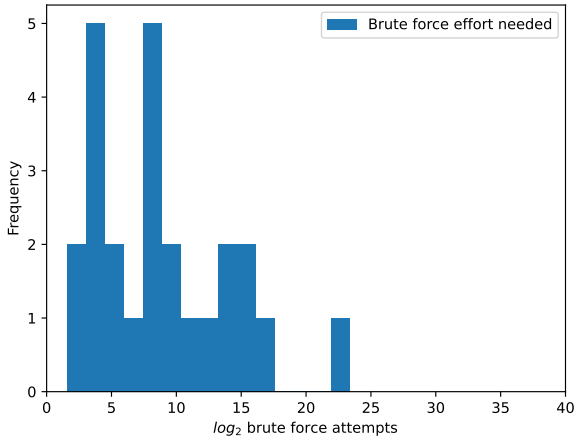

Figure 6: Brute force needed for key recovery after analyzing 25 different traces from one signal source.

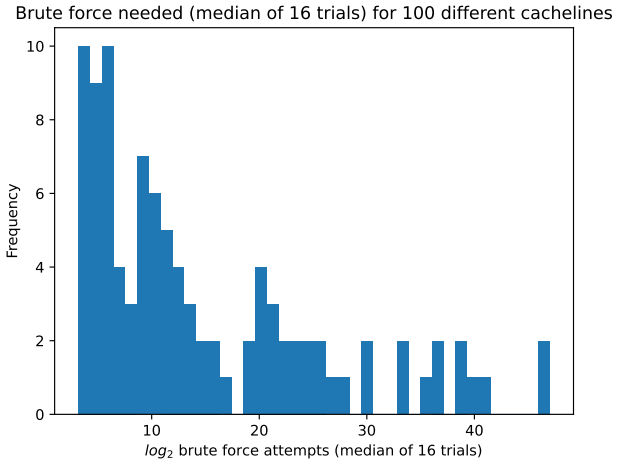

Figure 7: Histogram of median brute force effort of 16 different traces from 100 randomly selected cacheline signal sources.

latency signal and the smoothed signal. The keybit-dependent phase of the victim is indicated by the background color.

After training on 4 training traces, the classifier is able to correctly predict the starting moment of secret keybit phases (labels -1 and 1) in many cases on the 16 test traces. This can be seen in Figure 5, where the classifier output of each moment in the trace is shown: 0 for no boundary, 1 for the start of a 'duplicate,' or -1 for the start of an 'add.' There might be small clusters of 1 and -1 predicted around the boundaries. In this fragment, the predictions line up perfectly with the ground truth (the background shade). Finally, after running a peak detector on the classifier results, which detects a 1 or -1 prediction, and also limits the number of function boundary predictions in an area, we get a final recovered key bitstream.

To evaluate the validity of the recovered key bits, we compare the ground truth to the recovered key bits. Frequently, the key bits will be mis-predicted - either an extra keybit, missing keybit, or wrong keybit. Taking the reasoning from [3], we estimate the amount of brute force necessary to go from the initial guess to the final keybits, using the timing of the predictions as a hint. If the predictions are too close together, we guess the middle bit may be a spurious bit, or one of its neighbors might be, or none of these are (total of 4). Similarly, if we see a gap in the predictions, we guess we may have to insert a keybit there, or none (total of 3). If a bit is wrong, we assume we have to flip one of every bits (around 384 possibilities, depending on the number of 1 bits in the key). We simulate this process using the optimal Levenshtein edit sequence from guessed keybits to correct keybits, assign the corresponding work factor for each step, and multiply the possibilities to estimate the brute force work factor.

From one particular attacker cacheline, after training on 4 of its traces, we take 25 different traces and evaluate the key recovery performance on the test traces. The result can be seen in Figure 6. We see a histogram of the brute force effort to recover the real key each time. We repeat this for 100 randomly selected attacker cachelines: we evaluate the brute force effort needed to recover the key for 16 different traces of a single cacheline, and take the median of this effort. For each of 100 cachelines, we show this median in the histogram in Figure 7. We used a cutoff of 2^{60} as the brute force median work factor. 3 out of 100 cases were above this cutoff. The majority of the cachelines give a median work factor of 2^{20} or lower.

We show that with modest brute force effort, a majority of attacker cachelines are usable as side channel signal sources. This shows that the DRAM bank conflict signal gives a realistic side channel key recovery avenue for this victim.

V. Conclusion

We showed how an attacker can exploit timing differences caused by DRAM bank conflicts to implement complex side channel attacks: the page miss signal is an excellent source to leak information about memory accesses with fine-grained timing granularity. Our attack does not require the spy and victim to share DRAM rows — only banks — and shows that mitigation proposals in which sensitive code and data regions are marked as uncached, would be likely insufficient to stop all side channel attacks.

References

[1] C. Percival, "Cache missing for fun and profit," 2005.

[2] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, L3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 719–732.

[3] B. Gras, K. Razavi, H. Bos, C. Giuffrida *et al.*, "Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks." in *USENIX Security Symposium*, vol. 216, 2018.

[4] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level Cache Side-channel Attacks are Practical," in *IEEE Symposium on Security and Privacy*, 2015.

[5] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Composite-isa cores: Enabling multi-isa heterogeneity using a single isa," *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019. [Online]. Available: http://dx.doi.org/10.1109/hpca.2019.00026

[6] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[7] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[8] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 368–379.

[9] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 191–205.

[10] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with intel tsx," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 380–392.

[11] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "Aslr on the line: Practical cache attacks on the mmu," *NDSS (Feb. 2017)*, 2017.

[12] O. Acıiçmez and J.-P. Seifert, "Cheap hardware parallelism implies cheap security," in *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*. IEEE, 2007, pp. 80–91.

[13] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *Security and Privacy (SP), 2019 IEEE Symposium on*. IEEE, 2019.

[14] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," *arXiv preprint arXiv:1903.01843*, 2019.

[15] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Understanding and mitigating covert channels through branch predictors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, pp. 1–23, 2016.

[16] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. Ponomarev *et al.*, "Branchscope: A new side-channel attack on directional branch predictor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 693–707.

[17] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures." in *NDSS*, 2020.

[18] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 88–105.

[19] D. Kohlbrenner and H. Shacham, "Trusted browsers for uncertain times." in *USENIX Security Symposium*, 2016, pp. 463–480.

[20] Y. Cao, Z. Chen, S. Li, and S. Wu, "Deterministic browser," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 163–178.

[21] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, vol. 40, no. 3, pp. 118–129, 2012. [Online]. Available: http://dx.doi.org/10.1109/isca.2012.6237011

[22] M. Godfrey and M. Zulkernine, "Preventing cache-based side-channel attacks in a cloud environment," *IEEE transactions on cloud computing*, vol. 2, no. 4, pp. 395–408, 2014.

[23] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "Context: A generic approach for mitigating spectre." in *NDSS*, 2020.

[24] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 431–446.

[25] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "Cure: A security architecture with customizable and resilient enclaves." in *USENIX Security Symposium*, 2021, pp. 1073–1090.

[26] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 871–882.

[27] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *Proceedings of the 2009 ACM workshop on Cloud computing security*. ACM, 2009, pp. 77–84.

[28] B. Pinkas and T. Reinman, "Oblivious ram revisited," in *Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30*. Springer, 2010, pp. 502–519.

[29] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious ram," *arXiv preprint arXiv:1106.3652*, 2011.

[30] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious ram with o ((logn) 3) worst-case cost." in *Asiacrypt*, vol. 7073. Springer, 2011, pp. 197–214.

[31] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious {RAM}," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 415–430.

[32] D. Boneh, D. Mazieres, and R. A. Popa, "Remote oblivious storage: Making oblivious ram practical," 2011.

[33] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 406–418.

[34] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 217–233.

[35] R. Sprabery, K. Evchenko, A. Raj, R. B. Bobba, S. Mohan, and R. Campbell, "Scheduling, isolation, and cache allocation: A side-channel defense," *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 2018. [Online]. Available: http://dx.doi.org/10.1109/ic2e.2018.00025

[36] Intel, "Introduction to cache allocation technology in the intel xeon processor e5 v4 family," https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html, February 2016.

[37] ARM, "Memory system resource partitioning and monitoring (MPAM), for a-profile architecture," https://developer.arm.com/documentation/ddi0598/latest/, November 2022.

[38] Intel, "IntelÂ® transactional synchronization extensions (intelÂ® tsx) overview," https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/tsx-programming-considerations-ov.html.

[39] M. Kettenis, "Mailing list post with cvs message disabling smt (simultanious multi threading) by default," https://www.mail-archive.com/source-changes@openbsd.org/msg99141.html.

[40] Z. Zhang, S. Liang, F. Yao, and X. Gao, "Red alert for power leakage: Exploiting intel rapl-induced side channels," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 162–175.

[41] Y. Wang, R. Paccagnella, E. T. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner, "Hertzbleed: Turning power {Side-Channel} attacks into remote timing attacks on x86," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 679–697.

[42] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "Platypus: Software-based power side-channel attacks on x86," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 355–371.

[43] CVE-2020-8694, "Insufficient access control in the Linux kernel driver for some Intel(R) Processors may allow an authenticated user to potentially enable information disclosure via local access." http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2020-8694, 2020.
[44] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No. PR00306)*. IEEE, 1999, pp. 328–335.
[45] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' Track at the RSA Conference*. Springer, 2006, pp. 1–20.
[46] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers' Track at the RSA Conference*. Springer, 2007, pp. 225–242.
[47] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack Directories, Not Caches: Side-Channel Attacks in a Non-Inclusive World," in *IEEE Symposium on Security and Privacy*, 2019.
[48] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring (s): Side channel attacks on the cpu on-chip ring interconnect are practical." in *USENIX Security Symposium*, 2021, pp. 645–662.
[49] M. Dai, R. Paccagnella, M. Gomez-Garcia, J. McCalpin, and M. Yan, "Don't mesh around: Side-channel attacks and mitigations on mesh interconnects," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2857–2874.
[50] T. M. O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX security*, 2007.
[51] C. Helm, S. Akiyama, and K. Taura, "Reliable reverse engineering of intel dram addressing using performance counters," in *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2020, pp. 1–8.
[52] A. Barenghi, L. Breveglieri, N. Izzo, and G. Pelosi, "Software-only reverse engineering of physical dram mappings for rowhammer attacks," in *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*. IEEE, 2018, pp. 19–24.
[53] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "Drama: Exploiting dram addressing for cross-cpu attacks." in *USENIX Security Symposium*, 2016, pp. 565–581.
[54] G. Saileshwar, B. Wang, M. Qureshi, and P. J. Nair, "Randomized row-swap: mitigating row hammer by breaking spatial correlation between aggressor and victim rows," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 1056–1069.
[55] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," *arXiv preprint arXiv:1705.07289*, 2017.
[56] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip feng shui: Hammering a needle in the software stack." in *USENIX Security symposium*, vol. 25, 2016, pp. 1–18.
[57] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," ser. CCS'16.
[58] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 245–261.
[59] "Github repository for context-light." https://github.com/IAIK/contextlight., 2019.