Towards Efficient Evaluation of a Time-Driven Cache Attack on Modern Processors

Andreas Zankl[1](✉), Katja Miller[1], Johann Heyszl[1], and Georg Sigl[2]

[1] Fraunhofer Research Institution AISEC, Munich, Germany
{andreas.zankl,katja.miller,johann.heyszl}@aisec.fraunhofer.de
[2] Technische Universität München, Munich, Germany
sigl@tum.de

Abstract. Software implementations of block ciphers are widely used to perform critical operations such as disk encryption or TLS traffic protection. To speed up cipher execution, many implementations rely on pre-computed lookup tables, which makes them vulnerable to cache-timing attacks on modern processors. For *time-driven* attacks, the overall execution time of a cipher is sufficient to recover the secret key. Testing cryptographic software on actual hardware is consequently essential for vulnerability and risk assessment. In this work, we investigate the efficient and robust evaluation of cryptographic software on modern processors under a time-driven attack. Using a practical case study, we discuss necessary adaptations to the original attack and identify promising new micro-architectural side-channels for it. To leverage the leakage of multiple side-channels, we propose a simple, heuristic way to combine their corresponding attacks. As an additional benefit, combined attacks simplify a comprehensive evaluation of cryptographic software across multiple different processors. We finally formulate practical evaluation suggestions based on the results of our case study.

Keywords: ARM · New side-channels · Efficient evaluation · Vulnerability testing · Exploiting performance events · Rank estimation · AES

1 Introduction

Block ciphers are commonly used to protect bulk data. Their implementations provide high throughput and consequently focus on fast execution time. In software, processing steps can be saved by using pre-computed lookup tables. The transformation tables of AES are a prominent example of this speed-up technique. The disadvantage of lookup tables is that if they are accessed depending on a secret (e.g. a key), they can introduce a timing side-channel when the software is executed on a processor with cache. In the past, this gave rise to the field of cache attacks. In literature, cache attacks are typically split into three groups: access-driven, trace-driven, or time-driven. Access-driven cache attacks allow a spy program to precisely learn the part of the processor cache (e.g. which cache line) that was accessed by a victim program [18]. In trace-driven cache attacks,

I. Askoxylakis et al. (Eds.): ESORICS 2016, Part II, LNCS 9879, pp. 3–19, 2016.
DOI: 10.1007/978-3-319-45741-3_1

the spy is able to observe the results of a sequence of cache requests issued by the victim (e.g. hit, miss, miss, ...) [1]. For time-driven cache attacks, the spy requires only the overall time the victim needs to complete a cipher run [6–8].

In this work we focus on the time-driven cache attack proposed by Bernstein in 2005 [6], because it is a well-studied attack with minimalistic assumptions about the hardware under attack. It has been applied in settings ranging from mobile phones [14,15] and embedded systems [22,23] to virtual machines used in cloud computing [2]. As the attack relies on execution time measurements, the resolution and quality of the timing source is crucial to the success of the attack. In the original publication [6], time is measured with a hardware-based clock cycle counter. Similarly, Spreitzer and Plos [15], Spreitzer and Gérard [14], and Weiß et al. [22,23] successfully use the cycle count register of ARM Cortex-A8/-A9 processors in the attack. Atici et al. [5] use a level 1 (L1) data cache (D-cache) miss counter on various x86 processors in an adaptation of Bernstein's attack targeted at the last round of AES. In other work, Tiri et al. [17] use an L1 cache miss counter to verify their analytical model for time-driven cache attacks on multiple not further specified processors. Uhsadel et al. [19] investigate the L1/L2 D-cache miss counters as well as a clock cycle counter on x86 processors and apply them in the time-driven cache attack proposed by Bonneau and Mironov [8]. These publications show that so-called hardware performance events like clock cycles and cache misses are valuable side-channels for time-driven cache attacks. As a consequence, these performance events are also critical in an evaluation context, because they allow to construct a worst-case attack scenario. The more an implementation is resistant against attacks using high resolution performance events, the better it withstands less powerful attacker models that are more likely in practice. In addition, the better the side-channel source, the fewer measurements are required to identify leaks in the implementation.

Because of these benefits, we investigate hardware performance events known from literature and new events that have not yet been analyzed in the context of Bernstein's attack. For a fair comparison of the events, the original attack needs to be adapted, because it does not reliably determine the remaining entropy of the secret key after the attack. We therefore extend it with a recent key rank estimation algorithm. To further improve evaluation efficiency and robustness, we propose a new and heuristic way of combining multiple attacks. The combination of attacks is strongly advisable given that all of the performance events in our tests leak information about the secret key. Combined attacks thereby help to construct an improved worst-case test scenario, as they leverage the leakage of multiple performance events while filtering out noisy or poor-quality ones. Given that not all events leak equally on every processor, combined attacks can be used as a global measure to simplify testing across multiple different processors. For our case study, we use the modified attack by Bernstein to test a vulnerable AES implementation taken from the OpenSSL library on an ARM Cortex-A9 processor. Based on the results, we provide practical evaluation suggestions. To the best of our knowledge, our work is the first that discusses Bernstein's cache attack exclusively in an evaluation context and provides efficient ways to determine the vulnerability of a software component to the attack.

The rest of the paper is organized as follows. Background information about the time-driven cache attack by Bernstein is provided in Sect. 2. The application of key rank estimation and its benefits to the attack are discussed in Sect. 3. Our proposal for attack combination is given in Sect. 4. The selection of hardware performance events is discussed in Sect. 5. We shortly present our measurement setup in Sect. 6 before we discuss the results of our case study in Sect. 7. Based on the results, we provide practical suggestions in Sect. 8 before we finally conclude in Sect. 9. Further details about the AES implementation under attack are given in Appendix A.

2 Bernstein's Time-Driven Cache Attack

In 2005, Bernstein [6] proposed a profiled time-driven cache attack and successfully applied it to the T-table-based AES implementation that is part of the OpenSSL library v0.9.7a. The attack is embedded in a client-server scenario, with the client being the spy and the server being the victim. The attack itself consists of four phases: the *learn phase*, the *attack phase*, the *correlation phase*, and the *brute-force key search phase*.[1]

The *learn phase* is the profiling phase of the attack. The spy knows the secret key $\mathbf{k}$ and sends a set of plaintexts to the victim. The responses of the victim contain the overall encryption times. By randomly choosing the inputs and keeping track of the corresponding processing times, the spy creates a cache profile of the lookup tables under the known key $\mathbf{k}$. This is possible, because the lookup indices in the first round of AES are given by the XOR of plaintext and initial key: $\mathbf{p} \oplus \mathbf{k}$. The cache profile is written as matrix $\mathbf{CP}_{\mathbf{k}}[b][v]$ with one row for each input byte in the plaintext (indexed by b) and one column for each byte value (indexed by v). For AES, $\mathbf{CP}$ has the shape 16×256. Every timing measurement is added b times to $\mathbf{CP}$, while the plaintext is used to index the matrix. The input byte positions determine the row, the input byte values specify the column. After all measurements have been added, the average timing is computed for each matrix element and subtracted by the total average of all timings. After sufficient timing observations, the cache profile contains information about which input bytes and values cause longer or shorter processing times. Since the key $\mathbf{k}$ is known in this phase, the cache profile indicates which parts of the lookup tables are cached and which are not cached on average in the first round of AES. In the original work, the known key $\mathbf{k}$ is set to zero and the key length is 128 bits. In the *attack phase*, an unknown key $\tilde{\mathbf{k}}$ is used by the victim. The spy again sends a set of plaintexts, keeps track of the processing times and creates a second cache profile $\mathbf{CP}_{\tilde{\mathbf{k}}}[b][v]$, now for the unknown key.

In the *correlation phase*, the spy permutes the profile of the attack phase and correlates it with the one from the learn phase. Permutation is done by accessing the attack profile with indices that are XOR'ed with a possible key hypothesis

[1] Naming convention borrowed from the work by Neve et al. [13].

h: $\mathbf{CP}_{\tilde{\mathbf{k}}}[b][v \oplus h]$, $h \in \{0, \dots, 255\}$. The correlation is then calculated for each row in the profiles. The profile of the learn phase will be similar to the profile of the attack phase, if permuted by the correct key hypothesis h_c: $\mathbf{CP}_{\mathbf{k}}[b][v] \approx \mathbf{CP}_{\tilde{\mathbf{k}}}[b][v \oplus h_c]$. In this case the correlation will peak. The underlying assumption is that in the first round of AES the same parts of the lookup tables are cached or not cached. This causes both profiles to capture the same cache state and allows the correlation phase to compare them. The correlation values for each key byte b and each hypothesis value h are then entered in the correlation matrix $\mathbf{C}[b][h]$. The final step in this phase eliminates hypotheses with low correlation using a variance-based threshold decision. A *brute-force key search phase* is added to assemble key candidates from all left-over hypotheses and to test them against a known plaintext-ciphertext pair. For this step, the attack code iterates over the most likely key candidates and stops when the correct key has been found.

The original attack by Bernstein has a success rate limit that is determined by the cache line size of the target processor and the lookup table entry size of the cipher implementation. If multiple table entries fit on one cache line, they all exhibit the same timing behavior when accessed by the processor. This causes the values of each input byte to form groups with similar timing values in the cache profiles. The number of values per group, which equals the number of table entries per cache line, is denoted as L in this paper. For each hypothesis of a group, the correlation phase generates similar correlation values. As a consequence, a single hypothesis becomes indistinguishable from other members of its group and all of them have to be tested in the brute-force step. This introduces a minimum brute-force effort that cannot be reduced further with the original attack. More information about Bernstein's attack including the success rate limit can be obtained from the work by Neve et al. [13].

3 Key Rank Estimation in Bernstein's Attack

Instead of using the threshold decision in the correlation phase to eliminate unlikely key byte hypotheses, we implement a key rank estimation algorithm, which ranks the correct key against all other key candidates. The rank of the correct key represents the true brute-force effort an attacker would have to spend to recover it. Key rank estimation is necessary during evaluation, because the original threshold decision might eliminate the correct key byte hypothesis from each list. This can happen, if there is little statistical information for one key byte in the measurements. If a correct hypothesis is eliminated, the brute-force step cannot recover the key and fails. In this case there is no resulting brute-force effort that can be evaluated or compared to other attacks. As a consequence, attacks that are marginally successful will hardly ever show useful results. In an evaluation, this must be avoided, because the attack effort is required to determine the current level of security. Especially for testing an implementation on multiple different systems and comparing the results obtained from them, the attack effort must always be available.

In literature, the adverse impact of the threshold decision is noted by Spreitzer and Gérard [14], who rank key candidates instead of eliminating hypotheses with the threshold. In their work they discuss the key rank techniques proposed by Veyrat-Charvillon et al. [20,21]. In contrast, we implement the key rank estimation algorithm proposed by Glowacz et al. [10], because it has an improved time and memory efficiency and scales better to larger key sizes. Since the algorithm expects probabilities instead of correlations, two transformation steps have to be done. First, we calculate Pearson's correlation coefficient instead of taking the original values from the correlation phase. This is necessary, because the original values are not bound to the interval [-1, 1]. With the standard formula for Pearson's correlation coefficient we obtain

$$\mathbf{C}'[b][h] = \frac{\sum_{j=0}^{255} \mathbf{CP_k}[b][j] \cdot \mathbf{CP_{\tilde{k}}}[b][j \oplus h]}{\sqrt{\sum_{j=0}^{255} \mathbf{CP_k}[b][j]^2} \cdot \sqrt{\sum_{j=0}^{255} \mathbf{CP_{\tilde{k}}}[b][j \oplus h]^2}} \,. \tag{1}$$

$\mathbf{C}'[b][h]$ denotes the new correlation matrix with key byte position b and key byte hypothesis h. The second step is to convert the correlation coefficients to probabilities. Using the formula proposed by Gérard and Standaert [11] we obtain

$$\mathbf{P}[b][h] = \mathit{normalize}\left(e^{2 \cdot \mathbf{C}'[b][h]}\right), \tag{2}$$

which is a simplified Bayesian extension of the correlation coefficient distribution approximated with Fisher's transform. $\mathbf{P}[b][h]$ denotes the final probability matrix, which is the input to the key rank estimator. The function *normalize* scales all values such that each row b in $\mathbf{P}$ sums up to 1. The output of the estimation algorithm is the estimated rank r of the secret key that is limited by an upper (u) and a lower (l) estimation bound, all given in $\log_2$. An attacker would face an estimated brute-force effort of 2^r, but at least 2^l and at most 2^u, to recover the secret key. For a precise estimation, the bound tightness $(u - l)$ must be kept small. We choose the estimation precision such that $(u - l) \leq 1.07$ for all estimations in our experiments. As this is sufficiently small for our discussions, we only refer to the estimated key rank r (or the $\log_2$ thereof) in subsequent sections.

4 Proposal for Attack Combination

In the original work by Bernstein, multiple attacks are combined by assigning weights to leftover key byte hypotheses depending on the lengths of the lists they are on. This approach does not work with key rank estimation, because key byte hypotheses are not removed from their lists anymore. We therefore propose a new method for attack combination using the multiplication and normalization step discussed by Mather et al. [12]. To illustrate our method we start with M separate attacks. First, their probability matrices $\mathbf{P_m}[b][h]$, $m \in \{0, \dots, M-1\}$ are calculated as previously explained. Combining two attacks n and $n+1$ is done by multiplying the probability matrices $\mathbf{P_m}$ element-wise and normalizing the rows of the resulting matrix $\mathbf{P_{Comb}}$ such that they again sum up to 1. The multiplication and normalization step is defined as

$$\mathbf{P_{Comb}}[b][h] = \mathit{normalize}\left(\mathbf{P_n}[b][h] \cdot \mathbf{P_{n+1}}[b][h]\right). \tag{3}$$

This step affects probabilities differently based on their values. In theory, an attack with no information about a key byte will yield a uniform probability distribution for all 256 hypotheses values. All probabilities in one row of $\mathbf{P}$ will have the value $\frac{1}{256}$. To illustrate the combination effects, assume an attack m_0 was given with a probability for the correct hypothesis $p_{h_c} > \frac{1}{256}$. If this attack is combined with an attack m_1 with a uniform probability distribution, the combined attack will exhibit the identical probability values as m_0. If attack m_1 contains a probability $p_{h_c} > \frac{1}{256}$, then after the normalization step the combined attack exhibits a p_{h_c} that is higher than the maximum of m_0 and m_1. This is desirable, because the correct hypothesis is easier to find in the brute-force search. If attack m_1 contains a probability $p_{h_c} < \frac{1}{256}$, the combined attack exhibits a p_{h_c} that is smaller than the one in m_0. This should be avoided, as it increases the brute-force effort. Naturally, it is best to combine only those attacks that improve the probabilities of the correct hypotheses. If all "bad" attacks would yield a uniform probability distribution and all "good" attacks would exhibit a $p_{h_c} > \frac{1}{256}$, combination can be done by simply multiplying and normalizing all available probability matrices. Our practical experiments, however, show that bad attacks often have a slight non-uniform probability distribution with $p_{h_c} < \frac{1}{256}$. Including them in the combination would degrade the combined attack.

We thus propose a heuristic filter approach that excludes bad attacks from the combination. In the first step, all M attacks are ranked according to the sum of their L highest probabilities for a given key byte b. The reason why the L highest probabilities are taken into account is that on a processor where L lookup table entries fit on one cache line, good attacks arrange the hypotheses of one key byte into groups of size L. All hypotheses within a group have almost equal probabilities and are indistinguishable in the rest of the attack. Bad attacks do not exhibit this behavior in our experiments and are thus less likely to get a high rank, if the first L probabilities are considered. More details about the value L in Bernstein's original attack are provided in Sect. 2. Given the ranking of all M attacks, we start with attack m_{1st} that has the largest sum and combine it with attack m_{2nd} that has the second largest sum. We decide to keep the combination, if the probabilities of the L most likely hypotheses from attack m_{1st} increase in the combined attack. Otherwise, the combination is discarded and attack m_{1st} is combined with m_{3rd}. This step is repeated until all available attacks are processed. After the filter and combination step, the combined probabilities for key byte b are stored in matrix $\mathbf{P_{Comb}}[b][h]$, which is eventually used to estimate the key rank of the combined attack.

This method assumes that the best out of M available attacks (the one with the largest sum) is always a good one. This is consistent with our experiments, where good attacks have a higher deviation from the uniform distribution than bad attacks. In addition, considering L probabilities at once makes the filter step more robust against bad attacks. The combination step helps to achieve a better overall attack, which is desirable when evaluating software in a worst-case scenario. The advantage of our heuristic approach is that it allows to automatically combine multiple attacks by only providing the value L, which is easily derived from the cache line size of the processor and the lookup table entry size of the tested implementation. The simplicity of our approach is based on the observations about "good" and "bad" attacks, its effectiveness is shown by our practical experiments.

5 Performance Events on ARM Processors

Performance events provide detailed insight into the micro-architecture of a processor while it is working on a task. They are typically needed for debugging and performance evaluations on real hardware. Because of the fine granularity with which a processor can be observed, performance events can be powerful side-channels for the profiled cache attack by Bernstein. Previous literature illustrated that clock cycle and cache miss events allow to successfully perform the attack. Since these events are only a small subset of performance events that are typically available, we propose a more comprehensive study.

Table 1. Selection of performance events and corresponding descriptions according to the ARM manuals [3,4].

ID		Lit.	Description
ARMv7-A/R	`03`$_h$	√	Level 1 D-cache refill
	`04`$_h$	-	Level 1 D-cache access
	`05`$_h$	-	Level 1 data TLB refill
	`06`$_h$	-	Load instructions
	`11`$_h$	√	CPU cycles
	`CCNT`	√	Clock cycle counter
Cortex-A9	`50`$_h$	-	Coherent linefill miss
	`61`$_h$	-	D-cache stall cycles
	`65`$_h$	-	D-cache eviction requests
	`72`$_h$	-	Load/store instructions
	`85`$_h$	-	Data micro TLB stall cycles

Table 2. Selection of ARMv7-A/R processor cores and available hardware counters.

Core	#
Cortex-A5	2
Cortex-A7/8	4
Cortex-A9/15/17	6
Cortex-R4/5	3
Cortex-R7	8

Our target platform is an ARM Cortex-A9 processor, which belongs to the ARMv7-A/R architecture family. The ARMv7-A/R reference manual [4] defines hardware performance events that can be measured on all compliant processors. The ARM Cortex-A9 MPCore reference manual [3] specifies events that are additionally available on the Cortex-A9. Each event is identified by an event ID and can be counted by one of the hardware counters present on every processor core. Table 1 shows the two selections of performance events we analyze in our case study. The first set of events is common to all ARMv7-A/R compliant processors, the second one is specific to the ARM Cortex-A9. The events are displayed with their event IDs and descriptions as found in the reference manuals. All events previously used in literature in the context of Bernstein's attack are labeled with $\surd$ in the table, illustrating that most events are used for the first time.

Measuring multiple performance events in parallel is possible but limited by the number of hardware counters available. Table 2 shows a selection of current ARMv7-A/R processor cores and the number of available hardware counters taken from the corresponding MPCore reference manuals. Each counter can be configured to capture a specific event. The counter is then enabled and continuously counts occurrences of the configured hardware event. Configuration and access to the counter values is realized through registers of the co-processor 15. By default, this is only allowed from privileged (e.g. kernel) code. As the goal of our work is an efficient evaluation and not an improved attack, this poses no limitation. The subsequent paragraphs in this section discuss the selection of performance events in more detail. They are organized in categories *clock cycles*, *cache*, *TLB*, and *memory access*.

Clock Cycles. The clock cycle event labeled `CCNT` is counted by the register called `PMCCNTR`, which does not occupy any of the available hardware counters. The event is known from literature and for the purpose of comparison, we include it in our event selection. In addition, the clock cycle event can also be measured through event ID $\texttt{11}_h$. We analyze it in our experiments to compare it to the `CCNT`.

Cache. Cache miss events have also been investigated in Bernstein's attack. To provide a link to previous literature, we analyze level 1 data cache misses ($\texttt{03}_h$). In addition, we include events that have a close relation to L1 D-cache misses. Those events are cache requests that miss coherently in all processor cores ($\texttt{50}_h$), clock cycles the processor core is stalled because of a pending request from a cache miss ($\texttt{61}_h$), and the number of cache eviction requests that are caused by cache misses ($\texttt{65}_h$). All of them are likely to show similar key-dependent variations as those expected for L1 D-cache miss events.

TLB. Similar to the processor cache, the translation lookaside buffer (TLB) is also involved in fetching data from main memory. It is used by the memory management unit to speed up translations of virtual addresses. Since TLBs are buffers with limited size, lookups can result in hits or misses. On the ARM Cortex-A9, the *micro* TLB is a first level TLB that is separated into instruction and data part. The *main* TLB is a unified second level TLB, which catches the misses in the underlying micro TLBs. Because of the similarities to the processor cache, we include TLB-related events in our experiments. In particular, we measure misses in the data micro or main TLB ($\texttt{05}_h$) and stall cycles caused by misses in the data micro TLB ($\texttt{85}_h$).

Memory Access. Based only on the ARM reference manuals, it is difficult to conclude whether some events are applicable to the selected cache attack. Either not enough information is provided in the event descriptions or events are by definition counted approximately rather than precisely or the final counting behavior is defined by the processor implementation itself. Because of these uncertainties, we analyze memory read and write operations causing accesses to at least the L1 data or unified cache ($\texttt{04}_h$) as well as the number of load respectively load and store instructions ($\texttt{06}_h$ and $\texttt{72}_h$). Note that the tested AES software performs a constant number of memory accesses as discussed in Appendix A.

6 Measurement Setup

For our practical case study, we implement a client-server setup as proposed in the original attack by Bernstein. The client, or spy, establishes a network connection to the victim, which is running on a Linux server system (kernel v3.19.0) featuring an ARM Cortex-A9 quad-core processor. The measurements are performed with enabled L1 and L2 caches, program flow prediction, cache pre-fetcher and cache critical word first filling. We choose a full Linux operating system and leave all hardware acceleration features enabled to provide a realistic setting for our evaluation.

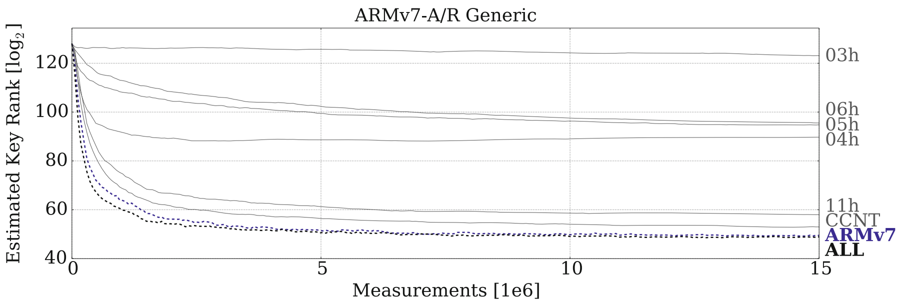
Direct access to the performance monitoring registers of the co-processor 15 is by default only allowed from kernel mode. User space access can either be enabled in the `PMUSERENR` register or realized by exposing the performance counter subsystem of the kernel with the `perf` tool set. While `perf` enables convenient user access to all the events tested in this work, using it adds another potential source of measurement noise. We avoid this by enabling direct user access in the `PMUSERENR` register with a custom system call. This has to be done once and gets reset when the system is powered off. As we conduct our experiments in an evaluation rather than an attack context, such low-level control over the target system is given.

As some of the measured performance events occur in core-private caches or TLBs, we force the victim program to run on one specific processor core. This is done with the `taskset` program from the Linux utilities. Restricting the victim to one core is no disadvantage but even beneficial. Letting the victim program float between processor cores adds noise to the measured performance events, which (1) prolongs the measurement phase until a stable attack success rate is achieved and (2) is no worst-case attack scenario that we aim to establish for this evaluation. During the measurements, the target system is idling and the victim program is competing for resources with itself and the system processes that run in the background. The AES-128 implementation under attack is written in ARM assembly, uses a 1 kiB lookup table, and executes a constant and key-value-independent number of instructions during encryption. More details about the implementation are given in Appendix A.

For every attack we take two sets of $30 \cdot 10^6$ ($\approx 2^{24.84}$) measurements, one for the learn phase and one for the attack phase. In the learn phase, we use a zero key, in the attack phase we use a random one. The attack is then performed with additional key rank estimation. For each of the selected performance events, we repeat this process 33 times in order to achieve a reasonable statistical significance for our practical experiments. We choose 33 repetitions to keep the measurement effort manageable. In the last step, the estimated key ranks of all 33 attacks are averaged for each event to form the final key ranks presented in the next section.

7 Discussion of Practical Results

Table 3 shows the attack results for the performance events in our case study. The left side of the table displays event IDs, coverage in previous literature, and descriptions. The right side of the table shows the average estimated key ranks in $\log_2$ that are achieved using the corresponding performance events. The combination `ARMv7` combines attacks from all tested ARMv7-A/R compliant events according to our method proposed in Sect. 4. The combination `ALL` combines all available attacks and additionally includes the events specific to the ARM Cortex-A9.
The first and most interesting observation from Table 3 is that all analyzed performance events reduce the entropy of the secret key. The average key ranks range between 2^{120} and 2^{51}. This is a significantly lower effort compared to searching for a full-entropy 128-bit key with an expected average key rank of 2^{127}. The best attacks with the current event selection are based on events $\texttt{11}_h$ and `CCNT`, which both count clock cycles. A further reduction of key entropy is only achieved by combining multiple attacks. The key ranks of the combinations `ARMv7` and `ALL` fall below those of the `CCNT` by 2^2 and 2^3, respectively. Although these improvements might seem moderate, they show that our proposed combination and filter method indeed excludes poor-quality side-channels and constructively combines the available leakage to improve the overall result. Events with poor attack results, such as $\texttt{03}_h$, $\texttt{50}_h$, and $\texttt{65}_h$, do not degrade the `ARMv7` and `ALL` combinations. Instead, the available side-channels improve the already good attack results retrieved from the `CCNT` measurements. The improvement is more significant, if fewer measurements are available to the attack. This is illustrated in Fig. 1, which shows the attack results over an increasing number of measurements.
Every plot in the figure shows the average estimated key ranks in $\log_2$ as the y-coordinate and the number of measurements as the x-coordinate. Note that the plots end at $15 \cdot 10^6$ measurements to better illustrate the early attack stages, in which fewer measurements are available. In this phase the combined attacks



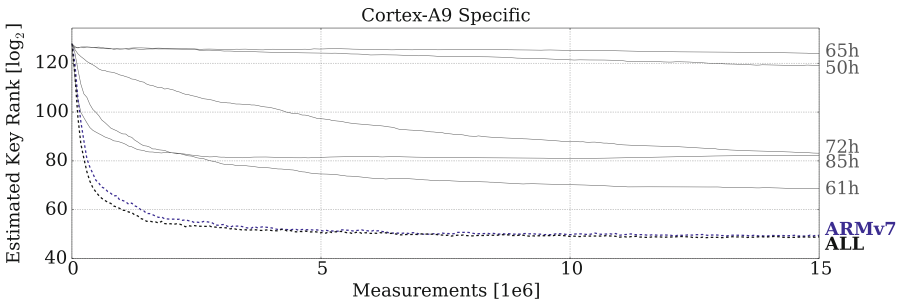


Fig. 1. Estimated key ranks over an increasing number of measurements. Top plot shows results for ARMv7-A/R events, bottom plot shows ranks for Cortex-A9 events. Combined attacks `ARMv7` and `ALL` are plotted with dashed lines and marked with bold labels.

Table 3. Estimated key ranks achieved with the selected performance events.

Performance Events				Estimated Key Ranks
	ID	Lit.	Description	[$\log_2$]
ARMv7-A/R	`03`$_h$	$\surd$	L1 D-cache refill	119
	`04`$_h$	-	L1 D-cache access	90
	`05`$_h$	-	L1 D-TLB refill	91
	`06`$_h$	-	Load instructions	93
	`11`$_h$	$\surd$	CPU cycles	57
	`CCNT`	$\surd$	Clock cycle counter	51
`ARMv7`		-	-	49
Cortex-A9	`50`$_h$	-	Coherent linefill miss	116
	`61`$_h$	-	D-cache stall cycles	67
	`65`$_h$	-	D-cache eviction requests	120
	`72`$_h$	-	Load/store instructions	76
	`85`$_h$	-	D-micro TLB stall cycles	83
`ALL`		-	-	48

Notes:

`ARMv7` .... Combination of ARMv7-A/R compliant events
`ALL` ..... Combination of all events measured on the Cortex-A9

`ARMv7` and `ALL` clearly exhibit the lowest average key ranks and consequently the best attack results in our experiments. Compared to the `CCNT`, the average key rank of the `ARMv7` combination is smaller by up to 2^{11}. The average rank of the `ALL` combination falls below the one of the `CCNT` by up to 2^{16}. These improvements decrease with more measurements, as previously noted. Further observations from both Table 3 and Fig. 1 are discussed in the following paragraphs.

Clock Cycles. The clock-cycle-based events `11`$_h$ and `CCNT` yield low average key ranks of 2^{57} and 2^{51}, respectively. We assume that the `CCNT` shows slightly better results, because once enabled, it is accessible with only one read request to the co-processor 15. All other events are counted such that their corresponding hardware counter has to be selected first in order to read its current value. This additional request to the co-processor 15 adds noise to the measurements, but is necessary if multiple events are counted in parallel. Because of its superior attack results and because it does not occupy any of the limited hardware counters in the processor, the `PMCCNTR` register is clearly recommended to measure the clock cycle event on our target system.

Cache. Among the cache-miss-related performance events, the D-cache stall cycles (`61`$_h$) yield the best attack result with an average key rank of 2^{67}. In contrast, cache misses, coherent linefill misses, and data eviction requests (`03`$_h$, `50`$_h$, and `65`$_h$) do not exhibit average key ranks below 2^{116}. One possible explanation is that stall cycles incorporate additional and more fine-grained key-dependent variations that are not reflected in the total number of cache misses or eviction requests counted by the other events.

TLB. The results of the TLB refill and TLB stall cycle events ($\texttt{05}_h$ and $\texttt{85}_h$) show that translation lookaside buffers offer a potent side-channel source for the implemented attack. These events perform superior to most cache-miss-related events on our target system. With an average key rank of 2^{83} compared to 2^{91}, the D-micro TLB stall cycles yield better results than the L1 D-TLB refills. We assume that the stall cycles again contain more exploitable information and that less noise in the micro TLB renders their results more successful.

Memory Access. The attacks based on L1 D-cache accesses as well as load and store instructions ($\texttt{04}_h$, $\texttt{06}_h$, and $\texttt{72}_h$) are successful, which is counter-intuitive given that the number of memory accesses in the tested AES implementation is constant and independent of the key value. This suggests that these events incorporate key-dependent variations such that the attack is able to reduce the secret key entropy. Since detailed information about the implementation of these events is not publicly available, further investigations are necessary to identify the source of their leakage.

Combined Attacks. The additional, Cortex-A9 specific events added to the `ALL` combination improve the average key rank of the `ARMv7` combination by at most 2^9. This maximum difference is shown in Fig. 1 before the attacks reach the $5 \cdot 10^6$ measurements mark. When more measurements are added, the improvement becomes approximately 2^1, as illustrated in Table 3. Given our event selection, this shows that good results can already be obtained using generic ARMv7-A/R events. On the other hand, combinations of platform-specific events not analyzed in this work may still be able to outperform ARMv7-A/R generic combinations.

8 Practical Evaluation Suggestions

Our case study shows that hardware performance events offer a promising pool of side-channels that can be exploited in the profiled cache attack by Bernstein. For a fair and complete assessment of the leakage contained in the events, key rank estimation has proven to be a useful tool. Although it adds complexity, determining the attack effort is unreliable in practice with the original approach. To control the computational cost, one can adapt the estimation bound tightness to one's specific requirements.

Within our selection of performance events, each one allows to reduce the entropy of the secret key. Among these events, clock cycles provide the best results on our ARM Cortex-A9 test system. If minimum measurement and post-processing effort are required, the evaluation can be limited to the `CCNT`. It is a high-resolution side-channel source that is available on all ARMv7-A/R compliant processor cores. It consequently allows to compare the leakage behavior of multiple systems in a simple way. However, further studies are necessary to verify that it also performs best on other ARM-Cortex-based systems. For now, we strongly recommend to consider more than just the clock cycle event.

Since attacking multiple performance events increases the measurement and post-processing effort, a minimum selection of events with maximum leakage is desirable. The lack of detailed information in the public ARM manuals and the uncertainty of how a processor actually counts certain events show that an optimal choice of performance events is not trivial to make. According to our case study, the ARMv7-A/R compliant performance events provide good results on the selected test system. Their combination is only slightly inferior compared to the combination of all tested events, including those specific to the ARM Cortex-A9. To maintain platform independence, we recommend to limit the tested events to ARMv7-A/R compliant ones. If fewer hardware counters are available on the processor under test, the results displayed in Table 3 and Fig. 1 can be a starting point for reducing the number of measured events. In order to compare the leakage behavior of multiple processors, we suggest to combine the attacks of each system with our proposed method. The resulting combinations represent robust overall attacks, regardless of which specific event leaks the most information on each processor. This allows to get a comprehensive view of the vulnerability of a cipher implementation on a range of different systems.

9 Conclusion

In this work, we studied the evaluation of a block cipher implementation on a modern processor with the profiled time-driven cache attack proposed by Bernstein. The application of key rank estimation as well as the combination of multiple attacks are generic extensions that might also be of interest in other cache attack work. In our case study, we identified new micro-architectural side-channels on our ARM-based test system that can successfully be exploited in Bernstein's attack. Since performance counters are not only available on ARM systems, we assume that new side-channels can also be found on other modern (e.g. x86) processors. As all tested events in the case study leak information about the secret key, the practical results strongly suggest that even more performance events might be exploitable than those analyzed both in literature and in our work. Together with the fact that the counting behavior of certain events is not properly documented and is even defined by the final processor implementation, a comprehensive study of performance events across multiple processors is a promising direction for future work. Furthermore, it is an open question whether a more effective filter approach exists that better separates good and bad attacks in the combination step. Eventually, these directions may lead to an optimal choice of events needed to evaluate the profiled cache attack by Bernstein with maximum efficiency.

Acknowledgements. This work was funded by the German Federal Ministry of Education and Research (BMBF) in the project SIBASE through grant number 01IS13020C.

A AES T-table Implementation

The profiled time-driven cache attack by Bernstein targets a software implementation of AES that uses so-called transformation tables. These T-tables are proposed by Daemon and Rijmen [9] to speed up AES in software. The tables reduce a round of AES to 16 table lookups and 16 XOR operations with 4-byte operands. The encryption process uses four 1 kiB T-tables $\mathbf{T}_{0..3}$ for all rounds except the initial and the last one, which lacks the `MixColumns` transformation. Encryption using T-tables is illustrated in Eq. 4. Note that it does not define the initial state, $\mathbf{s}^{(0)}$, and the final one, $\mathbf{s}^{(R)}$, because of their different treatment in the cipher. The cipher state in round $r \in \{1, \ldots, R-1\}$, $R \in \{10, 12, 14\}$ is given as $\mathbf{s}^{(r)}_{i..i+3}$ whereas the round key is given as $\mathbf{k}^{(r)}_{i..i+3}$, with $(i..i+3)$ denoting a consecutive 4-byte chunk of the state and the round key, respectively. The initial state $\mathbf{s}^{(0)}$ is a simple XOR operation between the plaintext $\mathbf{p}$ and the initial key $\mathbf{k}^{(0)}$, $\mathbf{s}^{(0)} = \mathbf{p} \oplus \mathbf{k}^{(0)}$.

$$\begin{aligned}
\mathbf{s}^{(r)}_{0..3} &= \mathbf{T}_0[\mathbf{s}^{(r-1)}_0] \oplus \mathbf{T}_1[\mathbf{s}^{(r-1)}_5] \oplus \mathbf{T}_2[\mathbf{s}^{(r-1)}_{10}] \oplus \mathbf{T}_3[\mathbf{s}^{(r-1)}_{15}] \oplus \mathbf{k}^{(r)}_{0..3}\\
\mathbf{s}^{(r)}_{4..7} &= \mathbf{T}_0[\mathbf{s}^{(r-1)}_4] \oplus \mathbf{T}_1[\mathbf{s}^{(r-1)}_9] \oplus \mathbf{T}_2[\mathbf{s}^{(r-1)}_{14}] \oplus \mathbf{T}_3[\mathbf{s}^{(r-1)}_{3}] \oplus \mathbf{k}^{(r)}_{4..7}\\
\mathbf{s}^{(r)}_{8..11} &= \mathbf{T}_0[\mathbf{s}^{(r-1)}_8] \oplus \mathbf{T}_1[\mathbf{s}^{(r-1)}_{13}] \oplus \mathbf{T}_2[\mathbf{s}^{(r-1)}_{2}] \oplus \mathbf{T}_3[\mathbf{s}^{(r-1)}_{7}] \oplus \mathbf{k}^{(r)}_{8..11}\\
\mathbf{s}^{(r)}_{12..15} &= \mathbf{T}_0[\mathbf{s}^{(r-1)}_{12}] \oplus \mathbf{T}_1[\mathbf{s}^{(r-1)}_{1}] \oplus \mathbf{T}_2[\mathbf{s}^{(r-1)}_{6}] \oplus \mathbf{T}_3[\mathbf{s}^{(r-1)}_{11}] \oplus \mathbf{k}^{(r)}_{12..15}
\end{aligned} \tag{4}$$

In order to reduce the storage space required by the T-table implementation, three of the T-tables can be exchanged for 12 extra rotations per round of AES. This is because each entry of a T-table is the byte-wise rotation of the same entry of any other table. The rotation factor remains constant for each table. Hence, we can rewrite Eq. 4 as follows, assuming $\mathbf{T} = \mathbf{T}_0$ is the only table available. The function *ror(v,n)* rotates the 4-byte value v by n number of bytes cyclically to the right.

$$\begin{aligned}
\mathbf{s}^{(r)}_{0..3} &= \mathbf{T}[\mathbf{s}^{(r-1)}_0] \oplus ror(\mathbf{T}[\mathbf{s}^{(r-1)}_5], 1) \oplus ror(\mathbf{T}[\mathbf{s}^{(r-1)}_{10}], 2) \oplus ror(\mathbf{T}[\mathbf{s}^{(r-1)}_{15}], 3) \oplus \mathbf{k}^{(r)}_{0..3}\\
\mathbf{s}^{(r)}_{4..7} &= \mathbf{T}[\mathbf{s}^{(r-1)}_4] \oplus ror(\mathbf{T}[\mathbf{s}^{(r-1)}_9], 1) \oplus ror(\mathbf{T}[\mathbf{s}^{(r-1)}_{14}], 2) \oplus ror(\mathbf{T}[\mathbf{s}^{(r-1)}_{3}], 3) \oplus \mathbf{k}^{(r)}_{4..7}\\
\mathbf{s}^{(r)}_{8..11} &= \mathbf{T}[\mathbf{s}^{(r-1)}_8] \oplus ror(\mathbf{T}[\mathbf{s}^{(r-1)}_{13}], 1) \oplus ror(\mathbf{T}[\mathbf{s}^{(r-1)}_{2}], 2) \oplus ror(\mathbf{T}[\mathbf{s}^{(r-1)}_{7}], 3) \oplus \mathbf{k}^{(r)}_{8..11}\\
\mathbf{s}^{(r)}_{12..15} &= \mathbf{T}[\mathbf{s}^{(r-1)}_{12}] \oplus ror(\mathbf{T}[\mathbf{s}^{(r-1)}_{1}], 1) \oplus ror(\mathbf{T}[\mathbf{s}^{(r-1)}_{6}], 2) \oplus ror(\mathbf{T}[\mathbf{s}^{(r-1)}_{11}], 3) \oplus \mathbf{k}^{(r)}_{12..15}
\end{aligned} \tag{5}$$

In our experiments we test a 1 kiB T-table implementation of AES that follows Eq. 5. It is part of the OpenSSL software library v1.0.2 and written in ARM assembly. The code is located under `crypto/aes/asm/aes-armv4.pl` in the GitHub repository of the library [16]. As suggested by the equation, the chosen implementation uses a constant and key-value-independent number of instructions. The only conditional branch in the code is used to realize the AES encryption loop. For each plaintext that is encrypted with AES-128, the processor performs 144 T-table lookups with `ldr` instructions and 16 S-box lookups with `ldrb` instructions, which are needed in the last encryption round. As it is common to many T-table-based implementations of AES, the secret key is leaked only through the table lookups themselves, because it is used to compute the indices of the tables.

References

1. Acıiçmez, O., Koç, Ç.K.: Trace-driven cache attacks on AES (Short Paper). In: Ning, P., Qing, S., Li, N. (eds.) ICICS 2006. LNCS, vol. 4307, pp. 112–121. Springer, Heidelberg (2006)
2. Apecechea, G.I., Inci, M.S., Eisenbarth, T., Sunar, B.: Fine grain Cross-VM attacks on Xen and VMware are possible! Cryptology ePrint Archive, Report 2014/248 (2014). http://eprint.iacr.org/
3. ARM: ARM Cortex-A9 MPCore Technical Reference Manual, June 2012. Revision r4p1
4. ARM: ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition, May 2014. Revision C.c
5. Atici, A., Yilmaz, C., Savas, E.: An approach for isolating the sources of information leakage exploited in cache-based side-channel attacks. In: 2013 IEEE 7th International Conference on Software Security and Reliability-Companion (SERE-C), pp. 74–83, June 2013
6. Bernstein, D.J.: Cache-timing attacks on AES. Tech. rep., The University of Illinois at Chicago (2005). http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
7. Bogdanov, A., Eisenbarth, T., Paar, C., Wienecke, M.: Differential cache-collision timing attacks on AES with applications to embedded CPUs. In: Pieprzyk, J. (ed.) CT-RSA 2010. LNCS, vol. 5985, pp. 235–251. Springer, Heidelberg (2010)
8. Bonneau, J., Mironov, I.: Cache-collision timing attacks against AES. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 201–215. Springer, Heidelberg (2006)
9. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer-Verlag New York Inc., Secaucus (2002)
10. Glowacz, C., Grosso, V., Poussier, R., Schueth, J., Standaert, F.X.: Simpler and more efficient rank estimation for side-channel security assessment. Cryptology ePrint Archive, Report 2014/920 (2014). http://eprint.iacr.org/
11. Gérard, B., Standaert, F.-X.: Unified and optimized linear collision attacks and their application in a non-profiled setting. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 175–192. Springer, Heidelberg (2012)
12. Mather, L., Oswald, E., Whitnall, C.: Multi-target DPA attacks: pushing DPA beyond the limits of a desktop computer. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8873, pp. 243–261. Springer, Heidelberg (2014)
13. Neve, M., Seifert, J.P., Wang, Z.: A refined look at bernstein's aes side-channel analysis. In: Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2006, pp. 369–369. ACM, New York (2006)
14. Spreitzer, R., Gérard, B.: Towards more practical time-driven cache attacks. In: Naccache, D., Sauveron, D. (eds.) WISTP 2014. LNCS, vol. 8501, pp. 24–39. Springer, Heidelberg (2014)
15. Spreitzer, R., Plos, T.: On the applicability of time-driven cache attacks on mobile devices. In: Lopez, J., Huang, X., Sandhu, R. (eds.) NSS 2013. LNCS, vol. 7873, pp. 656–662. Springer, Heidelberg (2013)

16. The OpenSSL Project: OpenSSL (2015). https://github.com/openssl/openssl
17. Tiri, K., Acıiçmez, O., Neve, M., Andersen, F.: An analytical model for time-driven cache attacks. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 399–413. Springer, Heidelberg (2007)
18. Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on aes, and countermeasures. J. Cryptology **23**(2), 37–71 (2010)
19. Uhsadel, L., Georges, A., Verbauwhede, I.: Exploiting hardware performance counters. In: 5th Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2008, pp. 59–67, August 2008
20. Veyrat-Charvillon, N., Gérard, B., Renauld, M., Standaert, F.-X.: An optimal key enumeration algorithm and its application to side-channel attacks. In: Knudsen, L.R., Wu, H. (eds.) SAC 2012. LNCS, vol. 7707, pp. 390–406. Springer, Heidelberg (2013)
21. Veyrat-Charvillon, N., Gérard, B., Standaert, F.-X.: Security evaluations beyond computing power. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 126–141. Springer, Heidelberg (2013)
22. Weiß, M., Heinz, B., Stumpf, F.: A cache timing attack on AES in virtualization environments. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 314–328. Springer, Heidelberg (2012)
23. Weiß, M., Weggenmann, B., August, M., Sigl, G.: On cache timing attacks considering multi-core aspects in virtualized embedded systems. In: Yung, M., Zhu, L., Yang, Y. (eds.) INTRUST 2014. LNCS, vol. 9473, pp. 151–167. Springer, Switzerland (2014)