J. Cryptol.
DOI: 10.1007/s00145-009-9049-y

Journal of
CRYPTOLOGY

Efficient Cache Attacks on AES, and Countermeasures

Eran Tromer

Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology,
32 Vassar Street, G682, Cambridge, MA 02139, USA
tromer@csail.mit.edu
and
Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100,
Israel

Dag Arne Osvik

Laboratory for Cryptologic Algorithms, Station 14, École Polytechnique Fédérale de Lausanne,
1015 Lausanne, Switzerland
dagarne.osvik@epfl.ch

Adi Shamir

Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100,
Israel
adi.shamir@weizmann.ac.il

Communicated by Lars R. Knudsen

Received 20 July 2007 and revised 25 June 2009

Abstract. We describe several software side-channel attacks based on inter-process leakage through the state of the CPU's memory cache. This leakage reveals memory access patterns, which can be used for cryptanalysis of cryptographic primitives that employ data-dependent table lookups. The attacks allow an unprivileged process to attack other processes running in parallel on the same processor, despite partitioning methods such as memory protection, sandboxing, and virtualization. Some of our methods require only the ability to trigger services that perform encryption or MAC using the unknown key, such as encrypted disk partitions or secure network links. Moreover, we demonstrate an extremely strong type of attack, which requires knowledge of neither the specific plaintexts nor ciphertexts and works by merely monitoring the effect of the cryptographic process on the cache. We discuss in detail several attacks on AES and experimentally demonstrate their applicability to real systems, such as OpenSSL and Linux's `dm-crypt` encrypted partitions (in the latter case, the full key was recovered after just 800 writes to the partition, taking 65 milliseconds). Finally, we discuss a variety of countermeasures which can be used to mitigate such attacks.

Key words. Side-channel attack, Cryptanalysis, Memory cache, AES.

Published online: 23 July 2009

1. Introduction

1.1. *Overview*

Many computer systems concurrently execute programs with different privileges, employing various partitioning methods to facilitate the desired access control semantics. These methods include kernel vs. userspace separation, process memory protection, filesystem permissions and `chroot`, and various approaches to virtual machines and sandboxes. All of these rely on a model of the underlying machine to obtain the desired access control semantics. However, this model is often idealized and does not reflect many intricacies of the actual implementation.

In this paper we show how a low-level implementation detail of modern CPUs, namely the structure of memory caches, causes subtle indirect interaction between processes running on the same processor. This leads to cross-process information leakage. In essence, the cache forms a shared resource which all processes compete for, and it thus affects and is affected by every process. While the *data* stored in the cache is protected by virtual memory mechanisms, the *metadata* about the contents of the cache, and in particular the memory access patterns of processes using that cache, are not fully protected.

We describe several methods an attacker can use to learn about the memory access patterns of another process, e.g., one which performs encryption with an unknown key. These are classified into methods that affect the state of the cache and then measure the effect on the running time of the encryption, and methods that investigate the state of the cache after or during encryption. The latter are found to be particularly effective and noise-resistant.

We demonstrate the cryptanalytic applicability of these methods to the Advanced Encryption Standard (AES, [39]) by showing a known-plaintext (or known-ciphertext) attack that performs efficient full key extraction. For example, an implementation of one variant of the attack performs full AES key extraction from the `dm-crypt` system of Linux using only 800 accesses to an encrypted file, 65 ms of measurements and 3 seconds of analysis; attacking simpler systems, such as "black-box" OpenSSL library calls, is even faster at 13 ms and 300 encryptions.

One variant of our attack has the unusual property of performing key extraction *without knowledge of either the plaintext or the ciphertext*. This is a particularly strong form of attack, which is clearly impossible in a classical cryptanalytic setting. It enables an unprivileged process, merely by accessing its own memory space, to obtain bits from a secret AES key used by another process, without any (explicit) communication between the two. This too is demonstrated experimentally, and implementing AES in a way that is impervious to this attack, let alone developing an efficient generic countermeasure, appears nontrivial.

This paper is organized as follows: Sect. 2 gives an introduction to memory caches and AES lookup tables. In Sect. 3 we describe the basic attack techniques, in the "synchronous" setting where the attacker can explicitly invoke the cipher on known data. Section 4 introduces even more powerful "asynchronous" attacks which relax the latter requirement. In Sect. 5, various countermeasures are described and analyzed. Section 6 summarizes these results and discusses their implications.

1.2. *Related Work*

The possibility of cross-process leakage via cache state was first considered in 1992 by Hu [24] in the context of intentional transmission via covert channels. In 1998, Kelsey et al. [27] mentioned the prospect of "attacks based on cache hit ratio in large S-box ciphers." In 2002, Page [47] described theoretical attacks on DES via cache misses, assuming an initially empty cache and the ability to identify cache effects with very high temporal resolution in side-channel traces. He subsequently proposed several countermeasures for smartcards [48], though most of these require hardware modifications and are inapplicable or insufficient in our attack scenario. Recently, variants of this attack (termed "trace-driven" in [48]) were realized by Bertoni et al. [11] and Acıiçmez and Koç [3,4], using a power side-channel of a MIPS microprocessor in an idealized simulation. By contrast, our attacks operate purely in software and are hence of wider applicability and implications; they have also been experimentally demonstrated in real-life scenarios.

In 2002 and subsequently, Tsunoo et al. devised a timing-based attack on MISTY1 [57,58] and DES [56], exploiting the effects of collisions between the various memory lookups invoked internally by the cipher (as opposed to the cipher vs. attacker collisions we investigate, which greatly improve the efficiency of an attack). Recently Lauradoux [32] and Canteaut et al. [18] proposed some countermeasures against these attacks, none of which are satisfactory against our attacks (see Sect. 5).

An abridged version of this paper was published in [45] and announced in [44].

Concurrently but independently, Bernstein [10] described attacks on AES that exploit timing variability due to cache effects. This attack can be seen as a variant of our Evict + Time measurement method (see Sect. 3.4 and the analysis of Neve et al. [42]), though it is also somewhat sensitive to the aforementioned collision effects. The main difference is that [10] does not use an explicit model of the cache and active manipulation, but rather relies only on the existence of some consistent statistical patterns in the encryption time caused by memory access effects; these patterns are neither controlled nor modeled. The resulting attack is simpler and more portable than ours, since its implementation is mostly oblivious to the fine (and often unpublished) details of the targeted CPU and software; indeed, [10] includes the concise C source code of the attack. Moreover, the attack of [10] locally executes only time measurement code on the attacked computer, whereas our attack code locally executes more elaborate code that also performs (unprivileged) memory accesses. However, the attack of [10] has several shortcomings. First, it requires reference measurements of encryption under *known* key in an identical configuration, and these are often not readily available (e.g., a user may be able to write data to an encrypted filesystem, but creating a reference filesystem with a known key is a privileged operation). Second, the attack of [10] relies on timing the encryption and thus, similarly to our Evict + Time method, seems impractical on many real systems due to excessively low signal-to-noise ratio; our alternative methods (Sects. 3.5 and 4) address this. Third, even when the attack of [10] works, it requires a much higher number of analyzed encryptions than our method.[1] A subsequent paper of

[1] In our experiments the attack code of [10] failed to get a signal from `dm-crypt` even after a 10 hours run, whereas in the same setup our Prime + Probe (see Sect. 3.5) performed full key recovery using 65 ms of measurements.

Canteaut et al. [18] describes a variant of Bernstein's attack which focuses on internal collisions (following Tsunoo et al.) and provides a more in-depth experimental analysis;[2] its properties and applicability are similar to Bernstein's attack.[3] See Sect. 6.5 for subsequent improvements.

Also concurrently with but independently of our work, Percival [50] described a cache-based attack on RSA for processors with simultaneous multithreading. The measurement method is similar to one variant of our asynchronous attack (Sect. 4), but the cryptanalysis has little in common since the algorithms and time scales involved in RSA vs. AES operations are very different. Both [10] and [50] contain discussions of countermeasures against the respective attacks, and some of these are also relevant to our attacks (see Sect. 5).

Koeune and Quisquater [30] described a timing attack on a "bad implementation" of AES which uses its algebraic description in a "careless way" (namely, using a conditional branch in the MixColumn operation). That attack is not applicable to common software implementations but should be taken into account in regard to certain countermeasures against our attacks (see Sect. 5.2).

Leakage of memory access information has also been considered in other contexts, yielding theoretical [22] and heuristic [63,64] mitigation methods; these are discussed in Sect. 5.3.

See Sect. 6.5 for a discussion of additional works following our research.

2. Preliminaries

2.1. *Memory and Cache Structure*

Over the past couple of decades, CPU speed (in terms of operations per second) has been benefiting from Moore's law and growing at rate of roughly 60% per year, while the latency of main memory has been decreasing at a much slower rate (7–9% per year).[4] Consequentially, a large gap has developed between the two. Complex multilevel cache architectures are employed to bridge this gap, but it still shows through during cache misses: on a typical modern processor, accessing data in the innermost (L1) cache typically requires amortized time on the order of 0.3 ns, while accessing main memory may stall computation for 50 to 150 ns, i.e., a slowdown of 2–3 orders of magnitude. The cache architectures are optimized to minimize the number of cache misses for typical access patterns but can be easily manipulated adversarially; to do so we will exploit the special structure in the association between main memory and cache memory.

[2] Canteaut et al. [18] claim that their attack exploits only collision effects due to microarchitectural details (i.e., low address bits) and that Bernstein's attack [10] exploits only cache misses (i.e., higher address bits). However, experimentally both attacks yield key bits of both types, as can be expected: the analysis method of [10] also detects collision effects (albeit with lower sensitivity), while the attack setting of [18] inadvertently also triggers systematic cache misses (e.g., due to the encryption function's use of stack and buffers).

[3] [18] reports an 85% chance of recovering 20 bits using 2^{30} encryptions after a 2^{30} learning phase, even for the "lightweight" target of OpenSSL AES invocation. In the same setting, our attack reliably recovers the full key from just 300 encryptions (Sect. 3.7).

[4] This relatively slow reduction in DRAM latency has proven so reliable, and founded in basic technological hurdles, that it has been proposed by Abadi et al. [1] and Dwork et al. [21] as a basis for proof-of-work protocols.

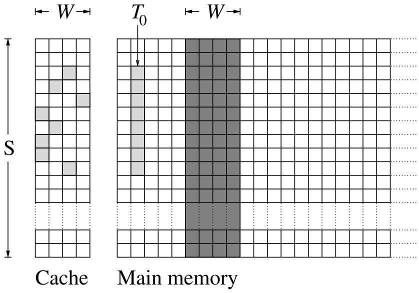

Fig. 1. Schematic of a single level of set-associative cache. Each column of memory blocks (*right side*) corresponds to $S \cdot B$ contiguous bytes of memory. Each row of memory blocks is mapped to the corresponding row in the cache (*left side*), representing a set of W cache lines. The *light gray blocks* represent an AES lookup table in the victim's memory. The *dark gray blocks* represent the attacker's memory used for the attack, which will normally be at least as big as the size of the cache.

Modern processors use one or more levels of *set-associative memory cache*. Such a cache consists of storage cells called *cache lines*, each consisting of B bytes. The cache is organized into S *cache sets*, each containing W cache lines,[5] so overall the cache contains $B \cdot S \cdot W$ bytes. The mapping of memory addresses into the cache is limited as follows. First, the cache holds copies of aligned blocks of B bytes in main memory (i.e., blocks whose starting address is 0 modulo B), which we will term *memory blocks*. When a cache miss occurs, a full memory block is copied into one of the cache lines, replacing ("evicting") its previous contents. Second, each memory block may be cached only in a specific cache set; specifically, the memory block starting at address a can be cached only in the W cache lines belonging to cache set $\lfloor a/B \rfloor \bmod S$. See Fig. 1. Thus, the memory blocks are partitioned into S classes, where the blocks in each class contend for the W cache lines in a single cache set.[6]

Modern processors have up to 3 levels of memory cache, denoted L1 to L3, with L1 being the smallest and fastest cache and subsequent levels increasing in size and latency. For simplicity, in the following we mostly ignore this distinction; one has a choice of which cache to exploit, and our experimental attacks used both L1 and L2 effects. Additional complications are discussed in Sect. 3.6. Typical cache parameters are given in Table 1.

2.2. *Memory Access in AES Implementations*

This paper focuses on AES, since its memory access patterns are particularly susceptible to cryptanalysis (see Sect. 6.2 for a discussion of other ciphers). The cipher is abstractly defined by algebraic operations and could, in principle, be directly implemented

[5] In common terminology, W is called the *associativity*, and the cache is called W*-way set associative*.

[6] CPUs differ in their policy for choosing which cache line inside a set to evict during a cache miss. Our attacks work for all common algorithms, but as discussed in Sect. 3.8, knowledge of the policy allows further improvements.

Table 1. Data cache parameters for popular CPU models.

CPU model	Level	B (cache line size)	S (cache sets)	W (associativity)	$B \cdot S \cdot W$ (total size)
Athlon 64/Opteron	L1	64 B	512	2	64 KB
Athlon 64/Opteron	L2	64 B	1024	16	1024 KB
Pentium 4E	L1	64 B	32	8	16 KB
Pentium 4E	L2	128 B	1024	8	1024 KB
PowerPC 970	L1	128 B	128	2	32 KB
PowerPC 970	L2	128 B	512	8	512 KB
UltraSPARC T1	L1	16 B	128	4	8 KB
UltraSPARC T1	L2	64 B	4096	12	3072 KB

using just logical and arithmetic operations.[7] However, performance-oriented software implementations on 32-bit (or higher) processors typically use an alternative formulation based on lookup tables, as prescribed in the Rijndael specification [19,20]. In the subsequent discussion we assume the following implementation, which is typically the fastest.[8]

Several lookup tables are precomputed once by the programmer or during system initialization. There are 8 such tables, T_0, T_1, T_2, T_3 and $T_0^{(10)}$, $T_1^{(10)}$, $T_2^{(10)}$, $T_3^{(10)}$, each containing 256 4-byte words. The contents of the tables, defined in [20], are inconsequential for most of our attacks.

During key setup, a given 16-byte secret key $\boldsymbol{k} = (k_0, \dots, k_{15})$ is expanded into 10 round keys[9] $\boldsymbol{K}^{(r)}$ for $r = 1, \dots, 10$. Each round key is divided into 4 words of 4 bytes each: $\boldsymbol{K}^{(r)} = (K_0^{(r)}, K_1^{(r)}, K_2^{(r)}, K_3^{(r)})$. The 0th round key is just the raw key $K_j^{(0)} = (k_{4j}, k_{4j+1}, k_{4j+2}, k_{4j+3})$ for $j = 0, 1, 2, 3$. The details of the rest of the key expansion are mostly inconsequential.

Given a 16-byte plaintext $\boldsymbol{p} = (p_0, \dots, p_{15})$, encryption proceeds by computing a 16-byte intermediate state $\boldsymbol{x}^{(r)} = (x_0^{(r)}, \dots, x_{15}^{(r)})$ at each round r. The initial state $\boldsymbol{x}^{(0)}$ is computed by $x_i^{(0)} = p_i \oplus k_i$ $(i = 0, \dots, 15)$. Then, the first 9 rounds are computed by updating the intermediate state as follows, for $r = 0, \dots, 8$:

$$\begin{aligned}
\big(x_0^{(r+1)}, x_1^{(r+1)}, x_2^{(r+1)}, x_3^{(r+1)}\big) &\leftarrow T_0\big[x_0^{(r)}\big] \oplus T_1\big[x_5^{(r)}\big] \oplus T_2\big[x_{10}^{(r)}\big] \oplus T_3\big[x_{15}^{(r)}\big] \oplus K_0^{(r+1)} \\
\big(x_4^{(r+1)}, x_5^{(r+1)}, x_6^{(r+1)}, x_7^{(r+1)}\big) &\leftarrow T_0\big[x_4^{(r)}\big] \oplus T_1\big[x_9^{(r)}\big] \oplus T_2\big[x_{14}^{(r)}\big] \oplus T_3\big[x_3^{(r)}\big] \oplus K_1^{(r+1)} \\
\big(x_8^{(r+1)}, x_9^{(r+1)}, x_{10}^{(r+1)}, x_{11}^{(r+1)}\big) &\leftarrow T_0\big[x_8^{(r)}\big] \oplus T_1\big[x_{13}^{(r)}\big] \oplus T_2\big[x_2^{(r)}\big] \oplus T_3\big[x_7^{(r)}\big] \oplus K_2^{(r+1)} \\
\big(x_{12}^{(r+1)}, x_{13}^{(r+1)}, x_{14}^{(r+1)}, x_{15}^{(r+1)}\big) &\leftarrow T_0\big[x_{12}^{(r)}\big] \oplus T_1\big[x_1^{(r)}\big] \oplus T_2\big[x_6^{(r)}\big] \oplus T_3\big[x_{11}^{(r)}\big] \oplus K_3^{(r+1)}
\end{aligned} \tag{1}$$

[7] Such an implementation would be immune to our attack but exhibit low performance. A major reason for the choice of Rijndael in the AES competition was the high performance of the implementation analyzed here.

[8] See Sect. 5.2 for a discussion of alternative table layouts. A common variant employs 1 or no extra tables for the last round (instead of 4); most of our attacks analyze only the first few rounds and are thus unaffected.

[9] We consider AES with 128-bit keys. The attacks can be adapted to longer keys.

Finally, to compute the last round, (1) is repeated with $r = 9$, except that $T_0, \dots, T_3$ is replaced by $T_0^{(10)}, \dots, T_3^{(10)}$. The resulting $\boldsymbol{x}^{(10)}$ is the ciphertext. Compared to the algebraic formulation of AES, here the lookup tables represent the combination of ShiftRows, MixColumns, and SubBytes operations; the change of lookup tables in the last round is due to the absence of MixColumns.

2.3. *Notation*

We treat bytes interchangeably as integers in $\{0, \dots, 255\}$ and as elements of $\{0, 1\}^8$ that can be XORed. Let δ denote the cache line size B divided by the size of each table entry (usually 4 bytes[10]); on most platforms of interest we have $\delta = 16$. For a byte y and table T_ℓ, we will denote $\langle y \rangle = \lfloor y/\delta \rfloor$ and call this *the memory block of* y in T_ℓ. The significance of this notation is as follows: two bytes y, z fulfill $\langle y \rangle = \langle z \rangle$ iff, when used as lookup indices into the same table T_ℓ, they would cause access to the same memory block,[11] i.e., such indices cannot be distinguished by a single memory access observed at block granularity. For a byte y and table T_ℓ, we say that an AES encryption *accesses the memory block of* y *in* T_ℓ if, according to the above description of AES, at some point during that encryption there is some table lookup of $T_\ell[z]$ where $\langle z \rangle = \langle y \rangle$.

In Sect. 3 we will show methods for discovering (and taking advantage of the discovery) whether the encryption code, invoked as a black box, accesses a given memory block. To this end we define the following predicate: $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y) = 1$ iff the AES encryption of the plaintext $\boldsymbol{p}$ under the encryption key $\boldsymbol{k}$ accesses the memory block of index y in T_ℓ at least once throughout the 10 rounds.

Also in Sect. 3, our measurement procedures will sample a *measurement score* from a distribution $M_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$ over $\mathbb{R}$. The exact definition of $M_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$ will vary, but it will approximate $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$ in the following rough sense: for a large fraction of the keys $\boldsymbol{k}$, all[12] tables ℓ and a large fraction of the indices $\boldsymbol{x}$, for random plaintexts and measurement noise, the expectation of $M_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$ is larger when $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y) = 1$ than when $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y) = 0$.

3. Synchronous Known-Data Attacks

3.1. *Overview*

Our first family of attacks, termed *synchronous attacks*, is applicable in scenarios where either the plaintext or ciphertext is known and the attacker can operate synchronously with the encryption on the same processor, by using (or eavesdropping upon) some interface that triggers encryption under an unknown key. For example, a Virtual Private Network (VPN) may allow an unprivileged user to send data packets through a secure

[10] One exception is OpenSSL 0.9.7g on x86-64, which uses 8-byte table entries. The reduced δ improves our attacks.

[11] We assume that the tables are aligned on memory block boundaries, which is usually the case. Non-aligned tables would *benefit* our attacks by leaking an extra bit (or more) per key byte in the first round. We also assume for simplicity that all tables are mapped into distinct cache sets; this holds with high probability on many systems (and our practical attacks can handle some exceptions).

[12] This will be relaxed in Sect. 3.7.

channel which uses the same secret key to encrypt all packets. This lets the user trigger encryption of plaintexts that are mostly known (up to some uncertainties in the packet headers), and our attack would thus, under some circumstances, enable any such user to discover the key used by the VPN to protect the packets of other users. As another example, consider the Linux `dm-crypt` and `cryptoloop` services. These allow the administrator to create a virtual device which provides encrypted storage on an underlying physical device, and typically a normal filesystem is mounted on top of the virtual device. If a user has write permissions to *any* file on that filesystem, he can use it to trigger encryptions of known plaintext, and using our attack, he is subsequently able to discover the universal encryption key used for the underlying device. We have experimentally demonstrated the latter attack and showed it to reliably extract the full AES key using about 65 ms of measurements (involving just 800 write operations) followed by 3 seconds of analysis. Note that, unlike classical known-plaintext attacks, in this scenario there is no access to the corresponding ciphertexts.

The attack consists of two stages. In the on-line stage, we obtain a set of random samples, each consisting of a known plaintext and the memory-access side-channel information gleaned during the encryption of that plaintext. This data is cryptanalyzed in an off-line stage, through hypothesis testing: we guess small parts of the key, use the guess to predict some memory accesses, and check whether the predictions are consistent with the collected data. In the following we first describe the cryptanalysis in a simplified form by assuming access to an ideal predicate Q that reveals which memory addresses were accessed by individual invocations of the cipher. We then adapt the attack to the real setting of noisy measurement M that approximate Q, show two practical methods for obtaining these measurements, report experimental results, and outline possible variants and extensions.

3.2. *One-Round Attack*

Our simplest synchronous attack exploits the fact that in the first round, the accessed table indices are simply $x_i^{(0)} = p_i \oplus k_i$ for all $i = 0, \dots, 15$. Thus, given knowledge of the plaintext byte p_i, any information on the accessed index $x_i^{(0)}$ directly translates to information on key byte k_i. The basic attack, in idealized form, is as follows.

Suppose that we obtain samples of the ideal predicate $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$ for some table ℓ, arbitrary table indices y, and known but random plaintexts $\boldsymbol{p}$. Let k_i be a key byte such that the first encryption round performs the access "$T_\ell[x_i^{(0)}]$" in (1), i.e., such that $i \equiv \ell \pmod 4$. Then we can discover the partial information $\langle k_i \rangle$ about k_i by testing candidate values $\tilde{k}_i$ and checking them the following way. Consider the samples that fulfill $\langle y \rangle = \langle p_i \oplus \tilde{k}_i \rangle$. These samples will be said to be *useful for* $\tilde{k}_i$, and we can reason about them as follows. If we correctly guessed $\langle k_i \rangle = \langle \tilde{k}_i \rangle$, then $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y) = 1$ for useful samples, since the table lookup "$T_\ell[x_i^{(0)}]$" in (1) will certainly access the memory block of y in T_ℓ. Conversely, if $\langle k_i \rangle \neq \langle \tilde{k}_i \rangle$, then we are assured that "$T_\ell[x_i^{(0)}]$" will *not* access the memory block of y during the first round; however, during the full encryption process, there is a total of 36 accesses to T_ℓ (4 in each of the first 9 AES rounds). The remaining 35 accesses are affected also by other plaintext bytes, so heuristically the probability that the encryption will not access that memory block in any round is

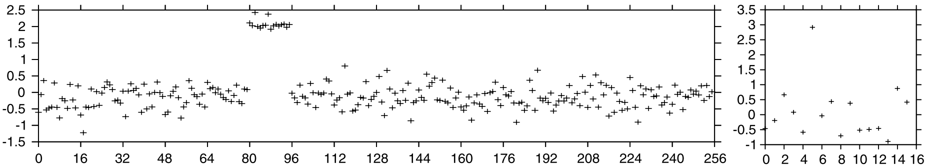

Fig. 2. Candidate scores for a synchronous attack using Prime + Probe measurements (see Sect. 3.5), analyzing a `dm-crypt` encrypted filesystem on Linux 2.6.11 running on an Athlon 64. *Left subfigure*: after analysis of 30,000 triggered encryptions. The horizontal axis is $\tilde{k}_5 = p_5 \oplus y$, and the vertical axis is the average measurement score over the samples fulfilling $y = p_5 \oplus \tilde{k}_5$ (in units of clock cycles). *Right subfigure*: after just 800 triggered encryptions, with the horizontal axis condensed to $\langle \tilde{k}_5 \rangle$. The encryption function always accesses $\langle k_5 \oplus p_5 \rangle$, and thus the high nibble of $k_5 = \text{0x50}$ is easily gleaned from the high points in either plot.

$(1 - \delta/256)^{35}$ (assuming sufficiently random plaintexts and avalanche effect). By definition, that is also the probability of $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y) = 0$, and in the common case $\delta = 16$, it is approximately 0.104.

Thus, after receiving a few dozen useful samples we can identify a correct $\langle \tilde{k}_i \rangle$—namely, the one for which $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y) = 1$ whenever $\langle y \rangle = \langle p_i \oplus \tilde{k}_i \rangle$. Applying this test to each key byte k_i separately, we can thus determine the top $\log_2(256/\delta) = 4$ bits of every key byte k_i (when $\delta = 16$), i.e., half of the AES key. Note that this is the maximal amount of information that can be extracted from the memory lookups of the first round, since they are independent and each access can be distinguished only up to the size of a memory block.

In reality, we do not have the luxury of the ideal predicate and have to deal with measurement score distributions $M_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$ that are correlated with the ideal predicate but contain a lot of (possibly structured) noise. For example, we will see that $M_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$ is often correlated with the ideal $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$ for some ℓ but is uncorrelated for others (see Fig. 5). We thus proceed by averaging over many samples. As above, we concentrate on a specific key byte k_i and a corresponding table ℓ. Our measurement will yield samples of the form $(\boldsymbol{p}, y, m)$ consisting of arbitrary table indices y, random plaintexts $\boldsymbol{p}$, and measurement scores m drawn from $M_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$. For a candidate key value $\tilde{k}_i$ we define the *candidate score of* $\tilde{k}_i$ as the expected value of m over the samples useful to $\tilde{k}_i$ (i.e., conditioned on $y = p_i \oplus \tilde{k}_i$). We estimate the candidate score by taking the average of m over the samples useful for $\tilde{k}_i$. Since $M_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$ approximates $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$, the candidate score should be noticeably higher when $\langle \tilde{k}_i \rangle = \langle k_i \rangle$ than otherwise, allowing us to identify the value of k_i up to a memory block.

Indeed, on a variety of systems we have seen this attack reliably obtaining the top nibble of every key byte. Figure 2 shows the candidate scores in one of these experiments (see Sects. 3.5 and 3.7 for details); the $\delta = 16$ key byte candidates $\tilde{k}_i$ fulfilling $\langle \tilde{k}_i \rangle = \langle k_i \rangle$ are easily distinguished.

3.3. *Two-Round Attack*

The above attack narrows each key byte down to one of δ possibilities, but the table lookups in the first AES round cannot reveal further information. For the common case $\delta = 16$, the key has 64 remaining unknown bits—still too much for exhaustive search.

We thus proceed to analyze the 2nd AES round, exploiting the nonlinear mixing in the cipher to reveal additional information. Specifically, we exploit the following equations, easily derived from the Rijndael specification [20], which give the indices used in four of the table lookups in the 2nd round:[13]

$$\begin{aligned}
x_2^{(1)} &= s(p_0 \oplus k_0) \oplus s(p_5 \oplus k_5) \oplus 2 \bullet s(p_{10} \oplus k_{10}) \oplus 3 \bullet s(p_{15} \oplus k_{15}) \oplus s(k_{15}) \oplus k_2 \\
x_5^{(1)} &= s(p_4 \oplus k_4) \oplus 2 \bullet s(p_9 \oplus k_9) \oplus 3 \bullet s(p_{14} \oplus k_{14}) \oplus s(p_3 \oplus k_3) \oplus s(k_{14}) \\
&\quad \oplus k_1 \oplus k_5 \\
x_8^{(1)} &= 2 \bullet (p_8 \oplus k_8) \oplus 3 \bullet s(p_{13} \oplus k_{13}) \oplus s(p_2 \oplus k_2) \oplus s(p_7 \oplus k_7) \oplus s(k_{13}) \\
&\quad \oplus k_0 \oplus k_4 \oplus k_8 \oplus 1 \\
x_{15}^{(1)} &= 3 \bullet s(p_{12} \oplus k_{12}) \oplus s(p_1 \oplus k_1) \oplus s(p_6 \oplus k_6) \oplus 2 \bullet s(p_{11} \oplus k_{11}) \oplus s(k_{12}) \\
&\quad \oplus k_{15} \oplus k_3 \oplus k_7 \oplus k_{11}
\end{aligned} \tag{2}$$

Here, $s(\cdot)$ denotes the Rijndael S-box function, and $\bullet$ denotes multiplication over GF(256).[14]

Consider, for example, (2) above and suppose that we obtain samples of the ideal predicate $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$ for table $\ell = 2$, arbitrary table indices y, and known but random plaintexts $\boldsymbol{p}$. We already know $\langle k_0 \rangle$, $\langle k_5 \rangle$, $\langle k_{10} \rangle$, $\langle k_{15} \rangle$, and $\langle k_2 \rangle$ from attacking the first round, and we also know the plaintext. The unknown low bits of k_2 (i.e., $k_2 \bmod \delta$) affect only the low bits of $x_2^{(1)}$ (i.e., $x_2^{(1)} \bmod \delta$), and these do not affect which memory block is accessed by "$T_2[x_2^{(1)}]$". Thus, the only unknown bits affecting the memory block accessed by "$T_2[x_2^{(1)}]$" in (1) are the lower $\log_2 \delta$ bits of k_0, k_5, k_{10}, and k_{15}. This gives a total of δ^4 (i.e., 2^{16} for $\delta = 2^4$) possibilities for candidate values $\tilde{k}_0, \tilde{k}_5, \tilde{k}_{10}, \tilde{k}_{15}$, which can be easily enumerated. To complete the recovery of these four key bytes, we can identify the correct candidate as follows.

Identification of a correct guess is done by a generalization of the hypothesis-testing method used for the one-round attack. For each candidate guess and each sample $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$, we evaluate (2) using the candidates $\tilde{k}_0, \tilde{k}_5, \tilde{k}_{10}, \tilde{k}_{15}$ while fixing the unknown low bits of k_2 to an arbitrary value. We obtain a predicted index $\tilde{x}_2^{(1)}$. If $\langle y \rangle = \langle \tilde{x}_2^{(1)} \rangle$, then we say that this sample is *useful* for this candidate, and reason as follows.

If the guess was correct, then $\langle y \rangle = \langle \tilde{x}_2^{(1)} \rangle = \langle x_2^{(1)} \rangle$, and thus "$T_2[x_2^{(1)}]$" certainly causes an access to the memory block of y in T_2, whence $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y) = 1$ by definition. Otherwise we have $k_i \neq \tilde{k}_i$ for some $i \in \{0, 5, 10, 15\}$, and thus

$$x_2^{(1)} \oplus \tilde{x}_2^{(1)} = c \bullet s(p_i \oplus k_i) \oplus c \bullet s(p_i \oplus \tilde{k}_i) \oplus \cdots$$

[13] These four equations are special in that they involve just 4 unknown quantities, as shown below.

[14] The only property of these functions that we exploit is the fact that $s(\cdot)$, $2 \bullet s(\cdot)$, and $3 \bullet s(\cdot)$ are "random-looking" in a sense specified below; this is needed for the analysis of the attack's efficiency. The actual attack implementation can be done in terms of S-box lookup tables.

for some $c \in \{1, 2, 3\}$, and since $\boldsymbol{p}$ is random, the remaining terms are independent of the first two. But for these specific functions, the above is distributed close to uniformly. Specifically, it is readily computationally verified, from the definition of AES [20], that the following differential property (cf. [13]) holds: for any $k_i \neq \tilde{k}_i$, $c \in \{1, 2, 3\}$, $\delta \geq 4$, and $z \in \{0, \dots, 256/\delta\}$, we always have

$$\Pr_{\boldsymbol{p}}\bigl[\bigl\langle c \bullet s(p_i \oplus k_i) \oplus c \bullet s(p_i \oplus \tilde{k}_i)\bigr\rangle \neq z\bigr] > 1 - (1 - \delta/256)^3$$

Thus, the probability that "$T_2[x_2^{(1)}]$" in (1) does not cause an access to the memory block of y in T_2 is at least $(1 - \delta/256)^3$, and each of the other 35 accesses to T_2 performed during the encryption will access the memory block of y in T_2 with probability $\delta/256$. Hence, $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y) = 0$ with probability greater than $(1 - \delta/256)^{3+35}$.

We see that each sample eliminates, on average, a $(\delta/256) \cdot (1 - \delta/256)^{38}$-fraction of the candidates—this is the probability, for a wrong candidate, that a random sample is useful for that candidate (i.e., yields a testable prediction) and moreover eliminates that candidate (by failing the prediction). Thus, to eliminate all the wrong candidates out of the δ^4, we need about $\log \delta^{-4} / \log(1 - \delta/256 \cdot (1 - \delta/256)^{38})$ samples, i.e., about 2056 samples when $\delta = 16$. Note that with some of our measurement methods the attack requires only a few hundred encryptions, since each encryption can provide samples for multiple y.

Similarly, each of the other three equations above lets us guess the low bits of four distinct key bytes, so taken together they reveal the full key. While we cannot reuse samples between equations since they refer to different tables ℓ, we can reuse samples between the analysis of the first and second round. Thus, if we had access to the ideal predicate Q, we would need a total of about 8220 encryptions of random plaintexts and an analysis complexity of $4 \cdot 2^{16} \cdot 2056 \approx 2^{29}$ simple tests, to extract the full AES key.

In reality we get only measurement scores from the distributions $M_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$ that approximate the ideal predicate $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$. Similarly to the one-round attack, we proceed by computing, for each candidate $\tilde{k}_i$, a candidate score obtained by averaging the measurement scores of all samples useful to $\tilde{k}_i$. We then pick the $\tilde{k}_i$ having the largest measurement score. The number of samples required to reliably obtain all key bytes by this method is, in some experimentally verified settings, only about 7 times larger than the ideal (see Sect. 3.7).

3.4. *Measurement via Evict + Time*

One method for extracting measurement scores is to manipulate the state of the cache before each encryption and observe the execution time of the subsequent encryption. Recall that we assume the ability to trigger an encryption and know when it has begun and ended. We also assume knowledge of the memory address of each table T_ℓ and hence of the cache sets to which it is mapped.[15] We denote these (virtual) memory addresses by $V(T_\ell)$. In a chosen-plaintext setting, the measurement routine proceeds as follows given a table ℓ, index y into ℓ, and plaintext $\boldsymbol{p}$:

[15] Also, as before, the cache sets of all tables are assumed to be distinct. See Sect. 3.6 for a discussion of possible complications and their resolution.

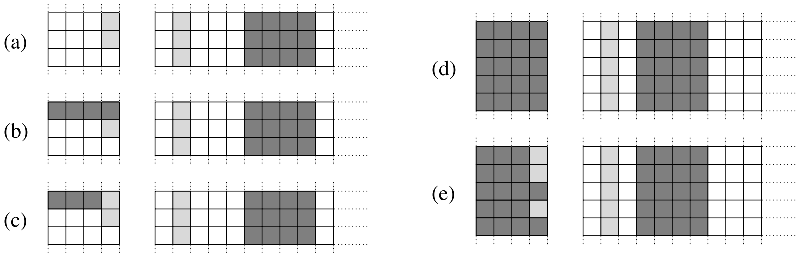

Fig. 3. Schematics of cache states, in the notation of Fig. 1. States (**a**)–(**c**) depict Evict + Time, and (**d**)–(**e**) depict Prime + Probe.

(a) Trigger an encryption of $\boldsymbol{p}$.
(b) (*evict*) Access some W memory addresses, at least B bytes apart, that are all congruent to $V(T_\ell) + y \cdot B/\delta$ modulo $S \cdot B$.
(c) (*time*) Trigger a second encryption of $\boldsymbol{p}$ and time it.[16] This is the measurement score.

The rationale for this procedure is as follows. Step (a) makes it highly likely that all table memory blocks accessed during the encryption of $\boldsymbol{p}$ are cached;[17] this is illustrated in Fig. 3(a). Step (b) then accesses memory blocks, in the attacker's own memory space, that happen to be mapped to the same cache set as the memory block of y in T_ℓ. Since it is accessing W such blocks in a cache with associativity W, we expect these blocks to completely replace the prior contents of the cache. Specifically, the memory block of index y in the encryption table T_ℓ is now not in cache; see Fig. 3(b). When we time the duration of the encryption in (c), there are two possibilities. If $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y) = 1$, that is, if the encryption of the plaintext $\boldsymbol{p}$ under the unknown encryption key $\boldsymbol{k}$ accesses the memory block of index y in T_ℓ, then this memory block will have to be re-fetched from memory into the cache, leading to Fig. 3(c). This fetching will slow down the encryption. Conversely, if $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y) = 0$, then this memory fetch will not occur. Thus, all other things being equal, the expected encryption time is larger when $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y) = 1$. The gap is on the order of the timing difference between a cache hit and a cache miss.

Figure 4 demonstrates experimental results. The bright diagonal corresponds to samples where $\langle y \rangle \oplus \langle p_0 \rangle = \langle k_0 \rangle = 0$, for which the encryption in step (c) always suffers a cache miss.

This measurement method is easily extended to a case where the attacker can trigger encryption with plaintexts that are known but not chosen (e.g., by sending network packets to which an uncontrolled but guessable header is added). This is done by replacing step (a) above with one that simply triggers encryptions of arbitrary plaintexts in order to cause *all* table elements to be loaded into cache. Then the measurement and its analysis proceed as before.

[16] To obtain high-resolution timing we use the CPU cycle counter (e.g., on x86 the RDTSC instruction returns the number of clock cycles since the last CPU reset).

[17] Unless the triggered encryption code has excessive internal cache contention, or an external process interfered.

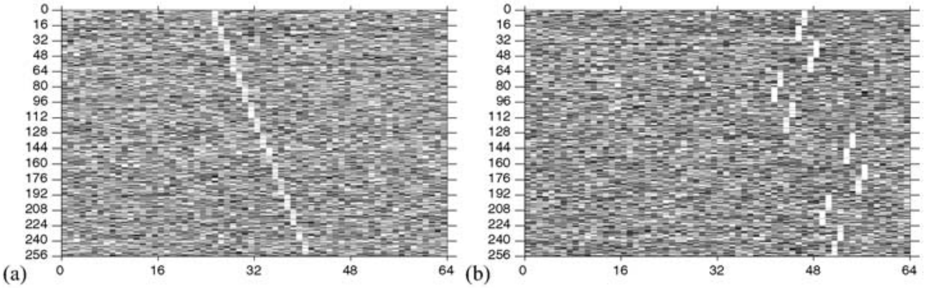

Fig. 4. Timings (lighter is slower) in Evict + Time measurements on a 2 GHz Athlon 64, after 10,000 samples, attacking a procedure that executes an encryption using OpenSSL 0.9.8. The horizontal axis is the evicted cache set (i.e., $\langle y \rangle$ plus an offset due to the table's location), and the vertical axis is p_0 (*left*) or p_5 (*right*). The patterns of bright areas reveal high nibble values of 0 and 5 for the corresponding key byte values, which are XORed with p_0.

The weakness of this measurement method is that, since it relies on timing the triggered encryption operation, it is very sensitive to variations in the operation. In particular, triggering the encryption (e.g., through a kernel system call) typically executes additional code, and thus the timing may include considerable noise due to sources such as instruction scheduling, conditional branches, page table misses, and other sources of cache contention. Indeed, using this measurement method, we were able to extract full AES keys from an artificial service doing AES encryptions using OpenSSL library calls,[18] but not from more typical "heavyweight" services. For the latter, we invoked the alternative measurement method described in the next section.

3.5. *Measurement via Prime + Probe*

This measurement method tries to discover the set of memory blocks read by the encryption *a posteriori*, by examining the state of the cache after encryption. This method proceeds as follows. The attacker allocates a contiguous byte array $A[0, \dots, S \cdot W \cdot B - 1]$, with start address congruent modulo $S \cdot B$ to the start address of T_0.[19] Then, given a plaintext $\boldsymbol{p}$, it obtains measurement scores for all tables ℓ and all indices y and does so using a *single* encryption:

(a) (*prime*) Read a value from every memory block in A.
(b) Trigger an encryption of $\boldsymbol{p}$.
(c) (*probe*) For every table $\ell = 0, \dots, 3$ and index $y = 0, \delta, 2\delta, \dots, 256 - \delta$:

- Read the W memory addresses $A[1024\ell + 4y + tSB]$ for $t = 0, \dots, W - 1$. The total time it takes to perform these W memory accesses is the measurement score for ℓ and y, i.e., our sample of $M_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$.

Step (a) completely fills the cache with the attacker's data; see Fig. 3(d). The encryption in step (b) causes partial eviction; see Fig. 3(e). Step (c) checks, for each cache set, whether the attacker's data is still present after the encryption: cache sets that were

[18] For this artificial scenario, [10] also demonstrated key extraction.

[19] For simplicity, here we assume that this address is known and that T_0, T_1, T_2, T_3 are contiguous.

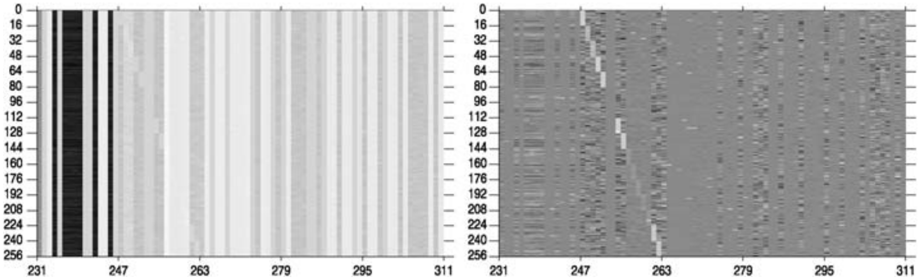

Fig. 5. Prime + Probe attack using 30,000 encryption calls on a 2 GHz Athlon 64, attacking Linux 2.6.11 `dm-crypt`. The horizontal axis is the evicted cache set (i.e., $\langle y \rangle$ plus an offset due to the table's location), and the vertical axis is p_0. *Left*: raw timings (lighter is slower). *Right*: after subtraction of the average timing of each cache set (i.e., column). The bright diagonal reveals the high nibble of $p_0 = 0\text{x}00$.

accessed by the encryption in step (b) will incur cache misses in step (c), but cache sets that were untouched by the encryption will not, and thus induce a timing difference.

Crucially, the attacker is timing a simple operation performed by *itself*, as opposed to a complex encryption service with various unknown overheads executed by something else (as in the Evict + Time approach); this is considerably less sensitive to timing variance, and oblivious to time randomization or canonization (which are frequently proposed countermeasures against timing attacks; see Sect. 5). Another benefit lies in inspecting all cache sets in one go after each encryption, so that each encryption effectively yields $4 \cdot 256/\delta$ samples of measurement score, rather than a single sample.

An example of the measurement scores obtained by this method, for a real cryptographic system, are shown in Fig. 5. Note that to obtain a visible signal it is necessary to normalize the measurement scores by subtracting, from each sample, the average timing of its cache set. This is because different cache sets are affected differently by auxiliary memory accesses (e.g., variables on the stack and I/O buffers) during the system call. These extra accesses depend on the inspected cache set but are nearly independent of the plaintext byte; thus they affect each column uniformly and can be subtracted away. Major interruptions, such as context switches to other processes, are filtered out by excluding excessively long time measurements.

3.6. *Practical Complications*

Above we have ignored several potential complications. One of these is that the attacker does not know where the victim's lookup tables reside in memory. It may be hard to tell in advance, or it might be randomized by the victim.[20] However, the attacker usually does know the layout (up to unknown global offset) of the victim's lookup tables, and this enables the following simple procedure: try each possible table offset in turn, and apply the one-round attack assuming this offset. Then pick the offset that gave the maximal candidate score. In our experiments this method reliably finds the offset even on a real, noisy system, e.g., a standard Linux distribution running its default background tasks (see Fig. 6). Moreover, when the machine is mostly idle, it suffices to simply look for a frequently-accessed range of memory of the right size (see Fig. 8).

[20] For example, recent Linux kernels randomize memory offsets.

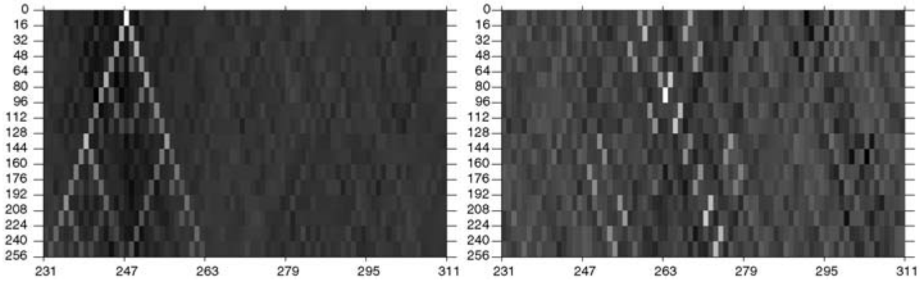

Fig. 6. Scores (lighter is higher) for combinations of key byte candidate (vertical axis) and table offset candidate (horizontal axis). The correct combinations are clearly identified as the bright spots at the head of the Sierpinski-like triangle (which is row-permuted on the right). Note the correct relative offsets of tables T_0 (*left*) and T_1 (*right*). This is the same dataset as in Fig. 5.

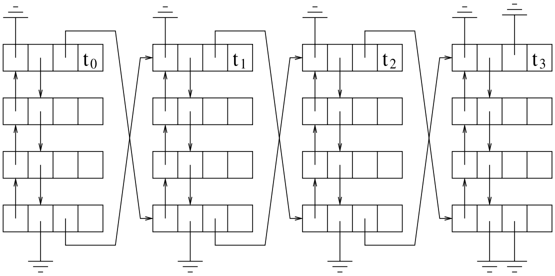

Fig. 7. Priming and probing using pointer chasing in doubly-linked list spanning cache lines. Each cache line is divided into several fields: a pointer to the next and previous cache line in the same set, a pointer to the next set, and time measurement t_i.

A naive implementation of the prime and probe steps (i.e., scanning the memory buffer in fixed strides) gives poor results due to two optimizations implemented in modern CPUs: reordering of memory accesses and automatic read-ahead of memory by the "hardware prefetcher." Our attack code works around both disruptions by using the following "pointer chasing" technique. During initialization, the attacker's memory is organized into a linked list (optionally, randomly permuted); later, priming and probing are done by traversing this list (see Fig. 7). To minimize cache thrashing (self-eviction), we use a doubly-linked list and traverse it forward for priming but backward for probing. Moreover, to avoid "polluting" its own samples, the probe code stores each obtained sample into the same cache set it has just finished measuring. On some platforms one can improve the timing gap by using writes instead of reads, or more than W reads.

The aforementioned prime and probe code is the main time-critical and machine-specific part of the attack, and was tailored by hand to the CPU at hand. The measurement obtained by this code can be read and analyzed at one's leisure (in our case, using C and Perl).

Another complication is the distinction between virtual and physical memory addresses. The mapping between the two is done in terms of full *memory pages* (i.e., aligned ranges of addresses). These can be of different sizes, even on a single system, but are usually large enough to contain all the tables used in the first 9 AES rounds. In the above descriptions, and in some of our attacks, we used the knowledge of both virtual and physical addresses of the victim's tables. Sometimes this is available (e.g., when the attacker and victim use the same shared library); it is also not a concern when the cache uses indexing by virtual address. When attacking a physically indexed cache, the attacker can run a quick preprocessing stage to gain the necessary knowledge about the mapping from virtual to physical addresses, by analysis of cache collisions between pages. Some operating systems perform page coloring [29], which makes this even easier. Alternatively, in both measurement methods, the attacker can increase the number of pages accessed to well above the cache associativity, thereby making it likely that the correct pages are hit; we have verified experimentally that this simple method works, albeit at a large cost in measurement time (a factor of roughly 300).

3.7. *Experimental Results*

We have tested the synchronous attacks against AES in various settings. To have an initial "clean" testing environment for our attack code, we started out using OpenSSL library calls as black-box functions, pretending we have no access to the key. In this setting, and with full knowledge of the relevant virtual and physical address mappings, using Prime + Probe measurements, we recover the full 128-bit AES key after only 300 encryptions on Athlon 64, and after 16,000 encryptions on Pentium 4E.[21] In the same setting but without any knowledge about address mappings (and without any attempt to discover it systematically), we still recover the full key on Athlon 64 after 8,000 encryptions.

We then proceeded to test the attacks on a real-life encrypted filesystem. We set up a Linux `dm-crypt` device, which is a virtual device that encrypts all data at the sector level. The encrypted data is saved in an underlying storage device (here, a loopback device connected to a regular file). On top of the `dm-crypt` device, we created and mounted an ordinary ext2 filesystem. The `dm-crypt` device was configured to use a 128-bit AES in ECB mode.[22] We triggered encryptions by performing writes to an ordinary file inside that file system, after opening it in `O_DIRECT` mode; each write consisted of a random 16-byte string repeated 32 times. Running this on the Athlon 64 with knowledge about address mappings, we succeeded in extracting the full key after just 800 write operations done in 65 ms (including the analysis of the cache state after each write), followed by 3 seconds of off-line analysis. Data from two analysis stages for this kind of attack are shown in Figs. 5 and 6 (the figures depict a larger number of samples, in order to make the results evident not only to sensitive statistical tests but even to cursory visual inspection).

The Evict + Time measurements (Fig. 4) are noisier, as expected, but still allow us to recover the secret key using about 500,000 samples when attacking OpenSSL on

[21] The Athlon 64 processor yielded very stable timings, whereas the Pentium 4E timings exhibited considerable variance (presumably, due to some undocumented internal state).

[22] Our tests used ECB mode in order to have, for each disk block encryption, identical known plaintexts in all AES invocations. In the recommended mode, namely CBC, the synchronous attack is less efficient since there is a lower probability that a given memory block in the S-box tables will remain unaccessed throughout the disk block encryption. It may still be feasible, e.g., if the tables are not aligned to memory blocks.

Athlon 64. Gathering the data takes about half a minute of continuous measurement, more than three orders of magnitude slower than the attacks based on Prime + Probe.

3.8. *Variants and Extensions*

There are many possible extensions to the basic techniques described above. The following are a few notable examples.

Known-Ciphertext Attacks So far we have discussed known-plaintext attacks. All of these techniques can be applied analogously in known-ciphertext setting. In fact, for AES implementations of the form given in Sect. 2.2, known-ciphertext attacks are more efficient than known-plaintext ones: the last round uses a dedicated set of tables, which eliminates the noise due to other rounds (assuming the two sets of tables map to disjoint subsets of the cache). Moreover, the last round has nonlinearity but no MixColumn operation, so the key can be extracted byte-by-byte without analyzing additional rounds. Indeed, this was demonstrated by [41] (subsequent to [44]); see Sect. 6.5. Since the round subkey derivation process in AES is reversible, recovering the last round's subkey yields the full key.

Also, even in the case of a known-plaintext attack, the final guess of the key can be efficiently verified by checking the resulting predictions for the lookups in the last round.

Note that in some scenarios, like the attacker having access to an encrypted partition, the ciphertext may not be available.

Attacking AES Decryption Since AES decryption is very similar to encryption, all of our attacks can be applied to the decryption code just as easily. Moreover, the attacks are also applicable when AES is used in MAC mode, as long as either the input or output of some AES invocations is known.

Reducing Analysis Complexity In the two-round attack, we can guess byte *differences* $\tilde{\Delta} = k_i \oplus k_j$ and consider plaintexts such that $p_i \oplus p_j = \tilde{\Delta}$, in order to cancel out pairs of terms $S(k_i \oplus p_i) \oplus S(k_j \oplus p_j)$ in (2). This reduces the complexity of analysis (we guess just $\tilde{\Delta}$ instead of both $\tilde{k_i}$ and $\tilde{k_j}$), at the cost of using more measurements.

Redundant Analysis To verify the results of the second-round analysis, or in case some of the tables cannot be analyzed due to excessive noise, we can use the other 12 lookups in the second round, or even analyze the third round, by plugging in partial information obtained from good tables.

Sub-cacheline Leakage Typically, loading a memory block into a cache line requires several memory transfer cycles due to the limited bandwidth of the memory interface. Consequently, on some processors the load latency depends on the offset of the address within the loaded memory block. Such variability can leak information on memory accesses with resolution better than δ, hence an analysis of the first round via Evict + Time can yield additional key bits. Cache bank collisions (e.g., in Athlon 64 processors) likewise cause timing to be affected by low address bits.

Detection of Eviction Depth The Prime + Probe measurement can be extended to reveal not only whether a given cache set was accessed but also the *number* of evictions from that cache set (i.e., the number of accessed distinct memory blocks mapped to that cache set). This means that the accesses of interest, such as S-box lookups, can be detected even when "masked" by accesses to the same cache set, whether accidental (e.g., to the stack or I/O buffers; see Fig. 5) or intentional (as an attempted countermeasure). For example, in the case of a set-associative cache employing an *LRU* eviction algorithm, each distinct memory block accessed by the victim will evict exactly one of the attacker's memory blocks from the corresponding cache set (if any are left); consequentially, the Probe time for a given cache set is roughly linear in the number of distinct accessed memory blocks mapped to that set (for up to W such blocks). For the *pseudo-LRU* (also called *tree-LRU*) eviction algorithm, this leakage is slightly weaker: the number of detected evictions equals the number of cache set accesses done by the victim for up to $\log_2(W) + 1$ such accesses;[23] beyond this bound, the two values are still highly correlated if accesses are sufficiently random. The attacker can also find the exact number of evictions from a pseudo-LRU cache by repeating the measurement experiment multiple times using different orderings of probed addresses.

Remote Attacks We believe that this attack can be converted into a remote attack on a network-triggerable cryptographic network process (e.g., IPsec [28] or OpenVPN [43]).[24] The cache manipulation can be done remotely, for example, by triggering accesses to the state tables employed by the host's TCP stack, stateful firewall, or VPN software. These state tables reside in memory and are accessed whenever a packet belonging to the respective network connection is seen. The attacker can thus probe different cache sets by sending packets along different network connections and also measure access times by sending packets that trigger a response packet (e.g., an acknowledgment or error). If a large number of new connections is opened simultaneously, the memory addresses of the slots assigned to these connections in the state tables will be strongly related (e.g., contiguous or nearly so), and can be further ascertained by finding slots that are mapped to the same cache set (by sending appropriate probe packets and checking the response time). Once the mapping of the state table slots to cache sets is established, all of the aforementioned attacks can be carried out; however, the signal-to-noise (and thus, the efficiency) of this technique remains to be evaluated.

4. Asynchronous Attacks

4.1. *Overview*

While the synchronous attack presented in the previous section leads to very efficient key recovery, it is limited to scenarios where the attacker has some interaction with the encryption code which allows him to obtain known plaintexts and execute code

[23] In a W-associative cache with binary-tree pseudo-LRU, the victim evicts its own data using $\log_2(W)+2$ accesses, but no fewer, assuming that the cache initially does not contain any data from the victim's memory space.

[24] This is ruled out in [18], though no justification is given.

just before and just after the encryption. We now proceed to describe a class of attacks that eliminate these prerequisites. The attacker will execute his own program on the same processor as the encryption program but without any explicit interaction such as inter-process communication or I/O, and the only knowledge assumed is about the nonuniform distribution of the plaintexts or ciphertexts (rather than their specific values). Essentially, the attacker will ascertain patterns of memory access performed by other processes just by performing and measuring accesses to its own memory. This attack is more constrained in the hardware and software platforms to which it applies, but it is very effective on certain platforms, such as the increasingly popular CPU architectures which implement simultaneous multithreading.

4.2. *One-Round Attack*

The basic form of this attack works by obtaining a statistical profile of the frequency of cache set accesses. The means of obtaining this will be discussed in the next section, but for now we assume that for each table T_ℓ and each memory block $n = 0, \dots, 256/\delta - 1$, we have a *frequency score* value $F_\ell(n) \in \mathbb{R}$ that is strongly correlated with the relative frequencies of the victim's table lookups.[25] For a simple but common case, suppose that the attacker process is performing AES encryption of English text, in which most bytes have their high nibble set to 6 (i.e., lowercase letters a through o). Since the actual table lookups performed in round 1 of AES are of the form "$T_\ell[x_i^{(0)}]$" where $x_i^{(0)} = p_i \oplus k_i$, the corresponding frequency scores $F_\ell(n)$ will have particularly large values when $n = 6 \oplus \langle k_i \rangle$ (assuming $\delta = 16$). Thus, just by finding the n for which $F_\ell(n)$ is large and XORing them with the constant 6, we get the high nibbles $\langle k_i \rangle$.

Note, however, that we cannot distinguish the order of different memory accesses to the same table and thus cannot distinguish between key bytes k_i involved in the first-round lookup to the same table ℓ. There are four such key bytes per table (for example, k_0, k_5, k_{10}, k_{15} affect T_0; see Sect. 2.2). Thus, when the four high key nibbles $\langle k_i \rangle$ affecting each table are distinct (which happens with probability $((16!/12!)/16^4)^4 \approx 0.2$), the above reveals the top nibbles of all key bytes but only up to four disjoint permutations of 4 elements each. Overall this gives $64 - \log_2(4!^4) \approx 45.66$ bits of key information, somewhat less than the one-round synchronous attack. When the high key nibbles are not necessarily disjoint we get more information, but the analysis of the signal is somewhat more complex.

More generally, suppose the attacker knows the first-order statistics of the plaintext; these can usually be determined just from the type of data being encrypted (e.g., English text, numerical data in decimal notation, machine code, or database records).[26] Specifically, suppose that the attacker knows $R(n) = \Pr[\langle p_i \rangle = n]$ for $n = 0, \dots, (256/\delta - 1)$, i.e., the histogram of the plaintext bytes truncated into blocks of size δ (where the probability is over all plaintext blocks and all bytes i inside each block). Then the partial key values $\langle k_i \rangle$ can be identified by finding those that yield maximal correlation between $F_\ell(n)$ and $R(n \oplus \langle k_i \rangle)$.

[25] Roughly, $F_\ell(n)$ is the average time (cycle count) it takes the attacker to access memory mapped to the same cache set as $T_\ell[n]$.

[26] Note that even compressed data is likely to have strong first-order statistical biases at the beginning of each compressed chunk, especially when file headers are employed.

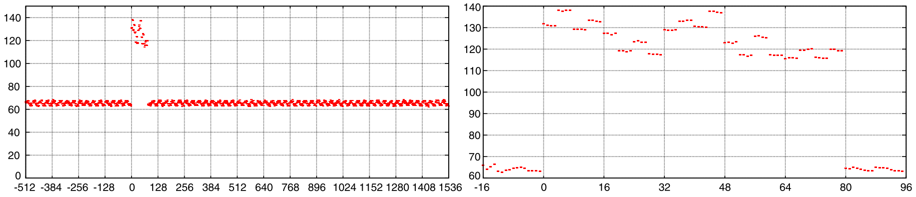

Fig. 8. Frequency scores for OpenSSL AES encryption of English text. Horizontal axis: cache set. Timings performed on 3 GHz Pentium 4E with HyperThreading. To the *right* we zoom in on the AES lookup tables; the pattern corresponds to the top nibbles of the secret key `0x004080C0105090D02060A0E03070B0F0`.

4.3. *Measurements*

One measurement method exploits the simultaneous multithreading (SMT, called "HyperThreading" in Intel Corp. nomenclature) feature available in some high-performance processors (e.g., most modern Pentium and Xeon processors, as well as POWER5 processors, UltraSPARC T1, and others).[27] This feature allows concurrent execution of multiple processes on the same physical processor, with instruction-level interleaving and parallelism. When the attacker process runs concurrently with its victim, it can analyze the latter's memory accesses in real time and thus obtain higher resolution and precision; in particular, it can gather detailed statistics such as the frequency scores $F_\ell(n) \in \mathbb{R}$. This can be done via a variant of the Prime + Probe measurements of Sect. 3.5 as follows.

For each cache set, the attacker thread runs a loop which closely monitors the time it takes to repeatedly load a set of memory blocks that exactly fills that cache set with W memory blocks (similarly to step (c) of the Prime + Probe measurements).[28] As long as the attacker is alone in using the cache set, all accesses hit the cache and are very fast. However, when the victim thread accesses a memory location which maps to the set being monitored, this causes one of the attacker's cache lines to be evicted from cache and replaced by a cache line from the victim's memory. This leads to one or (most likely) more cache misses for the attacker in subsequent loads and slows him down until his memory once more occupies all the entries in the set. The attacker thus measures the time over an appropriate number of accesses and computes their average, thus obtaining the frequency score $F_\ell(n)$.

4.4. *Experimental Results*

Attacking a series of processes encrypting English text with the same key using OpenSSL, we effectively retrieve 45.7 bits of information[29] about the key after gathering timing data for just 1 minute. Timing data from one of the runs is shown in Fig. 8.

[27] We stress that this attack can be carried out also in the absence of simultaneous multithreading; see Sect. 4.5.

[28] Due to the time-sensitivity and effects such as prefetching and instruction reordering, getting a significant signal requires a carefully crafted architecture-specific implementation of the measurement code.

[29] For keys with distinct high nibbles in each group of 4; see Sect. 4.1.

4.5. *Variants and Extensions*

This attack vector is quite powerful and has numerous possible extensions, such as the following.

Second-Round Analysis The second round can be analyzed using higher-order statistics on the plaintext, yielding enough key bits for exhaustive search.

Detecting Access Order If measurements can be made to detect the order of accesses (which we believe is possible with appropriately crafted code), the attacker can analyze more rounds, as well as extract the unknown permutations from the first round. Moreover, if the temporal resolution suffices to observe adjacent rounds in a single encryption, then it becomes possible to recover the complete key without even knowing the plaintext *distribution*, as long as it is sufficiently nonuniform.

Other Architectures We have demonstrated the attack on a Pentium 4E with Hyper-Threading, but it can also be performed on other platforms without relying on simultaneous multithreading. The essential requirements is that the attacker can execute its own code while an encryption is in progress, and this can be achieved by exploiting the interrupt mechanism. For example, the attacker can predict RTC or timer interrupts, and yields the CPU to the encrypting process a few cycles before such an interrupt; the OS scheduler is invoked during the interrupt, and if dynamic priorities are set up appropriately in advance, then the attacker process will regain the CPU and can analyze the state of the cache to see with great accuracy what the encrypting process accessed during those few cycles.[30]

Multi-Core and Multi-Processor On multi-core processors, the lowest-level caches (L1 and sometimes L2) are usually private to each core; but if the cryptographic code occasionally exceeds these private caches and reaches caches that are shared among the cores (L2 or L3), then the asynchronous attack becomes applicable at the cross-core level. In SMP systems, cache coherency mechanisms may be exploitable for similar effect.

Remote Attacks As in the synchronous case, one can envision remote attack variants that take advantage of data structures to which accesses can be triggered and timed through a network (e.g., the TCP state table).

5. Countermeasures

In the following we discuss several potential methods to mitigate the information leakage. Since these methods have different trade-offs and are architecture- and application-dependent, we cannot recommend a single recipe for all implementers. Rather, we aim to present the realistic alternatives along with their inherent merits and shortcomings. We focus our attention on methods that can be implemented in software, whether by

[30] This was indeed subsequently demonstrated by [41]; see Sect. 6.5.

operating system kernels or normal user processes, running under today's common general-purpose processors. Some of these countermeasures are presented as specific to AES but have analogues for other primitives. Countermeasures which require hardware modification are discussed in [10,47–50].

Caveat: due to the complex architecture-dependent considerations involved, we expect the secure implementation of these countermeasures to be a very delicate affair. Implementers should consider all exploitable effects given in [10] and carefully review their architecture for additional effects.

5.1. *Avoiding Memory Accesses*

Our attacks exploit the effect of memory access on the cache and would thus be completely mitigated by an implementation that does not perform any table lookups. This may be achieved by the following approaches.

First, one could use an alternative description of the cipher which replaces table lookups by an equivalent series of logical operations. For AES, this is particularly elegant, since the lookup tables have concise algebraic descriptions, but performance is degraded by over an order of magnitude.[31]

Another approach is that of bitslice implementations [12]. These employ a description of the cipher in terms of bitwise logical operations and execute multiple encryptions simultaneously by vectorizing the operations across wide registers. Their performance depends heavily on the structure of the cipher, the processor architecture, and the possibility of amortizing the cost across several simultaneous encryptions (which depends on the use of an appropriate encryption mode). For AES, bitsliced implementation on popular architectures can offer a throughput comparable to that of lookup-based implementations [26,31,34,35,51,52], but only when several independent blocks are processed in parallel.[32] Bitsliced AES is thus efficient for parallelized encryption modes such as CTR [35] and for exhaustive key search [62], but not for chained modes such as CBC.

Alternatively, one could use lookup tables but place the tables in registers instead of cache. Some architectures (e.g., x86-64, PowerPC AltiVec, and Cell SPE) have register files sufficiently large to hold the 256-byte S-box table, and instructions (e.g., AltiVec's VPERM and Cell's SHUFB) that allow for efficient lookups.

5.2. *Alternative Lookup Tables*

For AES, there are several similar formulations of the encryption and decryption algorithms that use different sets of lookup tables. Above we have considered the most common implementation, employing four 1024-byte tables $T_0, \ldots, T_3$ for the main rounds. Variants have been suggested with one 256-byte table (for the S-box), two 256-bytes tables (adding also $2 \bullet S[\cdot]$), one 1024-byte table (just T_0 with the rest obtained by rotations), and one 2048-byte table ($T_0, \ldots, T_3$ compressed into one table with nonaligned lookups). The same applies to the last round tables, $T_0^{(10)}, \ldots, T_3^{(10)}$. For encryption

[31] This kind of implementation has also been attacked through the timing variability in some implementations [30].

[32] Optimal throughput requires 64 parallel blocks in [34], 64/128/192 in [51], 128 in [35], 4 in [31], and 8 in [26].

(but not decryption), the last round can also be implemented by reusing one byte out of every element in the main tables.[33]

In regard to the synchronous attacks considered in Sect. 3, the effect of using smaller tables is to decrease the probability ρ that a given memory block will not be accessed during the encryption (i.e., $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y) = 0$) when the candidate guess $\tilde{k}_i$ is wrong. Since these are the events that rule out wrong candidates, the amount of data and analysis in the one-round attack is inversely proportional to $\log(1 - \rho)$.

For the most compact variant with a single 256-byte table and $\delta = 64$, the probability is $\rho = (1 - 1/4)^{160} \approx 2^{-66.4}$, so the synchronous attack is infeasible—we are unlikely to ever see an unaccessed memory block. For the next most compact variant, using a single 1024 bytes table, the probability is $\rho = (1 - 1/16)^{160} \approx 2^{-14.9}$, compared to $\rho \approx 0.105$ in Sect. 3.2. The attack will thus take about $\log(1-0.105)/\log(1-2^{-14.9}) \approx 3386$ times more data and analysis, which is inconvenient but certainly feasible for the attacker. The variant with a single 2 KB table (8 → 64 bit) has $\rho = (1 - 1/32)^{160}$, making the synchronous attack just 18 times less efficient than in Sect. 3.2 and thus still doable within seconds.

For asynchronous attacks, if the attacker can sample at intervals on the order of single table lookups (which is architecture-specific), then these alternative representations provide no appreciable security benefit. We conclude that overall, this approach (by itself) is of very limited value. However, it can be combined with some other countermeasures (see Sects. 5.3, 5.5, 5.8).

5.3. *Data-Independent Memory Access Pattern*

Instead of avoiding table lookups, one could employ them but ensure that the pattern of accesses to the memory is completely independent of the data passing through the algorithm. Most naively, to implement a memory access, one can read *all* entries of the relevant table, in fixed order, and use just the one needed. Modern CPUs analyze dependencies and reorder instructions, so care (and overhead) must be taken to ensure that the instruction and access scheduling, and their timing, are completely data-independent.

If the processor leaks information only about whole memory blocks (i.e., not about the low address bits),[34] then it suffices that the sequence of accesses to *memory blocks* (rather than *memory addresses*) is data-independent. To ensure this one can read a representative element from every memory block upon every lookup.[35] For the implementation of AES given in Sect. 2.2 and the typical $\delta = 16$, this means that each logical table access would involve 16 physical accesses, a major slowdown. Conversely, in the formulation of AES using a single 256-byte table (see Sect. 5.2), the table consists of only 4 memory blocks (for $\delta = 64$), so every logical table access (the dominant innermost-loop operation) would involve just 4 physical accesses; but this formulation of AES is inherently very slow.

[33] Beside memory saving, this has the benefit of foiling attacks based on the last round involving a separate set of cache sets; see Sect. 3.8.

[34] This assumption is false for the Athlon 64 processor (due to cache bank collision effects) and possibly for other processors as well. See Sect. 3.8 and [10].

[35] This approach was suggested by Intel Corp. [17] for mitigating the attack of Percival on RSA [50] and incorporated into OpenSSL 0.9.7h. In the case of RSA the overhead is insignificant, since other parts of the computation dominate the running time.

A still looser variant is to require only that the sequence of accesses to *cache sets* is data-independent (e.g., store each AES table in memory blocks that map to a single cache set). While this poses a challenge to the cryptanalyst, it does not in general suffice to eliminate the leaked signal: an attacker can still initialize the cache to a state where only a specific memory block is missing from cache, by evicting all memory blocks from the corresponding cache set and then reading back all but one (e.g., by triggering access to these blocks using chosen plaintexts); he can then proceed as in Sect. 3.4. Moreover, statistical correlations between memory block accesses, as exploited in the collision attacks of Tsunoo et al. [56,57], are still present.

Taking a broader theoretical approach, Goldreich and Ostrovsky [22] devised a realization of *Oblivious RAM*: a generic program transformation which hides all information about memory accesses. This transformation is quite satisfactory from an (asymptotic) theoretical perspective, but its concrete overheads in time and memory size are too high for most applications.[36] Moreover, it employs pseudo-random functions, whose typical realizations can also be attacked since they employ the very same cryptographic primitives we are trying to protect.[37]

Xhuang, Zhang, Lee, and Pande addressed the same issue from a more practical perspective and proposed techniques based on shuffling memory content whenever it is accessed [63] or occasionally permuting the memory and keeping the cache locked between permutations [64]. Both techniques require nontrivial hardware support in the processor or memory system and do not provide perfect security in the general case.

A simple heuristic approach is to add noise to the memory access pattern by adding spurious accesses, e.g., by performing a dummy encryption in parallel to the real one. This decreases the signal visible to the attacker (and hence necessitates more samples) but does not eliminate it.

5.4. *Application-Specific Algorithmic Masking*

There is extensive literature about side-channel attacks on hardware ASIC and FPGA implementations and corresponding countermeasures. Many of these countermeasures are implementation-specific and thus of little relevance to us, but some of them are algorithmic. Of particular interest are masking techniques, which effectively randomize all data-dependent operations by applying random transformations; the difficulty lies, of course, in choosing transformations that can be stripped away after the operation. One can think of this as homomorphic secret sharing, where the shares are the random mask and the masked intermediate values. For AES, several masking techniques have been proposed (see, e.g., [46,53], and the references therein). However, most (except [53]) are designed to protect only against first-order analysis, i.e., against attacks that measure some aspect of the state only at one point in the computation—our asynchronous attacks do not fall into this category. Moreover, the security proofs consider leakage only of

[36] The Oblivious RAM model of [22] protects against a stronger adversary which is also able to corrupt the data in memory. If one is interested only in achieving correctness (not secrecy) in the face of such corruption, then Blum et al. [14] provide more efficient schemes, and Naor and Rothblum [38] provide strong lower bounds.

[37] In [22] it is assumed that the pseudorandom functions are executed completely within a secure CPU without memory accesses. If such a CPU was available, we could use it to run the AES algorithm itself.

specific intermediate values, which do not correspond to the ones leaking via memory access metadata. Lastly, every AES masking method we are aware of has either been shown to be insecure even for its original setting (let alone ours) or is significantly slower in software than a bitslice implementation (see Sect. 5.1).

Finding an efficient masking scheme for AES on 32-bit (or wider) processors that is resilient to cache attacks is thus an open problem.

5.5. *Cache State Normalization and Process Blocking*

To foil the synchronous attacks of Sect. 3, it suffices to ensure that the cache is in a data-independent normalized state (e.g., by loading all lookup table elements into cache) at any entry to and exit from the encryption code (including interrupt and context switching by the operating system). Thus, to foil the Prime + Probe attack, it suffices to normalize the state of the cache after encryption. To foil the Evict + Time attack, one needs to normalize the state of the cache immediately before encryption (as in [47]) and also after every interrupt occurring during an encryption (the memory accesses caused by the interrupt handler will affect the state of the cache in some semi-predictable way and can thus be exploited by the attacker similarly to the Evict stage of Evict + Time). Performing the normalization after interrupts typically requires operating system support (see Sect. 5.11). As pointed out in [10, Sects. 12 and 14], it should be ensured that the table elements are not evicted by the encryption itself, or by accesses to the stack, inputs or outputs; this is a delicate architecture-dependent affair.

A subtle aspect is that the cache state which we seek to normalize includes a hidden state which is used by the CPU's cache eviction algorithm (typically Least Recently Used or variants thereof). If multiple lookup table memory blocks are mapped to the same cache set (e.g., OpenSSL on the Pentium 4E; see Table 1), the hidden state could leak information about which of these blocks was accessed last even if all of them are cached; an attacker can exploit this to analyze the last rounds in the encryption (or decryption).

All of these countermeasures provide little protection against the asynchronous attacks of Sect. 4. To fully protect against those, during the encryption one would have to disable interrupts and stop simultaneous threads (and perhaps also other processors on an SMP machine, due to the cache coherency mechanism). This would significantly degrade performance on SMT and SMP machines, and disabling interrupts for long durations will have adverse effects. A method for blocking processes more selectively based on process credentials and priorities is suggested in [50].

Note that normalizing the cache state frequently (e.g., by reloading all tables after every AES round) would merely reduce the signal-to-noise of the asynchronous attacks, not eliminate them.

5.6. *Disabling Cache Sharing*

To protect against software-based attacks, it would suffice to prevent cache state effects from spanning process boundaries. Alas, practically this is very expensive to achieve. On current single-threaded processors, it would require flushing all caches during every context switch. Alternatively, and necessarily on a processor with simultaneous multithreading, the CPU can be designed to allow separate processes to use separate logical caches that are statically allocated within the physical cache (e.g., each with half the size and half associativity). Besides reduced performance and lack of support in current processors, one would also need to consider the effect of cache coherency mechanisms in SMP configurations, as well as the caveats in Sect. 5.3.

A relaxed version would activate the above means only for specific processes, or specific code sections, marked as sensitive. This is especially appropriate for the operating system kernel but can be extended to user processes as explained in Sect. 5.11.

To separate two processes with regard to the attacks considered here, it suffices[38] to ensure that all memory accessible by one process is mapped into a group of cache sets that is disjoint from that of the other process.[39] In principle, this can be ensured by the operating system virtual memory allocator through a suitable page coloring algorithm. Alas, this fails on both of the major x86 platforms: in modern Intel processors every 4096-byte memory page is mapped to every cache set in the L1 cache (see Table 1), while in AMD processors the L1 cache is indexed by virtual addresses (rather than physical addresses), and these are allocated contiguously.

5.7. *Static or Disabled Cache*

One brutal countermeasure against the cache-based attacks is to completely disable the CPU's caching mechanism.[40] Of course, the effect on performance would be devastating, slowing down encryption by a factor of about 100. A more attractive alternative is to activate a "no-fill" mode[41] where the memory accesses are serviced from the cache when they hit it, but accesses that miss the cache are serviced directly from memory (without causing evictions and filling). The encryption routine would then proceed as follows:

(a) Preload the AES tables into cache
(b) Activate "no-fill" mode
(c) Perform encryption
(d) Deactivate "no-fill" mode

The section spanning (a) and (b) is critical, and attacker processes must not be allowed to run during this time. However, once this setup is completed, step (c) can be safely executed. The encryption per se would not be slowed down significantly (assuming that its inputs are in cache when "no-fill" is enabled), but its output will not be cached, leading to subsequent cache misses when the output is used (in chaining modes, as well as for the eventual storage or transmission). Other processes executed during (c), via multitasking or simultaneous multithreading, may incur a severe performance penalty. Breaking the encryption chunks into smaller chunks and applying the above routine to each chunk would reduce this effect somewhat by allowing the cache to be occasionally updated to reflect the changing memory working set.

[38] In the absence of low-address-bit leakage due to cache bank collisions.

[39] This was proposed to us by Úlfar Erlingsson of Microsoft Research.

[40] Some stateful effects would remain, such as the DRAM bank activation. These might still provide a low-bandwidth side channel in some cases.

[41] Not to be confused with the "disable cache flushing" mode suggested in [48], which is relevant only in the context of smartcards.

Intel's family of Pentium and Xeon processors supports such a mode,[42] but the cost of enabling and disabling it are prohibitive. Also, some ARM implementations allow cache lines to be locked (e.g., [25, Sect. 3.4.4]). We do not know which other processor families currently offer this functionality.

This method can be employed only in privileged mode, which is typically available only to the operating system kernel (see Sect. 5.11) and may be competitive performance-wise only for encryption of sufficiently long sequences. In some cases it may be possible to delegate the encryption to a co-processor with the necessary properties. For example, IBM's Cell processor consists of a general-purpose (PowerPC) core along with several "Synergistic Processing Element" (SPE) cores. The latter have a fast local memory but it is not a cache per se, i.e., there are no automatic transfers to or from main memory, thus, SPEs employed as cryptographic co-processors would not be susceptible to this attack.[43]

5.8. *Dynamic Table Storage*

The cache-based attacks observe memory access patterns to learn about the table lookups. Instead of eliminating these, we may try to decorrelate them. For example, one can use many copies of each table, placed at various offsets in memory, and have each table lookup (or small group of lookups) use a pseudorandomly chosen table. Ideally, the implementation will use S copies of the tables, where S is the number of cache sets (in the largest relevant cache). However, this means most table lookups will incur cache misses. Somewhat more compactly, one can use a single table but pseudorandomly move it around in memory several times during each encryption.[44] If the tables reside in different memory pages, one should consider and prevent leakage (and performance degradation) through page table cache (i.e., Table Lookaside Buffer) misses.

Another variant is to mix the order of the table elements several times during each encryption. The permutations need to be chosen with lookup efficiency in mind (e.g., via a linear congruential sequence), and the choice of permutation needs to be sufficiently strong; in particular, it should employ entropy from an external source (whose availability is application-specific).[45]

The performance and security of this approach are very architecture-dependent. For example, the required strength of the pseudorandom sequence and frequency of randomization depend on the maximal probing frequency feasible for the attacker.

[42] Enable the CD bit of CR0 and, for some models, adjust the MTRR. Coherency and invalidation concerns apply.

[43] In light of the Cell's high parallelism and the SPE's abundance of 128-bit registers (which can be effectively utilized by bitslice implementations), it has considerable performance potential in cryptographic and cryptanalytic applications (e.g., [54]).

[44] If the tables stay static for long, then the attacker can locate them (see Sect. 3.6) and discern their organization. This was prematurely dismissed by Lauradoux [32], who assumed that the mapping of table entries to memory storage will be attacked only by exhaustive search over all possible such mappings; the mapping can be recovered efficiently on an entry-by-entry basis.

[45] Some of these variants were suggested to us by Intel Corp and implemented in [16], following an early version of this paper.

5.9. *Hiding the Timing*

All of our attacks perform timing measurements, whether of the encryption itself (in Sect. 3.4) or of accesses to the attacker's own memory (in all other cases). A natural countermeasure for timing attacks is to try to hide the timing information. One common suggestion for mitigating timing attacks is to add noise to the observed timings by adding random delays to measured operations, thereby forcing the attacker to perform and average many measurements. Another approach is to normalize all timings to a fixed value, by adding appropriate delays to the encryption, but beside the practical difficulties in implementing this, it means that all encryptions have to be as slow as the worst-case timing (achieved here when all memory accesses miss the cache). Neither of these provide protection against the Prime + Probe synchronous attack or the asynchronous attack.

At the operating system or processor level, one can limit the resolution or accuracy of the clock available to the attacker; as discussed by Hu [23], this is a generic way to reduce the bandwidth of side channels but is nontrivial to achieve in the presence of auxiliary timing information (e.g., from multiple threads [50]), and will unpredictably affect legitimate programs that rely on precise timing information. The attacker will still be able to obtain the same information as before by averaging over more samples to compensate for the reduced signal-to-noise ratio. Since some of our attacks require only a few milliseconds of measurements, to make them infeasible the clock accuracy may have to be degraded to an extent that interferes with legitimate applications.

5.10. *Selective Round Protection*

The attacks described in Sects. 3 and 4 detect and analyze memory accesses in the first two rounds (for known input) or last two rounds (for known output). To protect against these specific attacks it suffices to protect those four rounds by some of the means given above (i.e., hiding, normalizing, or preventing memory accesses) while using the faster, unprotected implementation for the internal rounds.[46] This does not protect against other cryptanalytic techniques that can be employed using the same measurement methods. For example, with chosen plaintexts, the table accesses in the 3rd round can be analyzed by differential cryptanalysis (using a 2-round truncated differential). Nonetheless, those cryptanalytic techniques require more data and/or chosen data, and thus when quantitatively balancing resilience against cache-based attacks and performance, it is sensible to provide somewhat weaker protection for internal rounds.

5.11. *Operating System Support*

Several of the countermeasures suggested above require privileged operations that are not available to normal user processes in general-purpose operating systems. In some scenarios and platforms, these countermeasures may be superior (in efficiency or safety) to any method that can be achieved by user processes. One way to address this is to provide secure execution of cryptographic primitives as operating system services. For

[46] This was suggested to us by Intel Corp and implemented in [16], following an early version of this work.

example, the Linux kernel already contains a modular library of cryptographic primitives for internal use; this functionality could be exposed to user processes through an appropriate interface. A major disadvantage of this approach is its lack of flexibility: support for new primitives or modes will require operating system modifications (or loadable drivers) which exceed the scope of normal applications.

An alternative approach is to provide a secure execution facility to user processes.[47] This facility would allow the user to mark a "sensitive section" in his code and ask the operating system to execute it with a guarantee: either the sensitive section is executed under a promise sufficient to allow efficient execution (e.g., disabled task switching and parallelism, or cache in "no-fill" mode—see above), or its execution fails gracefully. When asked to execute a sensitive section, the operating system will attempt to put the machine into the appropriate mode for satisfying the promise, which may require privileged operations; it will then attempt to fully execute the code of the sensitive section under the user's normal permissions. If this cannot be accomplished (e.g., a hardware interrupt may force task switching, normal cache operation may have to be enabled to service some performance-critical need, or the process may have exceeded its time quota), then the execution of the sensitive section will be aborted, and prescribed cleanup operations will be performed (e.g., complete cache invalidation before any other process is executed). The failure will be reported to the process (now back in normal execution mode), so it can restart the failed sensitive section later.

The exact semantics of this "sensitive section" mechanism depend on the specific countermeasure and on the operating system's conventions. This approach, while hardly the simplest, offers maximal flexibility to user processes; it may also be applicable inside the kernel when the promise cannot be guaranteed to hold (e.g., if interrupts cannot be disabled).

5.12. *Hardware AES Support*

Several major vendors (including Intel, AMD, Sun, and Via) have recently announced or implemented specialized AES hardware support in their chips. Assuming that the hardware executes the basic AES operation with constant resource consumption, this allows for efficient AES execution that is invulnerable to our attacks. Other code running on the system may, of course, remain vulnerable to cache attacks.

Similarly, AES may be relegated to a hardware implementation in a secure coprocessor. In particular, Trusted Platform Module (TPM) chip are nowadays ubiquitous; alas, they are typically too slow for bulk encryption.

6. Conclusions and Implications

6.1. *Summary of Results*

We described novel attacks which exploit inter-process information leakage through the state of the CPU's memory cache. This leakage reveals memory access patterns which can be used for cryptanalysis of cryptographic primitives that employ data-dependent

[47] Special cases of this were discussed in [50] and [10], though the latter calls for this to be implemented at the CPU hardware level.

table lookups. Exploiting this leakage allows an unprivileged process to attack other processes running in parallel on the same processor, despite partitioning methods such as memory protection, sandboxing, and virtualization. Some of our methods require only the ability to trigger services that perform encryption or MAC using the unknown key, such as encrypted disk partitions or secure network links. Moreover, we demonstrated an extremely strong type of attack, which requires knowledge of neither the specific plaintexts nor ciphertexts, and works by merely monitoring the effect of the cryptographic process on the cache. We discussed in detail several such attacks on AES and experimentally demonstrated their applicability to real systems, such as OpenSSL and Linux's `dm-crypt` encrypted partitions (in the latter case, the full key was recovered after just 800 writes to the partition, taking 65 milliseconds). Finally, we proposed a variety of countermeasures.

6.2. *Vulnerable Cryptographic Primitives*

The cache attacks we have demonstrated are particularly effective for typical implementations of AES, for two reasons. First, the memory access patterns have a simple relation to the inputs; for example, the indices accessed in the first round are simply the XOR of a key byte and a plaintext byte. Second, the parameters of the lookup tables are favorable: there is a large number of memory blocks involved (but not too many to exceed the cache size), and thus many bits are leaked by each access. Moreover, there is a significant probability that a given memory block will not be accessed at all during a given random encryption.

Beyond AES, such attacks are potentially applicable to any implementation of a cryptographic primitive that performs key- and input-dependent memory accesses. The efficiency of an attack depends heavily on the structure of the cipher and chosen implementation, but heuristically, large lookup tables increase the effectiveness of all attacks: having few accesses to each table helps the synchronous attacks, whereas the related property of having temporally infrequent accesses to each table helps the asynchronous attack. Large individual table entries also aid the attacker in reducing the uncertainty about which table entry was addressed in a given memory block. This is somewhat counterintuitive, since it is usually believed that large S-boxes are more secure.

For example, DES is vulnerable when implemented using large lookup tables which incorporate the P permutation and/or to compute two S-boxes simultaneously. Cryptosystems based on large-integer modular arithmetic, such as RSA, can be vulnerable when exponentiation is performed using a precomputed table of small powers (see [50]). Moreover, a naive square-and-multiply implementation would leak information through accesses to long-integer operands in memory. The same potentially applies to ECC-based cryptosystems.

Primitives that are normally implemented without lookup tables, such as the SHA family [40] and bitsliced Serpent [9], are impervious to the attacks described here. However, to protect against timing attacks one should scrutinize implementations for use of instructions whose timing is key- and input-dependent (e.g., bit shifts and multiplications on some platforms) and for data-dependent execution branches (which may be analyzed through data cache access, instruction/trace cache access, or timing). Note that timing variability of non-memory operations can be measured by an unrelated process running on the same machine, using a variant of the asynchronous attack, via the effect of those operations on the scheduling of memory accesses.

We stress that cache attacks are potentially applicable to any program code, cryptographic or otherwise. Above we have focused on cryptographic operations because these are designed and trusted to protect information, and thus information leakage from within them can be critical (for example, recovering a single decryption key can compromise the secrecy of all messages sent over the corresponding communication channel). However, information leakage can be harmful also in noncryptographic context. For example, even knowledge of what programs are running on someone's computer at a given time can be sensitive.

6.3. *Vulnerable Systems*

At the system level, cache state analysis is of concern in essentially any case where process separation is employed in the presence of malicious code. This class of systems includes many multiuser systems, as well as web browsing, DRM applications, the Trusted Computing Platform [55],[48] and NGSCB [37]. The same applies to acoustic cryptanalysis, whenever malicious code can access a nearby microphone device and thus record the acoustic effects of other local processes.

Disturbingly, virtual machines and sandboxes offer little protection against the asynchronous cache attack (in which attacker needs only the ability to access his own memory and measure time) and against the acoustic attacks (if the attacker gains access to a nearby microphone). Thus, our attacks may cross the boundaries supposedly enforced by FreeBSD `jail()`, VMware [61],[49] Xen [59], the Java Virtual Machine [33], and plausibly even scripting language interpreters. Today's hardware-assisted virtualization technologies, such as Intel's "Virtualization Technology" and AMD's "Secure Virtual Machine", offer no protection either.

Remote cache attacks are in principle possible, and if proven efficient, they could pose serious threats to secure network connections such as IPsec [28] and OpenVPN [43].

Finally, while we have focused our attention on cryptographic systems (in which even small amount of leakage can be devastating), the leakage also occurs in noncryptographic systems and may thus leak sensitive information directly.

6.4. *Mitigation*

We have described a variety of countermeasures against cache state analysis attacks; some of these are generic, while others are specific to AES. However, none of these unconditionally mitigates the attacks while offering performance close to current implementations. Thus, finding an efficient and secure solution that is application- and architecture-independent remains an open problem. In evaluating countermeasures, one should pay particular attention to the asynchronous attacks, which on some platforms allow the attacker to obtain (a fair approximation of) the full transcript of memory accesses done by the cryptographic code.

[48] While the Trusted Computing Module (TPM) chip itself may be invulnerable to software attacks, it cannot effectively enforce information flow control in the rest of the system when side channels are present.

[49] This compromises the system described in the recent NSA patent 6,922,774 [36].

6.5. *Follow-up Works*

Since the initial publications of these results [44,45], numerous extensions and variants have been suggested, including the following.

Countermeasures Brickell et al. of Intel Corp. [16,17] implemented and experimentally evaluated several AES implementations that reduce the cache side-channel leakage (see discussion in Sect. 5), and Page [49] evaluated partitioned cache architectures as a countermeasure.

Survey and Extensions to Related Attacks In [18], Canteaut et al. survey and classify the various cache attacks, and proposes extensions and countermeasures.

Collision-Based Attacks As discussed in Sect. 1.2, [18] describes an attack on AES based on exploiting internal cache collisions, following the approach of Tsunoo et al. This was improved by Bonneau and Mironov [15] (attacking the first or last round) and by Acıiçmez et al. [7] (attacking the first round). These attacks still require many thousands of encryptions even for an in-process OpenSSL target.

Exploiting the OS Scheduler In [41], Neve and Seifert empirically demonstrate the effectiveness of an extension we have merely alluded to hypothetically: carrying out an asynchronous attack without simultaneous multithreading, by exploiting only the OS scheduling and interrupts. Indeed, they show that with appropriate setup their approach provides excellent temporal resolution. They also demonstrate the effectiveness of analyzing the last round of AES instead of the first one, where applicable (see Sect. 3.8).

Branch Prediction and Instruction Cache Attacks In [2,5,6], Acıiçmez et al. describe new classes of attacks that exploit the CPU *instruction* cache or its branch prediction mechanism, instead of the *data* cache considered herein. They demonstrate efficient RSA key recovery via contention for these resources. The measurement approaches (and hence attack scenarios) are similar to the data cache attack techniques described here, but the information obtained is about the execution path rather than data accesses. Veith et al. [60] presented a related attack, which monitors branch prediction via the CPU performance counters. Since the type of vulnerable code is different compared to data cache attacks, these attacks are complementary.

Multiplier Unit Contention Attacks In [8], Acıiçmez and Seifert demonstrate another microarchitectural side channel: contention for the multiplication unit when two processes are running concurrently on an Intel HyperThreading CPU. They exploit this to eavesdrop on modular exponentiation in RSA signing.

Indubitably, further side channels in all levels of system architecture will be created and discovered, as hardware grows in parallelism and complexity.

Acknowledgements

We are indebted to Ernie Brickell, Jean-Pierre Seifert, and Michael Neve of Intel Corp. for insightful discussions and proposal of several countermeasures, to Daniel J. Bernstein for suggesting the investigation of remote attacks, and to Eli Biham, Paul Karger, Maxwell Krohn, and the anonymous referees for their helpful pointers and comments.

References

[1] M. Abadi, M. Burrows, M. Manasse, T. Wobber, Moderately hard, memory-bound functions. *ACM Trans. Internet Technol.* **5**(2), 299–327 (2005)

[2] O. Acıiçmez, Yet another microarchitectural attack: exploiting I-cache, in *IACR Cryptology ePrint Archive, report 2007/164* (2007). http://eprint.iacr.org/2007/164

[3] O. Acıiçmez, Ç.K. Koç, *Trace driven cache attack on AES*. IACR Cryptology ePrint Archive, report 2006/138 (2006). http://eprint.iacr.org/2006/138; full version of [4]

[4] O. Acıiçmez, Ç.K. Koç, Trace driven cache attack on AES (short paper), in *Proc. International Conference on Information and Communications Security (ICICS) 2006*. Lecture Notes in Computer Science, vol. 4296 (Springer, Berlin, 2006), pp. 112–121. Short version of [3]

[5] O. Acıiçmez, Ç.K. Koç, J.-P. Seifert, *On the power of simple branch prediction analysis*. IACR Cryptology ePrint Archive, report 2006/351 (2006)

[6] O. Acıiçmez, Ç.K. Koç, J.-P. Seifert, Predicting secret keys via branch prediction, in *Proc. RSA Conference Cryptographers Track (CT-RSA) 2007*. Lecture Notes in Computer Science, vol. 4377 (Springer, Berlin, 2007), pp. 225–242

[7] O. Acıiçmez, W. Schindler, Ç.K. Koç, Cache based remote timing attack on the AES, in *Proc. RSA Conference Cryptographers Track (CT-RSA) 2007*. Lecture Notes in Computer Science, vol. 4377 (Springer, Berlin, 2007), pp. 271–286

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

[9] R.J. Anderson, E. Biham, L.R. Knudsen, *Serpent: A Proposal for the Advanced Encryption Standard*. AES submission (1998). http://www.cl.cam.ac.uk/~rja14/serpent.html

[10] D.J. Bernstein, *Cache-timing attacks on AES*. Preprint (2005). http://cr.yp.to/papers.html#cachetiming

[11] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, G. Palermo, AES power attack based on induced cache miss and countermeasure, in *Proc. International Conference on Information Technology: Coding and Computing (ITCC'05)* (IEEE, New York, 2005), pp. 586–591

[12] E. Biham, A fast new DES implementation in software, in *Proc. Fast Software Encryption (FSE) 1997*. Lecture Notes in Computer Science, vol. 1267 (Springer, Berlin, 1997), pp. 260–272

[13] E. Biham, A. Shamir, Differential cryptanalysis of DES-like Cryptosystems. *J. Cryptol.* **4**(1), 3–72 (1991)

[14] M. Blum, W. Evans, P. Gemmell, S. Kannan, M. Naor, Checking the correctness of memories, in *Proc. Conference on Foundations of Computer Science (FOCS) 1991* (IEEE, New York, 1991), pp. 90–99

[15] J. Bonneau, I. Mironov, Cache-collision timing attacks against AES, in *Proc. Cryptographic Hardware and Embedded Systems (CHES) 2006*. Lecture Notes in Computer Science, vol. 4249 (Springer, Berlin, 2006), pp. 201–215

[16] E. Brickell, G. Graunke, M. Neve, J.-P. Seifert, *Software mitigations to hedge AES against cache-based software side channel vulnerabilities*. IACR Cryptology ePrint Archive, report 2006/052 (2006). http://eprint.iacr.org/2006/052

[17] E. Brickell, G. Graunke, J.-P. Seifert, Mitigating cache/timing attacks in AES and RSA software implementations, in *RSA Conference 2006, San Jose, session DEV-203* (2006). http://2006.rsaconference.com/us/cd_pdfs/DEV-203.pdf

[18] A. Canteaut, C. Lauradoux, A. Seznec, *Understanding cache attacks*. Research report RR-5881, INRIA, April 2006. http://www-rocq.inria.fr/codes/Anne.Canteaut/Publications/RR-5881.pdf

[19] J. Daemen, V. Rijmen, *AES Proposal: Rijndael*, version 2, AES submission (1999). http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf

[20] J. Daemen, V. Rijmen, *The Design of Rijndael: AES—The Advanced Encryption Standard* (Springer, Berlin, 2001). ISBN 3-540-42580-2

[21] C. Dwork, A. Goldberg, M. Naor, On memory-bound functions for fighting spam, in *Proc. CRYPTO'2003*. Lecture Notes in Computer Science, vol. 2729 (Springer, Berlin, 2003), pp. 426–444

[22] O. Goldreich, R. Ostrovsky, Software protection and simulation on oblivious RAMs. *J. ACM* **43**(3), 431–473 (1996)

[23] W.-M. Hu, Reducing timing channels with fuzzy time, in *Proc. IEEE Computer Society Symposium on Research in Security and Privacy* (IEEE, New York, 1991), pp. 8–20

[24] W.-M. Hu, Lattice scheduling and covert channels, in *IEEE Symposium on Security and Privacy* (IEEE, New York, 1992), pp. 52–61

[25] Intel Corp., *Intel IXP42X Product Line of Network Processors and IXC1100 Control Plane Processor Developer's Manual*. Order Number 252480-006US (2006). http://www.intel.com/design/network/manuals/252480.htm

[26] E. Käsper, P. Schwabe, *Faster and Timing-Attack Resistant AES-GCM*. IACR Cryptology ePrint Archive, report 2009/129 (2009). http://eprint.iacr.org/2009/129

[27] J. Kelsey, B. Schneier, D. Wagner, C. Hall, Side channel cryptanalysis of product ciphers, in *Proc. 5th European Symposium on Research in Computer Security*. Lecture Notes in Computer Science, vol. 1485 (Springer, Berlin, 1998), pp. 97–110

[28] S. Kent et al., *RFC 4301 through RFC 4309*. Network Working Group Request for Comments. http://rfc.net/rfc4301.html etc. (2005)

[29] R.E. Kessler, M.D. Hill, Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.* **10**(4), 338–359 (1992)

[30] F. Koeune, J.-J. Quisquater, *A timing attack against Rijndael*. Technical Report CG-1999/1, Université catholique de Louvain, http://www.dice.ucl.ac.be/crypto/tech_reports/CG1999_1.ps.gz

[31] R. Könighofer, A fast and cache-timing resistant implementation of the AES, in *Proc. RSA Conference Cryptographers Track (CT-RSA) 2008*. Lecture Notes in Computer Science, vol. 4964 (Springer, Berlin, 2008), pp. 187–202

[32] C. Lauradoux, Collision attacks on processors with cache and countermeasures, in *Western European Workshop on Research in Cryptology (WEWoRC) 2005*. Lectures Notes in Informatics, vol. P-74 (2005), pp. 76–85. http://www.cosic.esat.kuleuven.ac.be/WeWorc/allAbstracts.pdf

[33] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification*, 2nd edn. (Prentice Hall, New York, 1999)

[34] M. Matsui, How far can we go on the x64 processors?, in *Proc. Fast Software Encryption (FSE) 2006*. Lecture Notes in Computer Science, vol. 4047 (Springer, Berlin, 2006), pp. 341–358

[35] M. Matsui, J. Nakajima, On the power of bitslice implementation on Intel Core2 processor, in *Proc. Cryptographic Hardware and Embedded Systems (CHES) 2007*. Lecture Notes in Computer Science, vol. 4727 (Springer, Berlin, 2007), pp. 121–134

[36] R.V. Meushaw, M.S. Schneider, D.N. Simard, G.M. Wagner, *Device for and method of secure computing using virtual machines*. US patent 6,922,774 (2005)

[37] Microsoft Corp., *Next-generation secure computing base*. Web page, http://www.microsoft.com/resources/ngscb

[38] M. Naor, G.N. Rothblum, The complexity of online memory checking, in *Proc. 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS) 2005* (IEEE, New York, 2005), pp. 573–584

[39] National Institute of Standards and Technology, *Advanced Encryption Standard (AES)*, FIPS PUB 197 (2001)

[40] National Institute of Standards and Technology, *Secure Hash Standard (SHS)*, FIPS PUB 180-2 (2002)

[41] M. Neve, J.-P. Seifert, Advances on access-driven cache attacks on AES, in *Proc. Selected Areas in Cryptography (SAC'06)*. Lecture Notes in Computer Science, vol. 4356 (Springer, Berlin, 2006), pp. 147–162

[42] M. Neve, J.-P. Seifert, Z. Wang, A refined look at Bernstein's AES side-channel analysis. in *Proc. ACM Symposium on Information, Computer and Communications Security* (2006), p. 369

[43] OpenVPN Solutions LLC, *OpenVPN—an Open Source SSL VPN Solution by James Yonan*. Web site, http://openvpn.net

[44] D.A. Osvik, A. Shamir, E. Tromer, *Other people's cache: Hyper Attacks on HyperThreaded processors*. Fast Software Encryption (FSE) 2005 rump session, Feb. 2005

[45] D.A. Osvik, A. Shamir, E. Tromer, Cache attacks and countermeasures: the case of AES, in *Proc. RSA Conference Cryptographers Track (CT-RSA) 2006*. Lecture Notes in Computer Science, vol. 3860 (Springer, Berlin, 2006), pp. 1–20

[46] E. Oswald, S. Mangard, N. Pramstaller, V. Rijmen, A side-channel analysis resistant description of the AES S-box, in *Proc. Fast Software Encryption (FSE) 2005*. Lecture Notes in Computer Science, vol. 3557 (Springer, Berlin, 2005), pp. 413–423

[47] D. Page, *Theoretical use of cache memory as a cryptanalytic side-channel*. Technical Report CSTR-02-003, Department of Computer Science, University of Bristol (2002). http://www.cs.bris.ac.uk/Publications/pub_info.jsp?id=1000625

[48] D. Page, *Defending against cache-based side-channel attacks*. Information Security Technial Report, vol. 8, issue 8 (2003)

[49] D. Page, *Partitioned cache architecture as a side-channel defence mechanism*. IACR Cryptology ePrint Archive, report 2005/280 (2005). http://eprint.iacr.org/2005/280

[50] C. Percival, *Cache missing for fun and profit*. BSDCan 2005, Ottawa (2005). See http://www.daemonology.net/hyperthreading-considered-harmful

[51] C. Rebeiro, D. Selvakumar, A.S.L. Devi, Bitslice implementation of AES, in *Proc. Cryptology and Network Security (CANS) 2006*. Lecture Notes in Computer Science, vol. 4301 (Springer, Berlin, 2006), pp. 203–212

[52] A. Rudra, P.K. Dubey, C.S. Jutla, V. Kumar, J.R. Rao, P. Rohatgi, Efficient Rijndael encryption implementation with composite field arithmetic, in *Proc. Cryptographic Hardware and Embedded Systems (CHES) 2001*. Lecture Notes in Computer Science, vol. 2162 (Springer, Berlin, 2001), pp. 171–184

[53] K. Schramm, C. Paar, Higher Order Masking of the AES, in *Proc. RSA Conference Cryptographers Track (CT-RSA) 2006*. Lecture Notes in Computer Science, vol. 3860 (Springer, Berlin, 2006), pp. 208–225

[54] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D.A. Osvik, B. de Weger, Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate, in *Proc. CRYPTO 2009* (to be published). http://www.win.tue.nl/hashclash/rogue-ca/

[55] Trusted Computing Group, *Trusted Computing Group: Home*. Web site, http://www.trustedcomputinggroup.org

[56] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, H. Miyauchi, Cryptanalysis of DES implemented on computers with cache, in *Proc. Cryptographic Hardware and Embedded Systems (CHES) 2003*. Lecture Notes in Computer Science, vol. 2779 (Springer, Berlin, 2003), pp. 62–76

[57] Y. Tsunoo, E. Tsujihara, K. Minematsu, H. Miyauchi, Cryptanalysis of block ciphers implemented on computers with cache, in *Proc. International Symposium on Information Theory and Its Applications 2002* (2002), pp. 803–806

[58] Y. Tsunoo, E. Tsujihara, M. Shigeri, H. Kubo, K. Minematsu, Improving cache attacks by considering cipher structure. *Int. J. Inf. Secur.* “Online First”, Springer, Nov. 2005

[59] University of Cambridge Computer Laboratory, *The Xen virtual machine monitor*. Web site, http://www.cl.cam.ac.uk/research/srg/netos/xen

[60] A.A. Veith, A.V. Belenko, A. Zhukov, *A preview on branch misprediction attacks: using Pentium performance counters to reduce the complexity of timing attacks*. CRYPTO’06 rump session (2006)

[61] VMware Inc., *VMware: virtualization, virtual machine & virtual server consolidation*. Web site, http://www.vmware.com

[62] J. Yang, J. Goodman, Symmetric key cryptography on modern graphics hardware, in *Proc. Asiacrypt 2007*. Lecture Notes in Computer Science, vol. 4833 (Springer, Berlin, 2007), pp. 249–264

[63] X. Zhuang, T. Zhang, H.-H.S. Lee, S. Pande, Hardware assisted control flow obfuscation for embedded processors, in *Proc. International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (ACM, New York, 2004), pp. 292–302

[64] X. Zhuang, T. Zhang, S. Pande, HIDE: An Infrastructure for Efficiently protecting information leakage on the address bus, in *Proc. Architectural Support for Programming Languages and Operating Systems* (ACM, New York, 2004), pp. 82–84