Cache Attacks and Countermeasures: the Case of AES

2005-08-14

Dag Arne Osvik[1], Adi Shamir[2] and Eran Tromer[2]

[1] dag.arne@osvik.no
[2] Department of Computer Science and Applied Mathematics,
Weizmann Institute of Science, Rehovot 76100, Israel
{adi.shamir, eran.tromer}@weizmann.ac.il

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 such 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 can be recovered after just 800 writes to the partition, taking 65 milliseconds). Finally, we describe several countermeasures which can be used to mitigate such attacks.

Keywords: side-channel attack, cache, memory access, cryptanalysis, AES

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 content of the cache, and hence the memory access patterns of processes using that cache, is not fully protected.

We describe several methods an attacker can use to learn about the memory access patterns of another process. 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, [10]) 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, 65ms of measurements and 3 seconds of analysis; attacking simpler systems, such as "black-box" OpenSSL library calls, is even faster at 13ms 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 an unusually strong form of attack in which an unprivileged process can, just by accessing its own memory space, obtain bits from a secret AES key used by another process, without any (explicit) communication between the two. This too is demonstrated experimentally.

Implementing AES in a way that is impervious to this attack, let alone developing an efficient generic countermeasure, appears non-trivial; in Section 5, various countermeasures are described and analyzed.

1.2 Related Works

The possibility of cross-process leakage via cache state has been mentioned in several previous works. It was considered in 1992 by Hu [6] in the context of intentional transmission via covert channels. In 1998, Kelsey et al. [7] mentioned the prospect of "attacks based on cache hit ratio in large S-box ciphers". In 2002, Page [8] described theoretical attacks using cache misses, but assumed the ability to identify cache misses with very high temporal resolution; its applicability in realistic scenarios is unclear. In 2003, Tsunoo et al. [13] described attacks using timing effects due to collisions in the memory lookups *inside* the cipher, as opposed to the cipher-attacker collisions we investigate.

Concurrently with but independently of our work, Bernstein [1] describes attacks on AES that exploit timing variability due to cache effects; his attack can be seen as a variant of our Evict+Time measurement method (see Section 3.4). The main the difference is that [1] does not use an explicit model of the cache and active manipulation, but rather relies only on the existence of some consistent statistical timing pattern due to various uncontrolled memory access effects; this makes it simpler and cross-platform, but has three major 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 [1] 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 (Sections 3.5 and 4) address this. Third, even when the attack of [1] works, it requires a much higher number of analyzed encryptions.[3]

Also concurrently with but independently of our work, Percival [12] describes a cache-based attack on RSA for processors with simultaneous multithreading. The measurement method is similar to one variant of our asynchronous attack (Section 4), but the cryptanalytic aspect is very different since the algorithms and time scales involved in RSA encryption are very different from those of AES. Both [1] and [12] contain discussions of countermeasures against the respective attacks, and some of these are also relevant to our attacks (see Section 5).

Koeune and Quisquater [5] 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 attack (see Section 5.2).

Leakage of memory access information has also been considered in other contexts, yielding theoretical [4] and practical [15][16] mitigation methods; these are discussed in Section 5.3.

2 Preliminaries

2.1 Memory and Cache Structure

Modern processors use one or more levels of *set-associative memory cache*. Such a cache consists of S *cache sets*, each containing A *cache lines*[4], where each cache line holds B bytes (the *cache line size*). Overall, the cache contains $S \cdot W \cdot B$ bytes, but 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, which we will term *memory blocks*; when a cache miss occurs, a full memory block is copied into one of the cache lines. 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 Figure 1. Thus, the memory blocks are partitioned into S classes, where the blocks in each class contend for the cache lines in a single cache set.

2.2 Memory Access in AES implementations

This paper focuses on AES, since its memory access patterns are particularly susceptible to cryptanalysis (see Section 6.1 for a discussion of other ciphers). The cipher is abstractly defined by algebraic operations and could be, in principle, implemented using just logical and arithmetic operations. However, performance-oriented software implementations on 32-bit (or higher) processors typically use the following alternative formulation, as prescribed in the Rijndael AES submission [3].[5]

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

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

[5] Some software implementations use variants of formulation with a different table layouts; see 5.2 for discussion. The most common variant employs a single table for the last round; most of our attacks analyze only the first rounds, and are thus unaffected.

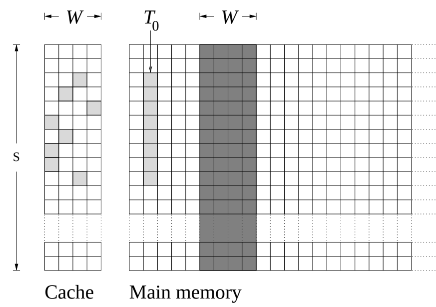

Fig. 1. Schematic of a set-associative cache. The light gray blocks represent a cached AES lookup table. The dark gray blocks represent the attacker's memory.

Several lookup tables are precomputed once by the programmer or during system initialization. There are 8 such tables, $T_0, \ldots, T_3$ and $T_0^{(10)}, \ldots, T_3^{(10)}$, each containing 256 4-byte words. The contents of the tables, defined in [3], are inconsequential for most of our attacks.

During key setup, a given 16-byte secret key $\boldsymbol{k} = (k_1, \ldots, k_{16})$ is expanded into 10 round keys[6], $\boldsymbol{K}^{(r)}$ for $r = 1, \ldots, 10$. 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 details of the expansion are also mostly inconsequential.

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

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

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

2.3 Notation

We treat bytes interchangeably as integers in $\{0, \ldots, 255\}$ 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

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

bytes[7]); 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[8]; they would therefore be impossible to distinguish based only on a single memory access. For a byte y and table T_ℓ, we say that an AES encryption with given inputs *accesses the memory block of* y *in* T_ℓ if, according to the above description of AES, at some point in the encryption there will be some table lookup to $T_\ell[z]$ where $\langle z \rangle = \langle y \rangle$.

In Section 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 Section 3, our measurement procedures will sample *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[9] 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

The first family of attacks, termed *synchronous attacks*, is applicable in scenarios where 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 may allow an unprivileged user to send data packets through a secure channel. 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 all users' packets. As another example, consider the Linux `dm-crypt` and `cryptoloop` services. These allow the administrator to create a virtual device which provides encrypted storage into 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 (actually chosen) plaintext, and using our attack he is subsequently able to discover the 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 65ms of measurements (involving just 800 write operations) followed by 3 seconds of analysis.

[7] One exception is OpenSSL 0.9.7g on x86-64, which uses 8-byte table entries. The reduced δ improves our attacks.
[8] 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 per key byte in the first round.
[9] This will be relaxed in Section 3.7.

The full attack obtains a set of random samples, and then performs off-line cryptanalysis. The latter proceeds, essentially, by guessing small parts of the key, using the guess to predict memory accesses, and checking whether the predictions are consistent with the collected data. In the following we first describe the cryptanalysis in an idealized form using the predicate Q, and adapt it to the noisy measurements of M. We then show two different methods for obtaining these measurements, detail some 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 $i = 0, \ldots, 15$. 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 x_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 "$T_\ell[x_i^{(0)}]$", 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 as follows. 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 indeed $\langle k_i \rangle = \langle \tilde{k}_i \rangle$ then we will always have $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y) = 1$ for useful samples, since the table lookup "$T_\ell[x_i^{(0)}]$" will indeed 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; however, during the full encryption process there will be $4 \times 9 - 1 = 35$ more accesses to T_ℓ. Those 35 accesses are affected by other plaintext bytes, so (for sufficiently random plaintexts) the probability that the encryption will not access that memory block in any round is $(1 - \delta/256)^{35}$. 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 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 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 Figure 5). We thus proceed by averaging over many samples. As above, we concentrate on a specific key x_i and a corresponding table ℓ. Our measurement will yield samples of the form (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

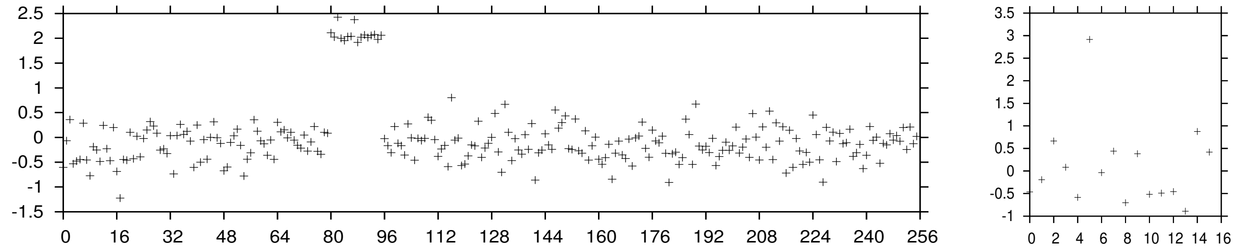

Fig. 2. Candidate scores for a synchronous attack using Prime+Probe measurements, analyzing a `dm-crypt` encrypted filesystem on Linux 2.6.11 running on an Athlon 64, after analysis of 30,000 (left) or 800 (right) triggered encryptions. The horizontal axis is $\tilde{k}_5 = p_5 \oplus y$ (left) or $\langle \tilde{k}_5 \rangle$ (right) 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). The high nibble of $k_5 = 0x50$ is easily gleaned.

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 Sections 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-Rounds Attack

The above attack narrows each AES key byte to one of δ possibilities, but the table lookups in the first AES round can not reveal further information. For the common case $\delta = 16$, the key still has 64 unknown bits. We thus proceed to analyze the 2nd AES round, exploiting the non-linear mixing in the cipher to reveal additional information. Specifically, we exploit the following equations, easily derived from the Rijndael specification [3], which give the indices used in four of the table lookups in the 2nd round:[10]

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

Here, $s(\cdot)$ denotes the Rijndael S-box function and $\bullet$ denotes multiplication over GF(256).[11]

Consider, for example, equation (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

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

[11] 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.

$\boldsymbol{p}$. 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 "$T_2[x_2^{(1)}]$" 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 are easily enumerated. We can identify the correct candidate as follows, thereby completing the recovery of the these four key bytes.

Identification of a correct guess is done by a generalization of the 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 "$T_2[x_2^{(1)}]$" 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

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

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 verified from [3] that for any $k_i \neq \tilde{k}_i$, $c \in \{1, 2, 3\}$, $\delta \geq 4$ and $z \in \{0, \ldots, 256/\delta\}$ we always have $\Pr_{\boldsymbol{p}}[\langle c \bullet s(p_i \oplus k_i) \,\oplus\, c \bullet s(p_i \oplus \tilde{k}_i)\rangle \neq z] > 1 - (1 - \delta/256)^3$. Thus, the probability that "$T_2[x_2^{(1)}]$" 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 $(1 - \delta/256)$. Hence, $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y) = 0$ with probability greater than $(1 - \delta/256)^{3+35}$.

We see that each sample is useful for a $\delta/256$-fraction of the candidates, and on average eliminates at least a $(1-\delta/256)^{38}$-fraction of the wrong candidates for which it is useful. Thus, to eliminate all the wrong candidates out of the δ^4, we need about $\log \delta^{-4} / \log(1-\delta/256 \cdot (1-\delta/256)^{37})$ samples, i.e., about 2055 samples when $\delta = 16$. [12]

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 samples and a complexity of $4 \cdot 2^{16} \cdot 2055 \approx 2^{29}$ 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 $\tilde{k}_i$, 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

[12] With some of our measurement methods the attack requires only a few hundred *encryptions*, since each encryption provides samples for multiple y.

to reliably obtain all key bytes by this method is, in some experimentally verified settings, only about 7 times larger than the ideal (see Section 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 (virtual) memory address of each table T_ℓ, denoted $V(T_\ell)$. [13] In a chosen-plaintext setting, the measurement routine proceeds as follows given a table ℓ, index y into ℓ and plaintext $\boldsymbol{p}$:

(a) Trigger an encryption of $\boldsymbol{p}$.
(b) *(evict)* Access some W memory addresses, at least B bytes apart, that are 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. This is the measurement score.[14]

The rationale for this procedure is as follows. Step (a) ensures that all table memory blocks accessed during the encryption of $\boldsymbol{p}$ are cached[15]; this is illustrated in Figure 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 K, we expect these blocks to completely replace the prior content of the cache. Specifically, this memory block of the encryption table is now not in cache; see Figure 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 Figure 3(c). This fetching will slow down the encryption. Conversely, if $Q_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y) = 1$ 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 proceeds as before.

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

[13] This, along with several other complications, will be addressed in Section 3.6.
[14] To obtain high-resolution timing we use the CPU cycle counter (e.g., RDTSC on x86).
[15] Unless the triggered encryption code has excessive internal cache contention.

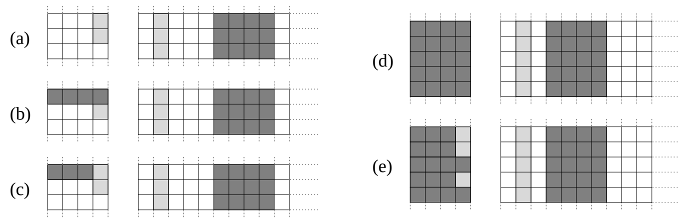

Fig. 3. Schematics of cache states, in the notation of Figure 2.1. States (a)-(c) depict Evict+Time and (d)-(e) depict Prime+Probe.

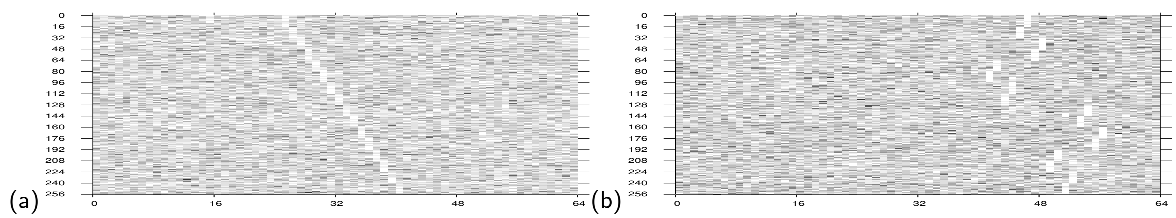

Fig. 4. Timings (lighter is slower) in Evict+Time measurements on a 2GHz Athlon 64, after 10,000 samples, attacking a procedure that executes an encryption using OpenSSL 0.9.8. The horizontal axis is the evicted cache set (i.e., $\langle y \rangle$ plus an offset due to the table's location) and the 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.

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 and 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[16], but not from more typical "heavyweight" services. For the latter, we had to invoke 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. Given a plaintext $\boldsymbol{p}$, it obtains measurement scores for all tables ℓ and all index y and does so using a *single* encryption. This method proceeds as follows. The attacker allocates a contiguous byte array $A[0,\ldots,S \cdot W \cdot B - 1]$, with start address congruent $\bmod S \cdot B$ to the start address of T_0 .[17]

[16] For this artificial scenario, [1] also demonstrated key extraction.

[17] Here we assume this address is known, and that T_0, T_1, T_2, T_3 are contiguous. Both assumptions can be eliminated.

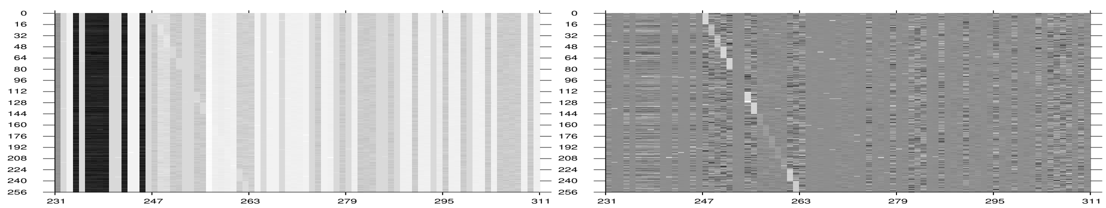

Fig. 5. Prime+Probe attack using 30,000 encryption calls on a 2GHz Athlon 64, attacking Linux 2.6.11 `dm-crypt`. The horizontal axis is the evicted cache set (i.e., $\langle y \rangle$ plus an offset due to the table's location) and the vertical axis is p_0. Left: raw timings (lighter is slower). Right: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

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

- Read the W memory addresses $A[1024\ell + 4y + tSB]$ for $t = 0, \ldots, W - 1$. The total time it takes to perform these reads is the measurement score, i.e., our sample of $M_{\boldsymbol{k}}(\boldsymbol{p}, \ell, y)$.[18]

Step (a) completely fills the cache with the attacker's data. Step (c) checks, for each cache set, whether the attacker's data is still present after the encryption: cache sets that were 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 induces a timing difference.

Crucially, the attacker is timing a simple operation performed by *itself*, as opposed to a complex encryption service with various overheads executed by someone 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 Section 5). Another benefit lies in inspecting all cache sets simultaneously 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 Figure 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., stack and I/O buffers) during the system call.

[18] We perform probing using pointer-chasing to ensure non-reorderable loads. To avoid "polluting" our samples, the probe code stores each obtained sample into the same cache set it measured. On some platforms one can improve the timing gap by using writes instead of reads, or more than W reads.

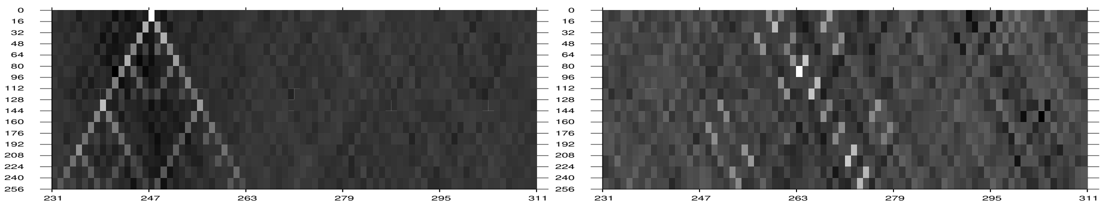

Fig. 6. Scores (lighter is higher) for combinations of key byte candidate (vertical axis) and table offset candidate (horizontal axis). The correct combinations are clearly identified as the bright rectangles at the head of the (permuted) Sierpinski triangles. Note the correct relative offsets of tables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

3.6 Complications

In the above we have omitted several practical 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[19]. 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 works very well, even on a real, noisy system (see Figure 6). Often, it even suffices to simply look for a frequently-accessed range of memory of the right size (see Figure 7).

Another complication is the distinction between virtual and physical memory addresses. The mapping between the two is done in terms of aligned ranges of addresses (*pages*). These can be of different sizes, even on a single system, but are usually large enough to contain the main AES tables. 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, 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).

Additional complications arise, such as methods for obtaining high-resolution, low-latency time measurements. These are all surmountable, but are omitted here for brevity.

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

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.[20] 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 set out to test the attacks on a real-life encrypted filesystem. We set up a Linux `dm-crypt` device, which is a virtual device which uses underlying storage (here, a loopback device connected to a regular file) and encrypts all data at the sector level (here, using 128-bit AES encryptions in ECB mode). On top of this we create and mount an ordinary ext2 filesystem. We then trigger 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 succeed in extracting the full key after just 800 write operations done in 65ms (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 Figures 5 and 6 (for visual clarity, the figures depict a larger number of samples).

The Evict+Time measurements (Figure 4) let us recover the secret key using about 500,000 samples when attacking OpenSSL on 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 ones (a more comprehensive account will appear in the full version of this paper).

We have discussed known-plaintext attacks. All of these techniques can be applied analogously in known-ciphertext setting. In fact the latter are significantly more efficient for AES implementations of the form given in 2.2, since the last round uses a dedicated set of tables and the noise due to other rounds is thus eliminated. Also, in the last round we have non-linearity but no MixColumn operation, so we can easily extract the full key without analyzing additional rounds. Note that even in the case of known-plaintext, the final guess of the key can be efficiently verified by checking the resulting predictions for the lookups in the last round.

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

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

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.

To verify the results of the second-round analysis, or in case some of the tables cannot be analyzed due to noise (e.g., see Figure 5), 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.

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 leaks information on memory accesses with resolution better than δ, hence 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.

We believe this attack can be converted into a remote attack on a network-triggerable cryptographic network process (e.g., IP/Sec or OpenVPN). The cache manipulation can be done remotely, for example by triggering accesses to the network stack's TCP connection table, but its efficiency 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 synchronously before and after 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 a non-uniform 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 processors with 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. For a simple but common case, suppose 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 `p`). 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_ℓ; see Section 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. 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[21]). Specifically, suppose that for $n = 0, \ldots, 256/\delta - 1$ the attacker knows $R(n) = \Pr[\langle p_i \rangle = n]$, i.e., the histogram of the plaintext bytes truncated into blocks of size δ (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 $R(n \oplus \langle k_i \rangle)$.

4.3 Measurements

One measurement method exploits the simultaneous multithreading feature available in some high-performance processors (e.g., Pentium and Xeon processors with HyperThreading). 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; in particular, it can gather 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 Section 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, i.e., W memory blocks mapped to that cache set (similarly to step (c) of the Prime+Probe measurements).[22] 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, that 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, giving us the frequency score $F_\ell(n)$.

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

[22] 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.

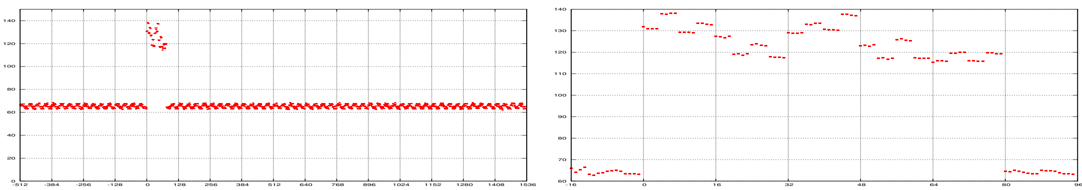

Fig. 7. Frequency scores for OpenSSL AES encryption of English text. Horizontal axis: cache set. Timings performed on 3GHz Pentium 4E with HyperThreading. To the right we zoom in on the AES lookup tables; the pattern corresponds to the top nibbles of the secret key `0x004080C0105090D02060A0E03070B0F0`.

4.4 Experimental Results

Attacking a series of processes encrypting English text with the same key using OpenSSL, we effectively retrieve 45.66 bits of information[23] about the key after gathering timing data for about 1 minute. Timing data from one of the runs is shown in Figure 7.

4.5 Variants and Extensions

This attack vector is quite powerful, and numerous extensions are possible. For example:

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

If measurements can be made to detect 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 key without even known plaintext *distribution*.

We have demonstrated the attack on a Pentium 4E with HyperThreading, but it can also be performed on other platforms without relying on simultaneous multithreading. The key is for the attacker to find a way to execute its own code midway through an encryption, and this can be achieved by exploiting the interrupt mechanism. For example, the attacker can predict RTC or timer interrupts and yield 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 what the encrypting process accessed during those few cycles.

On multi-core processors, the lowest-level caches (L1 and sometimes L2) are usually private to the 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.

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

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 implementors. 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 operating system kernels or normal user processes, running under today's common general-purpose processors. Some of these measures are presented as specific to AES, but have analogues for other primitives.

Caveat: due to the complex architecture-dependent considerations involved, we expect secure implementation of these countermeasures to be a very delicate affair. Implementors should consider all exploitable effects given in [1], and carefully review their architecture for additional effects.

5.1 Avoid 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[24].

Another approach is that of bitslice implementations [2]. 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 indeed amortizing cost across several simultaneous encryptions (i.e., the use of appropriate encryption mode). For AES, we expect (but have not yet verified) that amortized performance would be comparable to that of a lookup-based implementation, but its relevance is application-dependent.

Finally, one could use lookup tables but place the tables in registers instead of cache. Some architectures (e.g., x86-64 and PowerPC AltiVec) have register files sufficiently large to hold the 256-byte S-box table, but reasonable performance seems unlikely.

5.2 Alternative Lookup Tables

There are alternative formulations of AES, using a smaller set of tables. We have considered the most common implementation, employing four 1024-byte tables $T_0, \ldots, T_3$ for the main rounds.

[24] This kind of implementation has also been attacked through the timing variability in some implementations [5].

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 non-aligned lookups). The same applies to the last round tables, $T_0^{(10)}, \ldots, T_3^{(10)}$.

In regard to the synchronous attacks considered in Section 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 unfeasible – we're 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 Section 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 2KB table ($8 \to 64$ bit) has $\rho = (1-1/32)^{160}$, making the synchronous attack just 18 times less efficient than in Section 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 (this 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 Sections 5.3, 5.5, 5.8).

5.3 Data-Oblivious Memory Access Pattern

Instead of avoiding table lookup, one could employ them but ensure that the pattern of accesses to the memory is completely oblivious to 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 timing are completely data-independent.

More efficiently, one can read one representative element from each memory block.[25] For the implementation of AES given in Section 2.2 and the typical $\delta = 16$, this means each logical table access would involve 16 physical accesses and — a major slowdown. Note that in the formulation of AES using a single 256-byte table (see Section 5.2), the table consists of only 4 memory blocks (for $\delta = 64$), so every logical table access would involve just 4 physical accesses. However, this formulation is inherently very slow.

Goldreich and Ostrovsky [4] gave a generic program transformation for hiding memory accesses, which is quite satisfactory from an (asymptotic) theoretical perspective. However, its concrete overheads in time and memory size appear too high for most applications. Moreover,

[25] This is insufficient on processors which leak low address bits (see Section 3.8).

it employs pseudo-random functions, whose typical realizations employ the same cryptographic primitives we are trying to protect.[26]

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 [15] or occasionally permuting the memory and keeping the cache locked between permutations [16]. Both techniques require non-trivial 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. [11] and the references within). However, these 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, and our asynchronous attacks do not fall into this category. Moreover, the security proofs consider leakage only of specific intermediate values, which do not correspond to the ones leaking through accessed memory addresses. 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 Section 5.1). Thus, this venue presently seems unfruitful.

5.5 Cache State Normalization and Process Blocking

If one considers only the synchronous attacks of Section 3 then it suffices to simply normalize the state of the cache just before encryption (to prevent the initial cache state from affecting the encryption, as in Evict+Time) and just after the encryption (to prevent the encryption from affecting the final cache state, as in Prime+Probe). Normalization can be achieved, for example, by loading all lookup tables into the cache. However, as pointed out in [1, Sections 12 and 14], it should be ensured that the table elements are not evicted by the encryption itself, by accesses to the stack, inputs or outputs. Ensuring this is a delicate architecture-dependent affair.

[26] In [4] it is assumed that the pseudorandom functions are executed completely within a secure CPU, without memory accesses.

This method provides little protection against the asynchronous attacks of Section 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 is suggested in [12].

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 Disable 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 a single-threaded processor, it would require flushing all caches during every context switch. On a processor with simultaneous multithreading, it would also require the logical processors to use separate logical caches, statically allocated within the physical cache; some modern processors do not support such a mode. One would also need to consider the effect of cache coherency mechanisms in SMP configurations.

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 Section 5.11.

5.7 Static or Disabled Cache

One brutal countermeasure against the cache-based attacks is to completely disable the CPU's caching mechanism.[27] Of course, the effect on performance would be devastating. A more attractive alternative is to activate a "no-fill" mode 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 its inputs are in cache when "no-fill" is enabled), but its output will not be cached, leading to subsequent cache misses. Other processes executed during (c), via multitasking or simultaneous multithreading, will however

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

incur a 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 work set.

Intel's family of Pentium and Xeon processors supports such a mode, but the cost of enabling and disabling it are prohibitive. Also, some ARM implementations have a mini-cache which can be locked, but it is too small for the fastest table-based formulations. We do not know which other processors offer this functionality.

This method can be employed only in privileged mode, which is typically available only to the operating system kernel (see Section 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 no automatic transfers to or from main memory and thus, if used as cryptographic co-processors, would not be susceptible to this attack.[28]

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 memory several times during each encryption[29]. If the tables reside in different memory pages, one should consider and prevent leakage (and performance degradation) through page table cache (TLB) 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).[30]

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.

5.9 Hiding the Timing

All of our attacks perform timing measurements, whether of the encryption itself (in Section 3.4) or of accesses to the attacker's own memory (in all other cases). A natural countermeasure for timing

[28] 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 seems to have considerable performance potential in cryptographic and cryptanalytic applications.

[29] Recall that if the tables stay static for long, the attacker can easily locate them; see Section 3.6.

[30] Some of these variants were suggested to us by Intel Corp.

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 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 (as discussed in [12]) or accuracy of the clock available to the attacker. This will decrease the signal-to-noise ratio in the attacker's measurements, but by averaging a larger number of samples the attacker can still obtain the same information as before. Since some of our attacks require only a few milliseconds of measurements, to make them unfeasible the clock accuracy would have to be degraded to an extent that interferes with legitimate applications.

5.10 Selective Round Protection

The attacks described in Sections 3 and 4 detect and analyse 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 the means given above (i.e., hiding, normalizing or preventing memory accesses), while using the faster, unprotected implementation for the internal rounds.[31] 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). None the less, those cryptanalytic techniques require more data and/or chosen data, and thus when quantitatively balancing resilience against cache-based attacks and performance, it it sensible to provide somewhat weaker protection for internal rounds.

5.11 Operating System Support

Several of the above suggestions 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 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 an operating system upgrade.

An alternative approach is to provide a secure execution facility to user processes.[32] This facility would allow the user to mark a "sensitive section" in his code and ask the operating

[31] This was suggested to us by Intel Corp.

[32] Special cases of this were discussed in [12] and [1], though the latter calls for this to be implemented at the CPU hardware level.

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).

6 Conclusions

6.1 Vulnerable Cryptographic Primitives

Throughout the above we have concentrated on the AES cipher, both due to its importance and due to its high susceptibility to this form of attack. However, the attack is not specific to AES, or indeed to block ciphers. Any implementation of a cryptographic primitive that performs data-dependent memory accesses is, in principle, vulnerable. The efficiency of the attack depends heavily on the structure of the cipher and chosen implementation. Heuristically, large lookup tables increase the effectiveness of all attacks, as do large lookup entries; 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.

For example, DES is vulnerable when implemented using large lookup tables (to account for 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 [12]). 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 and bitsliced Serpent, are impervious to the attacks described here. However, to protect against timing attacks one should scrutinize implementations for use of instructions whose timing is data-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 could be measured by an unrelated process running on the same machine, by a variant of the asynchronous attack, through the effect on the *timing* of memory accesses.

6.2 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. Clearly this includes many multi-user systems. Disturbingly, virtual machines and sandboxes offer little protection, since for the asynchronous attack the attacker needs only the ability to access his own memory and measure time. Thus, the attack may cross the boundaries supposedly enforced by FreeBSD `jail()`, the Java Virtual Machine, VMware, Xen and NGSCB. Systems relying on impenetrable virtualization boundaries (and thus potentially compromised) include web browsers, various DRM schemes, a patent recently granted to the US National Security Agency [9], and numerous custom deployments.

Single-user machines are not exempt from this attack, as untrusted code included in web pages may employ the attack. This potentially includes ActiveX controls (which have other grave security concerns), Java applets (which can, e.g., time their accesses to a simple byte array), and plausibly even JavaScript code may be able to perform the attack with sufficient efficiency.

Remote attacks are in principle possible, and if proven efficient could pose serious threats to secure network connections such as IP/Sec and OpenVPN.

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 non-cryptographic systems and may, in some cases, leak sensitive information directly.

6.3 Mitigation

We have described a variety of countermeasures against cache state analysis attacks; some of these are generic, while some are specific to AES. However, none of these unconditionally mitigates the attacks while offering performance close to current implementations. Thus, finding an efficient 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.

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 for directing us to reference [6].

References

1. Dan Bernstein, *Cache-timing attacks on AES*, preprint, 2005, http://cr.yp.to/papers.html#cachetiming
2. Eli Biham, *A fast new DES implementation in software*, proc. FSE 1997, LNCS 1267, 260–272, Springer, 1997
3. Joan Daemen, Vincent Rijmen, *AES Proposal: Rijndael*, version 2, AES submission document, 1999, http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael-ammended.pdf
4. Oded Goldreich, Rafail Ostrovsky, *Software protection and simulation on oblivious RAMs*, Journal of the ACM, vol. 43 no. 3, 431–473, 1996
5. François Koeune, Jean-Jacques 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
6. Wei-Ming Hu, *Lattice scheduling and covert channels*, IEEE Symposium on Security and Privacy, 52–61, IEEE, 1992
7. John Kelsey, Bruce Schneier, David Wagner, Chris Hall, *Side channel cryptanalysis of product ciphers*, proc. 5th European Symposium on Research in Computer Security, LNCS 1485, 97–110, Springer-Verlag, 1998
8. Daniel 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
9. Robert V. Meushaw, Mark S. Schneider, Donald N. Simard, Grant M. Wagner, *Device for and method of secure computing using virtual machines*, US patent 6,922,774, 2005
10. National Institute of Standards and Technology, Advanced Encryption Standard (AES) (FIPS PUB 197), 2001
11. Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, Vincent Rijmen, *A side-channel analysis resistant description of the AES S-box*, proc. FSE 2005, Springer-Verlag, to appear
12. Colin Percival, *Cache missing for fun and profit*, BSDCan 2005, Ottawa, 2005; see http://www.daemonology.net/hyperthreading-considered-harmful/
13. Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, Hiroshi Miyauchi, *Cryptanalysis of DES implemented on computers with cache*, proc. CHES 2003, LNCS 2779, 62-76, 2003
14. Johannes Wolkerstorfer, Elisabeth Oswald, Mario Lamberger, *An ASIC implementation of the AES SBoxes*, proc. CT-RSA 2002, LNCS 2271, 67–78, Springer, 2002
15. Xiatong Zhuang, Tao Zhang, Hsien-Hsin S. Lee, Santosh Pande, *Hardware assisted control flow obfuscation for embedded processors*, proc. International Conferenfce on Compilers, Architectures and Synthesis for Embedded Systems, 292-302, ACM, 2004
16. Xiaotong Zhuang, Tao Zhang, Santosh Pande, *HIDE: An Infrastructure for Efficiently protecting information leakage on the address bus*, proc. Architectural Support for Programming Languages and Operating Systems, 82–84, ACM, 2004