Robust Final-Round Cache-Trace Attacks Against AES

Joseph Bonneau

Computer Science Department, Stanford University jbonneau@stanford.edu

Abstract. This paper describes an algorithm to attack AES using side-channel information from the final round cache lookups performed by the encryption, specifically whether each access hits or misses in the cache, building off of previous work by Acıiçmez and Koç [AK06]. It is assumed that an attacker could gain such a trace through power consumption analysis or electromagnetic analysis. This information has already been shown to lead to an effective attack. This paper interprets cache trace data available as binary constraints on pairs of key bytes then reduces key search to a constraint-satisfaction problem. In this way, an attacker is guaranteed to perform as little search as is possible given a set of cache traces, leading to a natural tradeoff between online collection and offline processing. This paper also differs from previous work in assuming a partially pre-loaded cache, proving that cache trace attacks are still effective in this scenario with the number of samples required being inversely related to the percentage of cache which is pre-loaded.

Keywords: AES, cryptanalysis, side-channel attack, power analysis, cache.

1 Introduction

Side-channel attacks have been demonstrated experimentally against a variety of cryptographic systems. Side-channel attacks utilize the fact that in reality, a cipher is not a pure mathematical function $E_K[P] \rightarrow C$, but a function $E_K[P] \rightarrow (C, t)$, where t is any additional information produced by the actual encryption operation. Often the additional data t leaks enough useful information for an attacker to fully recover the key. The attack presented in this paper focuses on obtaining a "trace" of cache accesses performed by the encryption by analyzing the power consumption of the encryption.

In 1997, Rijmen and Daemen proposed the Rijndael cipher to the National Institute of Standards and Technology (NIST) as a candidate for the Advanced Encryption Standard (AES). After four years of competition, Rijndael was chosen by NIST in October 2000 and officially became AES in 2001 with US FIPS 197. The cipher is now widely deployed and is expected to be the world's predominant block cipher over the next 25 years. In its final evaluation of Rijndael [NBB$^+$00], NIST stated that table lookup operations are "easy to defend against power attacks" and regarded Rijndael as the easiest among the finalists to defend against side-channel attacks in general.

In spite of this favorable review, side channel cryptanalysis has been the most effective approach to attacking AES so far. The heavy use of lookup tables has been widely demonstrated to be an exploitable cryptographic side-channel on computers with cached memory [TTMM02,TSS$^+$03,Pag02,Per05]. Side-channel attacks against AES have been based on direct cache observation [OST06], timing [Ber05,BM06], and most recently power consumption [BBM$^+$06,AK06]. The reliance on lookup tables is once again exploited in the attack described in this paper.

2 Overview of the AES cipher

A full description of the Rijndael cipher is provided in [DR02], but below is a brief description of the cipher's properties that are utilized in this study.[1] AES is an iterated cipher: Each round i takes a 16-byte block of input X^i and a 16-byte block of key material K^i, producing a 16-byte block of output X^{i+1}, using a temporary 16-byte state buffer T. Each round is carried out as follows:

$$\begin{aligned} T &:= \texttt{SubBytes}(X^i), \\ T &:= \texttt{ShiftRows}(T), \\ T &:= \texttt{MixColumns}(T), \\ X^{i+1} &:= T \oplus K^i. \end{aligned}$$

While the three functions `SubBytes`, `ShiftRows`, and `MixColumns` are specified as algebraic operations, performance-oriented software implementations of AES combine all three operations and pre-compute the values.[2] The values are stored in large lookup tables, T_0, T_1, T_2, T_3, each mapping one byte of input to four bytes of output. In software, these tables each require $2^8 \cdot 4 = 1024$ bytes of storage. Each round is carried out by splitting up the 16 bytes of X^i into $x_0^i, x_1^i, \ldots, x_{15}^i$, and the 16 bytes of K^i into $k_0^i, k_1^i, \ldots, k_{15}^i$. The encryption round is then carried out as:

$$\begin{aligned} X^{i+1} = \{&T_0[x_0^i\ ] \oplus T_1[x_5^i\ ] \oplus T_2[x_{10}^i] \oplus T_3[x_{15}^i] \oplus \{k_0^i\ , k_1^i\ , k_2^i\ , k_3^i\ \}, \\ &T_0[x_4^i\ ] \oplus T_1[x_9^i\ ] \oplus T_2[x_{14}^i] \oplus T_3[x_3^i\ ] \oplus \{k_4^i\ , k_5^i\ , k_6^i\ , k_7^i\ \}, \\ &T_0[x_8^i\ ] \oplus T_1[x_{13}^i] \oplus T_2[x_2^i\ ] \oplus T_3[x_7^i\ ] \oplus \{k_8^i\ , k_9^i\ , k_{10}^i, k_{11}^i\}, \\ &T_0[x_{12}^i] \oplus T_1[x_1^i\ ] \oplus T_2[x_6^i\ ] \oplus T_3[x_{11}^i] \oplus \{k_{12}^i, k_{13}^i, k_{14}^i, k_{15}^i\}\}. \end{aligned} \tag{1}$$

The round calculation can be performed very efficiently in software this way, using just 16 table lookups and 16 word-length x-or's. A complete encryption in 128-bit mode consists of an x-or with the first 16 bytes of key material, referred to as "input whitening," followed by 9 normal encryption rounds, plus

[1] This paper will focus exclusively on AES with a 128 bit key. 192 and 256 bit versions use a different key expansion algorithm and more rounds.

[2] This formulation was a part of the original Rijndael proposal [DR02]. Most public AES implementations have made no significant changes to this original source code.

a simplified final round. The final round performs no `MixColumns` operation as it might trivially be inverted by an attacker and would ostensibly slow down hardware implementations. This omission will prove crucial, as it causes software implementations to use a new table T_4 in the last round, which is just the AES S-Box used in `SubBytes`:

$$\begin{aligned} C = \{&T_4[x_0^{10}] \oplus k_0^{10}, T_4[x_5^{10}] \oplus k_1^{10}, T_4[x_{10}^{10}] \oplus k_2^{10}, T_4[x_{15}^{10}] \oplus k_3^{10}, \\ &T_4[x_4^{10}] \oplus k_4^{10}, T_4[x_9^{10}] \oplus k_5^{10}, T_4[x_{14}^{10}] \oplus k_6^{10}, T_4[x_3^{10}] \oplus k_7^{10}, \\ &T_4[x_8^{10}] \oplus k_8^{10}, T_4[x_{13}^{10}] \oplus k_9^{10}, T_4[x_2^{10}] \oplus k_{10}^{10}, T_4[x_7^{10}] \oplus k_{11}^{10}, \\ &T_4[x_{12}^{10}] \oplus k_{12}^{10}, T_4[x_1^{10}] \oplus k_{13}^{10}, T_4[x_6^{10}] \oplus k_{14}^{10}, T_4[x_{11}^{10}] \oplus k_{15}^{10}\}. \end{aligned} \tag{2}$$

A total of 10 rounds are used in 128-bit AES, but 11 16-byte blocks of key material are needed because of the input-whitening. These 176 bytes of key material are generated by taking the raw 16-bytes of the key and repeatedly carrying out a non-linear transformation which produces the next 16-byte block based on the previous 16-byte block until all 176 bytes are created. This key expansion structure has two important properties. First, the 16 bytes of key material used for input whitening prior to the first encryption round are just the raw key bytes. Second, the key expansion algorithm was explicitly chosen [DR02] to be invertible given any 16 consecutive bytes of the expanded key. This is useful to an attacker in that recovery of the final 16 bytes of the expanded key (or any other 16 bytes) is equivalent to recovery of the original key.

A complete AES encryption can be viewed in terms of the table lookups it performs: for each of the tables T_0, T_1, T_2, T_3, four lookups are made in each of the first nine rounds for a total of 36 lookups. In the final round 16 lookups are made into the table T_4, bringing the total number of lookups performed to 160. We will call the entire set of lookup indices $\{l_1, l_2, \dots l_{160}\}$ a *lookup trace* for the encryption operation. Knowing a complete lookup trace for a single encryption operation, along with either the ciphertext or plaintext, would instantly be enough information to recover the key [3]. Such a simple attack would require the attacker have control over the target machine's memory bus, which is unlikely in practice. However, there are multiple side-channels which leak enough partial information about an encryption's lookup trace to recover the key.

3 Cache Model and Attack Assumptions

The attacks in this paper assume the computer performing the encryption operation uses cached memory which can be described using a simple model of the cache. A cache is a small, fast storage area sitting between the CPU and main memory. When values are looked up in main memory, they are stored in

[3] In fact, it would be enough to simply know the lookup trace of the first round of one encryption, along with the plaintext (or alternatively, the lookup trace of the final round and the ciphertext).

the cache, usually evicting older values in the cache. Subsequent lookups to the same memory address can then retrieve the data from the cache, which is faster than main memory, this is called a “cache hit.” Because most software exhibits temporal locality in memory accesses, caches greatly improve performance.

Unfortunately, because a cache miss on a modern processor requires fetching the requested data from main memory, as well as possibly inserting a stall into the pipeline or attempting to execute further instructions out of order, the processor uses more electricity during a cache miss than a hit. As a result, cache misses are easily identifiable as spikes in the power consumption graph of a processor performing AES encryption [BBM$^+$06]. We assume that it is possible for an attacker to recover a *cache trace* for the final round of an encryption by such power analysis methods. In contrast with a lookup trace, a cache trace does not include the exact indices looked up, only whether the lookup hit in cache or missed in cache. We denote a cache trace for the final round by $\{t_1, t_2, \ldots, t_{16}\}$, where each t_i is either “hit” or ”miss” in correlation with the result of the i^{th} table lookup in the final round.

Modern caches do not store individual bytes, but groups of bytes from consecutive “lines” of main memory. Line size varies, but two common values are 32 bytes for a Pentium III and 64 bytes on more recent Pentium IV processors. Since the usual size of AES table entries is 4 bytes, groups of 8 consecutive table entries share a line in the cache on a Pentium III, this value is defined as δ in [OST06]. In general it will hold that $\delta = \frac{l}{b}$ where l is the processor’s . line size and b is the block size of table entries.

The value of δ is critically important to cache-based attacks because for any addresses a, b which are equal ignoring the lower $\log_2 \delta$ bits (notated as $\langle a \rangle = \langle b \rangle$ in [OST06]), looking up address a will cause an ensuing access to b to hit in cache instead of main memory. This has been noticed to be a significant problem for attacks on the first round of AES in that it makes recovery of the low-order key bits impossible [AK06,OST06,BM06]. In particular, this effect requires the attack of Acıiçmez and Koç to consider the second round and become significantly more complex.

Unlike the cache trace attacks described by Acıiçmez and Koç[AK06], we will not assume the cache is completely clean prior to encryption. For instance, if t_2 is “hit” for some encryption’s cache trace, we do not conclude that $\langle l_1 \rangle = \langle l_2 \rangle$. This could certainly be the case, but it could also be the case that the cache line needed by the lookup l_2 happened to be in cache prior to the start of the encryption. This would represent a *false hit* in that it did not hit in cache due to the lines loaded by the encryption itself, but due to the line being pre-loaded[4].

We do assume that any cache line loaded during encryption will remain there for the duration of the encryption. This is a safe assumption because the T_4 table, with a size of 1024 kB, should fit easily into the L1 cache of any modern processor. Since the table entries are the only values loaded during encryption, the encryption routine will not evict any of the table lines out of cache once

[4] If the assumption is made that the cache is clean prior to encryption, the attack presented in this paper can be significantly improved, as discussed in Section 8

they are loaded. It would be possible for lines to be evicted in the middle of an encryption if the process were context switched out, but this would be evident in the power usage of the encryption, and the sample could then be discarded. This assumption is critical because it allows to conclude, if t_i is "miss" for some encryption, that $\langle l_i \rangle \neq \langle l_j \rangle$ for any $j < i$.

4 Related Work

Side-channel attacks have been demonstrated against implementations of many cryptosystems, utilizing timing [Ber05,TSS$^+$03,Koc96,BB05,KQ99,BM06], power consumption [ABDM00,KJJ99], electromagnetic radiation [GMO01], etc. Side channel attacks have been most effective against public-key cryptographic implementations, particularly smart cards. Public key algorithms are vulnerable because they typically perform lengthy mathematical operations, whose time and power consumption are data-dependent. In the AES selection process, Rijndael was considered "favorable" to defend against side-channel attacks because it did not utilize conditional statements or data-dependent rotations, which were commonly the basis of the side-channel attacks on public key systems [DR99]. In the final evaluation, NIST agreed that Rijndael appeared safe against side-channel attacks, considering table-lookups a "safe" operation [NBB$^+$00].

To the surprise of its designers and NIST, AES has proved vulnerable to side-channel attacks on machines with cached memory due to its use of table lookups. Direct observation attacks, as demonstrated by Osvik, Shamir, and Tromer [OST06], discover specific information as to what values are present in an encryption's lookup trace by running code on the target machine before and after encryption. For example, if the attacker can determine that, whenever $p_0 = z$, the data in $T_0[z']$ is accessed during encryption, then it must be the case that $x_0^0 = z'$. Since it holds that $p_0 \oplus k_0 = x_0^0$, the attacker can conclude that $k_0 = z \oplus z'$. This principle is used to construct sophisticated, effective attacks which are successful with as few as 800 encryption samples.

An encryption's lookup trace also affects the overall time of the encryption. This effect was first used to construct attacks against DES by Tsunoo et al. [TSS$^+$03,TTMM02], who also described possible attacks against AES. A successful statistical timing attack against AES was presented by Bernstein [Ber05]. More efficient timing attacks against AES directly using cache effects were presented by Bonneau and Mironov [BM06]. These timing attacks are made possible because a pair of table lookups to the same index will be faster than a pair of lookups to two different indices, thus, the presence of collisions in an encryption's lookup trace will usually speed up the encryptions. The latter attacks found increased success by studying timing due to cache-collisions in the final round of AES encryption, similar to the approach in this paper. Timing attacks are attractive in that, theoretically, the can be carried out by a remote attacker, although so far this has not been possible against AES due to the precise timing information required.

Finally, information is leaked about the lookup-trace through power analysis. This possibility was mentioned by Page [Pag02] and by Kelsey et al [KSWH00]. Power analysis and electromagnetic analysis has previously been used as a side-channel to attack cryptographic smartcards [KJJ99,GMO01]. For AES, the critical point is that cache misses require the functioning of a large amount of circuitry to fetch the desired data from main memory. As a result, cache misses should show up as huge spikes in a graph of power usage. As demonstrated by Bertoni et al [BBM^{+}06], given a power usage trace of an AES encryption, it is easy to tell which table lookups resulted in cache misses and which resulted in cache hits. They used this information to construct an attack by specifically evicting a small part of the AES tables prior to encryption, then observing if a miss occurred to indicate that the evicted data was needed during the encryption.

More powerful attacks were recently described by Acıiçmez and Koç [AK06]. Ine one attack, they use the entire first and second round traces, along with known plaintext, to recover an AES key. For example, suppose the second access to T_0 in the first round results in a cache hit. Then it can be deduced that the first two lookups in T_0 must be the same. We can see from Equation 1 that these lookups are just $P_0 \oplus K_0$ and $P_4 \oplus K_4$, so we have learned that $K_0 \oplus K_4 = P_0 \oplus P_4$. Due to the use of cache-lines on modern processors (see Section 3), the attack must also examine the second round, which ultimately narrows the number of possible keys to around 2^{32} given sufficient cache traces. They also develop a similar final round attack, which is simpler and also effective.

These attacks are significant in that they can recover the full key after observing between 5 to 50 encryptions, with a varying amount of exhaustive search required. Acıiçmez and Koç correctly stress that the exhaustive search can be carried out “offline” after the encryption traces are obtained so it is often desirable to do more offline work if it means fewer traces are needed. The final round attack presented in this paper is very similar to the work of Acıiçmez and Koç, but views the attack in a different manner as a constraint satisfaction problem.

Finally, Acıiçmez and Koç assume a completely “clean” cache prior to encryption, that is, no AES table entries are already loaded into cache. If an attacker can run code on the target machine between encryptions, this can be achieved by reading enough garbage data from memory. However, if the attacker cannot run code, there may be no way to ensure all AES tables are out of cache. A single table entry left in cache is enough to foil the attack if it assumes hits are generated only from the encryption itself, this is a significant limitation. This paper does not make a clean cache assumption and explores the effects of a partially pre-loaded cache. In particular, this means ignoring hits in the cache trace, since it is unknown whether or not they are “false hits” due to a pre-loaded cache.

5 The Attack

5.1 Constraint inference

The goal of the attack is to use a series of final-round cache traces, as well as known ciphertexts, to recover the key. To do this, we wish to gather as many constraints as possible on the key bytes from each cache trace. We consider only binary constraints, that is, for each pair of key bytes (k_i^{10}, k_j^{10}), we wish to rule out as many possible values (x, y) for that pair. Given a cache trace $\{t_1, t_2, \ldots, t_{16}\}$ for the final round of some encryption, we consider Equation 2.

Suppose that some value $t_j =$ "miss", that is, we know the j^{th} lookup resulted in a cache miss. We know that it is then not equal to any of the previous cache lookups indices, for all t_i, with $i < j$, we know that $\langle l_i \rangle \neq \langle l_j \rangle$. Consider the corresponding ciphertext bytes c_i and c_j. We know that $c_i = k_i^{10} \oplus T_4[l_i]$ and $c_j = k_j^{10} \oplus T_4[l_j]$. Note the key bytes k_i^{10} and k_j^{10} are taken from the final round of the expanded key schedule. If we guess some value z for the value of k_i^{10}, we can compute[5] the conditional lookup index $l_i^z = T_4^{-1}[c_i \oplus z]$, this is the value of l_i which would have been accessed if z is correct. We know that the lookup index $\langle l_i \rangle \neq \langle l_j \rangle$, so we wish to compute all values of l_j which would be impossible if l_i^z is the correct lookup index, these are just any values for which $\langle l_i^z \rangle = \langle l_j \rangle$. There will be δ such values, which can easily be enumerated by computing $l_j^z(n) = l_i^z \oplus n$ for all $0 \leq n < \delta$.

Finally, for each $l_j^z(n)$, we compute $z'(n) = T_4[l_j^z(n)] \oplus c_j$, this corresponds to the value for the key byte k_j^{10} which would be required to be consistent with the known ciphertext bytes c_i, c_j and the value z for k_i^{10}. We can thus conclude that it is impossible to have $k_i^{10} = z$ and $k_j^{10} = z'(n)$, because this would have meant $\langle l_i \rangle = \langle l_j \rangle$ and l_j would have hit in the cache. So all pairs $(z, z'(n))$ for $0 \leq n < \delta$ are ruled out as possible values for (k_i^{10}, k_j^{10}).

We can repeat this process for every value of $0 \leq z \leq 255$, producing a total of $256 \cdot \delta$ constraints on (k_i^{10}, k_j^{10}). Also, we can produce constraints for all pairs i, j with $i < j$, so in total $j \cdot 256 \cdot \delta$ constraints are produced. Notice that the presence of the S-box between the known ciphertext values and the lookup indices actually improves the attack. Because the S-box is non-linear, the values $z'[n]$ will appear to be randomly distributed throughout the range $[0, 255]$ given any one z. Although the ruled-out lookup values $l_i^z, l_j^z(n)$ only differ in their low-order bits, this small differential is destroyed by the S-box, leading to a good distribution of constraints on (k_i^{10}, k_j^{10}) and avoiding the problems of the first round attacks in [AK06]. This effect was also useful in the timing attacks described in [BM06].

5.2 Constraint propagation

With ample cache traces, it would easy be to produce enough constraints to uniquely identify each key byte by the analysis above, eventually, the only un-

[5] Since T_4 is the AES S-box, a permutation, we know that T_4^{-1} is a well-defined function and trivial to compute

constrained possibility for (k_i^{10}, k_j^{10}) would be the correct pair of values. However, well before this occurs there will be enough constraints that only one possible assignment to $k_1^{10}, k_2^{10}, \cdots, k_{16}^{10}$ will satisfy all constraints, although each individual byte will appear to have multiple possible values. This is just the classic constraint satisfaction problem in artificial intelligence, and good search algorithms exist to find keys which solve the constraint sets produced in this attack. With enough constraints, a small enough number of keys will identified as possible, which can presumably be checked against a known plaintext/ciphertext pair.

To illustrate the problem, suppose after analyzing all of our cache traces we are left with $(k_1^{10}, k_2^{10}) \in \{$`(0x01, 0x02), (0x04, 0x05)`$\}$, $(k_1^{10}, k_3^{10}) \in \{$`(0x01, 0x03), (0x06, 0x07)`$\}$, $(k_2^{10}, k_3^{10}) \in \{$`(0x02, 0x03), (0x08, 0x09)`$\}$. In this example it is obvious that the only possibility is $(k_1^{10}, k_2^{10}, k_3^{10}) =$ `(0x01, 0x02, 0x03)`, even though by constraints alone it appears that k_1^{10} could be any of `(0x01, 0x04, 0x06)`. The value `0x04` is clearly impossible for k_1^{10} though, since there is no possible value for k_3^{10} which could then satisfy the constraints.

To check for consistency and eliminate all values, we can use the AC-3 algorithm [Mac77], which checks for consistency between possible values and the constraints, and then propagates constraints. The algorithm works as follows: First, the domain for every byte k_i^{10} is set to $\{0, \ldots, 255\}$. Next all pairs (i, j) are placed in a queue. As each (i, j) is de-queued, each value x remaining in the domain of k_i^{10} is checked for consistency. If for any other byte k_j^{10}there does not exist a possible value $(k_i^{10}, k_j^{10}) = (x, y)$ for some y remaining in the domain of k_j^{10}, then x is eliminated from the domain of k_i^{10}. If any values are eliminated, then all pairs (k_i^{10}, k_j^{10}) are re-added to the end of the queue. The algorithm stops when the queue is emptied.

AC-3 is a natural choice for this attack since it guarantees consistency for binary constraints, which are produced by the power analysis. AC-3 has asymptotic complexity $O(n^2 d^3)$, where n is the number of nodes (key bytes) and d is the initial domain (in this case 256). So, the asymptotic running time of 2^{32} is possible, although in practice the algorithm as tested against constraint sets which the attack would produce runs very efficiently, taking only seconds on a personal computer. Similar constraint propagation algorithms exist with better asymptotic complexity, but for this attack they would probably not provide a considerable speedup.

5.3 Search

The correct key is identified by search, using the constraint propagation algorithm to guide the search. First, a guess x_0 is made for k_0^{10}. Then, based on this assignment, forward propagation is used to eliminate values for the remaining 15 bytes. For each key byte k_j^{10} for $j > 0$, we eliminate the value y from the domain of k_j^{10} if we know from the constraint set that $(k_0^{10}, k_j^{10}) \neq (x_0, y)$. After forward propagation, the AC-3 constraint propagation algorithm is run again to remove inconsistent values from the domains of the remaining key bytes.

Next, a guess x_1 is made for k_1^{10} based on the key byte's remaining domain, and another round of forward propagation and AC-3 propagation is performed.

When a full guess for all key bytes is made, this guess is then checked against a known plaintext/ciphertext pair for correctness [6].

The search must backtrack if either a complete guess is made which is incorrect, or if at some point the domain of an unassigned key byte becomes empty. In either case, the last assignment made is incorrect, it must be undone and a new assignment made. This also requires unrolling the constraint propagation that may have occurred due to the assignment being removed, this is facilitated by storing in the domain of each key byte at what search depth constraints were placed. In this manner, backtracking only requires a single scan over each byte's domain to remove incorrect constraints.

Because AC-3 is an optimal constraint propagation algorithm for binary constraints, this search is optimal in that it will eventually find the correct key, and will search as few incorrect keys as is possible. In the extreme case of no constraints being known, the algorithm is equivalent to exhaustive search.

5.4 Decryption Attack

AES decryption in software is very similar to encryption, with a slightly different key schedule, and equally vulnerable to cache trace attacks. The attack is slightly simpler against decryption since it recovers constraints on the raw key, instead of the final section of the expanded key. In decrypt mode, the attacks require known plaintext instead of known ciphertext.

6 Experimental Results

6.1 Experimental Setup

As mentioned previously, the attack will always succeed given enough time, although this could mean exhaustive search of every possible key if no constraints are known and would not be a meaningful attack. To study the effectiveness of the attack, we set a threshold τ of offline work that the attacker is able to perform in attempting to recover the key given a set of n cache traces. It is impossible to precisely quantify "work" because different operations involved in the attack, for instance checking a key for correctness vs. updating one element in a table of constraints, will have a different fixed cost. We consider either checking and updating one table entry, or checking one key for correctness, as one "operation" and consider τ to be the maximum number of operations practical for the attacker to carry out. By stopping the search after τ has been exceeded, the amount of work done either in searching or propagating constraints to a fixed amount. For our experiments we use a relatively small value of $\tau = 2^{30}$, an level which puts the attack time on the order of one minute on a common personal computer. In a real attack τ might be much higher.

[6] Note that since guesses are being made about the final 16 bytes of the expanded key schedule, the key expansion algorithm must be inverted, as previously mentioned this is an easy process because the key expansion is specifically designed to be invertible.

The attack has been simulated against the optimized C implementation of AES, as submitted in the original Rijndael proposal [DR02]. This implementation is used without significant change in many open source cryptographic libraries such as OpenSSL v 0.9.8(a), Crypto++5.2.1, and LibTomCrypt 1.09, libgcrypt v 1.2.2, and Botan v 1.4.2. We have assumed that it is possible to produce a precise cache trace through power analysis, for experiments we have simulated this process by augmenting the encryption code to keep a record of hits and misses based on the indices looked up during the encryption operation. We track for each cache line in the table T_4 whether or not it is in cache, loading it in on the first access to it.

6.2 Data

The effectiveness of the attack is dependent on the value of δ as defined in Section 3. Common cache-line sizes are 32 bytes for Pentium III and earlier, 64 bytes for Pentium IV and other recent processors. These give values for δ of 8, and 16 if a four-byte wide T_4 table is used. Some AES implementations implement T_4 as a one-byte wide table[7], this gives values for δ of 32, and 64 respectively. Thus, we have simulated the attack for values of δ ranging from 8-64 and we present results in Figure 1.

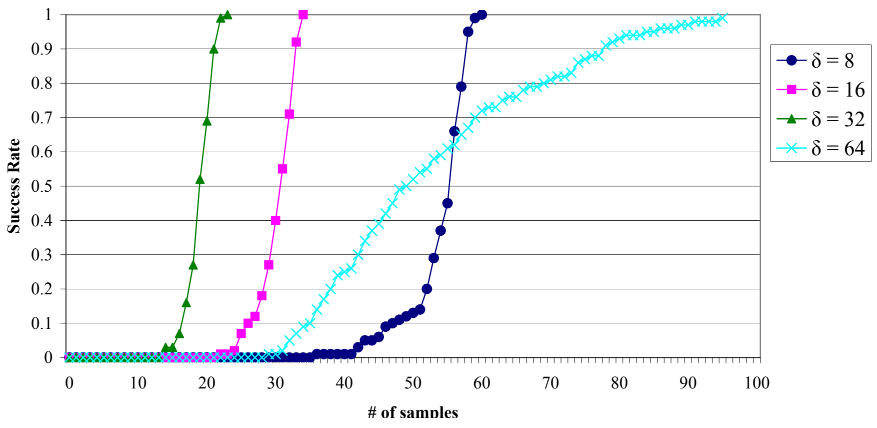

Fig. 1. Success rate of attack for common values of δ

[7] This is actually a more logical size. It is not clear why the original Rijndael code chose a four-byte wide table except for consistency with the other tables, nevertheless, this choice persists as an artifact in many implementations.

6.3 Analysis

It is interesting that the attack is less effective at high or low values of δ, specifically, the performance at $\delta = 16$ or 32 is best, and $\delta = 64$ is the worst. This can be understood because varying δ produces two simultaneous effects. Higher values of δ produce constraints more quickly, specifically each miss at position $1 \leq j \leq 15$ produces $j \cdot 256 \cdot \delta$ as outlined in Section 5.1 (a miss at position 0 is useless). However, at higher values of δ the number of misses is lower, especially at higher j, since the table is loaded into cache much more quickly. Specifically, since the table will occupy $\frac{256}{\delta}$ cache lines, the probability of a miss in position j can be estimated[8] at $(\frac{\frac{256}{\delta}-1}{\frac{256}{\delta}})^{j-1} = (1 - \frac{\delta}{256})^{j-1}$. This is problematic for high values of delta because it makes misses in the final cache accesses very unlikely, for example, with $\delta = 64$ the chance of the last cache access missing is just $0.75^{15} = 1.3\%$. Thus, each miss produces more constraints at higher value of δ, but the misses occur less frequently.

The results observed can be explained by a probabilistic model of the constraints produced by a set of cache traces. The attack algorithm takes a set of N cache traces and uses it to produce constraints by eliminating impossible values for pairs of key bytes, and then attempting to search for the key using those constraints. The success rate of the attack given a time limit τ should be well predicted by the number of constraints produced. For any cache trace, a miss occurs at position j with probability $(1-\frac{\delta}{256})^j$ and produces $j \cdot 256 \cdot \delta$ constraints, giving an expected value for the number of constraints as:

$$E(\#constraints) = \sum_{j=1}^{15} (1 - \frac{\delta}{256})^j \cdot j \cdot 256 \cdot \delta$$

δ	8	16	32	64
$E(\#constraints)$	178,976	263,303	292,626	204,235

This gives a rough explanation for the observed data, as samples are the most useful in terms of constraint production at $\delta = 32$. However, the constraints produced are not equally distributed but instead biased towards the earlier key bytes, because the earlier cache accesses are more likely to be misses. This bias is different for different values of δ, however, as seen in Figure 2, higher values of δ are more biased towards constraining the earlier key bytes. This bias is actually beneficial to the search, as values are guessed for the lower key bytes first. Since there are more constraints on these bytes, the search tree is pruned more quickly by assigning to these bytes, this is an application of the common "most constraining variable" heuristic used in constraint satisfaction [RN95].

[8] We model the accesses as independent random variables, which is reasonable because the output ciphertext should appear random if the cipher is secure.

Due to this effect, the higher values of δ perform better than the naive estimate at constraint generation would predict.

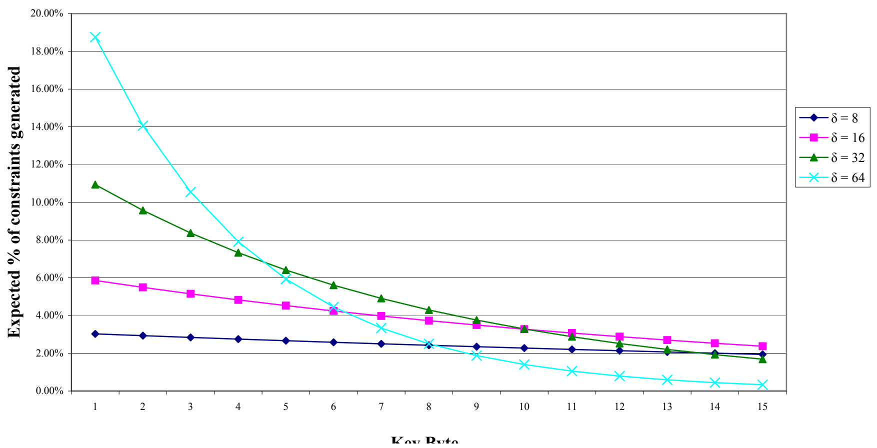

Fig. 2. Bias towards constraints on lower key bytes

The relatively long "tail" in the success rate for $\delta = 64$ in Figure 1 can also be explained by the random model. At $\delta = 64$, it is expected that a lower number of misses will be observed in the available cache traces, but this will compensated by the fact that each miss is more useful. For the higher key bytes the chance of a miss is very low at $\delta = 64$. There is a binomial distribution of the expected number of misses at each position. For $\delta = 64$, since the expected number of misses is very low in the later positions, the number of samples needed is much more variable. The absence of a few misses will greatly reduce the number of constraints produced. In contrast, at lower δ the number of misses expected is high, so individual misses are less important and the success rate spikes quickly.

7 Robustness to a Partially pre-loaded cache

The above experiments and random model are based on the cache being clean prior to encryption, although this assumption is not explicitly utilized. To simulate the effects of some table entries being pre-loaded into the cache, prior to the encryption each table line can be marked as being loaded into the cache with a probability σ. As a baseline, we set $\sigma = 0$ to simulate a totally clean cache, this is the best case scenario for the attack. We simulated the effects of a partially pre-loaded cache by varying the value of σ used in the experiments. The results were very similar to the above results, albeit requiring $\frac{1}{1-\sigma}$ times more samples to achieve the same success rates. The reason for this is that the production of constraints is decreased by $\frac{1}{1-\sigma}$. For any encryption operation in which some lookup index l_i does not collide with any of the previous lookups in the encryption, with probability σ the line accessed by l_i was randomly pre-loaded into the cache. In this case, the access is recorded as a false hit and its information is lost to the attack. Thus, constraints are only retained with probability $1 - \sigma$, so $\frac{1}{1-\sigma}$ times more traces must studied to obtain the same number of constraints as the optimal case where $\sigma = 0$.

This analysis indicates that the number of samples required is infinite as $\sigma \rightarrow 1$. This is an obvious condition of the attack model, if every line of the table is consistently in memory prior to encryption, the cache trace will be nothing but hits (many of them false hits) and no useful information is gained. Thus the attack, while not assuming a clean cache, will still fail if unless some portion of the table is not pre-loaded. This suggests that pre-loading the cache prior to encryption should be an effective countermeasure.

In order to try and get some of the table lines out of cache prior to encryption, an attacker may be able to run specific code on the machine[9], but more likely will rely on code running in between encryptions to read in other data which evicts the cache lines used by T_4. In practice, this means it is unlikely that a different, random portion of T_4 will be in cache prior to each encryption. It is far more likely that some certain subset of T_4 is being evicted by the code running in between encryptions, and that the same lines of T_4 tend to get knocked out over and over rather than different random lines. However, this is a problem because of the randomness produced by the cipher. The worst possible case is if only one line is out of memory prior to encryption, and it is the same line before every encryption. Experimentally, the attack still succeeds in this situation, and the performance numbers are indistinguishable from one random line being evicted from cache. This indicates that even if only one line of cache is invalidated between encryptions, the attack will still succeed. This scenario puts σ at $\frac{\delta-1}{\delta}$, meaning that the total number of samples required increases by $\frac{1}{1-\frac{\delta-1}{\delta}} = \delta$, which not a serious impediment to the attack for δ in the range 8-64.

8 Utilizing a Clean Cache Assumption

The attack can be strengthened if it is assumed that the cache is clean prior to encryption, because information is then revealed by cache hits as well as cache misses, this is then a very similar attack to the final round attack of Acıiçmez and Koç. Specifically, if we assume a clean cache and have a cache trace where l_j is a hit, we know that the cache line accessed by l_j must be equal to a cache line already accessed, that is, it must be that $\langle l_i \rangle = \langle l_j \rangle$ for some $0 \leq i < j$. These constraints are powerful because they indicate only a small number of possible values for l_j. Unfortunately, these constraints are not as easy to work with because they are not binary constraints as before, they are disjunctions potentially involving all 16 unknown key bytes. If l_j is a hit, it can be inferred that $\langle l_j \rangle = \langle l_0 \rangle \vee \langle l_j \rangle = \langle l_1 \rangle \vee \cdots \vee \langle l_j \rangle = \langle l_{j-1} \rangle$.

[9] Such methods are described by [OST06] and also [BM06]

For the case of l_1, a hit does produce a binary constraint with l_0, namely it directly indicates that $\langle l_0 \rangle = \langle l_1 \rangle$. This case can be added to the set of binary constraints as discussed in Section 5.1.

For other hits, they do not produce binary constraints and must be treated differently. Because a hit at lookup l_j produces a constraint which only relates to indices l_i with $i < j$, these constraints can be considered when guessing a new value for k_j^{10} in the search, as described in Section 5.3. Because a guess will have already been made for the bytes $k_0^{10}, k_1^{10}, \ldots, k_{j-1}^{10}$, the constraints based on hits can be used to avoid making a guess for l_j which is impossible.

Suppose the search algorithm is prepared to guess that $k_j^{10} = z$. For each cache trace where l_j is a hit, the value $l_j^z = T_4^{-1}[c_j \oplus z]$ is computed, this is the lookup l_j that would have occurred if the guess z is correct. Next, the set of previous lookups in that trace $l_0, l_1, \ldots, l_{j-1}$ can be computed since guesses have already been made for the key bytes $k_0^{10}, k_1^{10}, \ldots, k_{j-1}^{10}$. Specifically, if all of the key guesses were correct than each previous lookup l_i can be calculated as $T_4^{-1}[c_j \oplus k_i^{10}]$. If the guess z is correct, then it must be that $\langle l_j^z \rangle = \langle l_i \rangle$ for some previous l_i, so z is checked against each previous value. If no match is found, than the guess z is incorrect and another guess must be tried.

In this way, the constraints can be used to identify incorrect guesses earlier than would otherwise be possible, pruning significant portions of the search tree. This enables the attack to succeed faster or with fewer samples under a constant amount of search. To quantify this effect, the above approach was implemented and run in experiments similar to those for the attack without using hit information, with the same constant upper bound in the amount of search performed. This improves the success rate, as can be seen by comparing the experimental results in Figure 3 to those in Figure 1, and as summarized in Table 8.

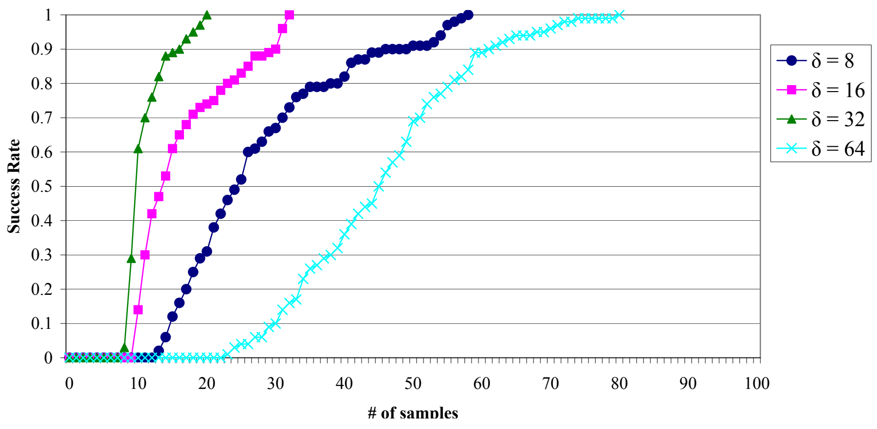

Fig. 3. Success rate of attack for common values of δ, with clean-cache assumption

	$\delta = 8$	$\delta = 16$	$\delta = 32$	$\delta = 64$
Med. samples, no cache assumption	56	31	19	49
Med. samples, clean cache assumption	25	14	10	45

Table 1. Comparison of performance with or without a clean-cache assumption

It is also useful to compare the above results with the analysis provided by Acıiçmez and Koç of their own attacks. This data appears to fit in well with their calculations, confirming both the accuracy of their model and the efficiency of this formulation of the attack.

9 Countermeasures

Defending AES against side channel attacks in general is a daunting task. For cache-trace attack based on power analysis, the best defense would be hardware-based obfuscation of the encryption's software's cache accesses. This could take the form of either designing a microprocessor to use a consistent amount of power regardless of the operation it is performing, or at the minimum reducing the power difference between cache hits and misses. Kocher concedes that it is probably impractical to ever build a semiconductor-based processor which did not leak information through its power usage [KJJ99]. It should be possible to insert noise in the power usage graph by randomly using additional power. This would probably be unacceptable in practice for many platforms for which minimizing power usage is crucial, such as laptop computers and other portable devices.

It has been suggested frequently that the encryption process obscure its cache accesses by either adding random lookups or preloading all elements of the table [Ber05,OST06,BM06,AK06,BBM$^+$06,BGNS06]. Pre-loading the table should eliminate cache trace attacks as the cache trace would be all hits for the actual encryption operation. This approach should slow down encryption by $30 - 50\%$, which may make it impractical for widespread use.

Randomly performing cache lookups during the final round in between the necessary lookups could also be effective against this attack if done properly. It is important to note that if the random lookups occur at deterministic positions in the lookup sequence, the attacker can simply ignore them, and the only effect will be to decrease the number of misses and slow down the attack. It is not sufficient that the addresses looked up are random, the lookups themselves must occur at random points in the sequence so that an attacker cannot tell if a cache miss in the sequence is a randomly added lookup or one of the necessary lookups. Another approach would be to add no extra lookups, but randomly permute the order lookups are performed within a round. Although it would come at a performance cost, Brickell et al. astutely note that it is probably sufficient to only protect the first and last rounds of encryption with such techniques, as the lookups performed in the middle rounds are much more difficult to correlate with ciphertext or plaintext data [BGNS06].

Another method, as discussed in [BM06], is to eliminate the use of a special T_4 table and instead re-use the previous four tables, each of which completely contains the S-box values. This is an attractive defense because it comes at no performance cost and would greatly reduce the number of constraints produced by the cache trace attack. Unfortunately, this protection is not be possible for the decryption routines since the inverse S-box table is not contained in any of the decryption tables.

10 Conclusions

This paper has presented an approach to attacking AES given a small set of cache traces by reduction to a simple constraint satisfaction problem. By not assuming a clean cache, the attack is more robust than previous attacks. The attack also demonstrates that such an attack is practical with a low number of samples and requires computations which are easily performed on a desktop computer.

If an attacker is able to gain access to the power consumption profile of a target machine, presumably by physical access and inserting a monitoring device, than this attack could be used for many malicious purposes. One example would be hijacking the AES key used to encrypt data in an SSL session. This would be likely if a target machine is encrypting a small number of packets, while doing some unrelated work in between to evict some AES tables from memory. Another possibility would be to attempt to recover the encryption key used in a target's encrypted file system, if the target is occasionally encrypting disk blocks.

A machine's power consumption data is a high amount of information to give to an attacker, but the fact it enables a relatively strong attack against AES further illustrates that cached memory is fundamentally insecure and should be avoided by future cryptographic primitives.

References

[ABDM00] Mehdi-Laurent Akkar, Régis Bevan, Paul Dischamp, and Didier Moyart. Power analysis, what is now possible.... In *Advances in Cryptology—ASIACRYPT 2000*, pages 489–502, 2000.

[AK06] Onur Acıiçmez and Çetin Kaya Koç Trace Driven Cache Attack on AES. IACR Cryptology ePrint Archive, April 2006.

[BAK98] Eli Biham, Ross J. Anderson, and Lars R. Knudsen. Serpent: A new block cipher proposal. In *Fast Software Encryption '98*, pages 222–238, 1998.

[BB05] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[BBM$^+$06] Guido Bertoni, Luca Breveglieri, Matteo Monchiero, Gianluca Palermo, and Vittorio Zaccaria. AES Power Attack Based on Induced Cache Miss and Countermeasure. International Conference on Information Technology: Coding and Computing - ITCC05, IEEE Computer Society, 2005.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES. April 2005. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[BM06] Joseph Bonneau and Ilya Mironov. Cache-collision Timing Attacks Against AES. April 2006. *Workshop on Cryptographic Hardware and Embedded Systems 2006.*

[BGNS06] Ernie Brickell and Gary Graunke and Michael Neve and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. IACR ePrint Archive, Report 2006/052, Feb 2006.

[DR99] Joan Daemen and Vincent Rijmen. Resistance against implementation attacks: A comparative study of the AES proposals. *Second AES Candidate Conference*, February 1999.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES—the advanced encryption standard*. Springer-Verlag, 2002.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems—CHES 2001*, pages 251–261, 2001.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology—CRYPTO '99*, pages 388–397, 1999.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO '96*, pages 104–113, 1996.

[KQ99] F. Koeune and J.-J. Quisquater. A timing attack against Rijndael. Technical Report CG-1999/1, June 1999.

[KSWH00] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. *Journal of Computer Security*, 8(2/3), 2000.

[Lau05] Cedric Laradoux. Collision attacks on processors with cache and countermeasures. Western European Workshop on Research in Cryptology—WEWoRC'05, C. Wolf, S. Lucks, and P.-W. Yau (editors), pp. 76–85, 2005.

[Mac77] Alan Mackworth Consistency in networks of relations *Artificial Intelligence 8*, 1977.

[Mat97] Mitsuru Matsui. New block encryption algorithm MISTY. In *Fast Software Encryption '97*, pages 54–68, 1997.

[NBB$^+$00] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback. Report on the development of the Advanced Encryption Standard (AES). October 2000. http://csrc.nist.gov/CryptoToolkit/aes/round2/r2report.pdf.

[NSW06] Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. A refined look at Bernstein's AES side-channel analysis. *ASIACCS*, p. 369, 2006.

[NS06] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *SAC'06*, to appear.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *CT-RSA*, pages 1–20, 2006.

[Pag02] Daniel Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical Report CSTR-02-003, University of Bristol, April 2002.

[Pag03] Daniel Page. Defending against cache based side channel attacks. Technical Report. Department of Computer Science, University of Bristol, 2003.

[Pag05] Daniel Page. Partitioned cache as a side-channel defense mechanism. IACR Cryptology ePrint Archive, Report 2005/280, August 2005.

[Per05] Colin Percival. Cache missing for fun and profit. Presented at BSDCan '05, 2005. http://www.daemonology.net/hyperthreading-considered-harmful/.

[RN95] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 1995.

[TSS+03] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *Cryptographic Hardware and Embedded Systems—CHES 2003*, pages 62–76, 2003.

[TTMM02] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Miyauchi. Cryptanalysis of block ciphers implemented on computers with cache. In *International Symposium on Information Theory and Applications 2002*, pages 803–806, 2002.