Cache-Collision Timing Attacks Against AES

Joseph Bonneau[1] and Ilya Mironov[2]

[1] Computer Science Department, Stanford University jbonneau@stanford.edu
[2] Microsoft Research, Mountain View, CA mironov@microsoft.com

Abstract. This paper describes several novel timing attacks against the common table-driven software implementation of the AES cipher. We define a general attack strategy using a simplified model of the cache to predict timing variation due to cache-collisions in the sequence of lookups performed by the encryption. The attacks presented should be applicable to most high-speed software AES implementations and computing platforms, we have implemented them against OpenSSL v. 0.9.8.(a) running on Pentium III, Pentium IV Xeon, and UltraSPARC III+ machines. The most powerful attack has been shown under optimal conditions to reliably recover a full 128-bit AES key with 2^{13} timing samples, an improvement of almost four orders of magnitude over the best previously published attacks of this type [Ber05]. While the task of defending AES against all timing attacks is challenging, a small patch can significantly reduce the vulnerability to these specific attacks with no performance penalty.

Keywords: AES, cryptanalysis, side-channel attack, timing attack, 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 physical implementation. The attacks presented in this paper use timing data.

In 1997, Rijmen and Daemen proposed the Rijndael cipher to the National Institute of Standards and Technology (NIST) as a candidate to become 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 "not vulnerable to timing attacks" and regarded Rijndael as the easiest among the finalists to defend against side-channel attacks.

In contrast to NIST's predictions, a number of side channel attacks have already been demonstrated against AES, including timing attacks by Bernstein [Ber05] and Tsunoo et al. [TSS$^+$03]. This paper introduces a new model for attacking AES by using the timing effects of cache-collisions to gather noisy information about the likelihood of relations between key bytes. This leads to a multivariate optimization problem, where the unknown key is an optimal value of a certain objective function. We solve for the key using a variety of AI methods, including belief propagation and iterated local search, as discussed in Appendix D. We also deviate from previous work in attacking the final round of encryption instead of the first round. Table 1 demonstrates the improvements of the attacks in this paper over previous attacks.

Table 1. Overview of timing attacks against AES.

Attack	Samples needed	Sample type	Goal
Bernstein [Ber05]	$2^{27.5}$	Plaintext/timing	Full key recovery
Tsunoo et al. [TSS+03]	2^{26}	Plaintext/timing	Full key recovery
First round attack	$2^{14.58}$	Plaintext/timing	60 key bits recovered
Final round attack	2^{15}	Ciphertext/timing	Full key recovery
Expanded Final round attack	2^{13}	Ciphertext/timing	Full key recovery

The AES cipher is described in Section 2, and previous attacks are discussed in Section 3. Our attack model is introduced in Section 4, and novel attacks based on this tactic are detailed in Sections 5, 6, 7, 8. Possible countermeasures are suggested in Section 9. Finally, Section 10 concludes the paper by discussing possible real-world attack scenarios and the need to limit timing vulnerabilities.

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 were utilized in this study.[3] 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}. Each round is carried out by performing the algebraic operations `SubBytes`, `ShiftRows`, and `MixColumns` on X^i, then taking the exclusive-or with the round key K^i. Performance-oriented software implementations of AES combine all three operations and pre-compute the values. 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. Each round is carried out by splitting up X^i into 16 bytes $x_0^i, x_1^i, \ldots, x_{15}^i$, and K^i into

[3] 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.

16 bytes $k_0^i, k_1^i, \ldots, k_{15}^i$. 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 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 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.

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 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.

This formulation was a part of the original Rijndael proposal [DR02]. The attacks in this paper are widely applicable as many AES implementations have made no significant changes to the original optimized Rijndael code In addition to OpenSSL v. 0.9.8.(a), which was used in our experiments, the AES implementations of Crypto++ 5.2.1 and LibTomCrypt 1.09 use the original Rijndael C implementation with very few changes and are highly vulnerable. The AES implementations in libgcrypt v. 1.2.2 and Botan v. 1.4.2 are also vulnerable, but use a smaller byte-wide final table which lessens the effectiveness of the attacks.

3 Related Work

Side-channel attacks have been demonstrated against implementations of many cryptosystems, utilizing timing [Ber05,TSS$^+$03,Koc96,BB05], power consumption [ABDM00,KJJ99], electromagnetic radiation [GMO01], etc. Public key algorithms have proved the most vulnerable to timing attacks because they typically perform lengthy mathematical operations, the running time of which depends directly on the data due to branch statements. Kocher demonstrated timing attacks against a variety of software public-key systems in 1996 [Koc96]. Brumley and Boneh demonstrated more advanced timing attacks against RSA in 2003 which were effective even against a remote SSL server [BB05], these attacks were improved by another order of magnitude in 2005 [ASK05].

A similar timing attack was demonstrated against the reference AES implementation which uses branch statements to perform multiplication in the field GF(2^8) [KQ99]. However, as noted above, performance AES implementations pre-compute this calculation, obviating this attack. During the AES selection process, it was believed that timing attacks were only applicable to software with a data-dependent execution path (i.e., branch statements, data-dependent shifts), although Kocher did suggest that timing attacks could be constructed against symmetric ciphers by studying "cache hit ratio" [Koc96], a conclusion also reached by Kelsey et al. [KSWH00]. Nevertheless, in an analysis of AES finalists done by Daemen and Rijmen, Rijndael was deemed a "favorable" candidate to secure against timing attacks, since it did not use branch instructions or data-dependent rotations [DR99]. Even by the final NIST evaluation [NBB$^+$00], it was not recognized that table lookups could lead to timing attacks due to the effects of cached memory and AES was considered to be safe.

The use of table lookups into cached memory has recently been recognized as an exploitable cryptographic side-channel [Pag02]. Recent attacks due to Osvik, Shamir, and Tromer demonstrate how specific information about what values in cached memory the encryption algorithm has accessed can quickly leak enough information to reconstruct an AES key [OST06]. 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 $k_0 = z \oplus z'$. These attacks are different from timing attacks because they require that the attacker gain direct knowledge about cache access patterns,[4] thus they are directly using cache accesses as a cryptographic side-channel instead of timing.

Another class of cache attacks focuses on the use of power consumption to detect whether lookups performed during AES encryption resulted in hits or misses. This technique was first demonstrated in [BBM$^+$06]. An attack using power analysis of the first round was also described by Lauradoux [Lau05]. Acıiçmez and Koç [AK06] extended this approach by considering the first two rounds of AES. Their attack requires a very low (~ 50) number of encryptions, but require physical access to a machine's power supply.

Cache access patterns also cause timing variation, which can be used to construct a timing attack against AES software without direct observation of the cache accesses. This principle was first demonstrated by Tsunoo et al. [TTMM02,TSS$^+$03] who demonstrated timing attacks against DES and MISTY. Tsunoo et al, assuming that cache hit ratio should be correlated with encryption time, collect a number of plaintexts with unusually long encryption times. These plaintexts are then used to infer information about key bytes by inferring that the correct key should be one that leads to the lowest cache-hit ratio when used with the set of "slow" plaintexts. While the authors focus on attacking DES, the possibility of an attack on AES is briefly mentioned in [TSS$^+$03]. Unfortunately, insufficient detail is provided to reproduce the attack, although a figure of 2^{18} plaintexts

[4] As implemented in [OST06], knowledge of cache accesses is gained by running attack code on the target computer before and after the encryption operation.

with long encryption times is presented for the attack. Assuming consistency with the attacks on DES, this means a total of 2^{24} plaintexts are needed.

Our approach is similar to that of Tsunoo et al. in utilizing the correlation between cache hits and encryption time. However, our attacks focus on individual cache-collisions during encryption, instead of overall hit ratio. Furthermore, we use the entire data set, instead of simply plaintexts resulting in long encryption, and we consider conditions which lead to a shorter encryption time, instead of a longer one. Our methods is similar to that used in recent attacks independently described by Acıiçmez [Acıi05], although our attacks differ in focusing on the final round of encryption as opposed to the first round.

Bernstein demonstrated a different type of timing attack against AES in 2005 [Ber05] which can be thought of as a *statistical* timing attack. Bernstein observed that since the input bytes to the first round of encryption are simply the bytes $x_i^0 = p_i \oplus k_i$, and these bytes are immediately used as indices into the lookup tables, the entire encryption time t can be affected by each of the values x_i^0. To carry out Bernstein's attack, first a large volume of timing data is collected for each value of an input byte x_i^0 using a reference machine, this data is then correlated with data from the target machine to recover the key.

Bernstein's attack is a generic attack because it does not utilize any specific knowledge of why the value of a specific x_i^0 affects the encryption time, only the empirical observation that certain values do cause time variation. This approach is widely applicable because, as Bernstein details, it is extremely difficult to achieve fast constant-time software, and any timing variation could potentially be exploitable. The statistical attack method can even be extended [CLS06] to exploit timing variation of individual bits of the key instead of whole bytes.

The first downside of the statistical approach is that it requires a large number of samples, approximately $2^{27.5}$ in Bernstein's experiments. More critically, the attack is very fragile because relies on subtle machine-specific cache effects, requiring that the attacker recreate the target platform exactly. In our own experiments with Bernstein's attack code, we found even small changes to the mix of background processes from the target machine to the reference machine were enough to make the attack fail, raising serious doubts on the practicality of the attack. Similar difficulty in reproducing the attacks was reported in [OST06] and [OT05]. A recent analysis by Neve et al.[NSW05] discusses the reasons the attack succeeds in some cases and why it is probably not practical.

In contrast, this paper focuses exclusively on white-box timing attacks, which use expected timing effects due to the structure of the cipher. This approach requires no specific information about the target platform, and is likely to require far fewer samples if encryption software lends itself to simple and predictable timing effects, as AES does.

4 Attack Model and Strategy

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 situated between the CPU and main memory. When values are looked up in main memory, they are stored in the cache, 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.

Complicating matters is the fact that modern caches do not store individual bytes, but groups of bytes from consecutive "lines" of main memory. Line size varies between 32 bytes for a Pentium III and 64 or 128 bytes on more recent Pentium IV or AMD Athlon 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]). So, for any bytes l, l' which are equal ignoring the lower $\log_2 \delta$ bits (notated as $\langle l \rangle = \langle l' \rangle$ in [OST06]), looking up address l will cause an ensuing access to l' to hit in cache.

We view an AES encryption as a sequence of 160 table lookups to indices $l_1, l_2, ..., l_{160}$. A "cache collision" occurs if two separate lookups l_i, l_j satisfy $\langle l_i \rangle = \langle l_j \rangle$. In this situation, l_j should always hit in the cache.[5] If it were the case that $\langle l_i \rangle \neq \langle l_j \rangle$, then the access to l_j may result in a cache miss if $T[l_j]$ was out of memory prior to the encryption and no previous access fetched it. This should, on the average, take more time as it will require a second cache lookup with non-zero probability. We formalize this assumption:

Cache-Collision Assumption. *For any pair of lookups i, j, given a large number of random AES encryptions with the same key, the average time when $\langle l_i \rangle = \langle l_j \rangle$ will be less than the average time when $\langle l_i \rangle \neq \langle l_j \rangle$.*

This assumption rests on the approximation that the individual table lookups in the sequence are effectively independent for random plaintexts, which seems to hold in practice.[6] This assumption greatly oversimplifies many the intricacies of modern caches, as discussed in Appendix B and Appendix C, but is well supported by experimental data as shown in Figure 1. Notice that there is a clear correlation, especially for ≤ 10 collisions, which is where 90% of the data lies. We fit the experimental data with a linear model where the unknowns are defined as bonuses due to collisions between table lookups in the final round, a total of 120 variables. Depending on the mix of the processes running in the background the model explains between 13% and 28% of the variance in the timing data (the results are supported by five-fold cross-validation).

The notion of using collisions in the cache is by no means unique to this paper. Because caches are specifically designed to behave differently in the presence of a collision a non-collision, they are a natural side channel for attacking AES. This general notion has been used in several other attacks on AES

[5] We are assuming that the AES encryption itself does not evict any table entries after loading them, a reasonable assumption given the large size of modern caches compared to the AES tables.

[6] This will not hold for the first round if plaintexts are not random. This should hold for the final round regardless of plaintext, since the output ciphertext should be statistically random in any secure cipher.

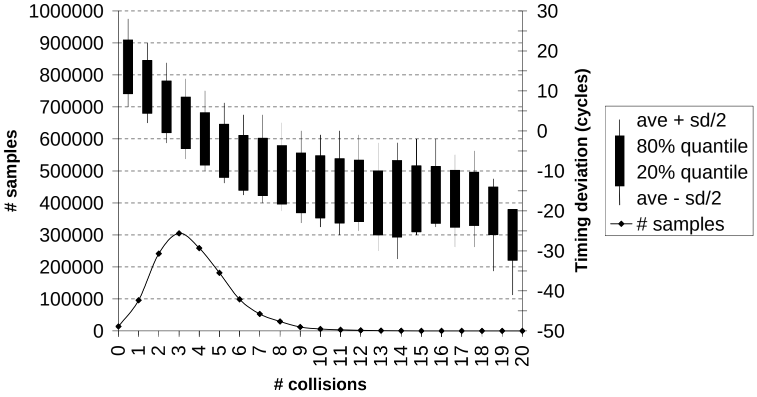

Fig. 1. Time deviation vs number of final round cache-collisions, Pentium III

[OST06,Lau05,Acii05,TSS$^+$03,TTMM02,Pag02], we seek to explicitly define the utility of cache collisions as they apply to timing attacks.

5 First Round Attack

A natural approach to attacking AES is to analyze table lookups performed in the first round, because they use the indices $x_i^0 = p_i \oplus k_i$, each of which depends on only one key byte and one plaintext byte. In equation (1), we can see that in the first round of encryption, the bytes $x_0^0, x_4^0, x_8^0, x_{12}^0$ are each used as an index into table T_0; they make up a "family" of four bytes in that they are all used to access the same table. There are three other families of bytes which share the tables T_1, T_2, and T_3 in round one. A cache collision occurs whenever two bytes x_i^0, x_j^0 in the same family satisfy $\langle x_i^0 \rangle = \langle x_j^0 \rangle$. This should occur when $\langle p_i \rangle \oplus \langle k_i \rangle = \langle p_j \rangle \oplus \langle k_j \rangle$, or after rearranging, $\langle p_i \rangle \oplus \langle p_j \rangle = \langle k_i \rangle \oplus \langle k_j \rangle$.

Plaintexts satisfying $\langle p_i \rangle \oplus \langle p_j \rangle = \langle k_i \rangle \oplus \langle k_j \rangle$ for a pair of bytes i, j should have a lower average encryption time due to the collision. The first round attack algorithm compiles timing data into a table $t[i, j, \langle p_i \rangle \oplus \langle p_j \rangle]$ of average encryption times for all i, j in the same table family. If a low average time occurs at $t[i, j, \varDelta]$, the algorithm estimates that $\langle k_i \rangle \oplus \langle k_j \rangle = \varDelta$. A t-test is used to identify values which are lower than the mean to a statistically significant degree. For each table family, the attacker will eventually have a redundant set of six equations, such as $\langle k_0 \rangle \oplus \langle k_4 \rangle = \varDelta_1$, $\langle k_0 \rangle \oplus \langle k_8 \rangle = \varDelta_2$, $\langle k_0 \rangle \oplus \langle k_{12} \rangle = \varDelta_3$, $\langle k_4 \rangle \oplus \langle k_8 \rangle = \varDelta_4$, $\langle k_4 \rangle \oplus \langle k_8 \rangle = \varDelta_5$, $\langle k_8 \rangle \oplus \langle k_{12} \rangle = \varDelta_6$ for table T_0.

The four sets of equations for key bytes within the same family are the only information gained by this attack; there is no way to gain exact key information.[7] Furthermore, there is no way to learn the lower $\log_2 \delta$ bits of each key byte. The attacker must still guess a value for one complete byte in each table family, plus

[7] Extending the attack two look at two rounds may fix this problem and produce a full-key recovery attack. See Section 8.3.

the low-order $\log_2 \delta$ bits of the other bytes, or a total of $4 \times (8 + 3 \cdot \log_2 \delta) = 68$ bits (for $\delta = 8$), which is impractical to search for almost any real attacker.[8] The attack does provides a significant speedup over previous attacks, in experiments with 50 random keys on a Pentium III the attack succeeded with an average of $2^{14.6}$ timing samples.

6 Final Round Attack

To design a fast attack which can recover the full key, we consider the final round of encryption. As noted previously, the final round of AES omits the `MixColumns` operation, reducing equation (1) to simply:

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

In this equation, C is the 16-byte output ciphertext, and T_4 is the AES S-box. The details of the S-box are inconsequential to this attack, the only important fact is that the S-box is a non-linear permutation over all 256 possible byte values. For any two ciphertext bytes c_i, c_j, it holds that $c_i = k_i^{10} \oplus T_4[x_u^{10}]$ for some u and $c_j = k_j^{10} \oplus T_4[x_w^{10}]$ for some w. Regardless of the actual values of u and w, whenever $x_u^{10} = x_w^{10}$, a cache collision occurs on T_4. Suppose $x_u^{10} = x_w^{10}$, and $T_4[x_u^{10}] = T_4[x_w^{10}] = \alpha$. Then it will hold that $c_i = k_i^{10} \oplus \alpha$ and $c_j = k_j^{10} \oplus \alpha$.

If, on the other hand, $c_i \oplus c_j \neq k_i^{10} \oplus k_j^{10}$, two different values α, β must have resulted from the table lookups. It would be true that $\alpha \oplus \beta = \gamma = c_i \oplus c_j \oplus k_i^{10} \oplus k_j^{10}$ with γ a constant for a fixed value of $c_i \oplus c_j$. Since α and β are the direct results of S-box lookups, though, a fixed differential γ does not guarantee a fixed offset of the lookup indexes used to produce them. Ironically, the non-linearity which is the raison d'être of the S-box also enables this attack to succeed. For the purposes of this attack, given values of α and β satisfying $\alpha \oplus \beta = \gamma \neq 0$, the indexes which were looked up in the S-box to produce α and β are essentially random. So, if $c_i \oplus c_j = k_i^{10} \oplus k_j^{10}$, then a cache collision occurs in T_4, otherwise, the lookups will be from two essentially random locations in T_4.

The goal of the attack is to record timing data for random ciphertexts at each value of $\Delta = c_i \oplus c_j$. For each ciphertext/time pair observed, the encryption time is used to update a table of average times $t[i, j, \Delta]$ for all values i, j. The goal to find one value $\Delta'_{i,j}$ for each i, j such that $t[i, j, \Delta'_{i,j}] < \bar{t}$ where $\bar{t}$ is the average encryption time over all ciphertexts. Eventually, the values of $\Delta'_{i,j}$ will become accurate guesses for the true values $\Delta_{i,j} = k_i^{10} \oplus k_j^{10}$, which should be the only values which cause significantly low encryption times.

[8] The attack is less effective in the face of larger values of δ. For $\delta = 16$, the attack can only reduce the search space to 2^{80}.

These values can be used by an attacker to construct a guess at the final 16 bytes of the expanded key in the presence of noise, as described in Appendix D. The authors of Rijndael made it a specific design goal to enable recovery of the entire key given any 16 consecutive bytes of the expanded key [DR02]. Thus, it is simple to revert the key expansion algorithm to recover the raw key K given the final 16 bytes $k_0^{10}, k_1^{10}, \ldots, k_{15}^{10}$ of the expanded key. For each guess at the final key bytes, the attack program reverts the key and checks it against one known plaintext/ciphertext pair. Table 2 presents statistical data for the number of (C, t) pairs seen before the attack recovers a full 128-bit AES key, from attacks against 50 random keys.

Table 2. Median samples required, Final round Attack.

CPU	L1 cache eviction	L2 cache eviction
Pentium III 1.0 GHz	2^{16}	2^{15}
Pentium IV Xeon 3.2 GHz	$2^{19.9}$	2^{16}
UltraSPARC-III+ 0.9 GHz	$2^{18.7}$	2^{15}

7 Expanded Final Round Attack

One problem with the simple final round attack is that it considers only cache collisions due to lookups on the same table index. When the number of table entries per cache line δ is 8 or 16, however, the majority of cache collisions will not be on the same index but on two different indices of the same cache line. To take advantage of all cache collisions, we consider all conditions for which there will be a cache collision in the final round on two bytes i, j. Recall that $c_i = k_i^{10} \oplus S[x_u^{10}]$ and $c_j = k_j^{10} \oplus S[x_v^{10}]$ for some u, v. A collision will occur whenever $\langle x_u^{10} \rangle = \langle x_v^{10} \rangle$, or equivalently:

$$\langle S^{-1}[c_i \oplus k_i] \rangle = \langle S^{-1}[c_j \oplus k_j] \rangle. \tag{3}$$

An attacker can utilize this relationship by guessing exact values (k'_i, k'_j) for each i, j, instead of guessing only a differential. For each guess, an average time is computed for all timing samples which satisfy equation (3) under the guessed key bytes. The correct value will eventually have a lower average time due to the cache collision. The memory and time requirements for analyzing timing data are higher in this attack because there are $256 \cdot 256 = 65,536$ possible guesses for each pair of bytes. However, the data processing can be done off-line by the attacker after the data is collected. In practice, an attacker will want to reduce the amount of samples needed at the expense of increasing off-line processing time. Appendix D describes details of the attack algorithm.

This greatly speeds up the attack because the data collection rate is effectively increased by δ, since all cache-line collisions over any two bytes i, j are detected instead of exact byte collisions. That is, the proportion of random samples which satisfy equation (3) is $\frac{\delta}{256}$, instead of $\frac{1}{256}$ for the simpler version of the attack. Additionally, the data is more precise in that a guess can be made about the probability of the exact value of a pair of key bytes, instead of simply a differential. These factors combine to give the following performance numbers over 50 random keys:

Table 3. Median samples required, Expanded Final Round Attack.

CPU	L1 cache eviction	L2 cache eviction
Pentium III 1.0 GHz	2^{14}	2^{13}
Pentium IV Xeon 3.2 GHz	$2^{18.6}$	$2^{13.6}$
UltraSPARC-III+ 0.9 GHz	$2^{17.3}$	$2^{14.3}$

8 Attack Variants

8.1 Decryption Attacks

AES decryption in software is very similar to encryption, with a slightly different key schedule, and equally vulnerable to timing attacks. Both final round attacks are effective against decryption with only minor modifications.[9] In decrypt mode, the attacks require known plaintext instead of known ciphertext.

8.2 Chosen Plaintext/Ciphertext Attacks

In many cases, an attacker may be able to get encryption times for chosen plaintext and/or chosen ciphertext. This ability could potentially be used to create more powerful attacks based on hypothesis testing. Attacks of this type are an area for further research.

8.3 Two-Round Attack

The problem of the first round attack only recovering partial key information is a common problem in cache-based attacks due to the use of cache lines on modern processors, the standard solution is to consider the second round of cache accesses as well [OST06,Acıi05,AK06,NSW05] The lookup indices in the second round depend on at least four key bytes, but do not depend on the entire

[9] They are actually slightly simpler in that they recover information about the raw key, instead of the final bytes of the expanded key, so key reversion is not necessary.

key, so they can be studied. This approach can also be used to strengthen the cache-collision based first round attack to a full-key recovery attack.

The first round attack produces relations on bytes within the same table family. Unfortunately, the design of the cipher ensures that every second-round byte depends on at least one byte from all four table families. Consider the byte x_0^1, which is used as a lookup index on table T_0 in the second round:

$$x_0^1 = 2 \bullet S(p_0 \oplus k_0) \oplus 3 \bullet S(p_5 \oplus k_5) \oplus S(p_{10} \oplus k_{10}) \oplus S(p_{15} \oplus k_{15}) \oplus S(k_{13} \oplus \texttt{0x01}) \oplus k_0. \tag{4}$$

This is the simplest byte to attack, its value depends only on five key bytes. Furthermore, since the value of $k_5 \oplus k_{13}$ is learned from the first round attack, only the lower $\log_2 \delta$ bits must be guessed for k_{13} (This can be written as $\lfloor k_{13} \rfloor$). Suppose a guess is made for $\{k_0, k_5, k_{10}, k_{15}, \lfloor k_{13} \rfloor\}$. Then, given any plaintext P, the value of x_0^1 can be easily computed. The value of $x_0^0 = p_0 \oplus k_0$ is also known, and using the information from the first round attack, the values $\langle x_4^0 \rangle = p_4 \oplus k_0 \oplus \langle k_0 \oplus k_4 \rangle$, $\langle x_8^0 \rangle = p_8 \oplus k_0 \oplus \langle k_0 \oplus k_8 \rangle$, $\langle x_{12}^0 \rangle = p_{12} \oplus k_0 \oplus \langle k_0 \oplus k_{12} \rangle$ are also known, all of which were used as lookup indices into T_0. So, if $\langle x_0^1 \rangle$ is equal to any of $\langle x_0^0 \rangle$, $\langle x_4^0 \rangle$, $\langle x_8^0 \rangle$, $\langle x_{12}^0 \rangle$, a collision occurred during the encryption.

This collision predicate can be used to check a guess for $\{k_0, k_5, k_{10}, k_{15}, \lfloor k_{13} \rfloor\}$. The problem is that checking a guess requires scanning over all plaintext/time samples to compute an average for sample on which the predicate holds. Equation 4 is too complicated to be simply stored in a table like the other predicates used in this paper. Since there are $2^{32+\log_2 \delta}$ possible guesses (2^{35} for a Pentium III), the amount of offline processing required to check all guesses is $2^{32+\log_2 \delta} \cdot |D|$ where D is the set of timing samples. After the correct guess is identified, the remaining 33 bits of the key can then be exhaustively searched.

In testing this attack, we found that on a Pentium III, 2^{16} samples appears to be enough to clearly distinguish the correct $\{k_0, k_5, k_{10}, k_{15}, \lfloor k_{13} \rfloor\}$ from incorrect guesses. For example, in looking at the average times estimated for Equation 4 based on 100,000 incorrect guesses, the average time was 1018.92 with a standard deviation of 0.32. The correct guess produced an average time of 1015.57, which is 10.5 standard deviations below the mean. If the incorrect guesses' times are assumed to have a roughly normal distribution, this means there is a vanishingly small chance that any of the 2^{35} incorrect guesses would be as low as the correct one.

Still, there are 2^{35} guesses to check, each requiring a scan of 2^{16} samples, so the attack has complexity 2^{51}. While this computation is entirely offline and could feasibly be searched by some attackers (it is easily parallelizable), it makes the attack impractical in many situations. It is probable that a better two-round attack could be developed, this is an area for further research. We conjecture that a chosen-plaintext attack may greatly simplify the offline processing by allowing batch processing of plaintext/timing pairs.

9 Countermeasures

The strongest approach to defending against cache-based side-channel attacks, including timing attacks, is modifying hardware to limit the amount of data leaked by the cache. Hardware-based defenses are surveyed in [Ber05,OST06,Pag02,Pag05]. The most pertinent idea for defending against timing attacks is the addition of an assembly instruction to lock the entire AES table into the cache after pre-loading it. It seems unlikely, though, that any hardware-based solution will eliminate all data leakage through cache accesses.

A more realistic approach may be software designed to prevent the data that is leaked from being useful. Bernstein favors "constant-time" encryption software and offers a number of suggestions on achieving it in [Ber05], although this is probably impractical for performance reasons. Both [Ber05] and [OST06] mention the possibility of software attempting to obscure its timing data, either by adding random lookups or other instructions, or forcing all encryptions to take a fixed amount of time. The second approach is probably impractical due to performance reasons, and the first approach does not provide any firm security guarantee because any amount of added noise can be overcome with enough additional samples. Brickell et al. [BGNS06] describe several modifications to AES code to limit the vulnerability to cache-based attacks, including pre-loading tables, using smaller tables, and permuting the tables randomly on each encryption to prevent meaningful data from being leaked. One very practical idea is to carefully guard the first and last rounds, but leave the middle rounds unmodified for performance reasons. These modifications would probably defeat the attacks in this paper, but at a slowdown of nearly two times they may also be considered impractical.

We add to the discussion two simple techniques to defend specifically against the final round attacks presented in this paper by eliminating or shrinking the final round lookup table. Neither invokes any performance penalty. The final round computation to produce four bytes of output is performed as follows:

$$\begin{aligned} C_0 = (T_4[x_0^{10}] \mathbin{\&} \texttt{0xFF000000}) \oplus (T_4[x_5^{10}] \mathbin{\&} \texttt{0x00FF0000}) \oplus (T_4[x_{10}^{10}] \mathbin{\&} \texttt{0x0000FF00}) \oplus \\ (T_4[x_{15}^{10}] \mathbin{\&} \texttt{0x000000FF}) \oplus \{k_0^{10}, k_1^{10}, k_2^{10}, k_3^{10}\}. \end{aligned} \tag{5}$$

We have modified OpenSSL's AES software to make no use of the special table T_4, instead re-using the tables T_0, T_1, T_2, T_3 from previous rounds and careful byte manipulations. More precisely, we observe that due to the choice of the polynomial of the `MixColumns` step, the following is equivalent to equation (5):

$$\begin{aligned} C_0 = (T_3[x_0^{10}] \mathbin{\&} \texttt{0xFF000000}) \oplus (T_0[x_5^{10}] \mathbin{\&} \texttt{0x00FF0000}) \oplus (T_1[x_{10}^{10}] \mathbin{\&} \texttt{0x0000FF00}) \oplus \\ (T_2[x_{15}^{10}] \mathbin{\&} \texttt{0x000000FF}) \oplus \{k_0^{10}, k_1^{10}, k_2^{10}, k_3^{10}\}. \end{aligned}$$

This makes the code somewhat less elegant, but also introduces some of the problems of the first round attack into the final round attack, the output bytes are now broken up into four "families" which share a table, instead of all bytes effectively using the same table.

A second approach is to use a byte-wide T_4 table. There is no apparent reason why the original implementation uses a four byte-wide T_4 table, except possibly to make the code consistent with the code for the normal round.[10] The smaller table requires changing the output code to use shifts instead of bitwise-& operations:

$$C_0 = (T_4[x_0^{10}] << 24) \oplus (T_4[x_5^{10}] << 16) \oplus (T_4[x_{10}^{10}] << 8) \oplus (T_4[x_{15}^{10}]) \oplus \{k_0^{10}, k_1^{10}, k_2^{10}, k_3^{10}\}.$$

This fix increases δ by a factor of four, which makes the probability of a cache collision being detectable through timing data much lower. For example, with $\delta = 64$, the probability that a collision saves a cache miss over the entire encryption is $(1 - \frac{\delta}{256})^{14} = 0.018$. So, less than 2% of cache misses should be detectable, compared to about 40% for $\delta = 16$.

Table 4. Countermeasures Against Expanded Final Round Attack

AES implementation	Pentium III 1.0 GHz	Pentium IV Xeon 3.2 GHz
Original Rijndael Code	2^{13}	$2^{13.6}$
No Final Round Tables	$2^{20.2}$	$2^{24.7}$
Small Final Round Tables	$2^{19.7}$	2^{25}

We have verified experimentally that either modification can still be attacked, requiring little change to the expanded final round attack. However, as seen in Table 4, both modifications make the attacks require significantly more samples with no performance cost.[11] Implementers of AES, such as OpenSSL, should consider deploying one of these "free" defense mechanisms until true defenses are widely adopted.

10 Conclusions

Side-channel attacks were not given adequate treatment in the AES selection process. Rijndael, in optimized form, makes heavier use of lookup tables than any of the other four AES finalists, which exposes it to multiple side-channel attacks, including timing. By comparison, Serpent [BAK98], the AES runner-up, uses only tiny 4-bit by 4-bit S-boxes, which are in fact implemented only

[10] The AES implementations in both the libgcrypt v. 1.2.2 and Botan v. 1.4.2 cryptographic libraries use the original Rijndael code, modified to employ a byte-wide T_4 table.

[11] Both changes slightly *decrease* the average time to perform an encryption.

by logical operations, making Serpent invulnerable to cache-based side-channel attacks. At the time this was not recognized as an advantage, but it should be clear now that table lookups should be avoided or used with extreme caution in future cryptographic software.

The attacks described in this paper represent a significant step towards developing realistic *remote* timing attacks against AES, which are to make use of less accurate data than the processor cycle counts available in the simulated environment used in this study. There are a number of environments where such an attack could potentially be employed where direct observation of the pattern of cache accesses is not possible:

- On an encrypted network file system, an attacker which could time encryptions of single disk blocks and attempt to recover the encryption key.
- In a virtual machine environment, the virtual machine monitor could force cache flushes between context switches. An attacker could attempt to time another virtual machine performing encryptions.
- As recently proposed by Page [Pag05], a computer could partition cache between separate processes. User-level processes could not access the cache used by a root-level daemon process doing encryptions, but could time encryptions being done by that process.
- An SSL server (or client) could be a source of timing data to an attacker listening on the network. It is possible that both encryption and decryption could be observed in this setting.

In principle, the attacks in this paper could be employed in such scenarios, since they only require timing data and known plaintext or ciphertext. It remains to be seen if the timing data which could be obtained is accurate enough to attack, and there are additional complications as discussed in Appendix A. Nevertheless, the timing attacks in this paper should make clear the need for software AES implementations to protect against timing variation due to cached memory. While AES has resisted conventional cryptanalysis so far, it will be rendered useless if practical timing attacks are developed.

11 Acknowledgements

We are grateful to Dan Boneh for his encouragement of this research as well as many helpful comments, as well as Andrew Morrison and anonymous CHES 2006 reviewers for comments on drafts of this paper.

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, Report 2006/138, April 2006.

[Acıi05] Onur Acıiçmez. “Remote Timing Attacks”. Given at Intel Corporation, Oregon, USA, December 2005. Available at: http://web.engr.oregonstate.edu/ aciicmez/osutass/

[ASK05] Onur Acıiçmez, Werner Schindler, and Çetin Kaya Koç. Improving Brumley and Boneh timing attack on unprotected SSL implementations ACM Conference on Computer and Communications Security, 2005.

[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.

[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 Cryptology ePrint Archive, Report 2006/052, Feb 2006.

[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.

[CLS06] Anne Canteaut, Cedric Lauradoux, and Andre Seznec. Understanding Cache Attacks. Technical Report, April 2006. Available at: ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-5881.pdf

[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 2005, C. Wolf, S. Lucks, and P.-W. Yau (editors), pages 76-85, 2005.

[LMV04] Hervé Ledig, Frédéric Muller and Frédéric Valette. Enhancing Collision Attacks. In *Cryptographic Hardware and Embedded Systems—CHES 2004*, pages 176–190, 2004.

[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.

[NSW05] Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. A Refined Look at Bernstein's AES Side-Channel Analysis. *ASIACCS*, 2006.

[OT05] Mairead O'Hanlan and Anthony Tonge. Investigation of Cache Timing Attacks on AES. School of Computing, Dublin City University, 2005.

[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/.

[SLFP04] Kai Schramm, Gregor Leander, Patrick Felke, and Christof Paar. A Collision-Attack on AES: Combining Side Channel- and Differential-Attack. In *Cryptographic Hardware and Embedded Systems—CHES 2004*, pages 163–175, 2004.

[SWP03] Kai Schramm, Thomas J. Wollinger and Christof Paar. A New Class of Collision Attacks and Its Application to DES. In *Fast Software Encryption—FSE'03*, pages 206–222, 2003.

[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.

[TTS+06] Yukiyasu Tsunoo, Etsuko Tsujihara, Maki Shigeri, Hiroya Kubo, and Kazuhiko Minematsu Improving cache attacks by considering cipher structure. In *International Journal of Information Security 2006*.

A Implementation Notes

All of the attacks described in this paper have been implemented as a UNIX command line program `aes_attack`, the source code of which is available at the author's website. The program can be recompiled to use any of the attack algorithms described, as well as options for decryption attacks and different cache eviction routines. The program first generates a large number of timing samples by repeatedly triggering one encryption for a random plaintext using an OpenSSL library call and recording the resulting ciphertext along with a processor cycle count. Each timing/ciphertext pair is added to a large buffer after being recorded, this allows a minimum of activity in between encryptions. An explicit cache eviction routine is called before each encryption, as described in Appendix B, no other work is done between encryptions. After each encryption, each byte of the resulting ciphertext is touched, this must be done to ensure the encryption has finished before recording the ending time on platforms such as the Pentium IV which support out-of-order instruction execution while waiting for cache misses.

After generating a large number of samples, the attack algorithm is called with a small set of the data. It is incrementally given more of the data until it succeeds in recovering the key. Samples are not used if their time is more than twice the lowest time seen, this eliminates noise due to page faults and context switches. These ignored samples are still counted when reporting the number of samples necessary for the attack to succeed.

B Cache Eviction

All of the attacks described in this paper require the AES lookup tables to be (at least partially) out of the cache prior to an encryption operation. If all tables are cached, which would occur during a long run of consecutive encryptions, then cache collisions will not reduce timing. In a real attack scenario, an attacker must have some ability to remove the tables from cache before an encryption. The most likely approach would be simply waiting. If the target machine is doing other work, the tables will probably be quickly evicted from memory as other processes load their own data. Also, it is assumed that the target program only performs key expansion once, then stores the expanded key in memory and uses it for subsequent encryptions. Otherwise, key expansion before each encryption would have the side effect of loading some of the AES tables into memory, since they are used in key expansion.

For the purposes of this study, we consider two cases, if the AES tables are fully evicted from Level 1 cache, and if the AES tables are fully evicted from Level 2 cache. It is also easy to verify that if only some random fraction of the table entries are out of cache, the attack will still succeed with additional samples. To simulate the eviction of tables from L2 cache, we sequentially accesses a continuous block of memory the size of the cache, which will evict all previous contents. To save time in experiments on Pentium IV, we use the `clflush` instruction. Eviction from L1 cache is similar, although we must be careful not to evict tables from L2 cache. To do this, we read in a small amount of data to evict only L1 cache, but not the AES tables in L2 cache.

C Pentium IV complications

The model discussed in Section 4 appears to be a very good approximation for the cache behavior of the Pentium III and UltraSPARC processors. From our experiments, we have seen that it does not fully capture the complexity of the Pentium IV's cache structure. The first complication is that Pentium IV "usually" loads cache lines in pairs, making the cache lines 128 bytes. In some experiments two indices being in neighboring cache lines produced a bigger time drop than a traditional collision. Second, Pentium IV has a hardware pre-fetch mechanism. If it notices "several" straight cache misses, it will begin pre-fetching data in the direction the accesses are going, assuming it is a large serial data read. The Intel documentation uses the word several, which it says could be "as few as 2." So, certain cache collisions may trigger the hardware pre-fetcher, while others may not. Finally, Pentium IV supports out-of-order instruction execution while waiting for cache misses. This means that in certain situations, cache misses may have little effect on the overall encryption if there are enough instructions to be executed which do not depend on the fetched value. The net result of these Pentium IV features is somewhat chaotic behavior when a simple model is assumed, this was also observed in [OST06].

D Final Round Optimizations

The final round attack looks at the average time for each possible value $\Delta_{i,j} = k_i^{10} \oplus k_j^{10}$ for all i, j, where the true value for each $\Delta_{i,j}$ should be lower than the average. The raw data is converted into a cost function, $c(i, j, \Delta)$, which should be low for values of Δ which represented low times. Eventually, the true values of each $\Delta_{i,j}$ should be the lowest values. However, in the presence of noise, the algorithm seeks to produce some guess K' at the key which minimizes the total cost function $C[K] = \sum_{i,j}[c(i, j, K_i \oplus K_j)]$. The guess K' will not be a guess of actual key bytes, but a set of offsets $\Delta_{0,i} = k_0^{10} \oplus k_i^{10}$ for all $1 \leq i \leq 15$. Two adapted AI algorithms can be used to attempt to minimize this function.

The first is a variant of local optimization search. The cost function used by this algorithm is simply $c(i, j, \Delta) = (\Delta - \Delta^*)^2$, where Δ^* is the lowest value observed for that particular i, j. After an initial guess K_0 is made at the key offsets, the total cost function is calculated for every key guess K_0' which can be obtained by changing one byte of K_0. The lowest cost K_0' then becomes the new key guess K_1. This process is repeated either until a local minimum is reached, or a preset maximum number of iterations is reached. Each guess K_i leads to 256 possible values for the actual key. These are obtained by guessing all values for k_0^{10}, the final 15 bytes of the key are then determined by the offsets $\Delta_{0,i}$. Finally, the guess at the final 16 bytes of expanded key is reverted to a guess at the original key, which can be checked against a known plaintext/ciphertext pair.

The second approximation algorithm used is belief propagation. For this approach, a probability approximation $\varphi(i, j, \Delta)$ can be made based on the observed data by mapping it to a normal distribution, since the average and standard deviation are known. This is used in place of a cost function. Next, an initial set of probabilities are guessed for each key offset $p_0(i, \Delta) = \Pr[k_0^{10} \oplus k_i^{10} = \Delta]$. These probability guesses $p_0(i, \Delta)$ are updated as follows: For each $j \neq i$, the maximum value of $p_0(j, \Delta') \cdot \varphi(i, j, \Delta \oplus \Delta')$ over all Δ' is added to p_1. The guesses p_1 are then normalized. This process is repeated, and the probabilities $p(i, \Delta)$ should eventually be higher for the correct values. After each iteration, the probabilities are used to construct a best guess for the key, as before. In this study, both algorithms were used, since we found experimentally that each was successful before the other for certain data sets.

The expanded final round attack provides slightly different raw data than the simple final round attack, namely, a set of average times $t(i, j, \alpha, \beta)$, low times should occur at the values $k_i^{10} = \alpha$ and $k_j^{10} = \beta$. Instead of using a cost function, each pair is given a weight w representing its likelihood of being correct. A threshold τ is chosen, times $t(i, j, \alpha, \beta)$ which are not among the τ lowest times for i, j are given weight $w = 0$. The lowest time is given weights $\tau - 1$, the next lowest time $\tau - 2$, and so on.

The goal of the approximation algorithm is to produce a key guess which has the highest sum of weights $W[K] = \sum_{i,j}[w(i, j, K_i, K_j)]$. After making an initial guess, the algorithm proceeds to perform simulated annealing, changing one byte in each round which raises the total weight of the key as much as possible. Heuristically, this approach performed better than belief propagation for this attack. For this study, the algorithm was used with $\tau = 16$.