A COMPUTATIONALLY FEASIBLE SPA ATTACK ON AES VIA OPTIMIZED SEARCH

Joel VanLaven[1], Mark Brehob[2], and Kevin J. Compton[3]
EECS Department
University of Michigan - Ann Arbor
Ann Arbor, MI 48109-2122, USA
[1]jvanlav@umich.edu, [2]brehob@umich.edu, [3]kjc@umich.edu

Abstract: We describe an SPA power attack on an 8-bit implementation of AES. Our attack uses an optimized search of the key space to improve upon previous work in terms of speed, flexibility, and handling of data error. We can find a 128-bit cipher key in 16ms on average, with similar results for 192- and 256-bit cipher keys. The attack almost always produces a unique cipher key and performs well even in the presence of substantial measurement error.

Keywords: AES, SPA, Rijndael, power attack.

1. Introduction

In 2001 the National Institute of Standards and Technology selected the block cipher Rijndael as the Advanced Encryption Standard (AES), making it the standard for private key encryption. A cryptographic attack on AES, such as linear or differential cryptanalysis, appears intractable at this time. Therefore, some researchers have investigated side-band attacks, which use information about the physical manifestation of the hardware or software implementing the algorithm [1, 2, 6].

Side-band attacks assume access to the hardware performing the encryption. Timing attacks, proposed in 1996, assume only the ability to time the encryption (or perhaps sub-portions of the encryption) [4]. Such attacks are relatively easy to thwart by writing encryption software that uses a fixed sequence of operations. Power attacks, another style of side-band attack, are more difficult to thwart. The most common power attacks assume

the ability to observe the power utilization of the processor or ASIC over time [5].

Power attacks provide both high-level information about the operations being performed on the chip and low-level information about the data being operated upon. The high-level information is similar to timing information and can be dealt with in a similar way. Low-level information about the data arises from sources such as asymmetry in the efficiency of the n and p transistors or the flipping of bits on a bus or in a register. Without great care in the chip design and addition of power inefficiencies, a CMOS chip will use a slightly different amount of power based on the data being calculated [8]. Microprocessors operate on a fixed number of bits at a time (usually words or bytes), so what is actually revealed is the sum of the bits, or the Hamming weight of the data.

The two main variants of power attacks are differential power analysis (DPA), which requires the plaintexts or ciphertexts in addition to the power traces for many encryptions with the same key; and simple power analysis (SPA), which exposes the secret key solely from power traces [5]. While in theory most SPA attacks could reveal the key from a single encryption, poor signal-to-noise ratio forces averaging of the error over many encryptions with the same key. DPA is applicable to most ciphers and implementing such an attack is relatively straightforward. SPA is greatly affected by the design of a cipher and the susceptibility of a cipher to this style of attack may not be obvious.

This paper details an SPA power attack on an 8-bit implementation of AES. We assume that the Hamming weights of the bytes of the expanded key can be measured, possibly with some error. Our approach exploits regularities in the AES key schedule, which could likely be utilized even if different information more specific to the implementation is exposed. It improves upon a previously published attack by Mangard [6] in terms of speed, flexibility, and handling of measurement error. Specifically our algorithm improves upon this work in four ways.

- It runs approximately 20000 times faster.
- It nearly always finds a unique solution rather than a handful of candidate solutions.
- It works on cipher-key sizes of 192 and 256 in addition to 128 bits.
- It performs well under a more realistic error model.

Table 1. Time and Discovery Rate of Cipher Keys

	128-bit key no error	*256-bit key no error*	*128-bit key with error ($\sigma = 0.25$)*
Average time	16ms	20ms	35s
% of attacks with a unique solution	100%	99.97%	96%

With regard to the second item, the previous work required a ciphertext/ plaintext pair to find the correct solution, thus negating a primary advantage of an SPA attack. While the algorithm in this paper cannot guarantee a unique solution, our results show that even with significant errors in the data, it almost always finds a unique solution. Table 1 provides a summary of our results.

2. AES

In private-key cryptosystems such as AES, both the sender and receiver of a message require access to the same secret key. Public-key cryptosystems allow the sender and receiver to use different keys, only one of which needs to be secret, but require significantly more computational power as well as a significantly longer key. AES (like the DES standard that it replaced) is an iterative block cipher. This means that the data is manipulated in series of "mini-encryptions," called rounds, each of which uses its own key. In order to generate these *round keys* the AES algorithm expands the 128-, 192-, or 256-bit private key (also called the *cipher key*) into the needed number of 128-bit round keys using the key expansion algorithm described below.

Key Expansion in AES

Our attack exploits the relationships between the round keys resulting from patterns in the key expansion algorithm. As such, it is necessary to carefully describe the algorithm found in the AES specification [3]. The key expansion algorithm is slightly different depending upon the cipher key size. Though our attack works on all three different key sizes, for simplicity we will discuss only the 128-bit key expansion (which is the most commonly used). The 192-bit and 256-bit key expansions are similar, and the results of attacks on those key sizes are summarized in section 5.

The 128-bit cipher key is expanded into eleven 128-bit round keys, each of which can be thought of as 16 bytes arranged in a 4-by-4 block. Each successive round key is simply a transformation of the previous round key. Define $RK[N, R, C]$ for $N = 0,...,10$, $R = 0,...,3$ and $C = 0,...,3$, to be the byte found in the N-th round key at row R and column C. The first round key (i.e.

the round key for $N = 0$) is a copy of the 128-bit cipher key. When $N > 0$, the round key $RK[N, R, C]$ is equal to [24]

$$\begin{cases} RK[N-1,R,C] \oplus RK[N,R,C-1], \text{if } C > 0; \\ RK[N-1,R,C] \oplus SB[RK[N-1,R-1,3]], \text{if } R > 0 \text{ and } C = 0; \\ RK[N-1,R,C] \oplus SB[RK[N-1,3,3] \oplus RC[N], \text{if } R = 0 \text{ and } C = 0. \end{cases}$$

Here $\oplus$ is the XOR function; SB is an invertible function, called the *subbyte* function, which maps bytes to bytes; and $RC[N]$ is the N-th round constant, a fixed value independent of the cipher key. The AES standard gives the precise definitions of SB and $RC[N]$. Each byte other than those in the cipher key is computed from exactly two other bytes. For example, when $N > 0$ and $C > 0$,

$$RK[N,R,C] = RK[N-1,R,C] \oplus RK[N,R,C-1]$$

but then we also have

$$RK[N-1,R,C] = RK[N,R,C] \oplus RK[N,R,C-1]$$
$$RK[N,R,C-1] = RK[N,R,C] \oplus RK[N-1,R,C]$$

That is, the computational relationship between bytes is symmetric in the sense that each of the bytes is computable from the other two. This is also true in the cases where $N > 0$ but $C \neq 0$, the only difference being that we have slightly more complicated expressions involving the SB function and $RC[N]$ constants. We picture all of these computational relationships in the hypergraph of Figure 1.

[24] The notation here is not the same as the round key function $W[i,j]$ in [3]. The relationship between the two notations is $RK[N,R,C] = W[-R \bmod 4, 4N + C]$. Our notation was chosen to make our description of the key schedule structure clearer.

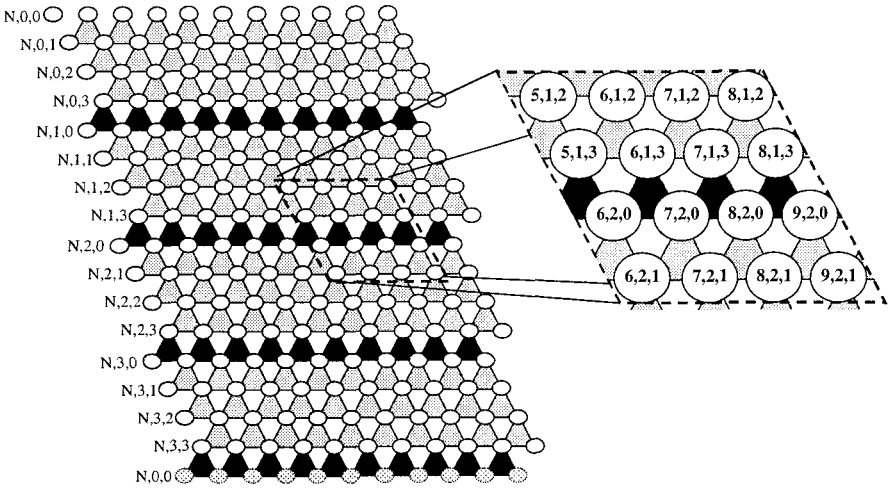

Figure 1. The Key Schedule Hypergraph for 128-bit Cipher Keys

A *hypergraph* is a pair (V, E) where V is the *vertex set* and $E \subseteq 2^V$ is the *hyperedge set*. If we require that all hyperedges contain exactly two vertices, we have the usual definition of a graph. In our hypergraph, the vertices are the bytes of the round keys and the hyperedges are the 3-element sets of computationally related bytes.

The ovals in the diagram represent the vertices of the hypergraph. For clarity, we have not labeled every vertex in the diagram. In the row labeled N,0,0, for example, the eleven vertices should be labeled

$$(0,0,0),(1,0,0),(2,0,0),...,(10,0,0)$$

and the bytes assigned to them are

$$RK[0,0,0],RK[1,0,0],RK[2,0,0],...,RK[10,0,0]$$

respectively. The figure shows a blowup of a small section of the hypergraph with the vertices labeled. The shaded triangles represent the hyperedges; the light shaded triangles represent the hyperedges where the subbyte relation is not used to compute the computational relationship. For example, since

$$RK[1,0,1] = RK[0,0,1] \oplus RK[1,0,0]$$

we have a light hyperedge $\{(1,0,1),(0,0,1),(1,0,0)\}$, which is the upper-leftmost shaded triangle. From

$$RK[1,1,0] = RK[0,1,0] \oplus SB[RK[0,0,3]]$$

we get a dark shaded hyperedge $\{(1,1,0),(0,1,0),(0,0,3)\}$.

The bytes of the cipher key $RK[0,0,0],RK[0,0,1],...,RK[0,3,3]$ are assigned to the vertices along the left edge of the diagram. The slightly fuzzy row of vertices at the bottom of the diagram is the same as the top row of vertices; that is, the diagram should "wrap around" and the first and last rows be identified.

3. Optimizing Search for a Cipher Key

We now give a precise statement of the SPA Key Schedule Problem (in the case of 128-bit cipher keys). Divide a cipher key of 128 bits into 16 bytes $RK[0,0,0]$ to $RK[0,3,3]$ and compute bytes $RK[N,R,C]$ as described in section 2. Given just the Hamming weights of these bytes, determine the original 128-bit cipher key.

An exhaustive search, cycling through the 2^{128} possible cipher keys, is clearly infeasible. Even if we cycle through only those keys where each byte of the cipher key has the correct Hamming weight, the number of possible keys could be as large as 2^{98}, still far too large to search. We need to utilize the Hamming weights of the entire expanded key to reduce the search space to a manageable number of keys.

A naive approach would be to search for the cipher key by sequentially assigning possible values for the bytes $RK[0,0,0]$ to $RK[0,3,3]$ (i.e., those bytes for which $N = 0$) and checking consistency with the Hamming weight information after each assignment. Inspection of Figure 1 shows that this is little better than an exhaustive search. After we have assigned values to $RK[0,0,0]$ and $RK[0,0,1]$, for example, we have no further information about Hamming weights of other bytes in the key schedule since they do not belong to a common hyperedge.

Suppose, instead, that we assign possible values for $RK[0,0,0]$ and $RK[1,0,0]$ corresponding to vertices in the bottom row of the hypergraph. We can then compute $RK[0,3,3]$ and check three values (rather than just two as before) for consistency with the Hamming weight information. This improves on exhaustive search because it eliminates many possible assignments. This is the main idea behind our search sequence optimization.

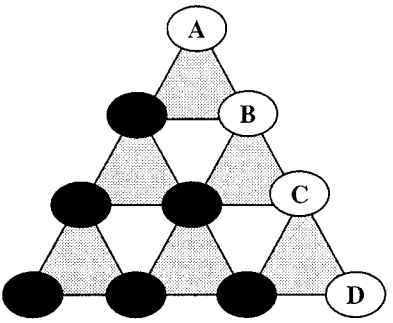

Figure 2. A Fragment of the Key Schedule Hypergraph

Systematic use of this idea results in a highly optimized search. Consider the small fragment of the hypergraph in Figure 2. Suppose we have assigned values consistent with the Hamming weight information for the six shaded vertices. If we then make an assignment to vertex A, we can compute values for vertices B, C, and D, then check that these values are consistent with the Hamming weights. Notice also that if we have values assigned to the six shaded vertices, assigning a value to any one of the vertices A, B, C or D allows us to compute values for the other three.

Thus, there are many ways to choose a sequence so that the maximum number of byte values can be computed after successive assignments to vertices in the sequence. However, it is not difficult to see that after each assignment (for at least the first 11 assignments), the pattern of computable values will be a triangular array of the type shown in Figure 2. That is, we can find a vertex sequence $S_0,S_1,...,S_{15}$ so that after values have been assigned to $S_0,...,S_i$, we can compute $(i+1)(i+2)/2$ byte values in a triangular array. When $i \geq 11$ a complete triangular array will not fit horizontally in the hypergraph shown in Figure 1 so the increase in the number of computable values is not as great. However, by this stage so many values are determined that maximizing the number of computable values is not so important (Figure 4 illustrates this). After assignments to $S_0,...,S_{15}$, all 176 bytes can be computed because of the wrap-around in the hypergraph.

Besides maximizing the number of computable values after each assignment, speedups can be gained by taking advantage of information available from the subbyte operation used in the dark hyperedges. Maximizing the number of dark hyperedges contained within the triangular array of computable values results in additional pruning in the early stages of the search. This can give approximately an order of magnitude speedup. Because the subbyte is applied to the top vertex in any dark hyperedge, it is possible to easily extract this information without determining the two lower vertices of such a hyperedge. Maximizing the number of top vertices of dark

hyperedges determined instead of whole dark hyperedges, allows a further speedup by a factor of about 2. The search sequence we use is: (9,0,3), (8,0,3), (7,0,3), (6,0,3), (5,0,3), (4,0,3), (3,0,3), (2,0,3), (1,0,3), (0,0,3), (0,1,0), (0,1,1), (0,1,2), (0,1,3), (0,2,0), (0,2,1). There are many other optimal sequences.

We can now give a precise description of the search algorithm. Let $S_0,S_1,...,S_{15}$ be a fixed optimal search sequence of 16 vertices as described above. Suppose that at some time during the search, values have been assigned to $S_0,...,S_i$ and are stored in a global array `A`. Let `consistent(i)` be a Boolean function that returns `true` precisely when the values computed from these $i+1$ values are consistent with the Hamming weight information and the information from dark hyperedges mentioned in the previous paragraph. Thus, when $i < 11$, `consistent` checks the consistency of $(i+1)(i+2)/2$ values.[25]

The search algorithm is a standard backtrack algorithm. Pseudocode for a recursive version of the algorithm is given in Figure 3. (Our implementation was iterative, to optimize performance, but the recursive version here is a little more transparent.) Function `search(n)` cycles through possible assignments to S_n, storing them in an array `A` at index `n`. For those bytes that are consistent with the Hamming weight information, the search goes on to `search(n+1)`. For those that are not consistent, it goes on to the next possible byte. If all the bytes have been checked, it returns to the last calling search. Whenever `n` reaches 16, it writes out a possible solution stored in `A`. To run the algorithm we initially call `search(0)`.

```
void search(n)
{
      if (n==16) write A;
      else
            foreach byte w
            {
                  A[n]=w;
                  if (consistent(n))
                        search(n+1);
            }
}
```

Figure 3. Pseudocode for the Recursive Search Algorithm

[25] In fact, it is really only necessary to check the consistency of the $i+1$ values added since the last consistency check.

4. The Attack in the Presence of Error

While work by Mayer-Sommer has shown that it is possible to determine Hamming weights in "an unequivocal manner" [7], measurement and data collection will inevitably have an associated error rate. We model this error by adding Gaussian noise with a mean of zero for each of the measured Hamming weights. This model is reasonable, because even if the actual distribution of noise for a single run is not Gaussian, averaging a number of independent noise measurements together should yield an overall distribution that approaches Gaussian. A mean of zero should be obtainable as part of the method used to calibrate the measurements of the Hamming weights.

The measured Hamming weights with this error assumption will be real-valued rather than discrete. Further, it is not possible to search for the exact key that matches the given Hamming weights: all keys match, but some keys are more likely than others. Our attack provides all cipher keys (if any exist) whose round key expansions have Hamming weights differing from the measured Hamming weights by less than some bound (using a sum of the squares metric). Given our assumptions about the nature of the error, the sum of squares difference is a maximum likelihood estimator. This means that any key not reported is less likely than any key that is reported. The bound can be chosen to guarantee with some confidence (e.g. 95%), given an expected amount of error, that the true key will be returned.

Changes to the Algorithm

When dealing with error, the definition of the function `consistent` from Figure 3 needs to be modified. Specifically, `consistent` will return `false` if the assignment to S_n gives a sum-of-squares difference greater than a certain bound. Our implementation computes an optimistic estimate of this value. It is computed as the sum of two values:

- The sum of squares difference between the Hamming weights of those bytes determined by S_n and their measured values.
- The minimum sum of squares difference between the measured values of all those bytes which are not determined by S_n and the integer values closest to those measured values.

The bound is determined by adding a fixed value based on desired confidence to the minimum sum-of-squares difference between the measured values and the integer values closest to those values. This method of determining the bound helps to make the work required by the algorithm more uniform than using a fixed bound based on desired confidence.

5. Results

We ran all of our simulations on a 500MHz Sun Blade 100 with 512MB of DRAM. A synopsis of our simulation results can be found in Table 2. As noted in section 4, the bound for the search is determined by the desired confidence given an expected amount of error. The last four entries in the table all assume a different amount of error. The expected amount of error is expressed as a standard deviation of the expected Gaussian noise. For all of the runs with error we targeted a confidence rate of 95%.

Table 2. Time and Discovery Rate of Cipher Keys

Attack type	*Average time per attack*	*% of attacks with a unique solution*
128-bit no error	16ms	100%
192-bit no error	60ms	100%
256-bit no error	20ms	99.97%
128-bit, $\sigma = 0.20$	4s	95%
128-bit, $\sigma = 0.25$	35s	96%
128-bit, $\sigma = 0.30$	38 min	**
128-bit, $\sigma = 0.35$	15 hours	**

*For the entries labeled **, not enough data was collected to provide a meaningful value*

Notice that the run time of our algorithm increases exponentially relative to the expected amount of noise. This is because a higher expected error rate forces us to have a looser bound, and that reduces the amount of pruning of the search space that can be accomplished. Figure 4 graphically shows how large an impact this has. An implication of Table 2 and Figure 4 is when a large amount of noise is present our algorithm's runtime will become untenable.

Another interesting result from Table 2 is that the 192-bit implementation takes longer than either the 128 or 256 bit implementations. This is because the 192-bit version of AES has fewer instances of *subbyte* per expanded key byte than the 128 or 256 bit versions.

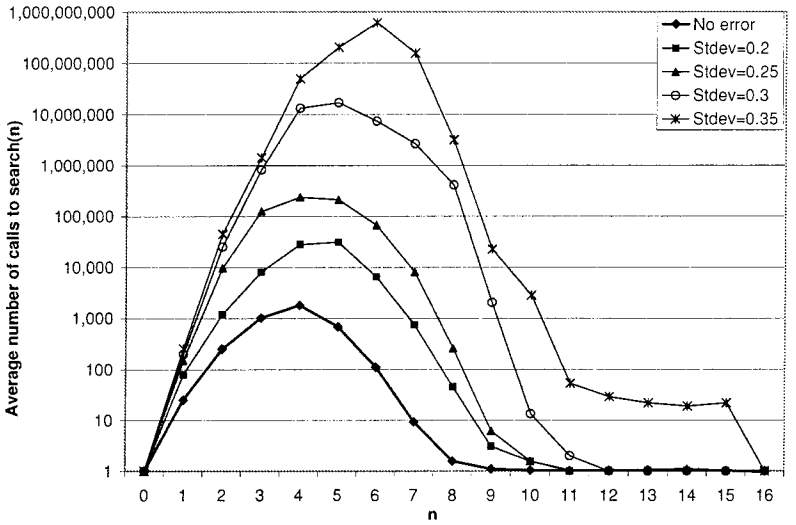

Figure 4. Average number of calls to search as a function of n

6. Conclusions and Future Work

We have shown that AES is susceptible to very efficient attacks based solely upon Hamming weights of the bytes of the expanded key. These Hamming weights would likely be exposed in the case of an 8-bit implementation, even if a pre-expanded key were used. Further, this algorithm works well even in the presence of significant Gaussian noise.

This work can be extended in the following ways:

- Modifying the algorithm to work with 16 or 32 bit implementations without a pre-expanded key.
- Proving information theoretic bounds about the feasibility of this SPA attack on 16 and 32 bit implementations with pre-expanded keys.
- Significantly improving the algorithm's efficiency in the face of error to handle larger errors.
- Gathering the data and performing the attack on a real system.

REFERENCES

[1] E. Biham and A. Shamir. Power analysis of the key scheduling of the AES candidates. In *Second Advanced Encryption Standard (AES) Candidate Conference*, 1999.

[2] S. Chari, C. Jutla, J.R. Rao, and P. Rohatgi. A cautionary note regarding evaluation of AES candidates on smart-cards. In *Second Advanced Encryption Standard (AES) Candidate Conference*, 1999.

[3] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Springer-Verlag, 2002.

[4] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, pages 104–113, 1996.

[5] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.

[6] Stefan Mangard. A simple power-analysis (SPA) attack on implementations of the aes key expansion. In *ICISC*, pages 343–358, 2002.

[7] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *CHES*, pages 78–92, 2000.

[8] Jan M. Rabaey. *Digital Integrated Circuits: a Design Perspective*. Prentice-Hall, Inc., second edition, 2002.