CACHE-COLLISION TIMING ATTACKS

AGAINST AES-GCM

by

Bonan Huang

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical & Computer Engineering

Fall 2010

Copyright 2010 Bonan Huang
All Rights Reserved

CACHE-COLLISION TIMING ATTACKS

AGAINST AES-GCM

by

Bonan Huang

Approved: ____

Xiaoming Li, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: ____

Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical & Computer Engineering

Approved: ____

Michael J. Chajes, Ph.D.
Dean of the College of Engineering

Approved: ____

Charles G. Riordan, Ph.D.
Vice Provost for Graduate and Professional Education

ACKNOWLEDGMENTS

I wish to thank my adviser, Xiaoming Li; and my friends and colleagues Murat Borat, Liang Gu, and Ryan Taylore.

I must also thank my spouse, Bin Wang, without whom none of this would ever have been possible.

TABLE OF CONTENTS

LIST OF TABLES v
LIST OF FIGURES vi
ABSTRACT vii

Chapter

1 OVERVIEW OF AES-GCM ALGORITHM 1
1.1 Detail Algorithm 1
1.2 Motivation 3

2 RELATED WORK 5

3 ATTACK MODEL AND STRATEGY 8
3.1 Attack Model 8
3.2 Second Round Attack 9
3.3 Final Round Attack 10

4 RESULTS 12

5 IMPLEMENTATION NOTES 16
5.1 Cache Eviction 16
5.2 Extended Euclidean Algorithm 17

6 COUNTERMEASURES AND CONCLUSION 18

REFERENCES 21

LIST OF TABLES

Table 4.1 Median Sample and Successful Rate... 12

LIST OF FIGURES

Figure 1.1 Analysis of AES-GCM mode 3

Figure 3.1 Final round attack model 10

Figure 4.1 Data collection graphs from 2^{18} through 2^{21}samples 13

Figure 4.2 The data collection results from 2^{22} samples. 14

Figure 4.3 The variance of real key average ranking during each iteration 15

ABSTRACT

Side-channel attacks that utilize timing, power consumption, and electromagnetic radiation to gain information about an encryption/decryption implementation have been demonstrated experimentally to be an effective attack against a variety of cryptographic systems. We define a general attack strategy against AES-GCM 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-GCM implementations and computing platforms, we have implemented them against Openssl-1.0.0-beta3 running on Intel(R) Xeon(R) CPU 5110 and Intel(R) Xeon(R) CPU 5520. This is the first time in publication to successfully attack the AES-GCM algorithm. While the task of defending AES-GCM against all timing attacks is challenging, a small patch can significantly reduce the vulnerability to these specific attacks with no performance penalty.

Chapter 1

OVERVIEW OF AES-GCM ALGORITHM

Galois Counter Mode (GCM) is a NIST-standardized mode of operation for symmetric key cryptographic block ciphers. Combined with Advanced Encryption Standard (AES) encryption, AES-GCM is an authenticated encryption algorithm designed to provide both authentication and privacy. This mode is defined for block ciphers with a block size of 128 bits. GCM core operation is the multiplication in 128-bit Galois field, and it's implemented using key-dependent lookup tables. AES-GCM mode is used in the IEEE 802.1AE (MACsec) Ethernet security, ANSI Fibre Channel Security Protocols (FC-SP) and etc. The most expensive operations for GCM authentication are the multiplications over the field GF(2^{128}). Since it takes time-memory tradeoffs, the standard implementation is through lookup tables, which are pre-computed for a particular value of the secret constant hash key, this kind of implementations are not guaranteed to be secure.

1.1 Detail Algorithm

AES operates on a 4x4 array of bytes, and is made up by four steps: *AddRoundKey, SubBytes, ShiftRows* and *MixColumns*. It scrambles a 16-byte input using a 16-byte key, and two constant 256-byte tables. Although AES has been under scrutiny by cryptanalysts for several years, side-channel attacks, especially cache-timing attacks proved that it's not guaranteed to be secure. For GCM, Figure 1.1 explains the whole operation mode, and each block represents 16-byte data. The 128-

bit hash key H is generated from the master encryption key K (AES key) as the plaintext is all-zero. Ek is the AES encryption function using master key K, multH() function calculates the multiplication by the hash key H in GF(2^{128}). Intr() is the counter increment function. There are four inputs of authenticated encryption operations: 1) master key K; 2) an initial vector IV; 3) a plaintext P and 4) additional authenticated data (AAD) A. There are two outputs: 1) a ciphertext C and 2) an authenticated tag T [1].

GF(2^{128}) uses the polynomial $f = 1 + x + x^2 + x^7 + x^{128}$. The core operation is the Galois field multiplication with hash key H, and the standard computation $Z = X \cdot H$ in software is:

$$Z = M_0[byte(X,0)] \oplus M_1[byte(X,1)] \oplus \ldots \oplus M_{15}[byte(X,15)] \quad (1.1)$$

where byte(X, i) denotes the ith byte of the element X. Each variable is 16 bytes, so the total table memory costs 64 KB. From [1], using the platform as Motorola G4 processor, the table-driven method takes 13.1 cycles per byte as the throughput, while no-table method takes 119 cycles per byte.

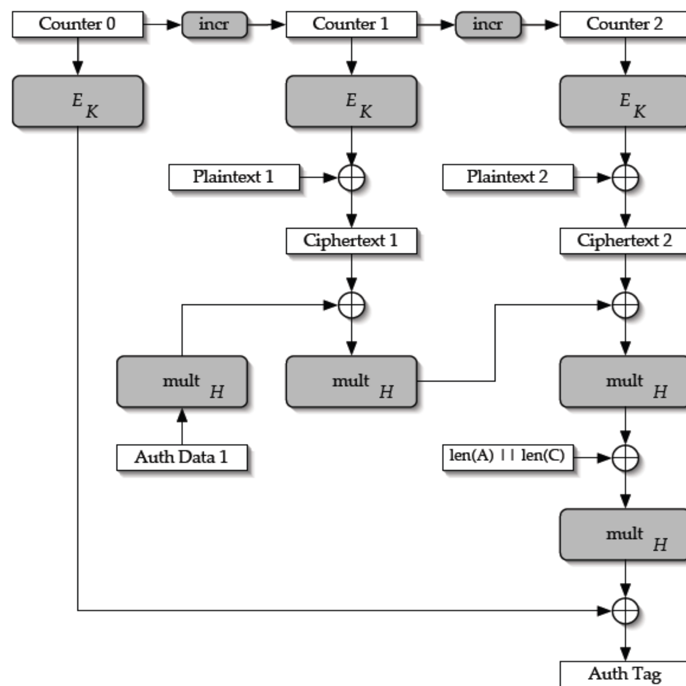

Figure 1.1 Analysis of AES-GCM mode

1.2 Motivation

In my thesis, the attacks assume that the computer uses cached memory which can be described using a simple model of the cache. Cache stores values which are looked up in main memory, while evicting older values in it. Subsequent lookups to the same memory address can then get the data directly from the cache, which is faster than main memory, this is called a "cache hit." Since most software exhibits temporal locality in memory accesses, caches greatly improve performance, however, at the same time it generates the imbalance of execution time. In GCM, the tables are generated by hash key H, the attacks in my thesis will focus on them, which may leak a great deal of secret information to a timing attack with the input of plaintexts and

ciphertexts. If H is compromised entirely, the authentication assurance will be completely lost. Attacker gaining information about this value can then easily deduce the secret value H necessary for a forgery attack [1].

Chapter 2

RELATED WORK

Side channel attack is any attack based on information gained from the physical implementation of a cryptosystem, rather than brute force or theoretical weaknesses in the algorithms. For example, timing information, power consumption, or even sound can provide an extra source of information which can be exploited to break the system. Some side-channel attacks require technical knowledge of the internal operation of the system on which the cryptography is implemented, although others such as differential power analysis are effective as black-box attacks.

Differential power analysis (DPA) is a side-channel attack which involves statistically analyzing power consumption measurements from a cryptosystem. The attack exploits biases varying power consumption of microprocessors or other hardware while performing operations using secret keys. DPA attacks have signal processing and error correction properties which can extract secrets from measurements which contain too much noise to be analyzed using simple power analysis. Using DPA, an adversary can obtain secret keys by analyzing power consumption measurements from multiple cryptographic operations performed by a vulnerable smart card or other device. The most powerful DPA attacks are based on statistical methods pioneered by Kocher [2].

Differential fault analysis is another type of side channel attack. The principle is to induce faults (unexpected environmental conditions) into cryptographic implementations, to reveal their internal states. Biham [3] and Boneh [4] found that a

smartcard containing an embedded processor might be subjected to high temperature, unsupported supply voltage or current, excessively high overclocking, strong electric or magnetic fields, or even ionizing radiation to influence the operation of the processor. The processor may begin to output incorrect results due to physical data corruption, which may help a cryptanalyst deduce the instructions that the processor is running, or what its internal data state is.

Cache-timing attacks are software side-channel attacks exploiting the timing variability of data loads from memory. This variability is due to the fact that all modern microprocessors use a hierarchy of caches to reduce load latency. Kocher [5] was the first to suggest cache-timing attacks against cryptographic algorithms that load data from positions that are dependent on secret information. Initially, timing attacks were mostly mentioned in the context of public-key algorithms until Kelsey et al. and Page [6] considered timing attacks, including cache-timing attacks, against secret-key algorithms.

Bernstein demonstrated a different type of timing attack against AES in 2005 [7] which can be thought of as a statistical timing attack. He found that the first round of encryption is simply the bytes $x = p \oplus k$, and these bytes are used as the indices of the lookup tables, so the running time is directly affected by each of the values. He first collected a large volume of timing data for each value of an input byte using both target machine and reference machine, and then he correlated the data to recover the key. Joseph Bonneau made a successful white-box timing attacks on AES [8], which used expected timing effects due to the structure of the cipher. He uses require timing data and known plaintext and ciphertext, and implement the last round attack using random walk and belief propagation methodology. Acıiçmez [9] found

that attack can be used to obtain secret AES keys of remote cryptosystems if the server under attack runs on a multitasking or simultaneous multithreading system with a large enough workload.

My thesis is the first publication to successfully attack AES-GCM. It focuses on an approach that requires no specific information about the target platform with only information of timing data and known plaintext and ciphertext. Since the GCM is built on GF(2^{128}), we also need the extended Euclidean algorithm for the calculation.

Chapter 3

ATTACK MODEL AND STRATEGY

There is a complicating fact that modern caches do not store individual bytes, but groups of bytes from consecutive “lines” of main memory. Line size varies between 64 or 128 bytes on more recent Intel or AMD processors. Since the usual size of GCM table entries is 16 bytes, groups of 4 consecutive table entries share a line in the cache on an Intel Xeon CPU. So, for any bytes l, l′ which are equal ignoring the lower 2 bits, looking up address l will cause an ensuing access to l′ to hit in cache.

3.1 Attack Model

We pick inputs of the two multiplications, for example: X, Y:

- $$Z = M_0[byte(X,0)] \oplus M_1[byte(X,1)] \oplus \ldots \oplus M_{15}[byte(X,15)] \quad (3.1)$$
- $$Z = M_0[byte(Y,0)] \oplus M_1[byte(Y,1)] \oplus \ldots \oplus M_{15}[byte(Y,15)] \quad (3.2)$$

A cache collision occurs when $byte(Y,0) = byte(X,0)$. From the complications we denote: $< x_i^0 >=< x_j^0 >$ ($<>$ represents the most significant bit) and now we formalize the assumption:

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

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. This assumption greatly oversimplifies many the intricacies of modern

caches. The notion of using collisions in the cache is by no means unique to my thesis. 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-GCM. This general notion has been used in several other attacks on AES.

3.2 Second Round Attack

A natural approach to attack AES-GCM is to analyze the table lookups performed in the second round, because it does multiplication with H for the first time in the process, and easily leaves the trace to search. From Fig 1.1, A is Auth Data1, C0 is Ciphertext 1, the cache collision occurs when $< A > = < (A \cdot H) \oplus C0 >$. To make the equation simplified, we set A = 0x00…01 so that $< H > = < A \oplus C0 >$.

The goal of the attack is to record timing data for random cipher texts at each value of$< H > = < A \oplus C0 >$. For each ciphertext/time pair observed, the encryption time is used to update a table of average times t[i, j] for all values i, j. The goal to find one value for each i, j such $that\ t[i,j] < \bar{t}[i]$ where $\bar{t}[i]$ is the average encryption time over all $C0[i]$.

Eventually, the value of j will become accurate guesses, which should be the only values which cause significantly low encryption times. However, it’s not always obvious to find the lowest timing value. The data processing can be done off-line by the attacker after the data is collected. Another t-test will check if the value is significantly lower of the mean. Meanwhile, a set of limited data will be saved in case the lowest timing data doesn’t come out the real hash key.

In experiments most of H bytes are not matched. One possible assumption is that the AES execution in the second round polluted the results. They may use look-up table based AES so that generating the big peaks.

3.3 Final Round Attack

To design a non polluted attack, we consider the final round of encryption. Since there’s no AES executed between last two multiplications, 16 bytes plaintext is used in the experiment so that the algorithm halts at “Counter 1”, see Fig 3.2.

To make it simplified, we set A = 0x00…0 here, so the first input is C0 (it was $(A \cdot H) \oplus C0$ before), and the second input is $(C0 \cdot H) \oplus lenA||lenC$.

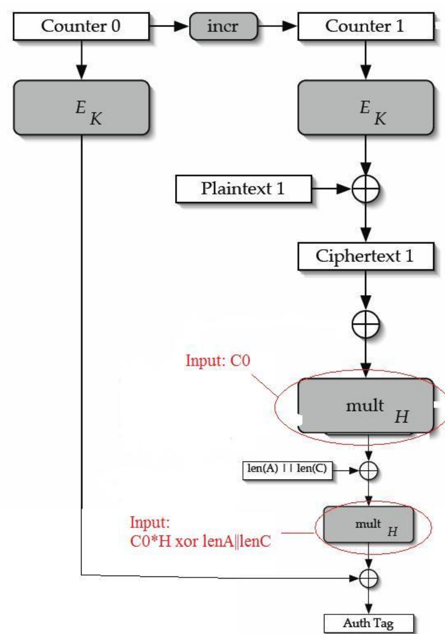

Fig 3.1 Final round attack model

To get cache collision: $< C0 > = < (C0 \cdot H) \oplus lenA||lenC >$, or after rearranging,

$$< H > = < (C_0 \oplus lenA||lenC) \cdot C_0^{-1} > \quad (3.3)$$

From (1), we find that $lenA||lenC$ is already fixed, so H can be found only through the ciphertexts. Plaintexts satisfying this equation should have a lower average encryption time due to the collisions, while the hard part is to calculate the inverse of C0 in $GF(2^{128})$. We implemented the code using extended Euclidean algorithm.

Chapter 4

RESULTS

Table 4.1 presents statistical data for the number of (C, t) pairs seen before the attack recovers a full 128-bit hash key H, from attacks against 10 random keys. The software platform is Openssl-1.0.0-beta3. For the 10 random keys, all the attacks on Intel 5110 succeeded, while there were 3 out of 10 failed on Intel 5520. The result shows that the latest Intel CPU has some complications which may be due to the hardware pre-fetch mechanism and out-of-order instruction execution.

Table 4.1 Median Sample and Successful Rate. Intel Xeron 5110 shows consistent successful rate while 70% were successful against Intel Xeon 5520 .

CPU	Median Sample	Successful Rate
Intel(R) Xeon(R) CPU 5110	$2^{21.5}$	100%
Intel(R) Xeon(R) CPU 5520	$2^{22.2}$	70%

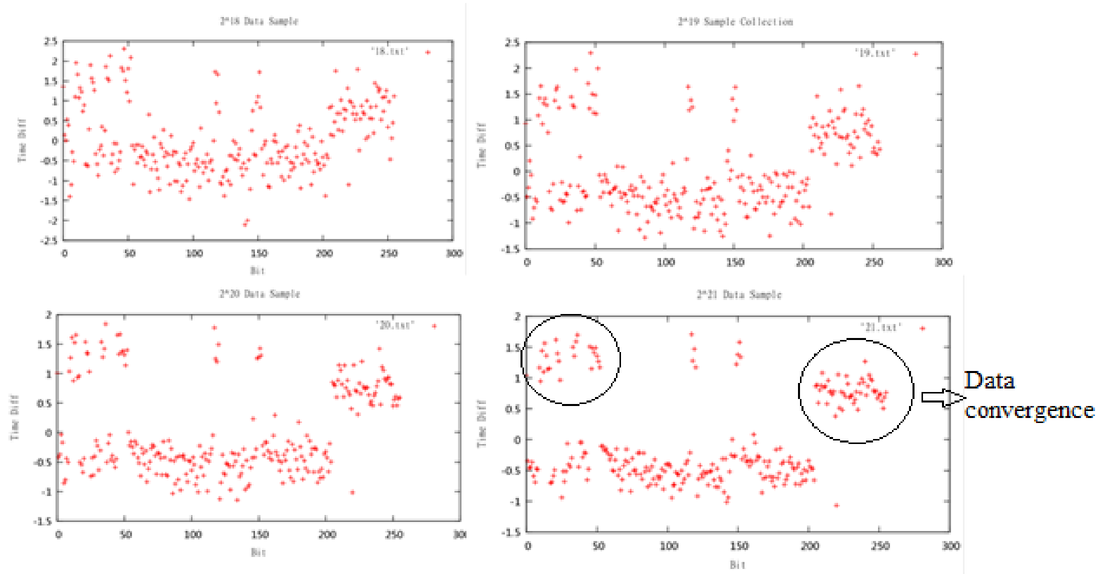

Fig 4.1 Data collection graphs from 2^{18} through 2^{21}samples, they show the convergence of one byte of the real key.

From 2^{18} through 2^{21}data sample collection, the lowest timing data varies, yet converge to the same point subsequently. From Fig 4.1, we can find that in 2^{18} data sample, the collection time gradually increases in the byte range of 150 to 250. In 2^{19} and 2^{20} data sample, however, the data block in the byte range of 200 to 250 is agglomerated more significantly and much higher than the time in the byte range of 150 to 200, indicating that the timing data converge to several blocks due to the effect of cache collision. Meanwhile, in 2^{18} data sample, we can find some timing data between 0 and 1 msec in the byte range of 0 to 50, while these data disappear in 2^{21} data sample. This infers that some correlated data gradually converge into a block rather than distribute uniformly. The least time happens when the byte is 220 (DC) in

2^{21} data sample, we'll run 2^{22} data sample once more to check if the least time condition holds.

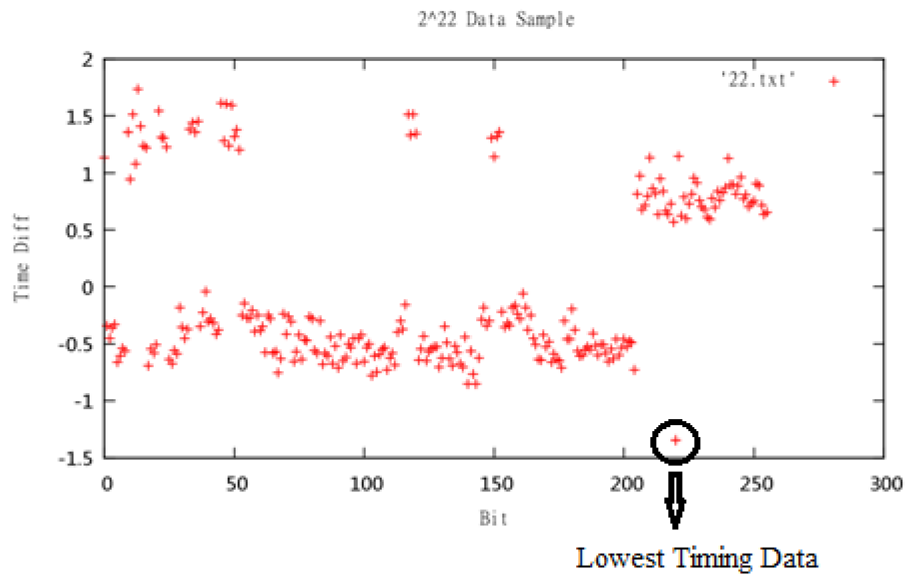

Fig 4.2 The data collection results from 2^{22} samples.

From Fig 4.2, after 2^{22} data sample collection, the lowest timing data locates at the same value 220, so the whole procedure stops and we conclude that the byte we mined is 220 (DC), meantime the pattern is classified significantly.

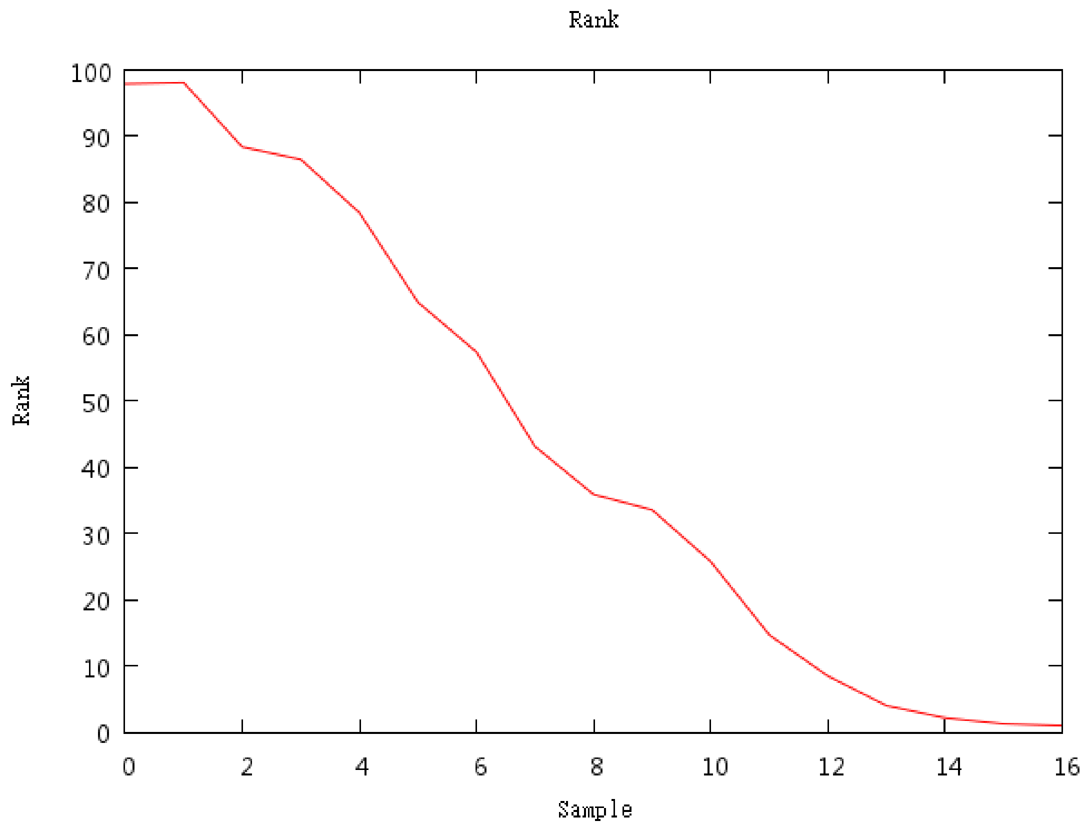

Fig 4.3 The variance of real key average ranking during each iteration

Fig 4.3 shows the average rank of the real key for each collecting iteration. Our research shows a solid improvement of the convergence running more samples. In principle, we conclude that the attack is robust due to its convergence.

Chapter 5

IMPLEMENTATION NOTES

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, no other work is done between encryptions.

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.

5.1 Cache Eviction

All of the attacks described in this paper require the AES-GCM lookup tables to be 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.

5.2 Extended Euclidean Algorithm

Currently there's no resource to support the expensive inverse calculation in GF(2^{128}). From our experiments, the extended Euclidean algorithm [10] is introduced. The extended Euclidean algorithm is an extension to the Euclidean algorithm for finding the greatest common divisor (GCD) of two integers. We define the polynomial $f = 1 + x + x^2 + x^7 + x^{128}$, and we also introduce the element a(x) whose inverse is desired, then an iterative method of the algorithm suitable for determining the inverse is given by the following.

```
remainder[1] := f(x)
remainder[2] := a(x)
auxiliary[1] := 0
auxiliary[2] := 1
i := 2
while remainder[i] > 1
   i := i + 1
   remainder[i] := remainder(remainder[i-2] / remainder[i-1])
   quotient[i] := quotient(remainder[i-2] / remainder[i-1])
   auxiliary[i] := -quotient[i] * auxiliary[i-1] + auxiliary[i-2]
inverse := auxiliary[i]
```

Chapter 6

COUNTERMEASURES AND CONCLUSION

Solutions requiring special hardware support are probably not practical, we cannot guarantee the encryption will take constant time without crippling performance. A more realistic approach may be software designed to prevent the data that is leaked from being useful. Köpf [11] gave algorithms that efficiently and optimally adjust the trade-off for given constraints on the side-channel leakage or on the efficiency of the cryptosystem. Kasper [12] presented the first constant-time implementation of AES-GCM thus offering a full suite of timing-analysis resistant software for authenticated encryption. However, some simple changes in the code will successfully achieve the countermeasures. The pseudo codes are listed as below:

Previous Pseudo Code:

```
static int GCM_mult_level(
                unsigned char *Z,
                unsigned char *X,
                unsigned char t[16][256][16])
{
  int i;
  unsigned char tmp[16];

  /* Everything has been precalculated, so just loop through each byte and
     the corresponding table and add the value at that point in the table
     onto the result in Z. */

  memcpy(tmp, X, 16);
  memset(Z, 0x00, 16);
  for (i = 0; i < 16; i++) {
```

```
    xor(Z, Z, t[i][tmp[i]], 16); //Using the 64kb table
  }
  return 0;
}
```

Modified Pseudo Code:

```
static int GCM_mult_level(
                              unsigned char *Z,
                              unsigned char *X,
                              unsigned char t[16][16])
{
  int i;
  int j;
  int tabi;
  int remi;
  unsigned char tmp[16];

  /* we need to shift everything over as we index the table (multiply).We
start at the high degrees and move down so that the values at the high
degree get multiplied over (shifted right) as we go.
*/
  memcpy(tmp, X, 16);
  memset(Z, 0x00, 16);
  for (i = 31; i > 0; i--) {
    tabi = (i & 0x01) ? tmp[i >> 1] & 0x0F : (tmp[i >> 1] >> 4) & 0x0F;
    xor(Z, Z, t[tabi], 16);

    remi = Z[15] & 0x0F;
    for (j = 31; j > 0; j--) {
      Z[j >> 1] = (j & 0x01)
     ? (Z[j >> 1] >> 4) & 0x0F
     : ((Z[(j >> 1) - 1] << 4) & 0xF0) | Z[j >> 1];
    }
    Z[0] &= 0x0F;
    Z[0] ^= rem_table_4bit[remi][0];
    Z[1] ^= rem_table_4bit[remi][1];
  } //Using exact calculation

  tabi = (tmp[0] >> 4) & 0x0F;
  xor(Z, Z, t[tabi], 16);

  return 0;
}
```

The original code function requires the pre-calculated 64kb table located in the cache to achieve the performance. By modifying the function using exact calculation, we can greatly increase resistance of the common AES-GCM implementation to last round attacks with no performance penalty by eliminating the special lookup table.

In principle, the attack in this paper only requires timing data and known plaintext and ciphertext. It makes clear the need for software AES-GCM implementations to protect against timing variation due to cached memory, although it remains to be seen if the timing data which could be obtained is accurate enough, also with some additional complications, especially for the latest CPU.

REFERENCES

[1] David A. McGrew, John Viega. The Galois/Counter Mode of Operation. 1997

[2] P. Kocher, J. Jaffe, B. Jun, "Differential Power Analysis," technical report, 1998

[3] Eli Biham, Adi Shamir: The next Stage of Differential Fault Analysis: How to break completely unknown cryptosystems 1996

[4] Dan Boneh, Richard A. DeMillo, Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults, Eurocrypt 1997

[5] F. Koeune and J.-J. Quisquater. A timing attack against Rijndael Tech. Jan. 1999

[6] Daniel Page. Theoretical use of cache memory as a cryptanalytic side-channel. June 2002

[7] Daniel J. Bernstein. Cache-timing attacks on AES. April 2005

[8] Joseph Bonneau Cache-Collision Timing Attacks Against AES 2006

[9] Onur Acıiçmez, Werner Schindler, Çetin K. Koç. Cache Based Remote Timing Attack on the AES, RSA 2007

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001

[11] Boris Köpf, Markus Dürmuth. A Provably Secure And Efficient Countermeasure Against Timing Attacks, IACR 2009

[12] Emilia Käsper, Peter Schwabe. Faster and Timing-Attack Resistant AES-GCM. CHES '09 Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems 2009