A NEW APPROACH FOR REDUCING CACHE TIMING ATTACK IN ADVANCED ENCRYPTION STANDARD USING DCF ALGORITHM

SOWMYA SREE DANNINA  
M.Tech Scholar  
Department of CSE,  
Chaitanya Engineering college,  
Visakhapatnam

K. RAJENDRA PRASAD  
Associate Professor  
Department of CSE,  
Chaitanya Engineering college,  
Visakhapatnam

Abstract:

Being accepted by many security related applications as the most secured cipher, The AES- Rijndael algorithm has a long successful existence in the field of global security. Despite of this, the AES is recently found to be broken theoretically. However, a few years back, Daniel Bernstein devised a cachetimingattack that was capable enough to break Rijndael’s seal that encapsulates the encryption key. Our paper proposes a new solution called Dynamic Cache Flushing (DCF) algorithm, which provides better security by encrypting key at a constant time over cache timing attack.

Keywords: AES, Timing attack, DCF

1. Introduction

In 1997, the National Institute of Standards and Technology put out a call for candidates to replace DES. There was particular emphasis on analyzing the candidates with respect to known attacks such as differential and linear cryptanalysis. The winning algorithm, which became the Advanced Encryption Standard (AES), was Rijndael. Rijndael was designed to be resistant against all known attacks.

A particular side-channel attack involves monitoring the movement of data into and out of cache. Recent advances in cache behavior analysis have shown that software implementations of AES are particularly vulnerable to cache attacks. A cache-timing attack is a subset of side channel attacks, which try to find flaws in the design of encryption algorithms. The attack takes advantage of these facts by measuring the time it takes to encrypt data with an unknown key and compares them to the time it takes to encrypt data with a known key, essentially creating two statistical profiles. Using statistical analysis of the profiles, it is possible to determine pieces of the unknown key and reduce the remaining key space to something that is more easily attacked using a brute force method. It has been prove that all 128-bit AES key can be recovered only through the first round attack using about 350 samples and two rounds attack using about 80 samples very efficiently[7].

2. Background Theory

2.1. The Advanced Encryption Standard (AES)

AES (Advanced Encryption Standard) is a symmetric algorithm in which encryption key is calculated from corresponding decryption and vice versa. AES has a fixed block size of 128 bits and a key size of 128, 192 or 256 bits. It operates on a 4x4 array of bytes, termed the state. There are 10 rounds for encryption with a 128 bit key, 12 for a 192-bit key and 14 for a 256-bit key. AES encrypts or decrypts the data block [1]. A block in AES is a group of 128 bits. Rijndael is an iterated block cipher whose key size can be 128, 192, or 256 bits based on rounds.
Number of round keys = Nr + 1

Each round consists of four different transformations namely SubBytes, ShiftRows, MixColumns, and AddRoundKey. Data block is referred as state. The State can be pictured as a rectangular array of bytes. State is made up of 16 bytes which is treated as matrices 4x4 bytes. In decryption process the reverse operations are executed in a slightly different order. AES [2] round composed of following four operations

SubBytes: A non-linear layer for resistance to differential and linear cryptanalysis. In the SubBytes step, each byte in the state is replaced with its entry in a fixed 8-bit lookup table;

\[ S; bij = S(aij). \]

ShiftRow: bytes in each row of the state are shifted cyclically to the left. The number of places each byte is shifted differs for each row. This causes diffusion of the bits over multiple rounds.

MixColumns: each column of the state is multiplied with a fixed polynomial \( c(x) \). This has a similar purpose to ShiftRow.

AddRoundKey — each byte of the state is combined with a byte of the round key using the XOR operation (\( \oplus \))[8].

The final round omits the MixColumns stage. In software implementations of AES, to speed up execution of the cipher, the SubBytes, ShiftRows and MixColumns steps are transformed into tables. These tables store pre-computed values, thus avoiding computation of the transforms during encryption and decryption. This is faster but uses more memory. At the beginning of encryption (and similarly for decryption) the input data (ni) is bitwise XORed with the secret key (ki). This is then used as an index into the SubBytes (S-Box) table, \( T_0[ni \oplus ki] \). This key-dependant lookup is particularly vulnerable to cache-based attacks.

2.2. Processor Cache

Cache memory is considerably faster in terms of access speed than main memory, and is used by the CPU to store the most frequently accessed memory locations. Each location in cache memory has an entry called a cache line. These cache lines comprise of an index, a tag and data. The index is a unique number of the particular cache line. The tag is the location of the data in main memory. And the data contains a duplicate copy of the data from main memory (see Figure 1)

![Figure 1 - main memory and cache memory](image)

When a CPU wishes to access data at particular memory location, it first checks if the memory location is in the cache. This is achieved by comparing the desired memory location against all the tags in the cache. If the memory location is contained in cache a cache-hit occurs. Otherwise the CPU must retrieve the data directly from main memory which takes longer. This is called a cachemiss.

3. Cache Timing Attack

Cache timing attack – the name speaks for itself. This belongs to a pattern of attacks that concentrates on monitoring the target cryptosystem, and analyzing the time taken to execute various steps in the cryptographic algorithm. In other words, the attack exploits the facts that every step in the algorithm takes a certain time to execute.

Although, the cache-timing attack is well-known theoretically, but it was only until April 2005 that a stout researcher named Daniel Bernstein [3, 4] published that the weakness of Rijndael can reveal timing information that eventually can be utilized to crack the encryption key. In his paper, Daniel announced a successful cache timing attack by exploiting the timing characteristics of the table lookups. Here is the simplest conceivable timing
attack on Rijndael. AES software implementations like Rijndael that uses look-up tables to perform internal operations of the cipher, such as Sboxes, are the one that are most vulnerable to this attack. For example, the variableindex array lookup T0[k[0] n[0]] near the beginning of the AES computation. A typical hacker might think that the time for this array lookup depends on the array index and the time for the whole AES computation is well correlated with the time for this array lookup. As a result, the AES timings leak information about k[0] n[0] and it can calculate the exact value of k[0] from the distribution of AES timings as a function of n[0]. Similar comments apply to k[1] n[1], k[2] n[2], etc. Assume, that the hacker watches the time taken by the victim to handle many n's and totals the AES times for each possible n[13], and observes that the overall AES time is maximum when n[13] is, say, 147. Suppose that the hacker also observes, by carrying out experiments with known keys k on a computer with the same AES software and the same CPU, that the overall AES time is maximum when k[13] n[13] is, say, 8. The hacker concludes that the victim's key k[13] is 147 8 = 155. This implies that a hacker can easily attack a variable time AES algorithm and can crack the encrypted data and eventually key [2]. Since in Rijndael algorithm all look up tables are stored in the cache, by putting another thread or some different way, attacker can easily get the encrypted data from the cache. Fig.2 shows that AES implementation in OpenSSL which does not take constant time. This was taken on a Pentium M processor. It is a 128 x 128 array of blocks where X axis shows one key for each row of blocks and Y axis shows one input for each column of blocks. Any combination of (key, Input) pair shows the encryption process for that particular pair by indicating the fix pattern of colors at that place. We can see the tremendous variability among blocks in Fig. 2. Due to this variability, attacker can easily determine the weak point, where the encryption took place by just analyzing the color pattern.

The cache timing attack problem has been tackled through various approaches [5]. Each solution has its own pros and cons. For instance, Intel released a set of compilers targeting their latest 64-bit processors. These compilers would take the C++ code as input and output a set of machine instructions that would not use CPU cache at all. In other words, the resultant code has a machine instruction that does not use CPU cache for temporary storage of data, in other words the cache is disabled automatically. The other suggestion was to place all the lookup tables in CPU registers rather than CPU cache, but this would affect performance significantly. Hardware approaches are also being considered. It has been suggested to have a parallel Field-Programmable Gate Array (FPGA) implementation or Application-Specific Integrated Circuits (ASIC) implementation with a separate coprocessor functioning with the existing CPU. This special coprocessor would contain special logical circuitry that would implement Rijndael. Timing attack can thus be avoided by barring other processes from accessing the special coprocessor [6].

4. Dynamic Cache Flushing (DCF) Algorithm

Numerous attempts have been made to address the timing attack loophole in AES. After a deep analysis of the logical steps involved in the Rijndael algorithm, we propose a novel technique to improvise the existing Rijndael algorithm. Our proposed algorithm follows variable-time AES algorithm by replacing it with a constant-time (but not high-speed) AES algorithm known as DCF (Dynamic Cache Flushing). Here, constant means totally independent of the AES key and input.
The resulting DCF algorithm would be capable enough to stand strong against the timing attacks. In order to determine the constant-time, first we need to collect timings and then look for input-dependent patterns. For example, we can repeatedly measure the time taken by AES for once (key; input) pair, convert the distribution of timings into a small block of colors, and then repeat the same color pattern for many keys and inputs. A constant-time AES algorithm would have the same block of colors for every key and input pair, as shown in Fig 3. Fig 3 is a 128 x 128 array of blocks. Here, X axis indicates the key for each row of blocks and Y axis shows the input for each column of blocks. The pattern of colors in a block reflects the distribution of timings for that (Key; Input) pair. Here, for all (Key, Input) pairs, the color patterns remains the same, due to the constant time. Hence, attacker cannot easily figure out at which point of time the encryption of key and data took place. DCF algorithm generates keys at a constant rate on today's popular dual-core CPUs.

DCF algorithm – as the name rightly suggests, flushes cache while the encryption of data is in progress. In other words, the data that is being copied by the program into the CPU cache during the encryption/decryption process is removed at periodic intervals. The major advantage of doing this is that, during a cache-timing attack, the spy process tries to tap the data stored in look up tables in the CPU cache. Since each instruction takes time to encrypt or decrypt the data, attacker can break the data by just taking difference of collected large body of timing data from the target machine for the plaintext byte and collected large body of reference timing data for each instruction. Fig. 2 shows that encryption/decryption takes place at random time and it can be easily determined by the spy process. If data in the CPU cache is flushed dynamically during the encryption or decryption process, it would make life more difficult for the spy process, when it tries to collect the data for sampling purposes. In addition, no data in the cache implies that there is no specific place or point that refers to the encryption process as shown in Fig. 3. It should be noted in Fig. 3 that the graph maintains a uniform pattern during the entire encryption/decryption process. Due to this uniformity, an attacker would face difficulty in tracking the exact time frame when encryption/decryption took place. This is possible by flushing the CPU cache at irregular intervals. Flushing the cache ensures that an attacker will not get enough insight into the data pattern during the encryption process by tapping the cache data. In order to increase the efficiency of this approach, one can increase the frequency of cache flushing. This would be a customizable parameter in the proposed DCF implementation. By further analyzing the DCF algorithm, it would lead to more “cache-misses” than “cache-hits”. The “cache-misses” would eventually be recovered by looking up into the RAM for data. The “cache-misses” is the performance penalty we pay with this approach. But with the computing capability we have today with the high-end dual core CPUs, this refetching of data elements from the RAM, can be dealt with. It should be noted that complete cache disabling is also an option [5], but in such scenarios the spy process might as well start tapping the RAM for encrypted data.Flushing the cache would rather confuse the spy process and

![Figure 3: Open SSL AES timings for 128 keys and 128 inputs on a Pentium M processor](image-url)

4.1 Description of the Proposed DCF Algorithm

The DCF algorithm is the improved version of Rijndael. In other words, the basic encryption/decryption process would remain unchanged. However, there are few additional steps injected into the Rijndael algorithm that would make it resilient to cache timing attack.
make life difficult for attackers to derive a fixed pattern of the timing information and encrypted data samples. Another feature intended in DCF algorithm is to implement random delays within the execution cycles during the encryption/decryption process. As a matter of fact that if bunch of the instructions from the encryption program repeats more than once, the execution time for those instructions remain constant all the time. By continuously monitoring the CPU instruction cycles, attacker can determine the time taken to execute a step in encryption algorithm. Attacker might be able to capture the entire process timeline and data patterns being encrypted or decrypted. In DCF, additional delays could be introduced while the algorithm steps are in progress. This would change the encryption/decryption timeline and make the algorithm more unpredictable. As a result, attacker will not be able to guess the timing pattern created by the encryption/decryption steps. Every time when the proposed DCF algorithm generates a unique timing pattern for encrypting the set of data, it makes things more difficult for an attacker who uses a key parameter (i.e., the time taken to encrypt a set of data) in his predictable brute-force approach for cracking the key. The delays in DCF could be made more unpredictable by randomizing the numeric values that defines the amount of delay caused. A good sturdy randomizer could achieve a fairly unpredictable pattern of Fig. 2 Open SSL AES timings for 128 keys and 128 inputs on a Pentium M processor and Fig. 3 AES timings using Constant-Time AES algorithm, for 128 keys and 128 input delays. The cache timing attack exploits the effect of memory access on the cache, and would thus be completely lessened by an implementation that does not perform any table lookups. Instead of avoiding table lookup, one could employ them by ensuring that the pattern of accesses to the memory is completely independent of the data passing through the algorithm. In its easiest form, is created, implementing a memory access for a relevant set of data, one can read all the data from the look-up table. In addition, one could use an alternative description of the cipher which replaces the table lookups by an equivalent series of the logical operations. For AES, this is particularly ideal since the lookup tables have concise algebraic descriptions, but performance is degraded by over an order of magnitude [5]. Flushing cache, random delays, and making data access independent of underlying data being processed, would make sense only if the DCF program is forced to run on a single thread. Single thread would also ensure that less data is being exposed to the spy process at any given point of time.

4.2 Simulation Results
Here is a brief description of DCF during execution of Rijndael algorithm. Assume that there is a huge data file that’s being encrypted using the DCF algorithm. The flowchart in Fig. 4 would portray a logical flow of events. A huge file is read into a user-defined variable, “buffer”.

![Dynamic Cache Flushing Algorithm Flowchart](image_url)
The password provided by the user is typically stored as the encryption key. Rijndael initializes itself by building the set of round tables and table lookups into its data structure which helps in processing the data in buffer. A timer is initialized just before Rijndael starts encrypting the data in the buffer. The time should be initialized in nanoseconds. During encryption, Rijndael puts the key and data together in the round operation. During various steps in the encryption process, the random delays are introduced using `Sleep(X)` function to ensure that the repeated set of instructions does not portray the same execution timeline. Here, the amount of time, the process needs to be suspended ‘X’, is directly proportional to the total amount of time ‘T’ taken to process the chunk of data of size ‘S’. If the timer becomes zero, flush or remove the data from the cache by using the `cacheflush()` function. The timer would be initialized with a random time that would make the encryption process time more unpredictable for the hacker. Reinitialize the timer with a random time and perform the encryption with random delay until all the data is processed (encrypted).

5. Conclusions:

Cache timing attack can be dangerous. Almost all computing systems have shared caches at some level and will be susceptible to this attack. The attacker process can be unprivileged and can even be a remote client. Main weaknesses detected in Rijndael algorithm are heavy use of table lookups which dominate the running time and table lookup indices are easily related to single plaintext and key bytes. The DCF algorithm discussed in this paper is nothing but - an improved Rijndael algorithm. DCF algorithm tries to simulate a scenario wherein the table lookups are accessed in constant-time rather than variable-time. This would disable any attacker from writing a spy program to brute force the key and data out of the cache data stored during execution of the DCF algorithm. In the implementation of DCF algorithm, cache is flushed periodically during encryption or decryption time. This would disable the attacker from tapping the cache for data. The DCF algorithm stands strong against the cache timing attack.

6. References:


7. About the Authors:

Sowmya Sree Dannina is currently pursuing her M.Tech in Computer Science and Engineering at Chaitanya Engineering College Kommadi, Visakhapatnam.

K.Rajendra Prasad is currently working as an Associate Professor in Computer Science and Engineering department, Chaitanya Engineering College Kommadi, Visakhapatnam. His research interests include Network Security, Data Mining.