Cache Attack on AES for Android Smartphone*

Bo Li
School of Computer Science and Engineering
Beihang Univeristy
Beijing, China
(+86)82338059
leborn@buaa.edu.cn

Bo Jiang†
School of Computer Science and Engineering
Beihang Univeristy
Beijing, China
(+86)82338059
jiangbo@buaa.edu.cn

ABSTRACT

Cache attack is mainly based on information leakage through the timing difference between cache hit and miss. It is an effective technique to attack AES implementations on x86 platform. However, since the cache architecture, instruction set of smartphone is different from that of the Intel x86 platform, effective cache attack on AES implementations for smart phones still faces several challenges. In this work, we realized an effective cache attack on AES implementations for Android smart phone based on the *Prime+Probe* strategy. We also proposed to use K-S statistical test to help rank private key assumptions in noisy execution environment. Our results show that cache attack on ASE implementations for Android platform is practical and countermeasures are needed to ensure mobile security.

CCS Concepts

Security and privacy → Mobile platform security;

Keywords

Cache attack; Android; AES; Side channel attack.

1. INTRODUCTION

With the rapid development of mobile Internet in recent years, mobile phones and other mobile devices have become an indispensable part of our lives. At one side, it makes our life more convenient: we can send mails, store personal data, communicate with each other, transfer money and do business with our smart phone. On the other side, the smart phone also brings security threats to us. Our personal data, bank account, confidential conversations are all subject to attack from the mobile phone. Therefore, with the prevalence of smart phones, more and more attention has been paid to their security problems.

The smart phone manufacturers and the Android operating system have used various measures to enhance mobile security, including trusted execution environment (TEE), virtual memory management, permissions management, etc. However, due to complexity of Android-based smart phones, security vulnerabilities are inevitable.

In modern computer architecture, although different processes are isolated in their own virtual address spaces, they still shared the same L2 cache. When the memory access time can be measured precisely, the information on Cache hit/miss will leak critical application execution information. Cache attack is such techniques that utilize precise memory access timing information to attack an application.

Cache attack can by synchronous or asynchronous. For synchronous attack, the attackers can directly invoke the interfaces of the application under attack through shared memory spaces. For asynchronous attack, there is no shared memory space between the attacker and the victim. And the attacker can only passively observe the memory access behaviors during execution. In another word, there is no direct interaction between the attacker and the victim when cache attack is conducted, and they execute in parallel on the same or different kernels to access their own address space. Therefore, the attacker does not need extra permissions.

In the last 10 years, more and more attention has been paid to cache attacks techniques. Kocher et al. [1] proposed a method to decipher the encryption algorithm in the computer by analyzing the information leaked by the cache at runtime. This idea has evolved rapidly under the attention of computer security professionals.

ICCSP 2018, March 16–19, 2018, Guiyang, China

In recent years, the cache attack technique has been applied on the Intel x86 platform. Cache attack techniques are used to monitor user keyboard inputs and to recover AES encryption keys, For example, the possibility of cross-process information leakage via cache attack was first proposed by Hu in 1992 [1] in the context of intentional transmission via covert channels. In 1998, Kelsey et al. [2] discussed the possibility of “attacks based on cache hit ratio in large S-box ciphers.” In 2002, Page [3] described theoretical attacks on DES via cache misses, assuming an initially empty cache and the ability to identify cache states with very high temporal resolution in side-channel traces. Tsunoo et al. [4] proposed a timing-based attack on DES, exploiting the effects of collisions between the various memory lookups invoked internally by the cipher. Furthermore, Gruss et al. [5] demonstrated the possibility to automatically exploit cache-based side-channel information based on the Flush＋Reload approach. Besides attacking cryptographic implementations like AES T-table implementations, they showed how to infer keystroke information and even how to build a keylogger by exploiting the cache side channel.

However, the CPUs of Android smart phone usually adopts the ARM architecture, which is different from the Intel x86 architecture in terms of instruction set, cache organization mode and cache replacement strategy. Therefore, effective cross-core cache attack on non-root mobile phones emerged until recently. Moritz Lipp et al. [6] proposed a cross core attack model for ARM processors without requiring root permissions. These models can effectively acquire privacy information based on statistical analysis of cache timing information leakage. However, there is no detailed implementation AES attacks on the Android platform. Because the cache structure of smart phones is different from Intel x86 architecture, the cache attack method of Intel x86 platform must be adapted for mobile platform. Frist, the cache replacement policy on the x86 platform uses the LRU strategy, so eviction of specified cache sets to memory is straightforward. However, since Android uses a pseudo random replacement policy, we need additional measures to evict data from a specified cache sets into memory. In addition, in order to obtain stable data access time, it is usually necessary to preheat the access memory or the cache operation. The previous attacks usually use the first access to cache or memory access time to measure cache hit or not, so it is easy to introduce errors, resulting in unsatisfactory results.

2. BACKGROUND

2.1 Determining the Best Eviction Strategy

The first thing to do is to determine the best eviction strategy. On Intel x86 platforms, we can use the cflush instruction to evict cache lines to memory. Although similar instructions exist on some of the android devices, they are only useful in privilege mode. So we need a more general strategy to evict the content of specified cache to memory.

In our design, we adopt the continuous address access strategy proposed in [6]. Continuous addresses access is a general strategy, which reads data from addresses that can be mapped into the same cache sets to evict the data in the cache. Although we can read large amount of addresses to guarantee eviction of data from cache, a large number of memory access operations will not only increase time, but also increase the related memory storage for addresses. Furthermore, since the cache of the mobile device uses pseudo random replacement strategy, continuous reading of a plurality of memory data does not guarantee the eviction cache content. We make use of the method proposed by Moritz Lipp et al. [6] to automatically generate eviction strategies and test its applicability. We have evaluated a large number of eviction patterns on our Lenovo k51c78 mobile phone, and identified the best eviction strategy.

2.2 The Prime + Probe Strategy

In order to obtain privacy information through the cache, the attacker must have the ability to get cache state. The prime＋probe approach allows an adversary to determine the cache sets used by the victim’s computation. It consists of the following three basic steps.

Prime＋Probe:

1. Occupy specific cache sets.
2. Victim program is scheduled.
3. Determine which cache sets are still occupied.

The following steps detail the three stages of Prime＋Probe. First, for a specified set, the Prime phase maps data to the cache and the previous cached data is evicted into memory. Then the victim program is executed. During its execution, the memory which was accessed may be mapped to some sets of the cache and took part of the line in the set. The last step is Probe, which checks whether the data putted into cache in the Prime phase is still in cache. It measures the access time for memory addresses. If the memory access time is large, it indicates a cache miss has occurred. Therefore, the victim program is highly probably to have accessed memory addresses mapped to the cache.

2.3 Precise Measurement of Time

An accurate timing method is crucial for successful cache attack. Because cache attack techniques must use time to distinguish cache hit and cache miss, which in turn helps the attacker to pinpoint which cache sets have been used by the victim program. In order to distinguish the situation between cache hit and cache miss, timing sources or dedicated performance counters can be used. Moritz Lipp et al. [6] has proposed several non-privileged timing methods. However, these interfaces are not supported on all Android versions and all processors. Therefore, it is necessary to determine the effective timing method that can accurately and stably measure memory access time for the device under attack. Besides reading the CPU register to obtain the CPU cycle to measure the time, there are three other measures. The first is clock_gettime syscall, which is the timing function of nanoseconds accuracy. The second is the *Perf* performance analysis tool. The availability of this feature depends on the Android kernel configuration. The third is a thread timing simulator. If no interface with sufficient accuracy is available, an attacker can run a thread that increments a global variable in a loop, providing a fair approximation of a cycle counter. In this work, we firstly evaluate the availability of different timing approach for our phone model. Then we choose the more precise one among those available approaches.

3. ATTACKING AES ALGORITHMS

3.1 The First Round Attack

In this paper, we focus on attacking AES implementations whose memory access patterns are particularly susceptible to cryptanalysis. Because many implementations of AES on 32-bit processors are based on lookup tables, as prescribed in the Rijndael specification [7][8]. In our cases, the victim AES implementation uses 8 T-tables, T_0, T_1, T_2, T_3, and $T_0^{(10)}$, $T_1^{(10)}$, $T_2^{(10)}$, $T_3^{(10)}$, and each T-table contains 256 4-byte words. During the process of AES encryption, key $\mathrm{k} = (k_0, \dots, k_{15})$ is expanded into 10 round keys $K^{(r)}$ for $\mathrm{r} = 1, \dots 10$. Each round key is divided into 4 words of 4 bytes each: $K^{(r)} = (K_0^{(r)}, K_1^{(r)}, K_2^{(r)}, K_3^{(r)})$. The 0th key is just the original key

$$K^{(0)} = (k_0, k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8, k_9, k_{10}, k_{11}, k_{12}, k_{13}, k_{14}, k_{15}).$$

Given a 16-byte plaintext $\mathrm{p} = (p_0, \dots, p_{15})$, the AES encryption progress need to calculate 4 intermediate index for each T-table at each round. The initial indices $x^{(0)}$ can be computed by $x^{(0)} = p_i \oplus k_i$ for $\mathrm{i} = 0, \dots, 15$. Then, the first 9 rounds are computed by calculating the intermediate state as follows, for round $\mathrm{r} = 0, \dots, 8$:

$$(x_0^{(r+1)}, x_1^{(r+1)}, x_2^{(r+1)}, x_3^{(r+1)}) \leftarrow T_0[x_0^{(r)}] \oplus T_1[x_5^{(r)}] \oplus T_2[x_{10}^{(r)}] \oplus T_3[x_{15}^{(r)}] \oplus K_0^{(r+1)}$$

$$(x_4^{(r+1)}, x_5^{(r+1)}, x_6^{(r+1)}, x_7^{(r+1)}) \leftarrow T_0[x_4^{(r)}] \oplus T_1[x_5^{(r)}] \oplus T_2[x_{14}^{(r)}] \oplus T_3[x_3^{(r)}] \oplus K_1^{(r+1)}$$

$$(x_8^{(r+1)}, x_9^{(r+1)}, x_{10}^{(r+1)}, x_{11}^{(r+1)}) \leftarrow T_0[x_8^{(r)}] \oplus T_1[x_{13}^{(r)}] \oplus T_2[x_2^{(r)}] \oplus T_3[x_7^{(r)}] \oplus K_2^{(r+1)}$$

$$(x_{12}^{(r+1)}, x_{13}^{(r+1)}, x_{14}^{(r+1)}, x_{15}^{(r+1)}) \leftarrow T_0[x_{12}^{(r)}] \oplus T_1[x_1^{(r)}] \oplus T_2[x_6^{(r)}] \oplus T_3[x_{11}^{(r)}] \oplus K_3^{(r+1)}$$

However, in the last round, AES replaces T_0, T_1, T_2, T_3, to $T_0^{(10)}$, $T_1^{(10)}$, $T_2^{(10)}$, $T_3^{(10)}$, and the result $x^{(10)}$ is the cipher text.

We can only distinguish a cache hit from cache miss by measuring memory access time. Furthermore, the structure of many smartphones' cache set is associative mapping. Such a cache consists of storage units called cache lines, each consisting of B bytes. The cache is organized into S cache sets, each containing W cache lines, so overall the cache contains $B * S * W$ bytes. The size of one T-table entry is 4 bytes and the cache line size B is usually 64 bytes, so every cache line can cache 16 T-table entries. If two T-table entries map to the same cache line, we say that the two entries are correlative. However Prime＋Probe only can evict and occupy a specified cache set, and then measure the Probe time. Since the cache set is the smallest eviction unit, we could not just evict and occupy a cache line of a specified set. So it is impossible to distinguish two different byte addresses or cache T-table entries mapped to the same cache set only by having knowledge of cache hit or cache miss. Since every cache set line can cache 64 bytes, and T-table entry is 4 bytes, 16 continuous T-table entries map to the same cache set if the start address of first entry exactly map to the start of cache line.

In this paper, we use Prime＋Probe technique to efficiently extract the full key. The approach is divided into the first round and the second round attack. Given a 16 byte key K = (k_0 ,...., k_{15}), it will be extended to the 10 round internal keys $K^{(\mathrm{r})}$ for r=1,... 10 in AES encryption process. Each round key is divided into 4 words of 4 bytes each: $K^{(r)} = \left(K_0^{(r)}, K_1^{(r)}, K_2^{(r)}, K_3^{(r)}\right)$. In the first round attack, we successfully gain the first 4 bits of each byte. The first round attack is based on the accessed indices of T-table which can be calculated simply through the key and plaintext namely $x^{(0)} = p_i \oplus k_i$ for all key index $\mathrm{i} = 0, \dots, 15$. Thus, if we know the value of plaintext byte p and any information on the accessed index $x^{(0)}$, we can directly translate these knowledge to information on key byte k . In the progress of AES encryption, the first round of access index of T-table can be obtained by the plaintext and key through XOR operation.

In this work, in order to obtain the key bits of the AES, we guess the value of the key firstly and then verify that whether they are the exact true key bytes. In the hypothesis process, we only need to assume the value of the first 4 bits of each byte because the last 4 bits could not be distinguished in the first round attack. There are two main step in the process of hypothesis testing, the first step is to guess the value of key bits of each byte, in which we should enumerate the possible value of the first 4 bits for every key byte separately from 0 to 15. In the second step, we can calculate the index i and the T-table index based on known plaintext and the guessed key. And then we can calculate the cache set index j, which T-table data in index i should map to. Finally we inspect whether the cache set j had been used in the process of AES encryption by Prime＋Probe. Finally, we used the K-S statistical test to determine whether the two distributions are the same or not. The data of the first distribution measures the time of Prime. While samples of the second distribution are the time of Probe. If the K-S test concludes that the two samples belong to different distribution, we can say that the victim AES implementation has accessed the specified sets during execution.

3.2 The Second Round Attack

The First-Round attack can only narrow each key byte down to 1/16 possibility, but the table lookups in the first round cannot reveal more information. In our experiment, the AES key contains 16 bytes, so there are still 64 unknown bits to search for. The second round attack mainly relies on the nonlinear relationship between the plaintext, the key and the cipher.

The indices accessed in the second round of encryption are not apparent as in the first round. We exploit the following equations derived from the Rijndael specification, which give the four indices of table lookups in the 2nd round. We can get 2nd access indices as follows:

$x_2 = s(p_0 \oplus k_0) \oplus s(p_5 \oplus k_5) \oplus 2 \bullet s(p_{10} \oplus k_{10}) \oplus 3 \bullet s(p_{15} \oplus k_{15}) \oplus s(k_{15}) \oplus k_2$
$x_5 = s(p_4 \oplus k_4) \oplus 2 \bullet s(p_9 \oplus k_9) \oplus 3 \bullet s(p_{14} \oplus k_{14}) \oplus s(p_3 \oplus k_3) \oplus s(k_{14}) \oplus k_1 \oplus k_{15}$
$x_8 = 2 \bullet s(p_8 \oplus k_8) \oplus 3 \bullet s(p_{13} \oplus k_{13}) \oplus s(p_2 \oplus k_2) \oplus s(p_7 \oplus k_7) \oplus s(k_{13}) \oplus k_0 \oplus k_4 \oplus k_8 \oplus 1$
$x_{15} = 3 \bullet s(p_{12} \oplus k_{12}) \oplus s(p_1 \oplus k_1) \oplus s(p_6 \oplus k_6) \oplus 2 \bullet\ s(p_{11} \oplus k_{11}) \oplus s(k_{12}) \oplus k_{15} \oplus k_3 \oplus k_7 \oplus k_{11}$
Here, s(•) denotes the Rijndael S-box function, and • denotes multiplication over GF(256).
The indices x_2, x_5, x_8, and x_{15} are indices of T-table 2, 1, 0, and 3, respectively. These indices are crucial to achieve AES attack. In this round, AES attack is performed by enumerating all possible values of all keys and observing the collected timing information statistically. Hypothesis testing is performed to identify a correct guess. For each candidate guess, we collect its timing information as samples, and then we perform statistical K-S test to help identify the best guess.

Since we already have grained the first 4 bits of each key byte, there are only $4*16$ bits needed to guess in round 2. Furthermore, since 4 key bytes together are needed to determine a second round T-table index. We must guess 4 half bytes simultaneously in second round hypothesis testing. For each guess, we can get a measurement score m. Based on m, we can determine which guess has the highest probability. Then we can get the index in T-table when AES encryption is executed by plaintext p with key k. Because the T-table access index reflects the cache set index, T-table access index may in turn lead to different access time for Prime＋Probe operation.

3.3 The K-S Statistical Test

In statistics, the Kolmogorov－Smirnov test [9] (K－S test) is a nonparametric test of the equality of continuous, one-dimensional probability distributions. It can be used to compare a sample with a reference probability distribution (one-sample K－S test), or to compare two samples (two-sample K－S test). It is named after Andrey Kolmogorov and Nikolai Smirnov. The two Sample K-S test is one of the most useful and general nonparametric methods for comparing two samples.

In this work, we adopt the K-S test to determine whether the victim program accessed the specified sets of cache when attack is scheduled. At this point, two samples that are crucial in cache attack are ready. The first sample is the collection of measurement data of Prime followed directly by the Probe operation. The second sample is the collection of measurement data of Prime operation, the memory access of victim, and the probe operation. If the memories accessed during the execution of the victim program are not mapped to the specified cache sets, the access time of the two samples should be similar and the result of the K-S test must confirm that the two samples belong to the same distribution. On the other hand, if the memories accessed during the execution of the victim program are mapped to the specified set, the time in the second sample will be greater than the time in the first sample in distribution. So with the help of K-S test we can automatically check whether the two samples belong to a same distribution or not.

4. EXPERIMENTAL STUDY

Table 1. Lenovo K51c78 information

System	Cache Size	Processor	Cache details
Android 5.0	512 KB	ARM MT6753 8 cores	512 sets 16-way set associative

In this section, we will perform the attack based on our proposed approach on Android phone. This paper mainly implements the side channel attack on AES at Lenovo K51c78. The details property of Lenovo K51c78 is shown in Table 1. We can see in Table 1 that the phone we use in the experiment is based on 8-core arm processor with 512KB of cache. The operating system of the phone is Android 5.0.

The attack processes can be divided into three steps: the preparation phase, the first round attack and the second round attack. The preparation phase mainly focused on finding the precise time of cache hit and cache miss thresholds and finding fast and effective eviction strategies for the target smartphone. Then we performed the first round and the second round of attack as described in previous sections. Within each round of attack, we obtain two samples for K-S test. Finally, we perform K-S test to compare the sample distributions. In order to obtain all key bit we firstly enumerate all possible value of AES bytes and then obtain the related measurement score. The measurement score reflects the suspiciousness of a certain key guess. So we can sort the guessed key based on the measurement score. To reveal all AES key, the first round attack is performed and the first four bits of each byte are revealed successfully.

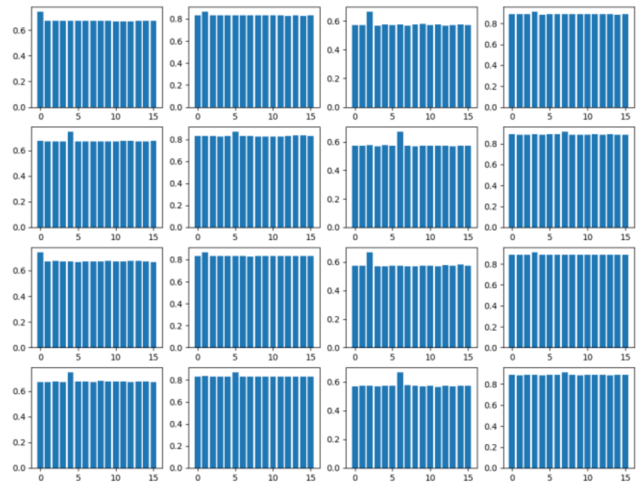

Figure 1. The Result of First Round Attack.

As shown in Figure 1, we can get the first 4 bits for each key byte (16 in total). Each diagram represents the first 4 bits of a key byte, plotted by 4-bit values (0-15) on the horizontal axis and its K-S measurement scores on the vertical. We can verify the first 4-bit values are exactly the first 4-bit key values for AES encryption.

In order to obtain the whole key bytes of the AES key, we conduct the second round of attack. Through the hypothesis testing of the remaining key bits, we calculate the corresponding cache set access indices, and then get the K-S value for the corresponding indices as the measurement score. Finally, we sort the score and find the highest score. We confirm again the assumption key bytes related to the highest measurement score are exact the key values used for AES. As shown in Figure 2-5, each diagram represents the last 4 bits of 4 key bytes, plotted by 4-bit values on the y-axis and measurement scores on the x-axis. As there is a large amount of combinations for every 4 half key bytes, only combinations gaining the highest measurement score are shown. Figure 2 shows the measurement scores of key byte indices 0, 5, 10, 15. So we can conclude that the hypothesis values for indices 0, 5, 10, 15 are separately 0x0, 0x5, 0x2, 0x7. Figure 3, Figure 4, Figure 5 can be interpreted similarly like Figure 2. So, we can get the last 4-bit value (0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7) for indices 0-15, which are exactly the real 4-bit key values, used for AES encryption.

The experiment results are shown as follows:

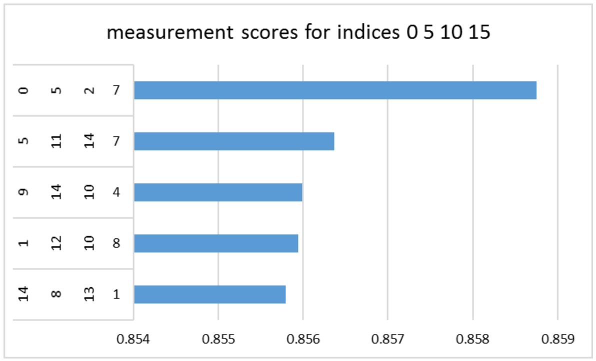

Figure 2. The Result of the Second Round Attack(0, 5, 10, 15).

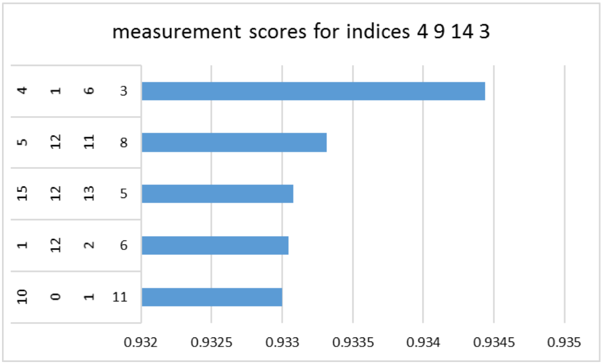

Figure 3. The Result of the Second Round Attack(4, 9, 14, 3).

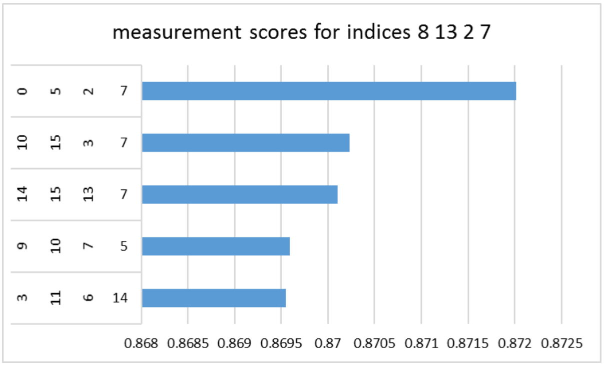

Figure 4. The Result of the Second Round Attack (8, 13, 2, 7).

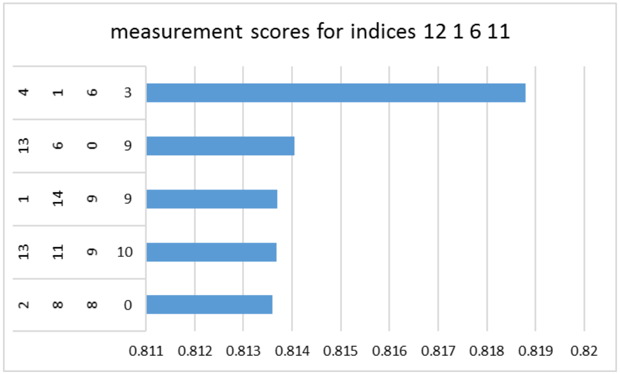

Figure 5. The Result of the Second Round Attack (12, 1, 6, 11).

As shown in Figure 2 to Figure 5, we can obviously get the last four bits of each key. The y-axis represents the 4 possible hypothesis values for the corresponding key byte indices. The x-axis represents their corresponding K-S scores. We only show combinations graining the highest measurement score. Figure 2 shows the measurement scores for key byte indices 0, 5, 10, 15. So we can conclude that the hypothesis values for indices 0, 5, 10, 15 are 0x0, 0x5, 0x2, 0x7, respectively. Figure 3 to Figure 5 can be interpreted similarly.

5. RELATED WORK

In this section, we review the closely related works.

Bernstein [10] exploited the total execution time of AES T-table to implement cache timing attacks. Percival [11] and Osvik et al. [12] proposed more fine-grained exploitations of memory accesses to the CPU cache than Bernstein's. Osvik et al.[12] proposed two concepts, namely *evict+time* and *prime+probe*, to determine which specific cache sets were accessed by a victim program. Yarom and Falkner [13] proposed *Flush+Reload* to attack cryptographic implementations and to build cross-VM covert channels [14], which are significantly more fine-grained attack that exploits three fundamental concepts of modern system architectures. Gruss et al. [15] proposed the method of *Evict+Reload*, which uses eviction instead of flush instruction. However, this method has been implemented on Intel platform.

Wei et al. [16] proposed that ARM architecture is different from Intel platform, the attack methods proposed for Intel x86 CPUs are not exactly suit for smartphones CPUs, so it is harder to hold a cache attack on Android devices. Spreitzer and Plos [17] proposed that although the cache attacks on mobile devices are more difficult, there are still information leaked by investigated the applicability of Bernstein's attack on smartphone. Similarly, Oren et al. [18] demonstrated the possibility to exploit cache attacks on Intel platforms from JavaScript and showed how to infer visited websites and how to track the user's mouse activity.

6. CONCLUSION

Cache attack was a successful technique to realize cross-process information leakage on Intel x86 platform. However, modern smartphones use one or more multi-core ARM CPUs that have a different cache organization from Intel x86 CPUs. Therefore, cache attack on Android smartphone poses different challenges. In this work we have realized a successful cache attacks on AES implementations based on Prime＋Probe and K-S statistical test. We have successfully recovered all AES key bytes after two rounds of attack. Furthermore, these attacks are practical as no privileged permissions are required. For future work, we will further explore asynchronous attacks on AES implementations to further enhance the applicability of our work.

7. ACKNOWLEDGMENTS

This research is supported in part by NSFC (project no. 61772056 and 61690202), the Research Fund of the MIIT of China (project no. MJ-Y-2012-07), and the research fund of the State Key Laboratory of Virtual Reality Technology and Systems.

8. REFERENCES

[1] P. C. Kocher, Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems, in Advances in Cryptology-CRYPTO’ 96, pp. 104 – 113, Springer, 1996.

[2] J. Kelsey, B. Schneier, D. Wagner, C. Hall, Side channel cryptanalysis of product ciphers, in Proc. 5th European Symposium on Research in Computer Security. Lecture Notes in Computer Science, vol. 1485, Berlin, pp. 97 – 110, Springer, 1998.

[3] D. Page, Theoretical use of cache memory as a cryptanalytic side-channel. Technical Report CSTR-02-003, Department of Computer Science, University of Bristol, 2002.

[4] Y. Tsunoo, E. Tsujihara, K. Minematsu, H. Miyauchi, Cryptanalysis of block ciphers implemented on computers with cache, in Proc. International Symposium on Information Theory and Its Applications, pp. 803 – 806, 2002.

[5] D. Gruss, R. Spreitzer, S. Mangard, Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In USENIX Security Symposium, USENIX Association, pp. 897 – 912, 2015.

[6] M. Lipp, D. Gruss, R. Spreitzer et al, ARMageddon: Cache Attacks on Mobile Devices. Mundo Electrónico, 6(1): pp 60-65, 2016.

[7] J. Daemen, V. Rijmen, AES Proposal: Rijndael, version 2, AES submission, 1999. http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf

[8] J. Daemen, V. Rijmen, The Design of Rijndael: AES—The Advanced Encryption Standard, Berlin, Springer, 2001.

[9] W. Feller, On the Kolmogorov – Smirnov Limit Theorems for Empirical Distributions. Selected Papers I. Springer International Publishing, pp. 177-189, 2015.

[10] D. J. Bernstein, Cache-Timing Attacks on AES, 2004. URL:http://cr.yp.to/papers.html#cachetiming.

[11] C. Percival, Cache Missing for Fun and Profit, 2005. URL: http://daemonology.net/hyperthreadingconsidered-harmful/.

[12] D. A. Osvik, A. Shamir, E. Tromer, Cache Attacks and Countermeasures: The Case of AES. In Topics in Cryptology – CT-RSA, vol. 3860 of LNCS, pp. 1 – 20, Springer, 2006.

[13] Y. Yarom, K. Falkner, FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In USENIX Security Symposium, USENIX Association, pp. 719 – 732, 2014.

[14] C. Maurice, C. Neumann, O. Heen, A. Francillon, C5: Cross-Cores Cache Covert Channel. In Detection of Intrusions and Malware, and Vulnerability Assessment – DIMVA, vol. 9148 of LNCS, pp. 46 – 64, Springer, 2015.

[15] D. Gruss, R. Spreitzer, S. Mangard, Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In USENIX Security Symposium, USENIX Association, pp. 897 – 912, 2015.

[16] M. Weiss, B. Heinz, F. Stumpf, A Cache Timing Attack on AES in Virtualization Environments. In Financial Cryptography and Data Security – FC, vol. 7397 of LNCS, pp. 314 – 328, Springer, 2012.

[17] R. Spreitzer, T. Plos, On the Applicability of Time-Driven Cache Attacks on Mobile Devices. In Network and System Security – NSS, vol. 7873 of LNCS, pp. 656 – 662, Springer, 2013.

[18] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, A. D. Keromytis, The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In Conference on Computer and Communications Security – CCS, ACM, pp. 1406 – 1418, 2015.