Cache-Attacks on the ARM TrustZone implementations of AES-256 and AES-256-GCM via GPU-based analysis

Ben Lapid and Avishai Wool

School of Electrical Engineering, Tel Aviv University, ISRAEL
ben.lapid@gmail.com, yash@eng.tau.ac.il

Abstract. The ARM TrustZone is a security extension which is used in recent Samsung flagship smartphones to create a Trusted Execution Environment (TEE) called a Secure World, which runs secure processes (Trustlets). The Samsung TEE includes cryptographic key storage and functions inside the Keymaster trustlet. The secret key used by the Keymaster trustlet is derived by a hardware device and is inaccessible to the Android OS. However, the ARM32 AES implementation used by the Keymaster is vulnerable to side channel cache-attacks. The Keymaster trustlet uses AES-256 in GCM mode, which makes mounting a cache attack against this target much harder. In this paper we show that it is possible to perform a successful cache attack against this AES implementation, in AES-256/GCM mode, using widely available hardware. Using a laptop's GPU to parallelize the analysis, we are able to extract a raw AES-256 key with 7 minutes of measurements and under a minute of analysis time and an AES-256/GCM key with 40 minutes of measurements and 30 minutes of analysis.

1 Introduction

1.1 Motivation

The ARM TrustZone [1] is a security extension helping to move the "root of trust" further away from the attacker. TrustZone is a separate environment that can run security dedicated functionality, parallel to the OS and separated from it by a hardware barrier. Recent Samsung flagship smartphones rely on Samsung's Exynos SoC architecture cf. [23]. The ARM cores in Exynos support the TrustZone security extension to create Trusted Execution Environments (TEEs).

In order to support cryptographic modules, the Android OS includes a mechanism for handling cryptographic keys and functions called the Keystore [8]. Keystore is used for several privacy related features such as full disk encryption and password storage. The Keystore depends on a hardware abstraction layer (HAL) module called the Keymaster to implement the underlying key handling and cryptographic functions; and many OEMs, including Samsung, choose to implement the Keymaster as a trustlet in the TrustZone.

1.2 Related Work

Lipp et al. [14] implemented cache attack techniques to recover secret keys from Java implementation of AES-128 on ARM processors, and exfiltrate additional execution information. In addition they were able to monitor cache activity in the TrustZone.

Zhang et al. [29] demonstrated a successful cache attack on a T-Table implementation of AES-128 that runs inside the TrustZone—however, their target was the C implementation that is part of OpenSSL while we focus on ARM32 assembly implementation found in Samsung Keymaster Trustlet for AES-256 and AES-256/GCM modes. Ryan et al. [15] demonstrated reliable cache side channel techniques that require loading a kernel module into the Normal World—which is disabled or restricted to OEM-verified modules on modern devices. To our knowledge no previous cache attacks on a standard devices' ARM TrustZone AES implementation using publicly available vulnerabilities have been published.

Recently, Green et al. [11] presented AutoLock, an undocumented feature in certain ARM CPUs which prevents eviction of cross-core cache sets. This feature severely reduces the effectiveness of cache side-channel attacks. The authors listed multiple CPUs that include AutoLock, and among them are the A53 and A57 used in the device we used (Samsung Galaxy S6).

Cache side channel attacks on AES were first demonstrated by Bernstein [3] with the target being a remote encryption server with an x86 CPU. Osvik et al. [21, 26] demonstrated the *Prime+Probe* technique to attack a T-Table implementation of AES which resides in the Linux kernel on an x86 CPU. Xinjie et al. [28] and Neve et al. [16] presented techniques which improve the effectiveness of cache side channel attacks. Spreitzer et al. [25] demonstrated a specialization of these attacks on misaligned T-Table implementations. Neve et al. [17] discussed the effectiveness of these attacks on AES-256 and demonstrated a successful specialized attack for AES-256.

1.3 Contributions

Our starting point is the observation of [13] that the ARM32 assembly-language AES implementation used by the Keymaster Trustlet uses a T-Table and is vulnerable to cache side-channel attacks. Furthermore, the Keymaster's T-Table is misaligned, which helps the attacker. Unlike prior works, which attacked evaluation boards or AES-128, we successfully demonstrate cache attacks on a real device, against the AES-256 and AES-256/GCM implementation used by the Keymaster trustlet. Beyond the larger keys in AES-256, GCM mode introduces additional challenges, since the cryptanalyst has no control over 4 of the 16 bytes of plaintext in an AES block.

A key aspect of our attack is that we extract the secret key using a *divide and conquer* strategy. In the AES-256/GCM case, rather than analyze all 256 key bits simultaneously, we identify them in 4 phases: we identify 84 bits in phase 1; based on them we identify the next 124 bits in phase 2, and so forth until all 256 bits are discovered.

In addition, we present our approach to implementing the analysis phase of our attacks on a GPU. Such an approach requires careful planning and, when implemented correctly, leads to a significant improvement in analysis speed.

Using a laptop's GPU to parallelize the analysis, we are able to extract a raw AES-256 key with 7 minutes of measurements and under a minute of analysis time and an AES-256/GCM key with 40 minutes of measurements and 30 minutes of analysis.

Organization: Section 2 describes the Keymaster trustlet and its cryptographic functions. Section 3 demonstrates cache side-channel attacks against the AES implementation used by the Keymaster trustlet in isolation. Section 4 describes the use of

GPU to mount the attacks and we conclude with Section 5. We provide GPU kernel examples in Appendix A.

2 Preliminaries

2.1 ARM TrustZone Overview

ARM TrustZone security extensions [2] enable a processor to run in two states, called Normal World and Secure World. This architecture also extends the concept of "privilege rings" and adds another dimension to it. In the ARMv8 ISA, these rings are called "Exception Levels" (ELs). The most privileged mode is the "Secure Monitor" which runs in EL3 and sits "above" the Secure and Normal Worlds. In the Secure World, the Secure OS kernel runs in EL1 and the Secure userspace runs in EL0. On Samsung devices, the Normal World OS is Android: the Linux kernel runs in EL1 and the user-space programs run in EL0.

The separation of Secure and Normal World allows that certain RAM ranges and bus peripherals may be indicated as "secure" and only be accessed by the Secure World. This means that compromised Normal World code (in userspace or kernel) will not be able to access these memory ranges or devices.

It's important to note that the world separation is completely "virtual". The same cores are used to run both Secure and Normal Worlds and they use the same RAM. Therefore, they use the same cache used by the core to improve memory access times; in [13] we describe how this design decision may be leveraged to mount cache side channel attacks.

In the Samsung ecosystem there are two major players in field of TrustZone implementations. One is Qualcomm, with the QSEE operating system [22] which is compatible with the Snapdragon SoC architecture used on many Samsung devices. The other is Trustonic, with the Kinibi operating system [27] which is used by Samsung in their popular Exynos SoC architecture as a part of the KNOX security system [24]. In this paper we focus on the Trustonic TrustZone.

These Trusted Execution Environments (TEEs) are used for various activities within the smart device: Secure boot, Keymaster implementation (see Section 2.2), secure UI, kernel protections, secure payments, digital rights management (DRM) and more.

2.2 Keystore and Keymaster Hardware Abstraction Layer (HAL)

The Android Keystore system [8], which was introduced in Android 4.3, allows applications to create, store and use cryptographic keys while attempting to make the keys themselves hard to extract from the device. The documentation advertises the following security features:

- Extraction Prevention: The keys themselves are never present in the application's memory space. The applications only know of *key-blobs* which cannot be used directly. The *key-blobs* are usually the keys packed with extra meta-data and encrypted with a secret key by the Keymaster HAL (Hardware Abstraction Layer).

– Key Use Authorizations: The Keystore system allows the application to place restrictions on the generated keys to mitigate the possibility of unauthorized use.

The Keystore system is implemented in the *keystored* daemon [9], which exposes a binder interface that consists of many key management and cryptographic functions. Under the hood, the *keystored* holds the following responsibilities:

– Expose the binder interface, listen and respond to requests made by applications.
– Manage the application keys. The daemon creates a directory on the filesystem for each application; the key-blobs are stored in files in the application's directory. Each key-blob file is encrypted with a key-blob encryption key (different per application) which is saved as the *masterkey* in the application's directory. The *masterkey* file itself is encrypted when the device is locked, and the encryption employs the user's password and a randomly generated salt to derive the *masterkey* encryption key.
– Relay cryptographic function calls to the Keymaster HAL device (covered below).

The Keymaster hardware abstraction layer (HAL) [7] is an interface between Android's *keystored* and the OEM implementation of a secure-hardware-backed cryptographic module. It requires the OEM to implement several cryptographic functions such as: key generation, init/update/final methods for various cryptographic primitives (public key encryption, symmetric key encryption, and HMAC), key import, public key export and general information requests. The implementation is a library that exports these functions and is implemented by relaying the request to the secure hardware system. The secure system usually encrypts generated keys with some key encryption key (which is usually derived by a hardware-backed mechanism). Therefore, the non-secure system does not know the actual key that is used, but may still save it in the filesystem and subsequently use it through the Keymaster to invoke cryptographic functions with the key. In practice - this is exactly how the *keystored* daemon uses the Keymaster HAL (with the aforementioned addition of an additional encryption of the key blobs).

An example of the usage of the Keymaster HAL is the Android Full Disk Encryption feature, implemented by the userspace daemon *vold* [10], which uses the Keymaster HAL as part of the key derivation.

2.3 Samsung's Keymaster HAL and Trustlet

Samsung's Keymaster HAL library exposes the aforementioned Keymaster interface and implements its functions by making calls to the Keymaster Trustlet. The trustlet itself has UUID: *ffffffff000000000000000000000003e*, and is located in the system partition (*/system/app/mcRegistry/<UUID>.tlbin*). The Trustlet code handles several tasks, of which the following are relevant to our work:

– Key generation of RSA/EC, AES and HMAC keys. Keys are generated using random bytes from the OpenSSL FIPS DRBG module, which seeds its entropy either from *keymaster_add_rng_entropy* calls from the Normal World or from a secure PRNG made available by the Secure World Crypto Driver. Key generation requests receive a list of key characteristics (as defined by the Keymaster HAL), which describe the algorithm, padding, block mode and other restrictions on the key. The

generated keys (concatenated with their characteristics) are encrypted by a key-encryption-key (**KEK**) which is unique to the Keymaster trustlet. The trustlet receives this key by making an IPC request along with a *constant* salt to a driver which uses a hardware-based cryptographic function to drive the key. The encryption used for key encryption is AES256-GCM128. The GCM IV and authentication tag are concatenated to the encrypted key before being returned to the user as a key blob. Therefore, an attacker that is able to obtain this KEK is able to decrypt all the key blobs stored in the file system—i.e., the KEK can be viewed as the "key to the kingdom", and it's encryption scheme is the target of our attacks in Section 3.

- Execution of cryptographic functions. The trustlet can handle begin/update/final requests for given keys created by the trustlet. It first decrypts the key-blobs and verifies the authentication tag, then verifies that the key (and the trustlet) supports the requested operation, and then executes it. The cryptographic functions are implemented using the OpenSSL FIPS Object Module [20]. In particular, we discovered that the AES code is a pure ARMv4 assembly implementation that uses a single 1KB T-Table. In general, AES implementations based on T-Tables are vulnerable to cache attacks [21, 26]. Our attacks (described in Section 3) explore cache side channel attacks on this AES implementation.
- The trustlet handles requests for key characteristics and requests for information on supported algorithms, block modes, padding schemes, digest modes and import/-export formats.

2.4 Attack model

The fundamental reason for the existence of the TrustZone is to provide a hardware-based root of trust for a trusted execution environment (TEE)—that is designed to resist even a compromised Normal World kernel.

Since the Normal World kernel, and all the kernel modules on Samsung's smartphones are signed by Samsung and verified before being loaded, injecting code into the kernel is challenging for the attacker. Our goal in this work is to demonstrate that weaker attacks, that do not require a compromised kernel, are sufficient to exfiltrate Secure World information—in particular secret key material.

Our attack has two stages: a data collection stage and an analysis stage. In the data collection stage we assume an attacker is able to execute code on a Samsung Galaxy S6 device, under **root privileges** and relevant **SELinux permissions**. Note that these privileges are significantly less than kernel privileges, since the attack code runs in EL0.

Root privileges are needed to access the */proc/self/pagemap* to identify cache sets, as described by Lipp et al. [14]. Our attack can theoretically be mounted without access to this file, but it will be substantially more difficult.

To achieve root privileges and the necessary SELinux permissions in our investigation we used the publicly known vulnerability called *dirtycow*. The rooting process is based on Trident [6], which uses *dirtycow*.

The main target of our attack is the Keymaster trustlet. The API to communicate with the trustlet expects a buffer which should hold a key blob. Valid key blobs typically include over 100 bytes of encrypted data, therefore an API call (e.g. to extract some meta-data from a key blob) uses the AES-256 block function at least 9 times (2

for initialization and at least 7 subsequent blocks). If we measure cache access effects only after the trustlet completes its work, the 9 block function invocations will induce too much noise and render our attacks infeasible. Therefore, instead we send *invalid* requests: having the key blob hold just one byte. Such API calls induce the two block function calls for GCM initialization, and a one more call to decrypt the single byte. The request then fails, therefore we do not have access to any ciphertext. Our attacks take this restriction into consideration by focusing on the first AES-256 rounds and knowledge of the plain-text and IV - and avoid relying on the resulting ciphertext.

In the subsequent analysis stage, the collected clock measurement data is analyzed on a separate machine - we utilized a Macbook Pro laptop using a Radeon Pro 460 GPU.

3 Cache Attacks Against the Keymaster AES

3.1 Overview

As stated before, the Keymaster key encryption uses AES256/GCM128, therefore we focused on AES side channel attacks. In this section we present our attack methods.

We begin our work by adapting prior cache attacks on AES to the ARM32 implementation used in the Keymaster trustlet. Our measurements were taken on a stock Samsung Galaxy S6 running original Samsung firmware.

Prior research [21, 26][28][16] demonstrated that the use of T-Tables in AES induces cache activity which leads to key leakage. In particular, when the T-Tables are misaligned in memory, better results have been achieved [25]. These methods exploit the fact that the implementation of the AES rounds use memory lookups which may be traced by evicting the T-Table from memory, running the AES encryption and then observing the cache access timing pattern. While the aforementioned methods assume the AES implementation uses four T-Tables, the AES implementation in the Keymaster trustlet uses one T-Table which is misaligned [13]. According to [21, 26] this design choice is still vulnerable but requires roughly 3000 times more data and analysis, which is still feasible.

The attack, presented by Osvik et al. [21, 26], assumes we can detect cache activity (on the cache sets which hold the T-Table) using the *Prime+Probe* method and focuses on the first round of AES. The *Prime+Probe* method measures cache activity by first *priming* a specific cache set (by writing memory to memory addresses which map to the same set - thereby *evicting* the cache set), then allowing the AES algorithm to run and finally *probing* the cache set (by accessing the *primed* memory addresses and measuring the time it took to fetch them). From the resulting measurements one can infer whether the AES algorithm has evicted a specific set - which would cause the *probing* phase to measure a higher value (due to some of its memory addresses being fetched from memory instead of the cache). In order to differentiate between *probe* measurements of evicted sets and non-evicted sets, a threshold value (denoted T_a below) is used. This value must be calibrated in advance for each hardware (CPU+Cache) that will be used as a target for the attack.

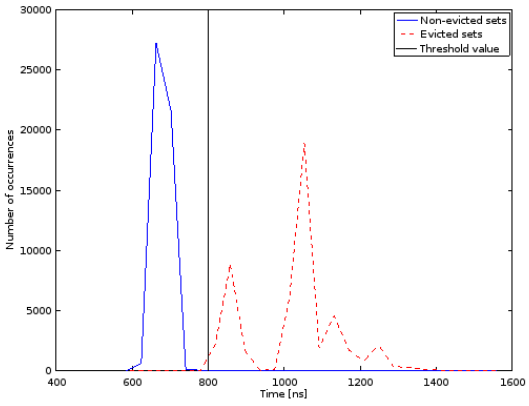

Fig. 1. Histogram of probe timing measurement for $50,000$ probes. Separation between evicted and non-evicted sets is visible at around 800ns.

If the *probe* measurement, for the cache set which holds T-Table entry number i, is below T_a, the entry was not accessed and therefore certain k_i values are incorrect. If, in-fact, one of these k_i values was correct, the T-Table entry would have been accessed and eviction would occur for one of our *primed* addresses, resulting in a *probe* measurement above the threshold. More precisely, due to noise in the system, we may only infer that they are more likely, therefore we give each k_i candidate a score based on how many times we deem it likely (0 for each time it is unlikely and 1 otherwise). The k_i values we infer from cache activity depends on p_i, the details of the cache and the alignment (or misalignment) of the T-Table with respect to cache lines.

3.2 Calibrating the probe measurement threshold

We selected the threshold value T_a through analysis of cache access times and eviction strategies as described by Lipp et. al. [14]. This method applies *Prime+Probe* to a single address multiple times in two manners: first, it is *primed* and *probed* consecutively and second, memory access is added after the *prime* and before the *probe*. This, essentially, creates statistics on probe measurements for a given eviction strategy on a given CPU and cache. T_a separates between probes on set indexes which were not evicted versus set indexes which at least one address was evicted. Figure 1 shows the results of this method on our Galaxy S6. The strategy we used (using the notation from Lipp et. al. [14]) is $N = 5$ (total eviction set size), $A = 5$ (shift offset), $D = 16$ (number of accesses per iteration) which we found to be the best strategy for our device after testing many alternatives; time measurements were made with linux's monotonic clock due to lack of better clock source available under our attack model. Based on the figure we set T_a to be $800[ns]$.

3.3 The analysis stage for AES-128 attacks

To begin with, we describe an attack on AES-128 in ECB mode. With the cache activity measurements gathered on the Galaxy S6 in the data collection stage, we implement the

analyzing stage on a GPU-equipped laptop. The analysis stage consists of two phases, described below.

Phase 1 In the first round of the AES implementation, each *i*-th plaintext byte p_i is XOR-ed with the *i*-th key byte k_i: $x_i^{(0)} = p_i \oplus k_i$. The value of $x_i^{(0)}$ is then used as an index to the T-Table which is accessed subsequently.

Because these calculations only rely on the value of p_i and k_i, it's possible to use a *divide-and-conquer* approach and consider each key byte independently. Given a probe measurement for T-Table entry x, we iterate through byte index *i* = 0, ..., 15 and let p_i be the i-th plaintext byte. For all possible values of k_i = 0, ..., 255, we check whether k_i is likely based on the method described above and update the scores. We end with a score matrix for the level of likelihood for each candidate value per key byte. We continue the measurements and analysis until for each key byte, one candidate has a z-score above 5 (i.e., 5 standard-deviations above the mean).

On the Samsung Galaxy S6 ARM A53 and A57 CPUs, each cache line is 64 bytes long; therefore each line holds 16 T-Table entries (4 bytes per entry). In the implementation present in the Keymaster trustlet, the T-Table has an 8 byte misalignment with respect to the cache lines, see [13]: So the T-Table actually spans over 17 cache lines, with the first line holding 14 entries and the last line holding 2 entries. This means that our best case resolution is 2: if we use the constraints based on a single AES round we are eventually left with 2 candidates for each key byte which are indistinguishable to us. This means we learn 7 out of 8 bits for each key byte, reducing the unknown key space from 128 bits to 16 bits.

Phase 2 Enumerating through 16 bits is trivial with modern hardware; however, we present the rest of the attack which continues to apply *divide-and-conquer* using analysis of subsequent rounds. It will be useful to understand the next sections in which we attack AES-256 and AES-256/GCM and it may also be of independent interest in cases where the misalignment is less favorable or nonexistent.

To identify the remaining AES128 key bits we focus on the second round of the AES implementation; specifically, the following equations, derived from the Rijndael specification[4], which give 4 of the entries accessed in the second round:

$$\begin{aligned}
x_2^{(1)} &= s(p_0 \oplus k_0) \oplus s(p_5 \oplus k_5) \\
&\quad \oplus 2 \bullet s(p_{10} \oplus k_{10}) \oplus 3 \bullet s(p_{15} \oplus k_{15}) \oplus s(k_{15}) \oplus k_2 \\
x_5^{(1)} &= s(p_4 \oplus k_4) \oplus 2 \bullet s(p_9 \oplus k_9) \\
&\quad \oplus 3 \bullet s(p_{14} \oplus k_{14}) \oplus s(p_3 \oplus k_3) \oplus s(k_{14}) \oplus k_1 \oplus k_5 \\
x_8^{(1)} &= 2 \bullet (p_8 \oplus k_8) \oplus 3 \bullet s(p_{13} \oplus k_{13}) \\
&\quad \oplus s(p_2 \oplus k_2) \oplus s(p_7 \oplus k_7) \oplus s(k_{13}) \oplus k_0 \oplus k_4 \oplus k_8 \oplus 1 \\
x_{15}^{(1)} &= 3 \bullet s(p_{12} \oplus k_{12}) \oplus s(p_1 \oplus k_1) \\
&\quad \oplus s(p_6 \oplus k_6) \oplus 2 \bullet s(p_{11} \oplus k_{11}) \oplus s(k_{12}) \oplus k_{15} \oplus k_3 \oplus k_7 \oplus k_1
\end{aligned} \tag{1}$$

Where $s(\cdot)$ denotes Rijndael S-box function and $\bullet$ denotes multiplication over GF(256). There are three properties of these equations which are important to note:

- Each equation refers to 4 "bound" k_i's (that are an input to $s(\cdot)$) and between 1 to 4 "free" k_i's that are simply XOR'ed. In fact, the keen reader may see that if we analyze the equations sequentially, each equation only has 1 "free" k_i: If we solve the equations in sequence then all but one of the "free" k_i is completely discovered by the previous equations.
- Because our measurement resolution is 2 entries, only the 7 most significant bits of the "free" k_i variables are relevant to the index calculations.
- Since the first 7 bits of every k_i are known from phase 1, each equation only has 4 unknown bits - the least significant bit of every "bound" k_i.

These properties allows us to apply *divide-and-conquer* once again, and consider each equation separately. For each pair of plaintext and *probe* measurements, we enumerate the 4 possible key bits, calculate the equation and check whether they are likely based on the cache accesses during the second AES round. Eventually, the most likely candidate of these 4 bits is selected.

Combining the results for the four equations, along with the result of the phase 1, yields the entire 128 bits of the key - full key recovery.

We implemented a cache side-channel attack against the AES-128 implementation used by the Keymaster trustlet after copying it to a user-space sandbox and using AES in ECB mode. We were able to successfully recover the entire 128 bits of the key using the method described above. Our experiment used 100,000 measurements: this amount of data can be collected in under a minute on a Samsung Galaxy S6 and analyzed in less than 15 seconds on a Radeon Pro 460 GPU. The amount of memory used by the GPU (in phase 1 of the attack) was 1GB. Details on the GPU analysis implementation in Section 4.

3.4 AES-256 attacks

Phases 1 & 2 As we saw in Section 2.3, Samsung's Keymaster trustlet uses AES-256. Attempting to use the attack described in the previous section on AES-256 is not enough for full key recovery. There are relatively few papers discussing the specifics of cache attacks against AES-256. The most relevant seems to be by Neve and Tiri [17]. They proposed an extension of a different attack—one that looks at the *last* two rounds of AES instead of the first, and requires knowing the ciphertext, in contrast to the requirement of knowing the plaintext in the attack we used in Section 3.3. As we discussed in section 2.4, relying on last AES round is difficult against the Keymaster trustlet. Therefore, we devised a method which extends the attack of Section 3.3 to recover 256 bit keys using the *first* three rounds.

The first part of the attack remains the same as phase 1 of the AES-128 attack (see Section 3.3): discover the 7 most significant bits of k_0 through k_{15}. We determine that the first round sieving has ended when for all 16 key bytes the most likely candidate value has a z-score above 5.

In order to recover the missing 16 bits in the lower half of the key, and most of the bits in the upper half, we rely on the second AES round. E.g., consider the first four indexes, which are derived from the Rijndael specifications:

$$\begin{bmatrix} x_0^{(1)} \\ x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{bmatrix} = \begin{bmatrix} 2\,3\,1\,1 \\ 1\,2\,3\,1 \\ 1\,1\,2\,3 \\ 3\,1\,1\,2 \end{bmatrix} \bullet \begin{bmatrix} s(p_0 \oplus k_0) \\ s(p_5 \oplus k_5) \\ s(p_{10} \oplus k_{10}) \\ s(p_{15} \oplus k_{15}) \end{bmatrix} \oplus \begin{bmatrix} k_{16} \\ k_{17} \\ k_{18} \\ k_{19} \end{bmatrix} \tag{2}$$

It's important to note that every 4 indexes depend on the same four key bytes from the lower half of the key (k_0, k_5, k_{10}, k_{15} in equation (2)) and each index depends on one byte of the upper half of the key. Another important property is that as in section 3.3, from the 8 bits of the bytes from the upper half of the key only the 7 most significant bits affect the measurement of the index. Therefore, each equation has only 11 unknown bits: 1 for each of the 4 lower-half-key bytes and 7 for the single upper-half-key byte.

Once again, we use *divide-and-conquer*; divide the problem to four four-equation subproblems, divide each subproblem to it's four equations and on each equation use the same methods described above to select the most likely candidate for the 1 least significant bits of the lower-half key bytes and the 7 most significant bits of the single upper-half key byte. Therefore, for a given equation e and measurement, iterate over all 2^{11} combinations of key-bit values. If the measurement is compatible with key-bit combination c, then increment $score[e][c]$. After this step, we have the entire lower-half key bytes (k_0 through k_{15}) and the 7 most significant bits of every upper-half key byte (k_{16} through k_{31}). Therefore, we have reduced the key space from 256 bits to 16 by using the first two rounds.

Phase 3 While enumeration of 16 bits is feasible, we present a third phase of the attack which may be applied to other misalingment circumstances. We do so by imitating the second phase of the attack on 128 bit AES (section 3.3). Consider equation (3) which is derived from the Rijndael specification with 256 key expansion (after substituding the first round indexes $x_i^{(1)}$)

$$\begin{aligned} x_2^{(2)} = {} & s(2 \bullet s(p_0 \oplus k_0) \oplus 3 \bullet s(p_5 \oplus k_5) \oplus s(p_{10} \oplus k_{10}) \\ & \quad \oplus s(p_{15} \oplus k_{15}) \oplus \underline{k_{16}}) \\ & \oplus s(s(p_4 \oplus k_4) \oplus 2 \bullet s(p_9 \oplus k_9) \oplus 3 \bullet s(p_{14} \oplus k_{14}) \\ & \quad \oplus s(p_3 \oplus k_3) \oplus \underline{k_{21}}) \\ & \oplus 2 \bullet s(s(p_8 \oplus k_8) \oplus s(p_{13} \oplus k_{13}) \oplus 2 \bullet s(p_2 \oplus k_2) \\ & \quad \oplus 3 \bullet s(p_7 \oplus k_7) \oplus \underline{k_{26}}) \\ & \oplus 3 \bullet s(3 \bullet s(p_{12} \oplus k_{12}) \oplus s(p_{11} \oplus k_{11}) \oplus s(p_6 \oplus k_6) \\ & \quad \oplus 2 \bullet s(p_{11} \oplus k_{11}) \oplus \underline{k_{31}}) \\ & \oplus s(\underline{k_{31}}) \oplus k_2 \end{aligned} \tag{3}$$

At first sight this equation may seem daunting. However, notice that we, in fact, know p_0 through p_{15}, k_0 through k_{15} and the 7 most significant bits of k_{16}, k_{21}, k_{26}, k_{31}. Therefore, only 4 bits are unknown in this equation. We then use a similar sieving

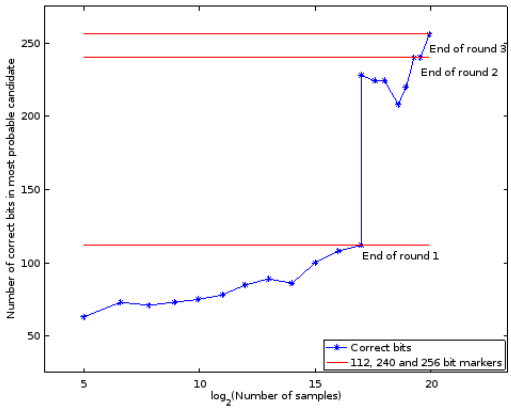

Fig. 2. Number of correct bits in the most likely candidate as function of samples used (log_2 scale).

method as in the previous phases to select the most likely candidate for these bits. Then, we apply the same technique to the equations for $x_5^{(2)}$, $x_8^{(2)}$ and $x_{15}^{(2)}$. Eventually, we arrive at full recovery of the AES 256 bit key.

Putting it all together We implemented this attack on the AES-256 code used by Samsung's Keymaster trustlet. Figure 2 shows the the number of correct bits in the most likely candidate as a function of the number of measurements used. The horizontal barriers mark the target of each phase of the attack: 112 bits for the 1^{st} phase, 240 bits for the 2^{nd} and 256 bits for the 3^{rd} phase. It's important to note that after we complete the first phase, we reuse the samples for the second phase which explains the sudden increase in known bits between phase one and phase two of the attack. It took 7 minutes to collect the one million measurements on the Galaxy S6. The sieving process took under a minute to complete (all three stages) and 3.5GB of memory, using a Radeon Pro 460 GPU on a laptop.

3.5 Galois counter mode (GCM) attacks

Challenges A further complication is that Samsung's Keymaster trustlet uses AES-256 in GCM mode [5]. Two factors make cache side channel attacks harder against GCM: the use of the block function in the initialization, and the lack of control over the 4 last bytes of the input to the block function.

According to the GCM specification, the computation of the authentication tag requires two invocations of the block function. When the input *initialization vector* (IV) is 96 bits long, the block function is invoked once with a plain-text of 0^{16} (a 16 zero byte string) and then with a plain-text of $IV||0^4$. Unless it is possible to distinguish between this initialization phase and subsequent encryption phases, the initialization induces substantial cache-access noise.

Furthermore, subsequent block function invocations made by GCM are called with the input $IV||Counter$, where IV is the original 96 bit IV and Counter is a four byte integer counter (starting with the value 2) which is appended (with big endianess) to the IV. This means that we have limited control over the input to the block function. While we control the 96 bits given as IV, the Counter bytes may only be changed by encrypting

additional data with the same GCM context. This implies that it is much more difficult to collect enough data to differentiate between key candidates for k_{12} through k_{15}.

Phases 1 & 2 We begin by attempting to apply the same technique used in the previous section to AES-256/GCM. We assume that we can distinguish between cache-access due to the first two block function invocations (initialization calls) and subsequent invocations. However, we continue limit ourselves to scenarios that allow only one encryption call and do not allow knowledge of the resulting ciphertext to allow use against the Keymaster trustlet. Because we do not have control over the last four bytes of the input, the first phase of the technique (section 3.4), which focuses on the first round of AES, only recovers the 7 most significant bits of k_0 through k_{11}, recovering only 84 bits of the key.

The second part of the technique, which focuses on the second round of AES-256, is more difficult under GCM. Instead of the 11 unknown bits we identified in Section 3.5, we now face 18 unknown bits: 1 least significant bit for k_0 through k_{11} (three of these per equation), 8 bits for k_{12} through k_{15} (one per equation) and 7 most significant bits for k_{16} through k_{31} (one per equation).

While an enumeration of 18 bits is feasible even with modest resources, another hurdle emerges. Consider the value $t = s(p_{15} \oplus k_{15}) \oplus k_{16}$ in the equation for $x_0^{(1)}$ in equation (2). Because p_{15} has a single value (typically p_{15}=2) which we cannot control, t has a constant value. We note that for each key byte candidate value x for k_{15} we can find a key byte candidate k_{16}, which will result in the same value x. More precisely, due to the resolution from the T-Table misalignment, we can find a 7 most significant bit candidate for k_{16}.

By applying the same sieving technique described above, we use pairs of IV (first 12 bytes of plaintext) and *probe* measurements to select the likely value of the 18 unknown bits. Due to the dependency between k_{15} and k_{16} described above, we expect to find 256 likely values: each having the correct least significant bit of k_0, k_5 and k_{10}, one of the 256 candidates for k_{15} and the 7 most significant bits of k_{16} candidate. This method allows us to gain full information on k_0, k_5 and k_{10}, and a constraint on k_{16} depending on k_{15}. This constraint may be visualized as a table indexed by k_{15} and having its value be the constraint on k_{16}.

We apply the same method for the rest of the equations shown in equation set (2) to gain information on the respective constraint between k_{15} and k_{17} through k_{19}. Note that in the case of k_{18} and k_{19} the constrained value t is $t = 3 \bullet s(p_{15} \oplus k_{15}) \oplus k_{18}$ and $t = 2 \bullet s(p_{15} \oplus k_{15}) \oplus k_{19}$ respectively. These four constraints may be grouped into a single table, Table 1 shows an example of such table.

The same method may be used to extract similar constraints between the three other bytes k_{12}, k_{13}, k_{14} and their respective four bytes from the upper half of the key.

To summarize, based on 2 rounds of AES in GCM mode we can extract the values of k_0 through k_{11}, and have four table which describe further constraints on the key. The remaining key space is 48 bits: 8 bits per table (32 total) and 1 additional bit per byte in the upper half of the key.

Table 1. The 7 most significant bits of upper key bytes for each possible k_{15} value

k_{15}	k_{16}	k_{17}	k_{18}	k_{19}
0	67	22	60	67
1	69	16	54	79
2	73	38	34	87
...				
253	115	38	109	34
254	32	117	21	9
255	82	7	14	96

Phase 3 We now shift our focus to round 3 and consider equations $x_{12}^{(2)}$ through $x_{15}^{(2)}$. Equation (4) shows one such equation: These equations are important to us for two reasons:

- The "bound" expressions holding k_{12} to k_{15} in these equations have appeared in our 2^{nd} round analysis and therefore we have a table that constraints the "free" upper-half key bytes to their values (e.g. $s(p_{15} \oplus k_{15}) \oplus k_{17}$ is known up to 1 bit). Thereby reducing the unknown bits in each of these expressions from 8 bits to 1.
- Due to the AES key expansion scheme, each of these equations includes one of the key bytes k_{12} through k_{15} in a "free" manner. Which allows us to receive different measurements for these bytes; note that this is the first round this is possible in.

$$
\begin{aligned}
&x_{12}^{(2)} = \\
&2 \bullet s(2 \bullet s(p_{12} \oplus \underline{k_{12}}) \oplus 3 \bullet s(p_1 \oplus k_1) \oplus s(p_6 \oplus k_6) \oplus s(p_{11} \oplus k_{11}) \oplus \underline{k_{28}}) \oplus \\
&3 \bullet s(s(p_0 \oplus k_0) \oplus 2 \bullet s(p_5 \oplus k_5) \oplus 3 \bullet s(p_{10} \oplus k_{10}) \oplus s(p_{15} \oplus \underline{k_{15}}) \oplus \underline{k_{17}}) \oplus \\
&s(s(p_4 \oplus k_4) \oplus s(p_9 \oplus k_9) \oplus 2 \bullet s(p_{14} \oplus \underline{k_{14}}) \oplus 3 \bullet s(p_3 \oplus k_3) \oplus \underline{k_{22}}) \oplus \\
&s(3 \bullet s(p_8 \oplus k_8) \oplus s(p_{13} \oplus \underline{k_{13}}) \oplus s(p_2 \oplus k_2) \oplus 2 \bullet s(p_7 \oplus k_7) \oplus \underline{k_{27}}) \oplus \\
&\underline{k_{12}} \oplus k_8 \oplus k_4 \oplus k_0 \oplus s(\underline{k_{29}}) \oplus 1
\end{aligned}
\tag{4}
$$

While Equation (4) might seem to have 48 unknown bits, using the knowledge from the previous phases we assert that it only has 13 unknown bits: 8 bits to choose k_{12}, 1 least significant bit for k_{29}, and 1 more least significant bit for each "bound" expression (4 additional bits).

Applying our sieving technique once again for equations $x_{12}^{(2)}$ through $x_{15}^{(2)}$ gives us the most likely value of the 7 most significant bits of k_{12} to k_{15} and the least significant bit of k_{28} through k_{31}. Thereby reducing the amount of unknown bits from 48 to 16 - 1 least significant bit of k_{12} to k_{15} and 1 least significant bit of k_{16} to k_{27}. Additional analysis of the 3^{rd} round accesses may reveal the remaining bits, but we chose to apply brute-force enumeration to find them.

It took five million measurements to mount this analysis which took 40 minutes to collect on the Galaxy S6. The analysis took 30 minutes and 3.5GB of memory to complete using a Radeon Pro 460 GPU on a laptop.

In summary, we see that the AES-256 GCM, with a single T-Table implementation used by Samsung's Keymaster trustlet, is vulnerable to cache side-channel attacks when it is used in isolation.

4 Analysis Acceleration using a GPU with OpenCL

4.1 Overview

Previous work [3][14][17][16][21, 26][25][29][28] goes into great details about the implementation of the attack phase and candidate sieving; however, little discussion is presented on the implementation of the analysis phase. While designing the attacks described in the previous sections, we found that the amount of data and time required by a sequential implementation of the attack is significant, so, we decided to leverage GPUs to expedite the analysis. The following sections provide detail into our design and implementation of a GPU based cache attack analysis method.

4.2 Programming the GPU

When programming a GPU, one must design a function that will be run in parallel on many data points; such a function is called a *kernel*. We used a GPGPU (general purpose GPU) programming framework called pyOpenCL [12]. This framework allowed us to write most of the analysis in python while easily deferring the heavy lifting to the GPU. pyOpenCL provides a very convenient way to write OpenCL kernels called "ElementwiseKernel": the programmer only needs to write the calculation for a single element while abstracting away most other details. A kernel is essentially a function, that receives at least one pointer to a GPU memory block (usually containing tables) and an argument i which is used as an index to that memory block. The framework instructs the GPU to run numerous copies of that kernel in parallel, each on a different GPU core, with each copy being allocated a different index i.

A kernel returns output by modifying the memory blocks received as arguments. In order to minimize the need to synchronize the kernels, usually each kernel writes to a separate cell in memory; thus avoiding memory contention and race conditions. The Radeon Pro 460 which we used in our analysis has 1024 cores and 4GB of internal memory.

4.3 Using the GPU in the attacks

Previous sections outlined the basic algorithm used for the analysis of the side channel artifacts: Use the plaintext bytes and side channel map to sieve through the possible candidates until one most likely candidate is found. This process can be broken down to the following steps: (i) thresholding the cache access patterns to discern between cache hits and misses, (ii) AES round calculations, (iii) matching between calculation and the cache access pattern and (iv) scoring.

i The thresholding step is straightforward: it receives a cache access timing matrix (rows are different measurements, columns are the relevant cache set indexes which were measured) and a threshold value T_a (recall section 3.2). Each matrix cell is compared against T_a and is set to either 1 if it's above T_a (miss) or 0 otherwise (hit).

ii The AES round calculation varies depending on the step of the analysis (as described in previous sections) but follows the same principles: receive the relevant

key candidates and plain-texts used in measurements, apply the relevant round calculation, apply the table misalignment and return the relevant cache set indexes for the given candidates for each given plain-text. Round calculations follow the equations presented in the previous sections, and use lookup tables to calculate the S-Box and GF(256) multiplications. The S-Box and multiplication tables are placed inside the GPU internal memory. The result is a matrix M of the candidates versus the plain-texts where each cell $M_{i,j}$ holds the result of the AES round calculation for i-th plain-text and the j-th key candidate. In other words, if key candidate j is correct, $M_{i,j}$ holds a cache index which we expect to measure as a miss for the i-th plaintext and its cache measurements.

iii The next step takes the thresholded cache access matrix and the cache set index matrix result from the AES round calculation step and returns an array which holds the score for each candidate. This is done in two steps: matching and summing. The match step takes the two input matrices and outputs a matrix of candidates versus plain-texts in which each cell is 1 if the index predicted by the AES round calculations (for a given plain-text and candidate) was a cache miss in our measurements (which implies the candidate is more likely) and 0 otherwise. The summing phase then sums the result by the plain-text axis, resulting in an array of scores for each key candidate.

iv Finally, we are left with the scores for each key candidate, all we have left is to choose the most likely one. Due to the large key enumeration space in phase 2 and 3 of the AES-256 and AES-256/GCM attacks, the memory on our GPU was not large enough to hold the plain-text over key candidate matrices when trying to analyze all of the plain-texts at once. Instead, we divided the plain-texts into batches, analyzed them separately and combined their result after each batch by simply adding the score array. This allowed us to analyze large amounts of data (over 2^{20} samples) over up to 18 bits of key candidates on a commodity laptop GPU within minutes.

4.4 Kernel implementation details

We used several such kernels and provide their code in the appendix:

1. Thresholding: The first kernel is used to reduce the measurements from a matrix of plain-texts over cache indexes which contains the cache timing measurements to a matrix of the same dimensions but with a value 1 if the measurement is above T_a which indicates a cache miss, or 0 otherwise. This is accomplished via a simple ternary operator. See Appendix A.1.
2. Round calculation kernel: The following explanation is relevant to the first four round bytes of the 2^{nd} AES256 round calculations (recall section 3.4), the same principles apply for the the rest of the round bytes and the 3rd round as well. This kernel receives our key candidates (5 key bytes serialized as a 64-bit integer), plain-text bytes, S-Box and GF(256) multiplication lookup tables, misalignment parameter and output matrices (plain-texts over candidates). Note that instead of calculating the round for each round byte separately, we optimize this kernel by reusing calculations to calculate four round bytes together (see Equation (2)). The kernel basically calculates the first AES round (SubBytes, ShiftRows and MixColumns) and then XORs the result with an upper-half key candidate byte. The result is the

index of the T-Table which will be access by the 2^{nd} round. It then applies the misalignment parameter and selects the bits which are relevant to the cache index and stores the results in the output matrices. These matrices then hold the cache set which we expect to measure as a miss for a each plain-text, *if* that the key candidates are correct. See Appendix A.2.

3. Round to hit matrix kernel: The previous kernel results in four matrices of cache sets. This kernel performs a pass through those matrices and merges their results with the actual measurements. It receives the four result matrices, and the thresholded measurements matrix. For each cell of the result matrices (which represent the index which we expect to see as 1 in the measurement matrix for a plain-text and a key candidate, if the candidate is correct), we retrieve the measurement of the relevant plain-text and the relevant cache index. This result will be 1 if the measurements support this candidate (cache index was indeed measured as a miss) and 0 otherwise. We write the result back to the result matrix to save memory. See Appendix A.3.
4. Sum axis kernel: The previous kernel results in four score matrices of plain-text over candidates. Since we are trying to calculate the candidate score, we then need to sum these matrices by the candidates axis. Special care must be taken when summing in GPU code as many cores may access the sum variable concurrently. Several solutions exist, such as: summing in CPU instead, logarithmic reduction kernels or using atomic OpenCL intrinsics. We compared the CPU solution (using the Python Numpy package sum by axis function) with a kernel which uses the "atomic_add" intrinsic and found that the kernel is about twice as fast. That being said, both solutions took negligible time compared to the other operations. We did not attempt to implement a more optimized sum kernel. See Appendix A.4.

5 Conclusions

The ARM TrustZone is a security extension which is used in recent Samsung flagship smartphones to create a Trusted Execution Environment (TEE) called a Secure World, which runs secure processes called Trustlets. The Samsung TEE includes cryptographic key storage and functions inside the Keymaster trustlet. The secret key material used by the Keymaster trustlet is derived by a hardware device and is inaccessible to the Android OS. However, the ARM32 AES implementation used by the Keymaster is vulnerable to side channel cache-attacks. The Keymaster trustlet uses AES-256 in GCM mode, which makes mounting a cache attack against this target much harder. In this paper we show that it is possible to perform a successful cache attack against this AES implementation, in AES-256/GCM mode using widely available hardware. Using a laptop's GPU to parallelize the analysis, we are able to extract a raw AES-256 key with 7 minutes of measurements and under a minute of analysis time and an AES-256/GCM key with 40 minutes of measurements and 30 minutes of analysis.

We conclude that cache side-channel effects are a serious threat to the current AES implementation inside the Keymaster trustlet. However, side-channel-resistant implementations, that do not use memory accesses for round calculations, do exist for the ARM platform, such as a bit-sliced implementation [19] or one using ARMv8 cryptographic extensions [18]. Using such an implementation would render most cache attacks, including ours, ineffective.

References

1. ARM. *Building a secure System using TrustZone Technology*. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.
2. ARM. ARM trustzone. https://www.arm.com/products/security-on-arm/trustzone, 2018.
3. Daniel J Bernstein. Cache-timing attacks on AES. 2005. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.
4. Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. In *AES submission document*. 1999. http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael-ammended.pdf.
5. Morris J Dworkin. SP 800-38D: Recommendation for block cipher modes of operation: Galois/counter mode GCM and GMAC, 2007. National Institute of Standards & Technology.
6. freddierice. Trident - temporary root for the Galaxy S7 active. https://github.com/freddierice/trident.
7. Google. Android keymaster HAL. https://source.android.com/security/keystore/implementer-ref.
8. Google. Android keystore. https://developer.android.com/training/articles/keystore.html.
9. Google. Android keystore - source code. http://androidxref.com/6.0.0_r1/xref/system/security/keystore/keystore.cpp.
10. Google. Android vold cryptfs. http://androidxref.com/6.0.0_r1/xref/system/vold/cryptfs.c.
11. Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. Autolock: Why cache attacks on ARM are harder than you think. In *26th USENIX Security Symposium*, pages 1075–1091, 2017.
12. Andreas Klöckner, Nicolas Pinto, Yunsup Lee, B. Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.
13. B. Lapid and A. Wool. Navigating the Samsung TrustZone with applications to cache-attacks on AES-256 in the Keymaster trustlet. In *Proc. 23rd European Symposium on Research in Computer Security (ESORICS)*, Barcelona, September 2018. To appear.
14. M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache attacks on mobile devices. In *USENIX Security conference*, pages 549–564, 2016. https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_lipp.pdf.
15. nccgroup. Cachegrab. https://github.com/nccgroup/cachegrab.
16. Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *International Workshop on Selected Areas in Cryptography*, pages 147–162. Springer, 2006.
17. Michael Neve and Kris Tiri. On the complexity of side-channel attacks on AES-256 – methodology and quantitative results on cache attacks. Technical report, 2007. https://eprint.iacr.org/2007/318.
18. OpenSSL. ARM AES implementation using cryptographic extensions. https://github.com/openssl/openssl/blob/master/crypto/aes/asm/aesv8-armx.pl.
19. OpenSSL. ARMv7 AES bit sliced implementation. https://github.com/openssl/openssl/blob/master/crypto/aes/asm/bsaes-armv7.pl.
20. OpenSSL. OpenSSL FIPS. https://www.openssl.org/docs/fips.html.

21. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers Track at the RSA Conference*, pages 1–20. Springer, 2006.
22. Qualcomm. Snapdragon security. https://www.qualcomm.com/solutions/mobile-computing/features/security, 2018.
23. Samsung. Mobile processor: Exynos 7 Octa (7420). http://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-7-octa-7420/, 2018.
24. Samsung. Platform security. http://developer.samsung.com/tech-insights/knox/platform-security, 2018.
25. Raphael Spreitzer and Thomas Plos. Cache-access pattern attack on disaligned AES T-tables. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 200–214. Springer, 2013.
26. Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.
27. Trustonic. Trustonic Kinibi technology. https://developer.trustonic.com/discover/technology.
28. Zhao Xinjie, Wang Tao, Mi Dong, Zheng Yuanyuan, and Lun Zhaoyang. Robust first two rounds access driven cache timing attack on AES. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 3, pages 785–788. IEEE, 2008.
29. Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. TruSpy: Cache side-channel information leakage from the secure world on ARM devices. *IACR Cryptology ePrint Archive*, 2016(980), 2016.

A OpenCL Kernels Code

A.1 Thresholding kernel

Listing 1.1. Thresholding Kernel

```
threshold_kernel = ElementwiseKernel(ctx,
'''
uint* in, uint thresh, uint* out
''',
'''
out[i] = (in[i] > thresh) ? (1) : (0)
''',
''threshold_kernel''
)
```

A.2 Round two kernel

Listing 1.2. Round Two Kernel

```
round_kernel = ElementwiseKernel(ctx,
'''
uint *x0, uint *x1, uint *x2, uint *x3, ulong *candidates, uint *p0,
uint *p5, uint *p10, uint *p15, uint row_size, uint *sbox,
uint *mult2, uint *mult3, uint disalignment
```

```
''',
'''
// Extract key byte candidate from serialized candidate and
//  apply SubBytes
uint t0 = SHIFT_RIGHT(candidates[i % row_size], 0) ^p0[i/row_size];
uint t5 = SHIFT_RIGHT(candidates[i % row_size], 8) ^p5[i/row_size];
uint t10 = SHIFT_RIGHT(candidates[i % row_size],16)^p10[i/row_size];
uint t15 = SHIFT_RIGHT(candidates[i % row_size],24)^p15[i/row_size];
uint k_e = SHIFT_LEFT(SHIFT_RIGHT(candidates[i % row_size], 32), 1);

t0  = sbox[t0 ];
t5  = sbox[t5 ];
t10 = sbox[t10];
t15 = sbox[t15];

// apply ShiftRows and MixColumns
//  also XOR with the upper key byte candidate
x0[i]=mult2[t0]^mult3[t5]^      t10 ^       t15 ^k_e;
x1[i]=      t0 ^mult2[t5]^mult3[t10]^       t15 ^k_e;
x2[i]=      t0 ^      t5 ^mult2[t10]^ mult3[t15]^k_e;
x3[i]=mult3[t0]^      t5 ^      t10 ^ mult2[t15]^k_e;

// apply disalignment
x0[i] = (x0[i] + disalignment) & 0xff;
x1[i] = (x1[i] + disalignment) & 0xff;
x2[i] = (x2[i] + disalignment) & 0xff;
x3[i] = (x3[i] + disalignment) & 0xff;

// select bits which affect cache index
x0[i] = SHIFT_RIGHT(x0[i], 4);
x1[i] = SHIFT_RIGHT(x1[i], 4);
x2[i] = SHIFT_RIGHT(x2[i], 4);
x3[i] = SHIFT_RIGHT(x3[i], 4);
''',
``round_kernel'',
preamble='''
#define SHIFT_RIGHT(X, Y) ((X >> Y) & 0xff)
#define SHIFT_LEFT(X, Y)  ((X << Y) & 0xff)
'''
)
```

A.3 Round to hit matrix kernel

Listing 1.3. Round To Hit Matrix Kernel

```
round_to_hits_kernel = ElementwiseKernel(ctx,
'''
```

```
uint *x0, uint *x1, uint *x2, uint *x3 ,
uint row_size, uint *sets_data_thresh,
uint sets_data_thresh_row_size
''',
'''
// (i/row_size) provides an index to the measurement row,
//  x[i] provides the offset to the cache set we wish to check
x0[i]=sets_data_thresh[(i/row_size)*sets_data_thresh_row_size+x0[i]];
x1[i]=sets_data_thresh[(i/row_size)*sets_data_thresh_row_size+x1[i]];
x2[i]=sets_data_thresh[(i/row_size)*sets_data_thresh_row_size+x2[i]];
x3[i]=sets_data_thresh[(i/row_size)*sets_data_thresh_row_size+x3[i]];
''',
``round_to_hits_kernel''
)
```

A.4 Axis sum kernel

Listing 1.4. Axis sum Kernel

```
sum_axis_column_kernel = ElementwiseKernel(ctx,
'''
uint *tmp, uint tmp_row_size, uint *out
''',
'''
// Use atomic_add to avoid data races,
//  not the fastest approach but the time it takes is negligible anyway
atomic_add(&out[i % tmp_row_size], tmp[i]);
''',
"sum_axis_column_kernel"
)
```