NCC Group Whitepaper

Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone

April 22, 2019 – Version 1.0

Prepared by

Keegan Ryan – Senior Security Consultant

Abstract

Trusted Execution Environments (TEEs) such as ARM TrustZone are in widespread use in both mobile and embedded devices, and they are used to protect sensitive secrets while often sharing the same computational hardware as untrusted code. Although there has been limited research in the area, the threat of microarchitectural attacks against ARM TrustZone has not been thoroughly studied. This is not the case for other TEEs, such as Intel SGX, where the security promises of the TEE have been violated numerous times by the academic community, showing that it is possible to use side-channel attacks to gain detailed insight into the microarchitectural behavior of trusted code. In this work, we show that TrustZone is susceptible to similar attacks, and we demonstrate the ability to achieve cache attacks with high temporal precision, high spatial precision, and low noise. These tools make it easy to monitor the data flow and code flow of TrustZone code with great resolution, and we apply our techniques to investigate the security of a real-world application. We examine ECDSA signing in Qualcomm's implementation of Android's hardware-backed keystore and identify a series of vulnerabilities that leak sensitive cryptographic information through shared microarchitectural structures. By using the powerful attacks developed in this paper, we are able to successfully extract this sensitive information and fully recover a 256-bit private key from Qualcomm's version of the hardware-backed keystore.

Table of Contents

1 Introduction

Modern devices are growing increasingly complex, with new security defenses being designed to defend against increasingly capable attackers. Even in the case of device compromise, defenders wish to find ways to secure sensitive information. Trusted Execution Environments (TEEs) aim to provide this level of protection by restricting secrets to a segregated domain. This environment may run on the same computational hardware that untrusted code uses, but the separation is distinct from the typical user-kernel or kernel-hypervisor boundaries. Even if an attacker fully compromises the operating system on a device, the secrets kept within the TEE should remain secure.

There are primarily two classes of TEE-supporting hardware which are commonly available today: Intel's Software Guard Extensions (SGX) and ARM's TrustZone. SGX is found on Intel processors and is available in both desktop and server environments [CD16]. SGX provides a way to run user-supplied code within an *enclave*, aiming to protect the memory contents and execution from all entities outside of the enclave. Even a malicious OS should be unable to leak secrets from an SGX enclave. However, Intel excludes side-channel attacks from its SGX threat model [Cor15, CD16], and the promised protections have been systematically and repeatedly broken by microarchitectural attacks, described below. However, despite this intense focus on SGX, the same attention has not been given to the exploitation of TrustZone.

This is a bit surprising, considering the widespread use of TrustZone. While practical applications for SGX are still being developed, ARM TrustZone is used daily by countless mobile devices and embedded systems. It is found in the flagship devices from Samsung, Google, and many others, and it is used for many sensitive operations such as full disk encryption, fingerprint processing, and device authentication [Sam16, CL16]. It is also advertised for use in smart home systems, digital currency wallets, and medical devices [Mar18].

That's not to say that the threat of compromising TrustZone has not been recognized. Both attackers and defenders have known for years that standard memory corruption bugs can be used to exploit vulnerable TrustZone implementations [lag15a, lag15b, Ros14], and so code is developed with these threats in mind. Other recent works have demonstrated that TrustZone TEEs can sometimes leak information through low-powered microarchitectural attacks, also described below. Despite these works, research into the microarchitectural security of TrustZone lags behind that of SGX, and the same level of powerful microarchitectural attacks have not been practically demonstrated against this ubiquitous TEE.

In this work, we show that powerful side-channel attacks are possible on ARM, and we use them to extract cryptographic keys from the hardened TrustZone-based keystore on Android phones.

1.1 Related Work

A number of recent publications have focused on TEEs, microarchitectural attacks, ARM, or Android keystores. In the controlled channel attack [XCP15], an untrusted operating system uses page faults to leak coarse control-flow information from SGX-based TEEs. The Cachezoom [MIE17] attack uses OS-scheduled timer interrupts to perform a cache attack on SGX with high temporal precision. Nemesis [VBPS18] targets multiple TEEs, including SGX, and infers execution information with single-instruction precision. Branch Shadowing [LSG+17] enables an attacker to target SGX enclave control-flow information with increased spatial resolution by monitoring the branch predictor. These attacks paved the way for compromising real-world cryptographic implementations in SGX [MES18, DDME+18].

ARMageddon [LGS+16] explores a cross-core cache attack on ARM-based mobile devices and considers the applicability to attacking TrustZone. However, this attack is performed from a compromised application, and it suffers from imperfect temporal resolution and measurement noise. TruSpy [ZSS+16] focuses on same-core cache attacks against TrustZone from both a compromised OS and compromised userland application. This attack recovers a secret key from an intentionally flawed AES implementation, but it also suffers from poor temporal resolution and measurement noise. CLKSCREW [TSS17] uses the malicious OS's access to energy management hardware to induce faults in TrustZone computations, extract an AES key from an intentionally flawed implementation, and bypass real-world code signing in the Nexus 6 cell phone.

Lastly, there has been research into the security of keys protected by Android's keystores. Researchers have found flaws in the lower-assurance software-based keystore [Cor14, ST16], but these do not impact TEE-based keystores. The security provided by these hardware-backed keystores is discussed elsewhere [CdRP14, CPVtG14], including analysis of Qualcomm's hardware-backed keystore. However, this research did not find ways to compromise keys in theses implementations. While hardware-backed keys can be extracted by compromising the entire TrustZone TEE [lag16b], we are unaware of any prior practical attacks on the hardware-backed keystore which do not require full compromise of the secure computing environment.

1.2 Contributions

This work significantly improves the power of microarchitectural attacks against ARM TrustZone. We take the concepts used to target Intel SGX and apply them to ARM TrustZone, demonstrating that a software-based attacker can gain substantial insight into the data flow and control flow of TrustZone applications. Our attacks improve on existing ARM microarchitectural techniques in three main categories.

1. *Temporal Resolution:* The TruSpy attack performs only one set of side-channel measurements per encryption in TrustZone [ZSS+16], meaning the order of memory accesses is indistinguishable by this attack. The ARMageddon attack demonstrates improved temporal resolution by executing the victim TrustZone code on one core and monitoring the last-level caches from a second core [LGS+16]. However, the temporal resolution of this style attack is limited, since increasing the sample rate leads to an increased probability of missing victim memory accesses [YF14]. In contrast, our attack is capable of achieving arbitrary temporal resolution, capturing TrustZone memory accesses which occur during very small time slices of victim execution.

2. *Spatial Resolution:* Existing attacks are limited by the dimensions of the memory caches. The TruSpy and ARMageddon attacks each have a spatial resolution of 64 bytes, since more granular accesses result in indistinguishable memory cache behavior. This applies to both attacks revealing control flow and data flow. Our attack reveals TrustZone control flow with 16-byte resolution, meaning code which was secure under the old attacks can be vulnerable to these new attacks.

3. *Noise:* The ARMageddon attacks must cope with cache thrashing, and this results in a compromise where, in one case, $\frac{1}{16}$ of victim accesses are missed. In addition, since unprivileged timing is used to distinguish cache behavior, there is a minute chance that the microarchitectural behavior is misidentified. Our attack improves upon this by drastically reducing the probability of missing certain victim behavior and virtually eliminating behavior misidentification.

While the limited information exposed by existing attacks is enough to compromise security in specific scenarios, our attacks are very general and capable of inferring fine-grained control-flow and data-flow in arbitrary TrustZone applications. This is in part due to our use of processor features accessible to our malicious but privileged OS, something which is a part of the TrustZone threat model [ARM15, Chapter 17] but has not been taken full advantage of in prior work.

We successfully implemented our attacks in a loadable kernel module, and we further built a set of extensible software tools around this kernel module to facilitate the collection, analysis, and automation of these side-channel attacks. Our software has been open sourced[1] and can be adapted to support new attacks, other

[1] https://github.com/nccgroup/cachegrab

sources of cache-based side-channel attack data, and new analysis methods.

Finally, to demonstrate the effectiveness of these attacks, this tool is used to compromise a 256-bit ECDSA key stored in Qualcomm's hardware-backed keystore. This attack would not have been possible without the development of our high-resolution microarchitectural attacks. The full attack completes in a matter of hours and shows that, despite the mitigations present in the library, small secret-dependent behavior is sufficient to completely compromise a private key.

1.3 Disclosure

We disclosed the key extraction vulnerability to Qualcomm in March 2018, and from then until October 2018, they developed, reviewed, and propagated a fix. Qualcomm notified affected OEMs and carriers at this point, triggering the start of a six-month recertification process. Finally, the issue was publicly disclosed in April 2018. This issue was assigned identifier CVE-2018-11976.

2 Background

2.1 TrustZone

Recent ARM Systems-on-Chip (SoCs) provide a Trusted Execution Environment (TEE) via TrustZone technology [ARM15, Chapter 17]. This TEE offers a way to run secure code in an isolated environment with the goal of protecting code execution from a potentially compromised operating system. TrustZone segments execution on the SoC into a Secure world and Non-secure world, and it imposes restrictions on how the Non-secure world can interact with the Secure world.

A processor can execute code within both of these worlds at different exception levels, starting at EL0 (Exception level zero). EL0 is used for userland code, EL1 is used for kernel code, and EL2 is used for hypervisor code. The exception to this split is EL3, or *monitor mode*, which only ever executes in the Secure world. Monitor mode handles interrupts and facilitates switching between the two worlds.

Code within the Non-secure world is typically feature-rich, such as the Android OS and applications in a cell phone or the Linux-based firmware in an embedded device. Code within the Secure world has limited functionality, as this reduces attack surface and makes it easier to audit the Secure world code for vulnerabilities.

All exception levels in the Secure world should be protected against the Non-secure world. That is, the untrusted OS in Non-secure EL1 should not be able to compromise the *trusted applications*, or *TAs*, running in Secure EL0, even though EL1 has access to a more privileged set of processor features than EL0. Even in the context of a fully compromised Non-secure OS, an attacker should be unable to access the memory or monitor the execution of a TA.

2.2 Hardware-Backed Keystore

One common use of TrustZone TAs on Android devices is the hardware-backed keystore. This feature frequently uses the Keymaster TA, which runs within the Secure world, to handle cryptographic operations such as generating private keys, importing keys, and creating signatures. Although the Android OS can direct the Secure world to sign arbitrary payloads, keys generated within the Keymaster TA should never be accessible to the Non-secure world. This work only focuses on the implementation of ECDSA within a particular version of Qualcomm's Keymaster TA, but other cryptographic algorithms are supported as well.

An Android application can use the hardware-backed keystore to interface with the Keymaster TA, enabling developers to create and use keys with the additional protections of TrustZone's Secure world. In the event that an attacker compromises the Android OS, they can steal keys which are only protected by the Non-secure world, but they should still be unable to extract keys generated within the hardware-backed keystore [Goo18a]. This property is called *device binding* [CdRP14], and it allows an application developer to be confident that all signatures from a bound key pair were generated by the same physical device.

This feature is useful for operations like device authentication. A device-bound private key can be used to sign a server-generated challenge, giving the server confidence that the request originated from the same physical device as prior requests. Using the hardware-backed keystore for this purpose is recommended by both Google [Wil17] and the FIDO Alliance [FID18]. This proof of device ownership for second-factor authentication requires that the device-binding of the keystore cannot be defeated.

The hardware-backed keystore contains additional mitigations against compromise, but these are not always used. First, it is possible to place restrictions on how frequently a hardware-backed key can be used or to require fingerprint verification before signing a challenge with a key [Goo18a]. Additionally, public keys in the hardware-backed keystore can be signed by factory-installed attestation keys, increasing confidence that a key was generated within the keystore and giving stronger guarantees than just device binding. Some devices, such as the Nexus 5X targeted by this work, lack a factory-installed attestation key. However, the lack of an attestation key is not considered a negative security indicator [Goo18c].

Whatever the use of the hardware-backed keystore, it is important that application developers can depend on this feature for security. Once a key has been installed within the hardware-backed keystore, this key should be close to impossible for an adversary to extract.

2.3 Cache Attacks

Caches are microarchitectural structures within processors which accelerate accesses to commonly used resources. When a vulnerable piece of code uses the cache in a way that depends on sensitive information, an attacker can sometimes monitor the cache and observe the change in speed and cache behavior to infer the victim's secrets. These attacks can be used to reveal the locations where the victim is accessing data, referred to as data flow, or which code the victim is running, referred to as control flow.

Our use of cache attacks focuses on set-associative caches. In a set-associative cache, the entire memory address space is segmented into a number of short *cache lines*, often 64 bytes in length. Each cache line is bucketed into one of typically a few hundred *cache sets*. When information from a cache line is accessed, the entire cache line is read into the assigned cache set. Each cache set can only hold a fixed number of cache lines, referred to as the number of *ways* or the *associativity*. That is, in a 4-way set-associative cache, each set can hold four cache lines. If a fifth entry belonging to the same set needs to be stored in the cache, one of the four existing entries in the set must be evicted to make room.

In many cache attacks, the attacker and victim share the same cache. This means that the process of storing a victim line in the cache can evict an attacker's line, and vice versa. Cache attacks also require that the attacker has a way of querying the cache state to know if a particular cache line is stored in the cache or not. There are a number of viable cache attacks, but our work only focuses on one.

2.3.1 Prime+Probe

Prime+Probe [OST06] is a cache attack where the attacker and victim possess distinct cache lines. The attack has two phases, and it begins with the attacker selecting a particular cache set to monitor. In the *priming* phase, the attacker accesses several particular locations in their own memory to fill up every way of the cache set with attacker data. The victim process then executes, potentially accessing a cache line belonging to the monitored set. If this happens, one of the attacker's lines is evicted to make room for the victim's entry.

In the *probing* phase, the attacker determines which of their cache lines remain in the cache. This could be done by timing accesses to the attacker's cache lines which were previously in the cache. If the cache line is still in the cache after the victim execution, the access is fast, while if the line has been evicted, the access is slow. Regardless of how the processor chooses which line in a set to evict, the attacker knows that if all of their lines are still cached, the victim has not accessed a line in the set. If one of the attacker's lines is not cached, the victim has accessed that set.

An attacker can monitor multiple cache sets simultaneously to learn more detailed information about which sets the victim accesses. However, note that the attacker does not learn which cache lines the victim accesses or the contents of the cache lines.

2.3.2 Repetition in Attacks

An attacker may perform a cache attack multiple times throughout the victim execution, partitioning victim execution into a sequence of disjoint time steps. Each time step reveals information about victim cache usage, so the attacker can observe how the victim's behavior changes over time.

There are multiple ways an attacker can perform this repetition [GYCH18]. In a hardware-threading attack such as [Per05], the attacker and victim run concurrently on the same core, simultaneously accessing a shared cache. However, such simultaneous attacks may miss victim accesses that occur during the attack setup or measurement phases, preventing an attack from having both low noise and high temporal resolution [YF14].

In a time-sliced attack, the attacker and victim are interleaved in time on the same core. Since the victim is not executing during the attack setup and measurement, the attack captures all victim accesses during a single time step. However, the attacker also captures accesses caused by switching contexts from attacker to victim code, introducing noise. Time slicing of victim execution can be controlled by the attacker [MIE17] or not, as in [GYCH18].

The result of a repeated Prime+Probe attack is a two-dimensional Boolean array where one dimension specifies the time step, the other specifies the list of monitored sets, and the data show whether a particular set was accessed by the victim during a given time step.

2.3.3 Types of Caches

There are multiple caches within a processor [GYCH18], but this work concentrates on three.

- *L1D:* The Level 1 Data cache is used by a processor to store memory which was recently accessed by code. This cache is typically unique to a single core, and an attack on the L1D cache reveals data flow.
- *L1I:* The Level 1 Instruction cache stores memory containing recently-executed instructions. This cache is also typically unique to a core. An attack on the L1I cache reveals control flow.
- *BTB:* The Branch Target Buffer is a cache-like structure which stores information about the branches in cache-line-sized sections of memory. This information is used to predict the direction branches are taken, potentially reducing the time spent speculatively executing the wrong branch and speeding up instruction execution. Instead of storing a copy of the line's memory contents within the cache, the cache stores prediction information such as prior branch results and destinations. It is possible to perform cache attacks on the BTB [AKS07b, AKS07a, EPAG16, ERAG+18] to reveal control flow.

2.4 ECDSA side-channel attacks

ECDSA private keys have been successfully targeted by side-channel attacks in the past [Rya19, vdPSY15, GB17]. ECDSA is a NIST-standardized digital signature algorithm [KSD13] that relies on the difficulty of the elliptic curve discrete logarithm problem. Signatures consist of two finite field elements $r, s \in \mathbb{F}_q$ and are calculated from a hash m of a message and private key x:

$$\begin{aligned} r &= [k * G]_X \\ s &= k^{-1}(m + rx) \end{aligned}$$

The value $k \in \mathbb{F}_q$ is the *nonce*, and G is a base point of order q on the elliptic curve. r is therefore calculated by taking the X coordinate of the scalar multiplication of k and G. k is also used when calculating s from r, x, and m.

Howgrave-Graham and Smart recognized that partial information about k can be reformulated into partial information about x via representation as an instance of the hidden number problem (HNP) [HGS01]. Solving this problem with the lattice-based approach of Boneh and Venkatesan [BV96] yields private key x and allows the attacker to forge signatures as the victim.

To perform this *nonce-leak* attack, the attacker must learn sufficient information about the value of k. In the original attack, this is accomplished by knowing a combination of the most-significant and least-significant bits in the $\lceil \log_2 q \rceil$-bit representation of q. This approach requires a single continuous block of unknown bits in the middle of k, but there are other approaches which enable key recovery when there are multiple blocks of unknown bits [HGS01, NS01, NS02, HR07, vdPSY15, FWC16a]. However, these cases require proportionally more bits of k to leak per signature or other special conditions in order to recover x.

A sufficient number of signatures with nonce leakage must be observed before the instance of HNP is solvable. The minimum number of bits leaked per nonce and signatures required depend on the key length and choice of HNP-solving algorithm. Lattice-based algorithms can occasionally recover a 256-bit key when three most- or least-significant bits of each nonce are known for around 100 signatures [LCL13], and they can reliably recover keys when at least four bits are known for around 80 signatures [Rya19]. Alternative methods to solving the HNP exist [DMHMP13, AFG+14, TTA18] that can recover a 256-bit key with three or even two bits leaked per nonce, but these require millions to even trillions of signatures.

Many attacks have been successfully performed using nonce leaks [FWC16b, BT11, BvdPSY14, BH09, GPP+16, BFMRT16]. A common approach is to target the point multiplication of G by k, as the behavior of the multiplication algorithm may depend directly on the values of the bits of k.

2.5 Fixed-Window Multiplication

Computing the point multiplication of G by k is central to ECDSA signing, and numerous algorithms exist to efficiently calculate this value. The properties of elliptic curves used for cryptography make it easy to flip the sign of a point (inversion), compute the sum of one point with another (addition), and compute the sum of a point with itself (doubling). Efficient multiplication algorithms combine these elementary operations to compute the final product.

Fixed-window multiplication (See [KMVOV96, Algorithm 14.82]), with window size four, separates the nonce k into l four-bit windows, with k_0 being the four most-significant bits, k_1 being the four next-most-significant bits, continuing up to k_{l-1}, the four least-significant bits. The fixed-window multiplication algorithm is described in Algorithm 1.

Algorithm 1: Fixed-Window Point Multiplication

Input: Nonce values $k_0, k_1, \ldots, k_{l-1}$
Output: Product of the nonce and base point G

```
$PrecomputedTable \leftarrow [0 * G, 1 * G, \ldots, 15 * G]$
$T \leftarrow 0 * G$
for $i \leftarrow 0$ to $l - 1$ do
    $T \leftarrow 16 * T$                          // Double point $T$ four times
    $P \leftarrow PrecomputedTable[k_i]$
    $T \leftarrow T + P$                           // Add to point $T$
return $T$
```

Note that this algorithm involves four doubling operations and one addition operation per iteration. Also note that this algorithm indexes into a precomputed lookup table. If an attacker can reveal the retrieved indices k_i of multiple consecutive iterations, the attacker learns many consecutive bits of k which can be used in a nonce leak attack.

3 Qualcomm ECDSA Implementation

To motivate the development of our microarchitectural attack techniques, we examine the implementation of ECDSA signing in a particular version of Qualcomm's Secure Execution Environment (QSEE). QSEE is Qualcomm's implementation of a TrustZone Secure world OS and backs operations which need to be protected from the Android OS [lag16a]. This ECDSA implementation is used by the hardware-backed keystore on recent Android devices with Qualcomm hardware.

In particular, we examine the QSEE ECDSA implementation on the Nexus 5X. We target firmware version 7.1.1 (N4F26T [Goo17]), which was released in March 2017, near the beginning of this research. However, this choice of target does not necessarily mean that implementation weaknesses only impact the Nexus 5X. Many other devices use QSEE to support hardware-backed keys, and so these devices may inherit the same weaknesses from shared QSEE source code.

3.1 Algorithm

QSEE's implementation of elliptic curve point multiplication was identified in the `cmnlib` block device[2] at offset `0x00ff04`. We analyzed this function in order to understand how QSEE performs the multiplication.

QSEE uses a variant of fixed-window multiplication techniques with a window size of four bits. However, there are important differences between QSEE's implementation and standard fixed-window techniques which alter how nonce bits are consumed by the algorithm. In the standard algorithm, every iteration of the loop consumes four bits of the nonce to select one of sixteen elliptic curve points $\{0 * G, 1 * G, \ldots, 15 * G\}$. The points are selected by referencing an index in a precomputed table with sixteen entries.

In QSEE's implementation, every iteration of the loop consumes four bits of the nonce to select one of sixteen elliptic curve points $\{-15 * G, -13 * G, \ldots, 13 * G, 15 * G\}$. These points are selected by referencing an index in a precomputed table with eight entries $\{1 * G, 3 * G, \ldots, 15 * G\}$ and conditionally flipping the sign of the precomputed point, which can be done quickly. It is not immediately clear why this difference exists, but the purpose may be to reduce the size of the lookup table or to avoid operations involving the point at infinity $0 * G$, which QSEE handles separately.

To support this smaller lookup table, the nonce is effectively recoded on the fly, converting four bits of nonce k_i to a 1-bit sign σ_i and three-bit index μ_i. To generate these k_i, a 256-bit nonce is first zero-extended at its beginning to 288 bits and then for the sake of the recoding is prepended with a single set bit. Besides extending the length to a multiple of four bits, the zero-extension has no bearing on the correctness of the algorithm. The four most significant bits of this 289-bit value are k_0, the next four bits are k_1, and so on, up to k_{71}. The recoding to σ_i and μ_i, for $i \in \{0, \ldots, 71\}$ is found by

$$\begin{aligned} k_i &= \lfloor (k + 2^{288}) / 2^{285-4i} \rfloor \bmod 16 \\ \sigma_i &= \lfloor k_i / 8 \rfloor \\ \mu_i &= \lfloor |2k_i - 15| / 2 \rfloor. \end{aligned}$$

Note that this approach consumes 288 bits of the 289-bit value, and the least significant bit has not been recoded. Let σ_{72} be the least significant bit of the nonce k.

The behavior of the QSEE multiplication algorithm is described in Algorithm 2. Note once again that, in a single iteration, there are four doublings, one lookup in a precomputed table, and one addition.

Although the QSEE implementation does not explicitly calculate the recoded σ_i and μ_i before running the

[2]This image is pulled from `/dev/block/platform/soc.0/f9824900.sdhci/by-name/cmnlib`.

Algorithm 2: QSEE Point Multiplication Algorithm

Input: Recoded nonce values $\sigma_0, \mu_0, \ldots, \sigma_{71}, \mu_{71}, \sigma_{72}$
Output: Product of the nonce and base point G

```
1  $PrecomputedTable \leftarrow [1 * G, 3 * G, \ldots, 13 * G, 15 * G]$
2  $T \leftarrow 0 * G$
3  for $i \leftarrow 0$ to $71$ do
4      $T \leftarrow 16 * T$                                  // Double point $T$ four times
5      $P \leftarrow$HardenedLookup($PrecomputedTable, \mu_i$)
6      $P \leftarrow$ConstantSelect($\sigma_i, P, -P$)
7      $T \leftarrow T + P$                                   // Add to point $T$
8  if $\sigma_{72} = 1$ then
9      $sel \leftarrow 0$
10 else
11     $sel \leftarrow 1$
12 $T \leftarrow$ConstantSelect($sel, T - G, T$)
13 return $T$
```

algorithm, the end result of having a 3-bit index and a possible inversion per loop iteration is still present.

Recall that an attacker is interested in recovering the least significant or most significant bits of nonce k. By our recoding definition, the least significant bit of k is σ_{72}, which specifies whether a conditional subtraction should take place. Due to the prepending of k with 32 unset bits and one set bit, the contents of $k_0, \ldots, k_7$ are known and constant, and recovering these values does not give additional information about the nonce. However, the lower three bits of k_8 are the most-significant bits of the original k, and the top bit of k_8 is unset due to the zero-extension. This means that k_8 takes one of eight values, and each value uniquely maps to a single possible value of μ_8. Thus observing μ_8 allows us to recover the three most-significant bits of k.

To make it clear that we are targeting the unknown bits of the nonce, we refer to the `HardenedLookup` call with argument μ_8 as the *first* table lookup. Even though there are prior lookups determined by the constant zero-extension bits, this is the first lookup operation within the precomputed table which involves the secret bits of k.

Learning whether or not the conditional subtraction occurs at the end of the algorithm and the index of the first table lookup therefore reveals σ_{72} and μ_8. This in turn reveals the least-significant bit and the three most-significant bits of k. By [HGS01], the ability to recover these four bits is sufficient to reliably compromise a 256-bit ECDSA key. The remainder of this section explores how to leak this information.

3.2 Conditional Subtraction Implementation

The comparison on line 8 of Algorithm 2 leaks the final bit via non-constant control flow. Although the algorithm uses a constant-time conditional selection method on line 12, this method takes a mask as an argument to determine which value to select. As found at offset `0x01038c`, the assignment of the mask has non-constant control flow.

Note that after evaluating the last bit σ_{72}, the code may proceed to either the if block or the else block, and control flow is eventually reunited at line 12. However, the behavior of this conditional leaks through microarchitectural state in two ways. Because of how the program was compiled, line 11 falls in a different L1I cache set than the other lines. Thus if a cache attack observes the victim using the L1I cache set of line 11, then the attacker infers that the last bit of the nonce is not set.

The branching behavior can also be observed by monitoring the BTB. There is an unconditional branch from line 11 to line 12, and this branch falls in a different BTB set than any of the neighboring branches. Concretely, when the last bit is set, the victim uses BTB sets 57, 58, and 59, but when the last bit is unset, they use sets 57, 58, 59, and 60.

Since the critical L1I and BTB sets are both used by the victim in non-secret-dependent ways elsewhere in the algorithm, a successful time-sliced attack must include a time step that captures only the secret-dependent usage of these sets. This requires high temporal resolution. However, provided the attacks are capable enough, this side-channel data leaks the last bit of the nonce in a straightforward way.

3.3 Table Lookup Implementation

The table lookup on line 5 is complicated and attempts to hide the index value with randomized masking, but it also leaks the index information through a combination of non-constant control flow and memory accesses. A more complete representation of the lookup function is given in Appendix A, but due to the complexity of this implementation, we will begin by describing a naive lookup implementation and explaining how Qualcomm's version differs.

Recall that the table has eight entries, which we refer to by the indices $1, 2, \ldots, 8$. Each entry has size 220 bytes, so this means that each entry occupies different sets in the L1D cache. We will consider the order of accesses to the memory of the entries in this table. Since memory operations access four bytes of memory at a time, the naive implementation reading entry 3 to an output buffer would have memory access pattern **33333** . . .. However, an attacker monitoring the L1D cache can easily observe that the secret index is 3, so this naive implementation is insufficient.

The Qualcomm implementation instead reads from all eight entries and uses eight logical masks to determine which values to keep and which to discard. These logical masks are applied in constant time. Here, the access pattern appears as **12$\underline{3}$456781 2$\underline{3}$4** . . . where the underline denotes that the mask used preserves the read value. By changing which of the eight masks is set, the lookup function no longer leaks information through the L1D cache. However, if the attacker learns that the third mask is set, the attacker has successfully leaked that the index is 3.

The Qualcomm implementation is further complicated by randomization. Every call to the lookup function randomly changes which entry is read from first. For example, the access pattern may actually look like **7812$\underline{3}$4567812$\underline{3}$4** . . .. Even though the first access is randomized, the progression from one access to the next stays the same, so, for example, 4 is always read following 3.

Even with this randomization, note that if the attacker learns that the first entry read is 7 and the fifth mask is set, then the attacker knows that the secret index is 3. The attacker therefore has the goals of detecting which entry is read first and which mask is set.

3.3.1 Recovering Entry Ordering

The first piece of information leaks through a time-sliced L1D cache attack. Note that, in the previous example, a time step at the beginning of the table accesses may observe L1D sets corresponding to entries 1, 2, 7, and 8 (from access pattern **7812**). Even though the cache attack does not directly reveal the order of these accesses in this time step, the attacker can use the knowledge of the fixed progression to infer that 7 was accessed first.

Not all possible time steps will successfully leak this information. Assume instead that the attack captures the first 11 table accesses **78123456781** in a single time step. The L1D cache will show accesses to all eight entries, and it will be impossible to disambiguate which one came first. It is therefore crucial that the L1D cache attack has high temporal resolution in order to capture as few table accesses as possible during the first time step of the table lookup.

3.3.2 Recovering Mask Values

Of the eight logical mask values used in the table lookup function, only one of them will be set, since only the values from one table entry need to be written to the output buffer. These masks are applied in constant time, but much like the flaw in Section 3.2, the mask values are not initialized in constant time.

Each of the eight logical masks is cleared or set by the if block or the else block of a conditional statement. Since there are eight conditional statements and only one of the masks is set, executing this part of the lookup function will result in executing seven if blocks and one else block. Unfortunately for the attacker, these eight possible control flows use the same L1I cache sets, so a L1I cache attack would be infeasible.

However, as before, the else blocks contain unconditional branches, and these branches are only taken when the else block is executed. These unconditional branches can then be observed in particular BTB sets, and identifying which of the eight rejoining unconditional branches is taken reveals which mask is set. In practice, not all possible control flows result in a unique BTB usage signature, but three of them do. This adds extra attack overhead, but the attacker may just collect samples until one of these three cases is encountered.

3.4 Key Recovery

A combination of powerful data-flow and control-flow time-sliced cache attacks is therefore capable of revealing the final nonce bit and the index of the first table lookup, exposing four nonce bits and making key recovery possible. The following section details how Cachegrab is able to accomplish such attacks with these desirable properties.

4 Cachegrab

We developed the Cachegrab tool[3] in order to facilitate performing practical and powerful microarchitectural attacks on ARM. Since TrustZone implementations are commonly on mobile and embedded devices, and these devices often run a Linux variant for the Non-secure OS, the heart of Cachegrab is implemented as a loadable kernel module. This is in line with TrustZone's threat model of a compromised Non-secure world, and it makes it possible to perform these attacks with minimal changes to the Non-secure OS.

Cachegrab also contains a companion userland application and visualization software to assist in these attacks. It currently supports passive time-sliced Prime+Probe attacks on AArch64 ARMv8 systems, but it is flexible and allows expansion to support new passive time-sliced attacks or other platforms. Using Cachegrab, multiple attacks can be performed simultaneously, exposing the usage of multiple caches over the same time steps.

4.1 Temporal Resolution

In order to achieve high temporal resolution, the attacker's code, running in the Non-secure world, must be capable of frequently examining the state of shared caches during the execution of the victim's code in the Secure world. The caches targeted by Cachegrab are unique to individual cores and used exclusively by a single thread, so the only way to poll the cache state during victim execution is by halting the victim and executing attacker code on the victim's core before resuming the victim. This repeats until the victim has completed execution.

This interleaving of the victim execution between priming and probing means that the attacker does not miss any victim accesses. It also means that the cache accesses from switching into and out of the TA will be mixed in with the desired TA cache information. Since some of these extra accesses come from the attacker's interrupt handling code, the attacker's memory can be marked as uncacheable, partially mitigating this issue.

In order to achieve this interleaving, we adopt the techniques of [MIE17]. This attack uses timer interrupts to halt SGX enclave execution on Intel processors and transfer control back to the malicious OS, which performs the Prime+Probe attack. By increasing the frequency of interrupts, the authors could achieve arbitrary temporal resolution.

This attack does not necessarily work on TrustZone-capable ARM devices. When a interrupt occurs during Secure world EL0 execution, control transfers to Secure world EL3 monitor mode, not to the attacker's code. This was observed by [ZSS+16] as a limitation of ARM systems and an indication that time-sliced attacks on TrustZone cannot use interrupts.

However, a Non-secure Linux OS relies upon interrupts to function, so in practice, the monitor mode may just transfer control directly to the Non-secure world [ARM15, Figure 17-1]. This occurs with minimal processing to ensure Non-secure interrupts are fast.

This behavior is exploited by Cachegrab to achieve high temporal granularity. One physical core is assigned to run the victim code and another is assigned to trigger the interrupts. As the victim executes the TA, the second core uses `smp_call_function_single` to trigger an inter-processor interrupt on the first core. This halts execution of the TA, transfers control to the monitor mode interrupt handler, and receives the forwarded interrupt within the untrusted Linux OS. This calls the attacker's code on the first core, and victim execution resumes when the attacker's interrupt finishes. This achieves the attacker-victim interleaving at an attacker-controlled frequency, making high-temporal resolution Prime+Probe attacks possible.

[3] https://github.com/nccgroup/cachegrab

4.2 Noise Reduction

An attacker must be able to reliably detect the state of shared caches in order to mount a successful microarchitectural attack. In the Prime+Probe attack, the attacker must detect if their cache lines have been evicted from the cache by a victim entry. Traditionally, this has been done by using a timer to measure the latency of a memory read operation [Yar16, GYCH18]. Cache misses take longer to complete than cache hits, so a timer can usually reveal if an entry is in the L1 caches.

Timers are useful in these attacks because they are accessible from low-privileged execution contexts, enabling one user process to attack the kernel or another process. However, this limitation does not exist in the TrustZone threat model, since the attacker controls the high-privileged but Non-secure EL1. Cachegrab takes advantage of this fact by querying the processor directly for cache state information.

ARMv8 processors contain performance counters which can be configured to monitor for various events. Among these are L1D cache refills (misses), L1I cache refills, and branch mispredictions [ARM16, Section 11.4.2].

For attacks on the L1D cache, we read the relevant performance counter to record the number of L1D misses, load a previously primed attacker cache line, and reread the number of misses. If this number incremented, the probed address had been evicted from the cache.

For the L1I cache, we place the performance counter read instructions in separate L1I cache lines and prime the L1I cache by executing this measurement code. During probing, executing the measurement code again causes potential misses as the L1I cache lines are reread, and these misses are captured by the performance counter reads.

For the BTB, we use the experimentally observed fact that branches not in the BTB are predicted to be not taken. We prime the BTB by executing several branches and training the predictor that these branches are always taken. We probe the BTB by executing the same branches with periodic performance counter reads. If the entry is still in the BTB, the branch is correctly predicted as taken, but if not, the branch is mispredicted. While the timing difference of a single mispredicted branch may be small [LSG+17], the use of performance counters gives a very low noise signal for BTB behavior.

4.3 Spatial Granularity

Similar to [LSG+17], we use the branch predictor to improve the spatial resolution of control-flow attacks. Attacks on the L1D and L1I caches are limited by the size of cache lines, which is typically 64 bytes. However, the effective cache line size of the cache-like BTB may be even smaller.

In [LSG+17], the aliasing behavior of Intel's BTB means that an attacker could place a branch at a particular address so that executing the branch would reference the BTB entry of a branch in the victim's address space. This is analogous to performing a cache attack on shared memory, since the attacker and victim branches alias to the same BTB entry.

Our attack follows [AKS07b] in filling the BTB with attacker entries which map to the same BTB set as a victim branch but not to the same BTB entry. Their approach is analogous to a Prime+Probe attack, where the attacker and victim share a cache, but do not share cache entries. Unlike [AKS07a], we find that the number of attacker branches per set should equal the associativity of the BTB and not more. Since we do not explicitly consider the eviction policy of the BTB, it is possible that priming one attacker branch could evict another primed attacker branch, leaving some victim entries in the BTB set after priming completes. We find that executing all attacker branches several times is enough to ensure that only attacker entries are likely to be the only ones in the BTB.

5 Results

We used these side-channel attacks to successfully extract an unknown ECDSA P-256 private key from Qualcomm's TrustZone keystore. To perform the test, we installed Cachegrab on a rooted Nexus 5X, reflecting the level of access of an attacker who has compromised the Non-secure Android kernel. Note that rooting the device does not affect the operation of the keystore in the Secure world.

We performed the attack in two stages. First, we imported a known 256-bit key x into the hardware-backed keystore. By the ECDSA equations, we can use x and a generated signature (r, s) to recover nonce k by $k = s^{-1}(m + rx)$. We can then compare this actual nonce value against the nonce bits recovered by the side-channel attacks on that signature. We then tuned the attack and analysis to reduce the probability of recovering an incorrect value.

Next, we generated an unknown 256-bit ECDSA key within the hardware-backed keystore, and we ran the tuned attack against this unknown key. When the side-channel data of a single signature captured the needed data, the attack would output its guess of the partial nonce bits for that signature. When enough nonce-bit guesses were recovered, we used the lattice-based key recovery algorithm to generate candidate values for the private key. These were checked against the public key reported by the keystore until the correct private key was found. Finally, to analyze the attack performance, we used the recovered private key to calculate the nonces of every signature and determine the frequency of successful nonce-bit leakage.

The Nexus 5X uses the Qualcomm Snapdragon 808, which has four Cortex-A53 cores and two Cortex-A57 cores [Qua15]. We configured Cachegrab to run the victim process on CPU 5, and to trigger interrupts from CPU 4, the two Cortex-A57 cores. The time step was configured to value 2500 with a maximum number of 20000 samples during the ECDSA signing operation of the Keystore TA.

The attack simultaneously monitored the L1D cache and the BTB. The L1D cache is 2-way set associative with 256 sets and a line size of 64 bytes. The BTB behaves as a 2-way set associative cache with 2048 cache sets and an effective cache line size of 16 bytes, as we determined by using the techniques in [UM09]. The L1D module in Cachegrab collected sets 89 through 118 inclusive, which contain the precomputed table. The BTB module collected sets 1875 through 2047 and 0 through 102 inclusive.

5.1 Initial Processing

An example capture with these settings is shown in Figure 1. As progress within the algorithm increases from left to right, the repetitive behavior in the L1D and BTB traces reveals that the majority of the ECDSA signing operation is spent within a loop. This corresponds to the loop of the QSEE elliptic curve multiplication algorithm. We can also identify the portions of the trace which correspond to setting up the precomputed table and the conditional subtraction based on the final bit.

Note that this image also contains horizontal lines and vertical lines artificially rendered in a lighter color. The horizontal lines correspond to cache sets that are used in every time step, a phenomenon which was also observed in the Cachezoom attack [MIE17]. The behavior is likely due to cache sets which are hit by the context switching code as the target core transitions between the attacker's code and the victim's. These horizontal lines can be easily filtered out, and have little effect on the final analysis.

The vertical lines correspond to the cache attack being mounted against the wrong victim code. For example, when Cachegrab completes the priming and returns from the interrupt, the monitor mode could run other code instead of transferring control back to the Keymaster TA. Therefore, these time steps contain no useful information about the ECDSA private key. Empirically, it is easy to identify these time steps by which BTB sets are activated, and so this source of noise can be filtered out as well.

After removing the noise, our processing script attempts to identify important locations within the traces.

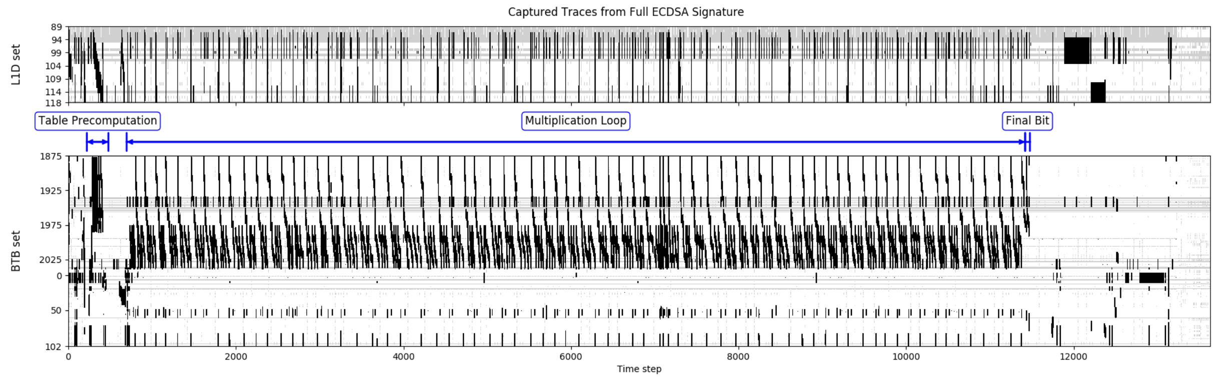

Figure 1: L1D and BTB capture of a single ECDSA signature. The connection to the multiplication algorithm is apparent, and it is clear that the majority of the signature generation is spent performing the elliptic curve point multiplication. A combination of manual and automatic processing is used to highlight the footprint of ECDSA signing on the caches, and the irrelevant accesses are rendered in light gray. This is simply to produce a clearer representation of ECDSA behavior; the processing in the attack itself is fully automated.

Since the number of victim instructions executed per time step varies slightly, we cannot simply use an absolute offset to know which time steps contain key data. We first attempt to identify the 72 table lookups by searching for the lookup function's branch signature within the BTB trace. If the script fails to find 72 lookups, the sample is discarded. This gives enough information to confidently know where the first three and final one nonce bit are processed within the traces.

5.2 Recovering the Final Bit

The leaky control flow involving the last bit of the nonce occurs not long after the final table lookup. This is shown in Figure 2. It is clear that during this part of the trace, the implementation always uses BTB sets 57, 58, and 59, but only uses set 60 when the final bit is unset. Recovering the final bit is as simple as detecting when sets 57, 58, and 59 are used and then checking to see if set 60 is used at the same time. Observe that about 30 time steps after this leakage, both samples use BTB set 60 for a later branch. If not for the temporal resolution of Cachegrab, the secret-dependent use would be combined with the later unconditional use of set 60, and we would not be able to leak the final bit.

5.3 Recovering the First Three Bits

The final hurdle in our key-recovery attack is determining the index of the first table lookup. A single iteration of the loop in the QSEE multiplication algorithm is shown in Figure 3, and the relationship to Algorithm 2 is apparent. Four doubling operations are followed by a lookup operation and an addition. The lookup operation consists of both branching behavior in the BTB cache to initialize the logical masks and activity in the L1D cache as all the entries in the precomputed table are accessed. This matches the expected behavior from analyzing the QSEE binary.

Since we are interested in recovering the index of a particular table lookup, we examine this part of the traces more closely. Two table lookups from two independent samples are shown in Figure 4. Recall from Section 3.3 that in order to recover the index, the attacker must detect which entry in the table was accessed first, and which of the eight logical masks was set. In Sample 1 in the figure, the L1D trace reveals that entry 7 was accessed first, and the BTB trace reveals that the eighth mask was set. From this, we infer that the

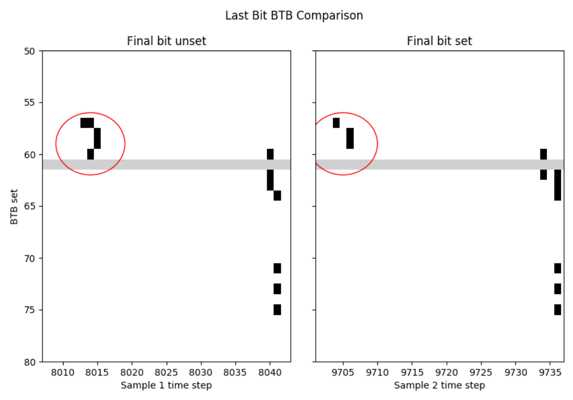

Figure 2: Comparison between BTB traces when the last bit is set verses unset. The circled region highlights the variation in control flow when setting the mask prior to the final addition. Although both samples use set 60, this is about 30 time steps after the nonce-dependent usage, so the nonce bit is easily distinguishable. Once again, noise that is independent of the multiplication algorithm has been drawn in gray.

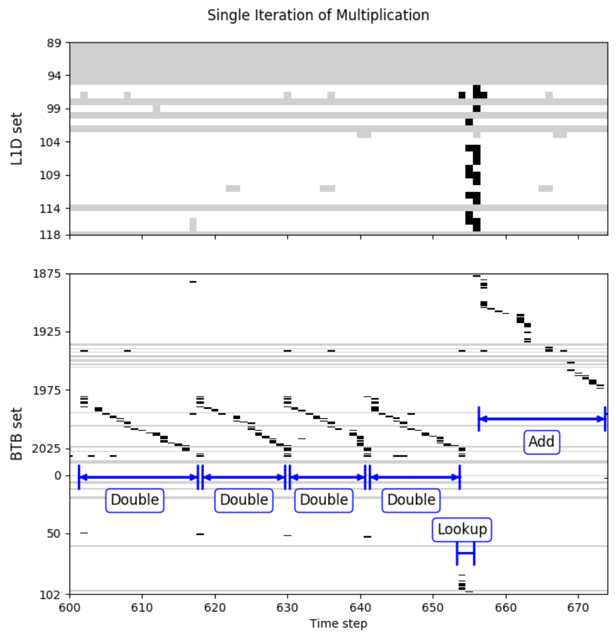

Figure 3: Single iteration of the multiplication loop. The BTB reveals the locations of the four doubling operations, the L1D cache reveals the numerous accesses to the precomputed lookup table, and the BTB reveals the location of the addition of the retrieved value. Note that the doubling operations are visually similar, but not identical. This is due to small variations in how many victim instructions are executed between attacker interrupts.

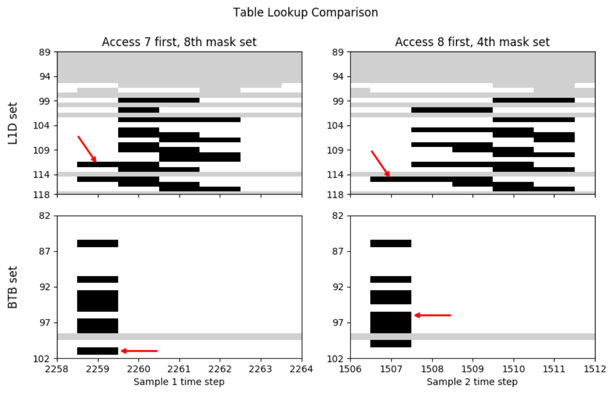

Figure 4: Comparison between the L1D and BTB trace behavior of the lookup function under different inputs. In these examples, the L1D behavior reveals which entry in the precomputed table was accessed first, and the BTB behavior reveals which mask was set. The arrows indicate the varying data points which correspond to the particular index retrieved by these executions.

access pattern was **781234567812** . . ., and therefore the sixth entry in the precomputed table was retrieved. Similarly, Sample 2 shows that entry 8 was accessed first, and the fourth mask was set, so the access pattern was **8123456** . . .. Thus the secret index of the lookup in this sample was 3.

Note that the presence of horizontal lines means that some of the victim cache behavior is not visible to Cachegrab. In some situations, it wasn't possible to uniquely determine both the first entry accessed and which mask was set, so these samples were simply discarded.

5.4 Analyzing Recovered Bits

The end result of this processing step for a single sample is either a guess of four nonce bits or discarding the sample. If a guess is made for a given signature sample, it can be combined with the signature values to create an HNP inequality. The lattice reduction method for solving the HNP is susceptible to erroneous HNP inequalities, so we employed random subsampling to improve the odds of a successful key recovery [BT11]. This technique still requires that the error rate of nonce-bit recovery is low, so the processing was tuned to err on the side of not guessing at nonce bits when there was uncertainty. The lattice reduction method successfully recovered the private key, and then we reanalyzed the collection data to determine our accuracy of recovering nonce bits.

In total, 12000 signature samples were collected against the unknown private key. This took 14 hours for an average rate of four seconds per sample. It took an additional 25 minutes to analyze these samples in an attempt to recover HNP inequalities. 3488 samples were discarded because the analysis did not detect the correct number of table lookups. The last bit was incorrectly recovered from five of these (0.06% error rate). Of the 8512 samples remaining, the analyzer attempted to recover the index for 70 of the table lookups in each sample. The analyzer was confident enough to guess at the index value for 7209 of these $8512 \times 70 = 595840$ lookups (1.21%); 7109 were guessed correctly (1.4% error rate). To generate an HNP inequality, the guessed index must correspond to the first block of the nonce, which occurred 102 times. Two of these inequalities were inaccurate: one recovered the wrong index, and one recovered the wrong last bit. The HNP solver used subsamples of 84 inequalities per lattice reduction, giving an approximately 3% chance that the subsample is completely free of errors. This process found the correct private key after five attempted reductions and about two minutes of processing time.

6 Discussion

Through the development of powerful side-channel attacks against ARM TrustZone, we have demonstrated that it is possible to break the security of Qualcomm's hardware-backed keystore. However, despite the significant attack resolution provided by Cachegrab, there are currently some drawbacks to these attacks, limiting the scenarios in which a practical attack can be performed.

Since every time step of captured victim execution requires executing the attacker's full Prime+Probe code, increasing the temporal resolution of the attack increases the total amount of time spent executing attacker code. This dramatically slows TA operations, as was observed in the four-second-long ECDSA signature operation during our attack. Practical exploits must balance this cost with the required resolution of the attack. Additionally, there appears to be a practical limit to temporal resolution. Due to the variability in the time it takes to execute the monitor mode management code, reducing the time between attacker-controlled interrupts increases the probability that the interrupt fires before the monitor mode has transferred control back to the TA. That time step contains no useful information and must be discarded, but it still incurs the timing cost of the Prime and Probe steps. This makes trace captures unacceptably slow and so single-instruction time steps are currently impractical.

The interrupt-handling code which executes every time step also introduces a limitation by systematically polluting certain cache sets after an interrupt fires. Although the contribution of these effects from interrupt-handling code in the Non-secure OS can be mitigated by mapping uncacheable memory, the same cannot be done for the interrupt code in the Secure world. This means that we are unable to reliably observe the TA's usage of certain cache sets. However, the Secure world interrupt code experimentally has a small cache footprint, and this may be found in other systems as well, since interrupt code may intentionally include limited cache accesses as a way to improve system performance.

Our full attack against Qualcomm's hardware-backed keystore is highly tailored for a particular software version on an individual device. Although the same effort could be replicated for other devices, many properties of our chosen system coincidentally aligned with the requirements of the attack. The BTB of the Cortex-A57 core in the Nexus 5X had a line size of 16 bytes, which was just barely small enough to distinguish the secret-dependent branches in the table lookup code. The precomputed table was also reliably aligned with cache sets which saw little noise from interrupt handling code, reducing the difficulty of identifying table accesses. Different versions of Qualcomm's TrustZone code will have different offsets, potentially changing the rate at which the secret information leaks. However, the goal of our research is not to show a weaponized exploit against hardware-backed keystores, but rather to demonstrate the ability to compromise sensitive information using powerful microarchitectural attacks.

We further analyzed firmware version 7.1.0 (NDE63P [Goo16]) of Google's Pixel XL phone. We found that, while the same algorithm is used, the implementation does not have the same secret-dependent control flow. Mask initialization uses non-branching instructions, showing that not every device with this algorithm is vulnerable to our attack. It is infeasible for us to perform the full analysis for every combination of device and firmware, but according to Qualcomm, 35 chipsets are affected by this vulnerability [Qua19]. By working with Qualcomm on this issue and following their timeline to allow OEMs and vendors to patch, we believe that the risk of publicly disclosing this issue has been effectively mitigated.

7 Conclusion

7.1 Countermeasures

A number of countermeasures may be applied to prevent our attacks from succeeding. Since Cachegrab relies on the monitor mode interrupt handler transferring control from the victim TA to the attacker's code in the Non-secure world, the monitor mode could be rewritten to prevent malicious interrupts from suspending TA execution. However, it may be infeasible to efficiently classify interrupts as legitimate or malicious, and failing to promptly deliver an interrupt to the Non-secure OS may unacceptably degrade performance.

Another approach is to flush the shared microarchitectural structures when transferring between the Secure and Non-secure worlds. This would prevent our Prime+Probe attack, since the probing phase would always indicate that all of the attacker's entries were evicted regardless of victim behavior. In fact, this flushing behavior was experimentally observed for the BTB in the most recent firmware[4] for the Nexus 5X. This is presumably a response to the Spectre disclosures [KGG+18] and mitigates the ability of our attack to target keys via the BTB. Notably, the L1I and L1D caches are not flushed, and Cachegrab is still capable of performing our Prime+Probe attacks on these caches.

In addition to stymieing the ability to collect side-channel information from TrustZone execution, some countermeasures can specifically harden the Keymaster TA and other TrustZone software. Secret-dependent branches can be fully eliminated, replacing them with constant-time alternatives. This requires a high level of understanding which information is considered secret, but it is an effective method for blocking the success of cache attacks, and it can be distributed to affected devices in the form of a software update.

Application developers can also set more restrictive usage policies for hardware-backed keys. Our attack created 12000 signatures over 14 hours, and rate-limiting the usage of this key, requiring fingerprint authentication, or limiting the number of key uses per boot would have dramatically slowed the attack and reduced the risk of key exposure.

Long-term countermeasures to this type of microarchitectural attack can be provided by physically separating the trusted code from untrusted code, eliminating shared microarchitectural structures. Android 9 introduces support for *StrongBox Keymasters* [Goo18a] which store keys in dedicated hardware security modules (HSMs). Such an HSM is present in Google's Pixel 2 devices [Xin17], which were released in late 2017. Since calculations involving the keys does not use the microarchitectural structures of the main processor, microarchitectural attacks within the processor will fail.

7.2 Future Work

Although the microarchitectural attacks proposed in this work are powerful, there is still room for future research. It may be possible to infer more detailed information about victim execution by targeting other shared microarchitectural structures, such as the translation lookaside buffer [GRBG18] or last-level cache (LLC). Since the LLC is organized differently from the L1 caches, two distinct addresses which alias to the same set in the L1 caches may be distinguishable in the LLC. Additionally, it may be possible to adapt the techniques in [VBPS18] to TA execution to gain side-channel information at single-instruction granularity. Further efforts may also find ways to more efficiently implement our side-channel attacks, acquiring the same information but without the associated slowdown of victim code.

Our work only considers the applicability of passive cache attacks to ARM TrustZone, but a natural extension here is demonstrating Spectre-style attacks [KGG+18, KKSAG18]. Cachegrab currently manipulates the branch predictor while interleaving Secure and Non-secure code execution on the same core, so this platform may aid in the exploration of the feasibility of performing transient execution attacks against TrustZone.

[4]Version 8.1.0, OPM7.181205.001 [Goo18b]

Despite the many open research directions it proposes, this work still represents a significant advance in the capability of practically targeting ARM TrustZone with microarchitectural attacks. Our attacks have high temporal precision, high spatial precision, and low noise, and we successfully demonstrated that these attacks are capable of breaking a real-world cryptographic implementation. These powerful abilities show that ARM TrustZone is not necessarily any more secure than Intel SGX when it comes to microarchitectural attacks, and they substantiate decisions to harden sensitive code with side-channel countermeasures or to execute trusted code in entirely separate hardware. Although these mitigations come with engineering challenges of their own, our attacks against ARM TrustZone demonstrate that these countermeasures are imperative to defending a product and improving the security of mobile and embedded devices.

Acknowledgments

I would like to express my gratitude to Daniel Genkin for his helpful advice and guidance on drafts of this paper, and I would also like to thank Thomas Pornin, Jennifer Fernick, and Audrey Erpelding for providing further reviews of the document. Their feedback was invaluable in forming the final product.

[AFG$^+$14] Diego F Aranha, Pierre-Alain Fouque, Benoit Gérard, Jean-Gabriel Kammerer, Mehdi Tibouchi, and Jean-Christophe Zapalowicz. GLV/GLS decomposition, power analysis, and attacks on ECDSA signatures with single-bit nonce bias. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 262–281. Springer, 2014. 10

[AKS07a] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 312–320. ACM, 2007. 9, 16

[AKS07b] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference*, pages 225–242. Springer, 2007. 9, 16

[ARM15] ARM, 110 Fulbourn Road, Cambridge, England CB1 9NJ. *ARM Cortex-A Series: Programmer's Guide for ARMv8-A*, version 1.0 edition, March24 2015. http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf. 5, 7, 15

[ARM16] ARM, 110 Fulbourn Road, Cambridge, England CB1 9NJ. *ARM Cortex-A57 MPCore Processor*, r1p3 edition, February1 2016. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0488h/DDI0488H_cortex_a57_mpcore_trm.pdf. 16

[BFMRT16] Pierre Belgarric, Pierre-Alain Fouque, Gilles Macario-Rat, and Mehdi Tibouchi. Side-channel analysis of Weierstrass and Koblitz curve ECDSA on Android smartphones. In *Cryptographers' Track at the RSA Conference*, pages 236–252. Springer, 2016. 10

[BH09] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, pages 667–684, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. 10

[BT11] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In Vijay Atluri and Claudia Diaz, editors, *Computer Security – ESORICS 2011*, pages 355–371, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. 10, 20

[BV96] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 129–142, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. 9

[BvdPSY14] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh aah... just a little bit": A small amount of side channel can go a long way. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 75–92, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. 10

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016. 4

[CdRP14] Tim Cooijmans, Joeri de Ruiter, and Erik Poll. Analysis of secure key storage solutions on Android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 11–20. ACM, 2014. 5, 7

[CL16] Paul Crowley and Paul Lawrence. Pixel security: Better, faster, stronger. https://blog.google/products/android-enterprise/pixel-security-better-faster-stronger/, November 2016.

Accessed: 2019-04-21. 4

[Cor14] The MITRE Corporation. CVE-2014-3100. Available from MITRE, CVE-2014-3100, April 29 2014. Accessed: 2019-04-21. 5

[Cor15] Intel Corporation. Intel software guard extensions. Reference number 332680-001, https://software.intel.com/sites/default/files/332680-001.pdf, June 2015. 4

[CPVtG14] Tim Cooijmans, E Erik Poll, ER Eric Verheul, and T Ties Pull ter Gunne. Secure key storage and secure computation in Android. *Master's thesis, Radboud University Nijmegen*, 2014. 5

[DDME$^+$18] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 171–191, 2018. 4

[DMHMP13] Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, pages 435–452, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 10

[EPAG16] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016. 9

[ERAG$^+$18] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 693–707, New York, NY, USA, 2018. ACM. 9

[FID18] FIDO. Hardware-backed keystore authenticators (HKA) on Android 8.0 or later mobile devices. https://fidoalliance.org/white-paper-hardware-backed-keystore-authenticators-hka-on-android-8-0-or-later-mobile-devices/, June 2018. 7

[FWC16a] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking OpenSSL implementation of ECDSA with a few signatures. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1505–1515. ACM, 2016. 10

[FWC16b] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking OpenSSL Implementation of ECDSA with a Few Signatures. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, pages 1505–1515, 2016. 10

[GB17] Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 83–98, 2017. 9

[Goo16] Google. Firmware update 7.1.0 for Pixel XL. https://dl.google.com/dl/android/aosp/marlin-nde63p-factory-dcdaaa51.zip, October 2016. 21

[Goo17] Google. Firmware update 7.1.1 for Nexus 5X. https://dl.google.com/dl/android/aosp/bullhead-n4f26t-factory-8eed1d9f.zip, March 2017. 11

[Goo18a] Google. Android keystore system. https://developer.android.com/training/articles/keystore, 2018. Accessed: 2019-04-21. 7, 22

[Goo18b] Google. Firmware update 8.1.0 for Nexus 5X. https://dl.google.com/dl/android/aosp/bullhead-opm7.181205.001-factory-5f189d84.zip, December 2018. 22

[Goo18c] Google. Verifying hardware-backed key pairs with key attestation. https://developer.android.com/training/articles/security-key-attestation, 2018. Accessed: 2019-04-21. 8

[GPP$^+$16] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1626–1638, New York, NY, USA, 2016. ACM. 10

[GRBG18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, 2018. 22

[GYCH18] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–37, 2018. 8, 9, 16

[HGS01] N. A. Howgrave-Graham and N. P. Smart. Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography*, 23(3):283–290, Aug 2001. 9, 10, 12

[HR07] Martin Hlaváč and Tomáš Rosa. Extended hidden number problem and its cryptanalytic applications. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography*, pages 114–133, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. 10

[KGG$^+$18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018. 22

[KKSAG18] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018. 22

[KMVOV96] Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996. 10

[KSD13] Cameron F. Kerry, Acting Secretary, and Charles Romine Director. FIPS PUB 186-4 Federal Information Processing Standards publication Digital Signature Standard (DSS), 2013. 9

[lag15a] laginimaineb. Exploring Qualcomm's TrustZone implementation. http://bits-please.blogspot.com/2015/08/exploring-qualcomms-trustzone.html, 2015. Accessed: 2019-04-21. 4

[lag15b] laginimaineb. Full TrustZone exploit for MSM8974. http://bits-please.blogspot.com/2015/08/full-trustzone-exploit-for-msm8974.html, 2015. Accessed: 2019-04-21. 4

[lag16a] laginimaineb. Exploring Qualcomm's Secure Execution Environment. http://bits-please.blogspot.com/2016/04/exploring-qualcomms-secure-execution.html, 2016. Accessed: 2019-04-21. 11

[lag16b] laginimaineb. Extracting Qualcomm's KeyMaster keys - breaking Android full disk encryption. https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html, 2016. Accessed: 2019-04-21. 5

[LCL13] Mingjie Liu, Jiazhe Chen, and Hexin Li. Partially known nonces and fault injection attacks on SM2 signature algorithm. In *International Conference on Information Security and Cryptology*, pages 343–358. Springer, 2013. 10

[LGS+16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016. 4, 5

[LSG+17] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, 2017. 4, 16

[Mar18] Kobus Marneweck. Enhancing embedded device security with new TrustZone-enabled microcontrollers. https://community.arm.com/developer/ip-products/processors/trustzone-for-armv8-m/b/blog/posts/enhancing-embedded-device-security-with-new-trustzone-enabled-microcontrollers, September 2018. Accessed: 2019-04-21. 4

[MES18] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations in SGX. In *Cryptographers' Track at the RSA Conference*, pages 21–44. Springer, 2018. 4

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017. 4, 9, 15, 17

[NS01] Phong Q. Nguyen and Jacques Stern. The two faces of lattices in cryptology. In Joseph H. Silverman, editor, *Cryptography and Lattices*, pages 146–180, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. 10

[NS02] Nguyen and Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology*, 15(3):151–176, Jun 2002. 10

[OST06] 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. 8

[Per05] Colin Percival. Cache missing for fun and profit, 2005. 8

[Qua15] Qualcomm. Qualcomm snapdragon 808 processor. https://www.qualcomm.com/media/documents/files/snapdragon-808-processor-product-brief.pdf, 2015. Accessed: 2019-04-21. 17

[Qua19] Qualcomm. April 2019 Qualcomm Technologies, Inc. security bulletin, April 1 2019. Accessed: 2019-04-21. 21

[Ros14] Dan Rosenberg. Reflections on trusting TrustZone. BlackHat USA: https://www.blackhat.com/docs/us-14/materials/us-14-Rosenberg-Reflections-on-Trusting-TrustZone.pdf, 2014. 4

[Rya19] Keegan Ryan. Return of the hidden number problem. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 146–168, 2019. 9, 10

[Sam16] Samsung. Device-side security: Samsung Pay, TrustZone, and the TEE. https://developer.samsung.com/tech-insights/pay/device-side-security, 2016. Accessed: 2019-04-21. 4

[ST16] Mohamed Sabt and Jacques Traoré. Breaking into the keystore: A practical forgery attack against Android keystore. In *European Symposium on Research in Computer Security*, pages 531–548. Springer, 2016. 5

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of security-oblivious energy management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, 2017. 5

[TTA18] Akira Takahashi, Mehdi Tibouchi, and Masayuki Abe. New Bleichenbacher records: Practical fault attacks on qdsa signatures. *IACR Cryptology ePrint Archive*, 2018:396, 2018. 10

[UM09] Vladimir Uzelac and Aleksandar Milenkovic. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 207–217. IEEE, 2009. 17

[VBPS18] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–195. ACM, 2018. 4, 22

[vdPSY15] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In Kaisa Nyberg, editor, *Topics in Cryptology – CT-RSA 2015*, pages 3–21, Cham, 2015. Springer International Publishing. 9, 10

[Wil17] Shawn Willden. Keystore key attestation. https://android-developers.googleblog.com/2017/09/keystore-key-attestation.html, 2017. Accessed: 2019-04-21. 7

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015. 4

[Xin17] Xiaowen Xin. Lock it up! New hardware protections for your lock screen with the Google Pixel 2. https://security.googleblog.com/2017/11/lock-it-up-new-hardware-protections-for.html, November 2017. Accessed: 2019-04-21. 22

[Yar16] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit. *Retrieved from School of Computer Science Adelaide: http://cs.adelaide.edu.au/~yval/Mastik*, 2016. 16

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014. 5, 9

[ZSS$^+$16] 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. 4, 5, 15

Appendices

A QSEE Table Lookup Code

The QSEE table lookup procedure for precomputed elliptic curve points was recovered from the ARM binary to roughly equivalent C code. This is shown in Figure 5. The complexity of this implementation, compared to naively copying a value from a table, indicates that the function was specifically designed to protect against various side channels, including timing attacks, memory cache attacks, and electromagnetic attacks which leak the Hamming weight of registers.

```c
typedef char bignum[0x48];
typedef struct {
  bignum x;
  bignum y;
  bignum z; // Used for projective representation
  uint32_t flags;
} ECPoint;

void lookup (ECPoint* result, ECPoint TABLE[8], int index, int randval) {
  uint32_t mask0 = (index == (randval + 0) % 8) ? 0xffffffff : 0;
  uint32_t mask1 = (index == (randval + 1) % 8) ? 0xffffffff : 0;
  uint32_t mask2 = (index == (randval + 2) % 8) ? 0xffffffff : 0;
  uint32_t mask3 = (index == (randval + 3) % 8) ? 0xffffffff : 0;
  uint32_t mask4 = (index == (randval + 4) % 8) ? 0xffffffff : 0;
  uint32_t mask5 = (index == (randval + 5) % 8) ? 0xffffffff : 0;
  uint32_t mask6 = (index == (randval + 6) % 8) ? 0xffffffff : 0;
  uint32_t mask7 = (index == (randval + 7) % 8) ? 0xffffffff : 0;

  uint32_t* dest = (uint32_t*)result;
  uint32_t* src0 = (uint32_t*)&TABLE[(randval + 0) % 8];
  uint32_t* src1 = (uint32_t*)&TABLE[(randval + 1) % 8];
  uint32_t* src2 = (uint32_t*)&TABLE[(randval + 2) % 8];
  uint32_t* src3 = (uint32_t*)&TABLE[(randval + 3) % 8];
  uint32_t* src4 = (uint32_t*)&TABLE[(randval + 4) % 8];
  uint32_t* src5 = (uint32_t*)&TABLE[(randval + 5) % 8];
  uint32_t* src6 = (uint32_t*)&TABLE[(randval + 6) % 8];
  uint32_t* src7 = (uint32_t*)&TABLE[(randval + 7) % 8];

  // Copy X and Y coordinates 4 bytes at a time
  for (int i = 0; i < 2 * sizeof(bignum) / sizeof(uint32_t); i++) {
    uint32_t newval, oldval = dest[i];
    newval  = (src0[i] ^ oldval) & mask0;
    newval |= (src1[i] ^ oldval) & mask1;
    newval |= (src2[i] ^ oldval) & mask2;
    newval |= (src3[i] ^ oldval) & mask3;
    newval |= (src4[i] ^ oldval) & mask4;
    newval |= (src5[i] ^ oldval) & mask5;
    newval |= (src6[i] ^ oldval) & mask6;
    newval |= (src7[i] ^ oldval) & mask7;
    dest[i] = newval ^ oldval;
  }
  result->flags = AFFINE_POINT;
}
```

Figure 5: QSEE table lookup for precomputed points.