iTimed: Cache Attacks on the Apple A10 Fusion SoC

Gregor Haas, Seetal Potluri, Aydin Aysu
North Carolina State University

Abstract

This paper proposes the first cache timing side-channel attacks on one of Apple's mobile devices. Utilizing a recent, permanent exploit named *checkm8*, we reverse-engineered Apple's BootROM and created a powerful toolkit for running arbitrary hardware security experiments on Apple's in-house designed ARM systems-on-a-chip (SoC). We integrate two additional open-source tools to enhance our own toolkit, further increasing its capability for hardware security research. Using this toolkit, which is a core contribution of our work, we then implement both time-driven and access-driven cache timing attacks as proof-of-concept illustrators. In both cases, we propose statistical innovations which further the state-of-the-art in cache timing attacks. We find that our access-driven attack, at best, can reduce the security of OpenSSL AES-128 to merely 25 bits, while our time-driven attack (with a much weaker adversary) can reduce it to 48 bits. We also quantify that access-driven attacks on the A10 which do *not* use our statistical improvements are unable to deduce the key, and that our statistical technique reduces the traces needed by the typical time-driven attacks by 21.62 million.

1 Introduction

With rare exceptions [10], side channel attacks (SCAs) on Apple's line of mobile devices have been largely unexplored in the literature for two core reasons. First, Apple's "It Just Works" design philosophy is based on tight vertical integration and hiding their devices' underlying complexities from both users and application programmers. Apple either designs or integrates all components in the system stack, and does not release detailed documentation about the system. Therefore, security research on iPhones typically starts with reverse engineering the target subsystem or application. Researchers rely on high-level overviews of security components [19], partial source code releases [16], or even illegally leaked source code [7] to aid their reverse engineering efforts.

Second, while reverse engineering and other forms of static analysis are partially possible on iPhones, dynamic analysis tends to be even more difficult. Dynamic analysis involves observing applications while they are running, typically under a debugger or another control tool. Apple's proprietary development tool (Xcode) does ship with a debugger, but it is not possible to debug arbitrary applications without first compromising the operating system and removing certain security restrictions [4]. Even in these cases, the kernel can often not be debugged—rare exceptions include the Apple A11 SoC, which contains proprietary debug registers that were accidentally left enabled in production devices [2], and development-fused devices which cannot be obtained legally [8]. Additionally, Apple ensures that applications cannot arbitrarily interact with other applications or the operating system by strictly enforcing the allowed inter-process communication (IPC) interfaces. As shown in the literature [6], even determining which interfaces exist is a challenging research problem.

In the context of hardware security research on iPhones, useful resources such as documentation or development tools are even rarer compared to software security research. For one, Apple does not release *any* detailed documentation for their in-house designed hardware modules. Some information can be found in Apple's patents for a dynamic voltage frequency modulation (DVFM) module [18], secure co-processor [17], etc., but even such references only provide high-level views of system components rather than the technical implementation details. Even with detailed knowledge of the hardware, interfaces to useful modules are often not exposed to application programmers. For example, without an attacker-controllable interface to the DVFM module, fault attacks such as CLKSCREW [30] or VoltJockey [26] are not possible. Likewise, since the operating system's scheduling interfaces are not exposed to programmers, it is arguably harder to apply timing-based SCAs which depend on thread-shared and core-shared state. In fact, to date, there is no successful demonstration of timing SCAs on Apple SoCs.

The iPhone security research community exhibits closed-source tendencies that are similar to, and partially caused by, Apple's closed-source design philosophy. Powerful exploit chains, especially ones which can modify the kernel, are often used to bootstrap further security research. To that extent, the most powerful class of iPhone exploits is based on vulnerabilities in the BootROM, a region of read-only memory (ROM) that contains Apple's first-stage bootloader [19]. Security researchers can use BootROM exploits to create an arbitrary kernel modification primitive that is *permanent*, and cannot be patched by Apple short of recalling all vulnerable devices. Public BootROM exploits thus are very rare. Currently, only eight exploits are known across all iPhone models [20].

Contributing to the ongoing effort to build quality, open-source iPhone research tools [2] [6], we present *iTimed*: a novel research toolkit built on *checkm8*, a new publicly disclosed BootROM vulnerability. Most modern iPhones, from

the iPhone 5 to the iPhone X, are permanently vulnerable to checkm8 which creates a unique opportunity for hardware security research on those devices. Our toolkit exposes many useful interfaces for general hardware security research, many of which tie directly into Apple's hardware drivers. We also enhance our toolkit by integrating two open-source tools developed by other groups that were not originally intended for hardware security. Finally, we present two proof-of-concept cache timing SCAs—one access-driven as in [24], and the other time-driven as in [3]. For both of these attacks, we propose novel statistical extensions that let us break standard implementations of t-table AES-128. To the best of our knowledge, these are the first such SCAs on an Apple SoC.

1.1 Contributions

The central contributions of this paper are:

- We reverse-engineered the iPhone 7 BootROM and directly show the root-cause vulnerability of checkm8.
- Incorporating useful functions and primitives obtained from our reverse-engineering, we implement an extensible BootROM toolkit to base extensive hardware security experiments on. Source code is currently available on request to ghaas@ncsu.edu .
- We integrate two additional open-source tools into our toolkit, creating an end-to-end solution for full-stack (hardware, drivers, operating system) hardware security experiments on iPhone hardware.
- Using this toolkit, we develop and implement an access-driven timing SCA. We show that a straightforward implementation of `PRIME+PROBE` fails due to Apple's cache replacement policies and prefetcher.
- To successfully implement this attack, we propose novel attack and statistical techniques that specifically account for the A10's microarchitecture. We find that our modified `PRIME+PROBE` attack can lower the AES-128 security level to 25 bits, while the standard `PRIME+PROBE` attack fails completely. To the best of our knowledge, this is the first such access-driven SCA on an Apple SoC.
- We then weaken the adversarial model and develop a time-driven, profiled cache timing SCA on the same SoC. We find that, even when an adversary does not manipulate the architectural state, the A10 SoC still exhibits vulnerable timing behaviors at both the BootROM stage of boot, as well as with a full operating system. To the best of our knowledge, this is the first such time-driven SCA on an Apple SoC.
- To improve the time-driven attack, we propose *multi-profile* attacks which utilize timing information from multiple known AES keys to deduce the value of an unknown AES key. We find that multi-profile attacks require up to 21.62 *million* less traces than single-profile attacks to reduce the AES key search space to 2^{48}.

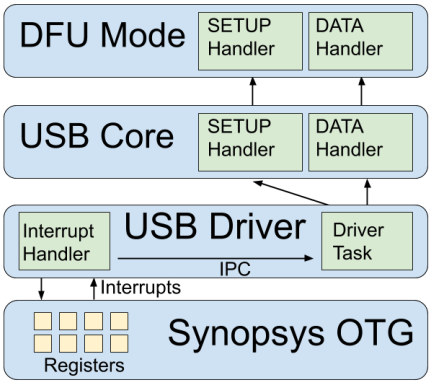


Figure 1: Simplified diagram of the A10 Fusion SoC's BootROM USB stack based on our reverse-engineering. The `SETUP` and `DATA` handlers are separated, which is a primary cause for the checkm8 vulnerability. The diagram also shows communication between modules—the USB driver communicates with the Synopsys OTG controller via interrupts, the interrupt handler communicates with the driver task via IPC, and all other communication consists of function calls.

2 Background

In this section, we introduce relevant background information to contextualize our work. We begin with a discussion on the iPhone system integrity infrastructure and show how the checkm8 vulnerability naturally leads to an arbitrary kernel modification primitive. We then give a brief architectural overview of the Apple A10 Fusion SoC, which is the target of our attack. Finally, we review recent and relevant advances in cache timing SCAs to further motivate our attacks.

2.1 iPhone System Integrity

Apple's design philosophy is that all software on their devices should "just work" exactly as distributed, such that any non-Apple modifications are neither necessary nor allowed. To that extent, system integrity is a core guarantee provided by Apple's security infrastructure. For iPhones specifically, this guarantee is rooted in the secure boot chain.

Apple's boot chain ensures that only the software signed by Apple can execute on the device. The chain begins in the BootROM, a small area of ROM that contains Apple's first-stage bootloader and public key. The BootROM loads the second-stage bootloader (named *iBoot*) from nonvolatile storage and verifies that its signature is valid. If either the load or the signature check fails, the BootROM instead enters a direct firmware upgrade (DFU) mode. In this mode, a signed firmware image can be sent to the device over USB and booted instead of iBoot—typically, such an image would restore the device to a working state by reinstalling both iBoot and the kernel. Once entered, the device will loop in DFU mode until it is powered off or receives a valid image.

Once iBoot is successfully loaded and verified, the BootROM will transfer control to a special *boot trampoline*. The trampoline is an intermediate stage of the boot process,

designed to reset the device to a known state by clearing all memory from the previous boot stage, and disabling hardware devices where applicable. The trampoline ensures that a vulnerability in one boot stage will not leak information about the previous boot stages. After exiting, the trampoline transfers control to iBoot which loads and verifies the kernel using the same general process as the BootROM. Thus, the integrity of the operating system is closely tied to the immutability and correctness guarantees of the BootROM.

2.2 checkm8

Understanding the DFU mode's USB implementation is essential for understanding the checkm8 exploit. Therefore, we thoroughly reverse-engineered[1] this interface and present our findings in Figure 1, as well as Listings 1, 2, and 3 in Appendix A.1. If the DFU mode is entered, the USB core subsystem is initialized first. This subsystem serves as a bridge between the high-level USB interfaces (such as the DFU mode implementation) and the low-level driver for the Synopsys OTG USB controller. Then, the USB DFU interface is registered with the USB core, providing it with a standard set of callback functions for various USB events. The DFU interface allocates an IO buffer to hold chunks of the downloaded firmware image, after which the USB data transfer stage begins. When a new packet is received, the USB driver task checks if it is a `SETUP` packet, containing metadata for a new transaction, or a `DATA` packet, containing the data from a previously specified transaction. In either case, the packet is then forwarded to the USB core and the interfaces for further processing

On September 27th, 2019, independent iOS security researcher axi0mX released a proof-of-concept tool named *checkm8*[2], which exploits a use-after-free vulnerability in the USB core implementation. Our reverse-engineering reveals the root-cause of this vulnerability. When a setup packet is received, the DFU interface sets the USB core's data phase buffer pointer (`dp_buf`) to the previously allocated `io_buffer` (Listing 2, line 15). As the transaction's data packets are received, they are copied to the `dp_buf` (Listing 1, line 16). Once all of the transaction's data has been received, the USB core calls the DFU interface's data handler and resets its static internal state (Listing 1, line 27). The DFU interface also copies the chunk's data to its final location `img_buf` (Listing 3, line 5). axi0mX found that this implementation has a vulnerability—if a setup packet is received, but USB is reset before any data packets are processed, the `dp_buf` pointer is never set to `NULL`. Since no valid image was received, the BootROM will reinitialize the DFU mode and allocate a new IO buffer. However, before any setup packets are received, the `dp_buf` pointer will still point to the last DFU iteration's IO buffer—a classic use-after-free vulnerability.

Using this vulnerability, it is possible to craft a *buffer overflow attack* that overwrites a function pointer on the heap (for details, see [22]). Then, using the DFU mode's normal functionality, an arbitrary shellcode can be uploaded and executed with full permissions in the BootROM. This breaks the integrity-related security guarantees provided by the BootROM—for example, checkm8 can be used to patch iBoot after it is loaded and verified, but before it is booted. Then, in iBoot, checkm8 can similarly patch the XNU kernel or transfer control to another operating system entirely. These claims are not theoretical—both our own toolkit and various other open-source tools, which we discuss in Section 4, implement this functionality. By default, axi0mX's shellcode simply adds stubs for reading, writing, and executing arbitrary addresses (triggered by specially formatted USB requests). However, we will show in Section 4 that these read/write/execute primitives can be combined and extended in powerful ways to build a more advanced research platform.

2.3 Apple A10 Fusion SoC

The checkm8 exploit works on most iPhone models, from the iPhone 5 to the iPhone X. Therefore our toolkit can be trivially extended to all these models. However, we specifically target an iPhone 7 (model *A*1778) containing an A10 SoC because, when this project began in 2019, the iPhone 7 was the most common Apple mobile device in the consumer market[3]—it is still commonly used with over 80 million sold copies. The iPhone 7's SoC contains four ARMv8-A cores arranged in a standard big.LITTLE[4] configuration: two power efficient cores, and two high performance cores [15]. However, only one of these core types can be active at a time which means the SoC appears as a dual-core processor. This SoC has a four-level memory hierarchy - a 64KB L1 data cache, a 64KB L1 instruction cache, a unified 3MB L2 cache, a unified 4MB L3 cache, and (depending on the boot stage) either 2MB of SRAM or 2GB of DRAM.

The SoC integrates a variety of devices that enable advanced functionality, but we will focus only on the ones relevant to our attacks. The Synopsys OTG USB controller is integrated directly into the SoC, providing the iPhone with both USB host and device capabilities. The SoC also includes a 24MHz hardware timer, accessible via the standard ARMv8 register `CNTPCT_EL0`. We also use the ARMv8 registers `CLIDR_EL1`, `CSSELR_EL1`, and `CCSIDR_EL1` to help reverse-engineer the SOC's cache structure and implementation. We note that we can only read these privileged registers because of checkm8.

2.4 Target Algorithm for SCAs

For this work, we use AES-128 [5] to encrypt a 16-byte input M using a 16-byte key K. A short, formalized description of this algorithm is given in Appendix B.1.

[1] Using standard tools, such as Ghidra (https://ghidra-sre.org/)
[2] https://github.com/axi0mX/ipwndfu
[3] https://deviceatlas.com/blog/most-popular-iphones
[4] https://www.arm.com/why-arm/technologies/big-little

2.5 Timing Attacks on AES

Cache timing attacks can be broadly classified into three categories: time-driven attacks, trace-driven attacks, and access-driven attacks. Time-driven and trace-driven attacks were first theoretically discussed in [25], experimentally demonstrated later by Bernstein in [3] and by Acıiçmez et al. in [1], respectively, and then improved with better statistical analysis [27, 28]. These attacks mainly define their threat model adversary as a passive observer—for time-driven attacks, the adversary can only observe the overall time for a full encryption operation. For trace-driven attacks, the adversary can observe each cache hit or miss within that encryption operation. Correspondingly, these types of attacks are more generalizable between architectures and cryptographic ciphers, but often require many traces to extract secret information.

Access-driven attacks, by contrast, assume that the adversary can actively manipulate the state of the cache. These were first demonstrated with the `PRIME+PROBE` technique [24], but have become much more popular recently. Discoveries of techniques such as `FLUSH+RELOAD` [34], `EVICT+RELOAD` [14], and `FLUSH+FLUSH` [13], among others, have led to a wealth of powerful, high-resolution attacks that require significantly less traces than time and trace-driven attacks.

Timing attacks typically target x86 desktop and server computers, but have also been demonstrated on ARM mobile devices running Android. Bernstein's attack was first shown on three Android devices [29] and subsequently extended to others [28]. More recently, Lipp et al., at USENIX 2016, ported access-driven attacks to ARM Android devices [23]. These papers inspire us to pursue this work as to date, and to the best of our knowledge, no cache attacks have been demonstrated on the Apple-designed ARM SoCs found in iPhones.

In addition to attacks, many timing SCA *defenses* have also been proposed [9]. Details of these lie out of the scope of this work since none have been confirmed on Apple's devices, except the ones which we identify in this work.

3 Threat Model

Both attacks assume that the attacker has the ability to synchronously trigger AES encryptions. We also assume known-plaintext attacks. Furthermore, our attacks do not need precise timing of intermediate states: while the exact format of the timing measurements varies between our attacks, we always treat the underlying AES implementation as a black box.

Our access-driven attack (Section 5), follows the standard synchronous threat model for access-driven timing SCAs [24]. For this attack, we require that the attacker is co-located on the same core as the victim since this attack, like other access-driven attacks, hinges on precise manipulations and measurements of the victim's cache state. Furthermore, we require knowledge of the virtual address of the t-tables—however, this is a standard assumption in the literature [24] and is not difficult to learn in practice.

Figure 2: Arduino with MAX3421E USB host shield connected to our test iPhone 7. The Arduino is a USB proxy between the host and device and generates correct partial requests when necessary. This setup successfully addresses checkm8's reliability issues. This iPhone is running *pongoOS*, discussed in Section 4.2.

Our time-driven attack (Section 6), again, follows the standard threat model of profiled time-driven timing SCAs [3]. We assume that the attacker can trigger AES encryptions on the target device with at least one known key, as well as the capability to trigger AES encryptions using the key under attack. These encryptions can be either triggered by a process on the same device or through remote interaction [3]. Time-driven attacks do not assume that the adversary has to be co-located on the same core to manipulate the micro-architectural state, making them *stronger* than access-driven attacks.

4 Tooling

While checkm8 is a powerful exploit technique, the proof-of-concept released by axi0mX is not suitable for extensive hardware security research. We present our novel extensions to checkm8 toolkit and describe how we improve the reliability, extensibility, and supported execution models of the base checkm8. We also discuss two novel tools which were not originally intended for hardware security research, and show how we incorporate these into our research platform.

4.1 Expanding the checkm8 Toolkit

We created our own open-source toolkit based on checkm8 that addresses a core set of usability issues. When discussing our toolkit, we refer to the computer that runs the toolkit as the *host* and the target iPhone as the *device*.

4.1.1 Reliability

We first improved the tool's success rate, both in terms of successfully exploiting checkm8 and system stability. As

described in Section 2.2, successfully exploiting the use-after-free vulnerability depends on partial USB transactions containing only a `SETUP` packet. However, no standard USB host controller drivers support generating such transactions. axi0mX's solution involves asynchronously canceling a normal USB request, which probabilistically results in a correct partial request. Several of these requests must be made (correctly) for the exploit to succeed, so the exploit's success rate becomes probabilistic as well. Furthermore, partial requests can be correct *enough* for the exploit to succeed but will silently corrupt memory in the background, crashing the device at some point in the future. This is a major challenge for hardware security research, which often depends on long profiling phases or precise hardware manipulation.

Figure 2 shows how we solved the reliability problem. Modifying a standard USB host controller driver (such as XHCI) to support partial requests would be challenging—each layer of the host's USB stack, from the driver to the user-space interface, would need to be changed. Therefore, we have opted to implement the required functionality on the Arduino platform. Arduinos are a family of low-cost 8-bit microcontrollers. They can be extended with functionality-specific "shields"—the one shown here has a USB host shield[5] with a MAX3421E USB host controller. The Arduino driver for this chip is open source, so we modified it to support checkm8 partial requests. Then, the Arduino acts as a proxy between the host and the device, forwarding all USB communication and generating correct partial requests when necessary.

4.1.2 Extensibility

Our checkm8 toolkit also addresses the issue of extensibility. Ideally, we would like to easily write, load, and execute complex programs that implement extensive hardware security experiments. For many such experiments, it is convenient to interact directly with the hardware; however, writing drivers for Apple's proprietary modules could be extraordinarily difficult. Luckily, the BootROM includes fully functional, if somewhat minimal, interfaces to many of these modules. For our toolkit, we reverse-engineered much of the BootROM and exposed a core set of useful functions which can be used by experimental programs to interact with the BootROM and the iPhone's hardware. Listing 4 is part of the source code for our time-driven attack's profiler, which contains several calls to BootROM functions. For example, line 22 reads 16 random bytes from the BootROM's pseudorandom number generator—one of the BootROM's many cryptographic interfaces. In line 26, we retrieve a high resolution timestamp from the SoC's 24MHz cycle counter, which we can also use to program interrupts some number of cycles in the future. Finally, line 34 interacts with the BootROM's simple tasking system, which implements both process-like and semaphore-like constructs. All these extensions are crucial to run successful side-channel experiments.

[5] https://store.arduino.cc/usa/arduino-usb-host-shield

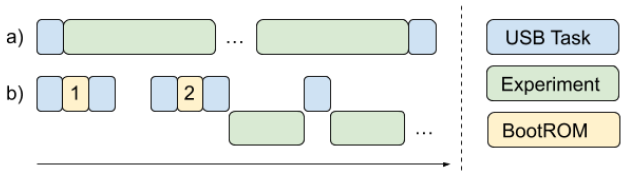


Figure 3: Flow of synchronous (a) and asynchronous (b) execution models over time. Blue shapes are the USB task, green shapes are the experimental program, and yellow shapes are calls to BootROM task initialization functions—`task_new` (1) and `task_run` (2) specifically.

Our tool also includes a full build system for these experimental programs. The BootROM does not support common executable formats, such as ELF, so all programs must consist of raw assembly instructions when they are installed on the device. We include cross compiler support, so that programs can be written in C rather than assembly. Our build system then automatically strips the compiled binary and places the program's entry point at the beginning of the file. All BootROM functions are called using absolute jumps rather than relative jumps, so programs can be installed anywhere in the memory space. The CPU's memory management unit (MMU) is still inactive at the BootROM stage of boot, so there is no address space protection. Since we do not want to overwrite any existing heap, stack, or global variables when installing programs, we use the BootROM's dynamic memory allocator to get a safe address. Finally, we write the program to the safe address and store its pointer for subsequent executions.

4.1.3 Execution Models

To complete our programming model and improvements to checkm8, we use the BootROM's tasking system to support two different execution models as shown in Figure 3. As mentioned in Section 2.2, axi0mX's checkm8 implements arbitrary execution by adding a stub to the USB controller which can be triggered with a specially formatted USB request. Therefore, any program executed with this technique will run synchronously *within* the USB controller task as shown in Figure 3a. For certain hardware security experiments, this execution mode might reduce the quality of the gathered data. For example, our time-driven attack (Section 6) depends heavily on maintaining the state of the L1 cache between iterations. If our profiling program were executed synchronously, the L1 cache would be polluted by the USB driver stack because a USB request must be processed between each iteration.

We instead execute such programs *asynchronously* using the reverse engineered BootROM tasking functions and synchronization primitives. The semantics of this execution model are similar to POSIX `pthreads`—a new task is created and scheduled as shown in Figure 3b, and receives its initial arguments through a pointer (Listing 4, line 3). The new task starts executing after the USB task finishes the `task_run`

USB execution request, and can explicitly yield control back to USB in case IO is required. The BootROM also includes a semaphore-like synchronization primitive which allows cooperation between tasks—this is useful for hardware security experiments that depend on core-shared state, particularly relevant to timing-based side channels as defined in [9].

4.2 checkra1n

We employ two external groups' toolkits that nicely complement ours and, used together, create a much richer hardware security research environment for iPhones. The first of these toolkits, *checkra1n* [31], is maintained by axi0mX, Luca Todesco, and other iPhone security researchers, aiming to create a user-land jailbreak via checkm8 as an exploit primitive. Jailbreaks allow ordinary users to install non-approved applications or system tweaks on their devices, and may be repurposed as a research tool. The checkra1n toolkit was first released on November 10th, 2019, and was partially open-sourced on March 1st, 2020 as *pongoOS*[6]. pongoOS is a simple, task based operating system used by checkra1n to patch the XNU kernel. It runs in the boot trampoline (discussed in Section 2.1) after iBoot, so it must first configure the hardware to support a proper execution environment.

Todesco reverse-engineered and reimplemented (without Apple's proprietary code) a large number of drivers for iPhone hardware. These drivers are tied together with a simple command line interface to control the system. pongoOS is modular and provides a straightforward SDK to compile custom modules against, as well as support for dynamically loading these modules at runtime. While pongoOS does include some useful features (such as access to the device's full 2GB of DRAM), we primarily use pongoOS to bootstrap the second open-source tool used in our hardware security experiments.

4.3 Project Sandcastle

Corellium, LLC, is the creator of a line of *virtualized* iPhones, functionally identical to real, physical devices. These are sold primarily to security researchers who do not want to risk irreversibly damaging real devices while searching for exploits. Corellium open-sourced the first version of Project Sandcastle on March 4th, 2020. Project Sandcastle consists of a set of supporting pongoOS modules and tools, a patched Linux kernel capable of booting on an iPhone 7, and a buildroot project that automatically compiles a bootable image with this kernel. This image includes, among other things, a full `glibc` and compiler toolchain, network drivers, support for both CPUs on A10, and many more features. pongoOS includes functionality for receiving and booting these images. This environment then makes it possible to fully explore advanced SCAs on iPhones, including industry-standard cryptographic implementations such as OpenSSL (see Section 7.3).

[6] https://github.com/checkra1n/pongoOS

5 Access-Driven Attacks

We now present an access-driven SCA on the OpenSSL t-table AES-128 implementation. This attack is motivated by the synchronous `PRIME+PROBE` attack from the seminal work of Osvik et al. in [24], but makes several key modifications which address the A10's specific microarchitecture. We emphasize that we pursue the `PRIME+PROBE` attack as it forms a canonical example—our statistical method is not only limited to this particular attack style, and extends easily to the ever-evolving iterations of access-driven attacks. For example, we have observed vulnerable cache flush timings which would enable attacks such as `FLUSH+RELOAD` [34] and `FLUSH+FLUSH` [13].

5.1 Notation

We closely follow the notation of `PRIME+PROBE` [24]. Access-driven attacks must account for cache configuration, so we model caches as tuples (S,W,B), which represent the number of sets, associativity, and block size (in bytes) of the cache respectively. To model the t-tables, for simplicity, we assume that the tables are contiguous in memory and that the start address is known.

The t-tables map into the cache based on two further parameters (s,o) which denote the size of an individual table entry (typically 4 bytes) and the offset of the first table within the cache (i.e., the memory address of $T_0[0] \bmod SB$). For indices $y \in [0,256)$ within a given t-table L, we can define the cache set of y in L:

$$C(L,y) = \lfloor \frac{o + s(256L + y - (o \bmod B))}{B} \rfloor + 1 \qquad (1)$$

Encrypting a message M with a key K will cause memory accesses to certain t-table entries. We define an oracle $Q_K(M,L,y)$, which equals 1 if encrypting M with K will access index y in t-table L. Thus, by repeatedly querying Q with known plaintexts, tables, and indices, we can learn some information about the unknown key K. Osvik et al. note that access to a perfect oracle Q is unrealistic and instead base their attack on an unreliable oracle $\mathcal{M}(M,L,y)$. Specifically, $\mathcal{M} \approx Q$ such that, for many (K,M,L,y), the expectation of $\mathcal{M}$ is higher when $Q_k(M,L,y) = 1$ than when $Q_k(M,L,y) = 0$:

$$E[\mathcal{M}_K \mid Q_K = 1] > E[\mathcal{M}_K \mid Q_K = 0], \forall (K,L,y) \qquad (2)$$

5.2 Standard `PRIME+PROBE` Fails

For our attack, we use `PRIME+PROBE` measurements to query the aforementioned oracle. For this strategy, we must allocate a probe array A of size $S \times W \times B$ bytes such that the start of the array is congruent to the start of the cache—that is, the address of $A[0] \bmod SB = 0$. Then, to obtain a `PRIME+PROBE` measurement for a message M, we:

1. *Prime:* read from every memory block in A. This step has the dual purpose of evicting the t-tables from memory, while also filling the cache with the attacker's data.

2. Encrypt M with the unknown key K.

3. *Probe:* if $Q_K(M,L,y) = 0$, we would expect that all of the attacker's data is still present in cache set $x = C(L,y)$. By contrast, if $Q_K(M,L,y) = 1$, one of the ways in cache set x would have been evicted and replaced with the corresponding t-table block. We can thus extract a measurement of $\mathcal{M}_K(M,L,y)$ by reading the W array entries

$$A[Bx], A[Bx+BS], \ldots, A[Bx+(W-1)BS]$$

and saving the total time taken for these accesses as T_x^M. A full trace consists of the set $\left\{T_x^M \mid x \in S\right\}$.

5.2.1 `PRIME+PROBE` Challenges

The general attack described above serves as a good theoretical base, but adjustments must often be made to account for specific processor microarchitecture. In the case of the A10 SoC, we must make two key modifications for the attack to succeed. The first has been well studied in the literature. `PRIME+PROBE` attacks on ARM processors were first shown in ARMageddon [23]. In this work, the authors note that ARM processors often use pseudorandom cache replacement policies (rather than deterministic least-recently-used-based policies) and, as such, the *prime* step must be modified. To successfully evict the t-tables and fill the target cache with the attacker's probe array, Lipp et. al. use the work of [12] to automatically search for fast eviction strategies. We employ a similar, yet simplified, approach in our attack—we simply access the W array entries in each set enough times that we have a good probability of filling the cache with our data.

The second modification has not been extensively researched from the attacker's perspective in the literature, with rare exceptions [33] that lack generality. We refer to Figure 4 to motivate this extension. The figure shows an example trace obtained using our attack's final `PRIME+PROBE` technique. For traces such as these, we define four measurement categories:

1. *True positives:* Cache sets which *are* accessed during the encryption, and are measured as if they *were* accessed. These are the most common and cluster near $y = 0$.
2. *True negatives:* Cache sets which *are not* accessed, and are measured as if they *were not* accessed. Visually, these are outliers of various magnitudes.
3. *False positives:* Cache sets which *are not* accessed, yet are measured as if they *were* accessed. These are caused by prefetching of the t-tables and cluster near $y = 0$.
4. *False negatives:* Cache sets which *are* accessed, yet are measured as if they *were not* accessed. These would be caused by prefetching of the probe array, but are mostly eliminated by our attack technique.

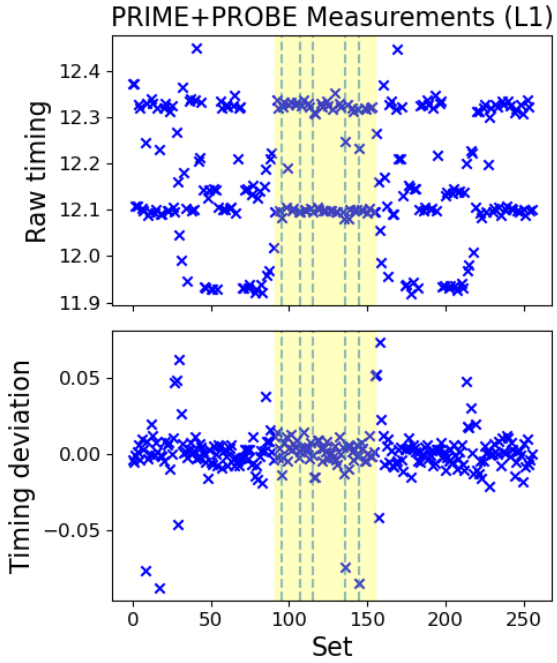


Figure 4: `PRIME+PROBE` timing measurements for each set with a known plaintext and key. The top figure shows raw timing measurements, while the bottom figure normalizes each measurement by the average timing of a set. The highlighted region shows the range of the t-tables (that is, $C(0,0) \rightarrow C(3,255)$). Dashed vertical lines show t-table sets which are *not* accessed during this encryption.

5.2.2 A10 SoC Hardware Prefetcher

In the top chart of Figure 4 we can visually identify the sets which correspond to t-table entries. In the bottom (normalized) figure we can clearly see several low outliers, which represent cache sets that are probed faster than usual, and thus correspond directly to t-table entries that are not accessed during the encryption. These outliers are useful for deducing information about the unknown key. However, there are also several cache sets which *should* be outliers, but are instead very close to the mean. We argue that these false positives are potentially caused by hardware prefetching of adjacent t-table entries, resulting in cache misses in the probe array. We therefore reveal that the attack has to modify both the `PRIME+PROBE` attack technique and statistical analysis to defeat Apple's hardware prefetcher. We discuss these two aspects in the next two subsections respectively.

5.3 Platform-Specific Attack Modifications

Our key insight is that prefetching not only affects the targeted t-tables, but also the attacker's probe array. Generally, without supposing a specific prefetcher implementation, we can assume that sequentially priming the array A will train the prefetcher to tightly associate the addresses in A. Then, when the attacker later probes the array, the A10 will prefetch

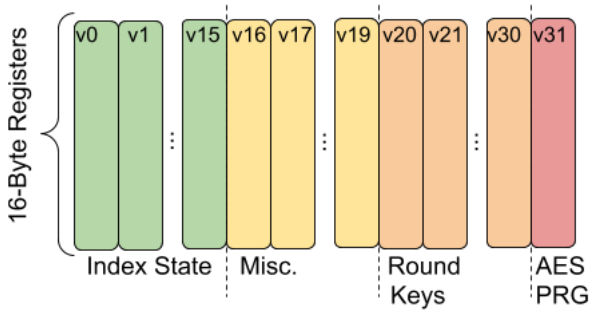


Figure 5: Register map of our randomized PRIME+PROBE implementation. By using the A10's 512 bytes of floating-point registers as state storage and a PRG, we can defeat the prefetcher without any additional memory accesses.

further entries into the cache—potentially evicting t-table entries which were accessed during the encryption and causing false negatives. This problem is further exacerbated by the advanced eviction strategies required to defeat the pseudorandom replacement policy, since such strategies often rely on accessing the entries in A in structured, repetitive ways.

In order to defeat the prefetcher, we must minimize the amount of information which it can learn about A. The most straightforward method to do so, which we employ, is to prime and probe with a uniform random distribution. Generating random numbers and keeping track of state (such as which cache sets have been primed or probed already) would, however, induce many auxiliary memory accesses and add further noise to the measurements. Instead, we rely on an architectural feature of the A10—the ARM NEON floating-point unit (FPU). This FPU includes 32 16-byte registers, as well as an implementation of the ARM AES instructions.

Figure 5 shows details of our randomized PRIME+PROBE implementation. We use the index state (`v0 - v15`) to track sets which have been primed or probed. The miscellaneous registers (`v16 - v19`) hold various counters which help us measure the performance of our technique. We load a full set of AES-128 round keys into registers `v20 - v30`, and then use these round keys (and the ARM AES instructions) to repeatedly encrypt the value in `v31`. Finally, we use the individual bytes in `v31` as a source of randomness for our prime and probe methods. In this way, we can fully randomize our PRIME+PROBE attack without adding any additional noise due to auxiliary memory accesses.

5.4 Statistical Modifications

While the original PRIME+PROBE statistical technique [24] is functional on the A10 SoC, its efficacy is greatly reduced. The hardware prefetcher causes false positives when encrypting, which decreases the distance between $E[\mathcal{M}_K \mid Q_K = 1]$ and $E[\mathcal{M}_K \mid Q_K = 0]$. Instead, we rely on a novel constraint-based statistical technique which is based on exploiting information from true negatives while remaining tolerant of false positives. We note that prefetcher-aware PRIME+PROBE attacks have

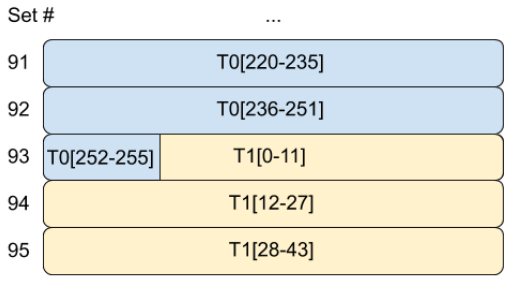


Figure 6: Visual representation of t-tables which are not aligned on the cache block boundary. We find that such t-tables often leak more information than aligned t-tables.

been previous explored [33], but these approaches are tailored to specific prefetchers and lack generality.

First, we heuristically determine two thresholds T^+ and T^-. We use these thresholds to categorize normalized measurements into positives and negatives—a measurement greater than T^+ is categorized as a positive, while a measurement less than T^- is categorized as a negative. For the dataset in Figure 4, for example, we would set $T^+ = -0.02$ and $T^- = -0.10$. We then define a scoring function E for individual timing measurements T_x^M:

$$E(T_x^M) = \begin{cases} 0 & T_x^M > T^+ \\ 1 & T_x^M < T^- \\ (T_x^M - T^+)/(T^- - T^+) & \text{otherwise} \end{cases} \quad (3)$$

Higher values of $E(T_x^M)$, then, indicate stronger evidence that the cache set x is not accessed when encrypting M with the unknown key K. We can determine exactly which tables L and indices y this measurement is useful for by inverting $x = C(L,y)$—we denote the set of these table-index pairs as $\mathcal{Y}$. Finally, we exploit the fact that the t-table lookup indices in the first round are simply equal to $T_\ell[M_i \oplus K_i], i \equiv \ell \bmod 4$. Thus, by definition, we know that higher values of $E(T_x^M)$ indicate stronger evidence that:

$$M_i \oplus y_i \neq k_i, \forall(\ell,y) \in \mathcal{Y} \wedge i \equiv \ell \bmod 4 \quad (4)$$

Using equation 4 we can iteratively eliminate potential key candidates by accumulating many such constraints from various random plaintexts. We note that this attack is particularly effective when the t-tables are not aligned to the cache block size (i.e., $o \bmod B \neq 0$). Figure 6 illustrates the concept. Assume we obtain a measurement for set 92 with a plaintext $M = 0^{128}$. If we obtain strong evidence that this set is not accessed, we would have $\mathcal{Y} = \{(0,236),\ldots,(0,251)\}$ and we could thus build the constraints $k_0,k_4,k_8,k_{12} \notin \{236,\ldots,251\}$. However, if we obtain such evidence for set 93 instead, we would have $\mathcal{Y} = \{(0,252),\ldots,(0,255),(1,0),\ldots,(1,11)\}$. We could then build the constraints $k_0,k_4,k_8,k_{12} \notin \{252,\ldots,255\} \wedge k_1,k_5,k_9,k_{13} \notin \{0,\ldots,11\}$, which eliminates potential key candidates for twice the number of key indices.

Our statistical attack proceeds as described above, iterating over random plaintexts M_i and the corresponding traces $T^{M_i} = \{T_0^{M_i}, \ldots, T_S^{M_i}\}$ and accumulating evidence against certain key candidates. True negatives contribute useful information for eventually deducing the correct key, while false positives do not actively reduce the quality of the attack. False negatives would add incorrect evidence, leading to incorrect rejections of key candidates. However, our randomized attack technique discussed in the last section largely eliminates such effects.

6 Profiled Time-Driven Attacks

We now consider a weaker adversary and additionally present a profiled, time-driven SCA on a t-table AES-128 implementation. The attack works on both our own, casual AES implementation as well as the industry-standard OpenSSL `1.1.1d` AES implementation. While both concrete instantiations of our attack use slightly different tools (described in Section 4), both follow the same general strategy.

6.1 Notation

We first introduce the notation which we use to describe this attack. We consider plaintext messages M and AES encryption keys K, where both $M, K \in \{0,1\}^{128}$. We further add an indexing scheme on plaintexts and keys, parameterized by a division factor d. For a given division factor d, the set of valid indices is I_d and the set of valid values at each index is $\mathcal{V}_d$ such that:

$$I_d = \left\{ i \mid 0 \leq i < \frac{128}{d} \wedge i \in \mathbb{Z} \right\}, |I_d| = \frac{128}{d} \tag{5}$$

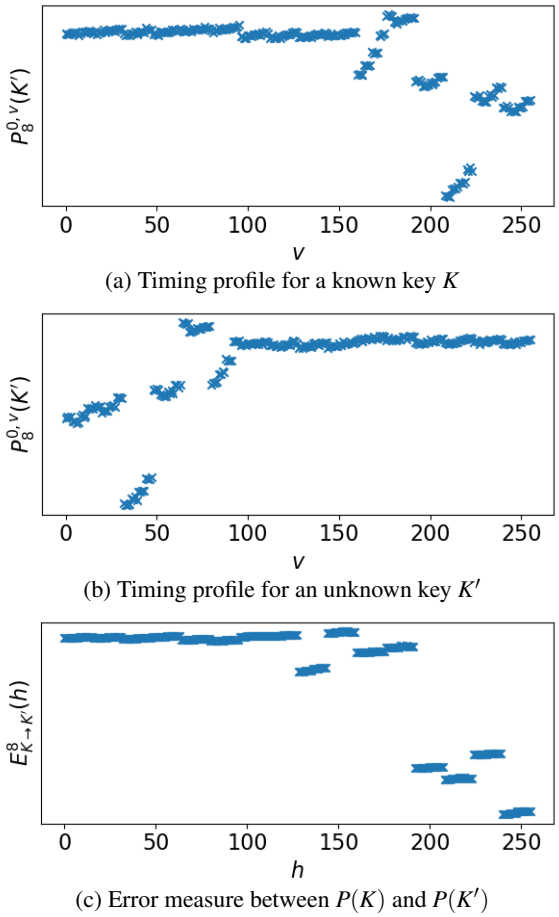
$$\mathcal{V}_d = \left\{ v \mid 0 \leq v < 2^d \wedge v \in \mathbb{Z} \right\}, |\mathcal{V}_d| = 2^d \tag{6}$$

$M_d^i = v$ denotes that the i^{th} index of plaintext M, under division factor d, has value v. The same notation applies to keys K. For example, a division factor of $d = 8$ would index plaintexts and keys at the byte level. A division factor $d = 4$ would index both at the nibble level, while $d = 1$ would index both at an individual bit level. This indexing scheme is similar to the one used in both [27] and [28].

For both attacks, we repeatedly encrypt random plaintexts M with a constant key K and time the duration of the entire encryption process (not including key expansion). We denote the amount of time taken to encrypt M with K as $T_K(M)$, and aggregate these timing measurements into so-called timing profiles $P(K)$. Timing profiles capture information about the average encryption time given a plaintext's value at a specific index. Generally:

$$P_d(K) = \left\{ P_d^{i,v}(K) \mid \forall i \in I_d, \forall v \in \mathcal{V}_d \right\} \tag{7}$$

$$P_d^{i,v}(K) = \langle \{ T_K(M) \mid M_d^i = v \} \rangle \tag{8}$$



(a) Timing profile for a known key K

(b) Timing profile for an unknown key K'

(c) Error measure between $P(K)$ and $P(K')$

Figure 7: $P(K)$, $P(K')$, and $E^d_{K\to K'}(h)$ for an example dataset at the BootROM stage of execution. For this example, we have $d = 8$ and $i = 0$. Timing attacks at this stage of execution are much more successful because of the profiles' *regularity* (clear and predictable clustering).

where $\langle \ldots \rangle$ is the average of a set. For conciseness and without loss of generality, we sometimes drop the parameter d from the notation when it is clearly implied.

6.2 Single-Profile Attacks

In the simplest instantiation of our attack, we collect two profiles $P(K)$ and $P(K')$ for a known key K and an unknown key K'. Our attack then proceeds to extract a set of candidate keys $C^{i,L}_{K\to K'}$ at each index i and with a complexity parameter L. A candidate set is defined as:

$$C^{i,L}_{K\to K'} = \left\{ h \oplus K^i \mid \min_{h \in \mathcal{V}_d}^{L} E^d_{K\to K'}(h) \right\} \tag{9}$$

Given the complexity parameter L, individual key hypotheses h, and an error measure $E^d_{K\to K'}(h)$ on these key hypotheses, the candidate set $C^{i,L}_{K\to K'}$ contains the L key hypotheses $h \oplus K^i$ with the lowest measured error. For this attack, we

define the error measure simply as the mean squared error (MSE):

$$E^d_{K\to K'}(h) = \frac{1}{|\mathcal{V}_d|}\sum_{v=0}^{|\mathcal{V}_d|-1}\left(P^{i,v}_d(K) - P^{i,v\oplus h}_d(K')\right)^2 \quad (10)$$

Given a key hypothesis h, the error measure calculates the differences between the known-key profile $P(K)$ and *permutations* of the unknown-key profile $P(K')$. Figure 7 illustrates this concept. The three charts show $P(K)$, $P(K')$, and $E^d_{K\to K'}(h)$ respectively—it seems that $P(K')$ is exactly reversed from $P(K)$. Such a transformation could be achieved by XOR-ing the indices of elements in $P(K')$ with some value that contains many high 1 bits. The bottom chart of Figure 7 shows the values of $E^d_{K\to K'}(h)$. Clearly, the key hypotheses with the lowest error measure are of the form $h = \texttt{0b1111XXXX}$. The correct key byte here happens to be $\texttt{0xF1} = \texttt{0b11110001}$, confirming our XOR rearrangement hypothesis: the transformation from $P(K') \to P(K)$ is explained by a key guess of $\texttt{0xF1}$ in this case.

Using these extracted candidate sets, we can now begin to reason about the complexity of a simple brute-force attack. Let $G_\mathcal{L}$ be the set of 128-bit key guesses of order $\mathcal{L} \geq 1$ and $||$ be the bitwise concatenation operator. Then, $G_\mathcal{L}$ contains all combinations of key guesses from the candidate sets of all positions:

$$G_\mathcal{L} = \left\{ \Big|\Big|_{i=0}^{|I_d|-1} h^i \mid \forall h^i \in C^{i,\mathcal{L}}_{K\to K'} \right\} \quad (11)$$

The attack strategy proceeds as follows. We begin with G_1, which contains only one key guess - the concatenation of the key hypotheses with the lowest error measure at each index. If this is not the correct key, we then search the set G_2 which contains all key guesses such that the key hypotheses at each index have the lowest or second-lowest error measure. Since $G_1 \subset G_2$, we only test the guesses in $G_2 - G_1$ to avoid repetition. We then iterate, and at each iteration j in the attack we test only the key guesses $G_j - G_{j-1}$. Now, let $\mathcal{L}'_s$ equal the smallest value of $\mathcal{L}$ such that $K' \in G_{\mathcal{L}'_s}$. We now have a brute-force complexity of:

$$\left|G_{\mathcal{L}'_s}\right| = {\mathcal{L}'_s}^{|I_d|} \quad (12)$$

Note that $\mathcal{L}'_s$ is also the maximum error ranking of a correct key hypothesis among all indices i. This is clearly a polynomial worst case bound parameterized by $\mathcal{L}'_s$, and therefore already a feasible attack in a theoretical sense. We note that this attack is bounded by the complexity of a regular brute-force attack on AES (which has complexity 2^{128}) by observing that the maximum possible value of $\mathcal{L}'_s$ is $\left|\mathcal{V}_d\right|$. The worst case brute-force complexity then becomes $\left|\mathcal{V}_d\right|^{|I_d|} = 2^{128}$. In Section 7, we use the value of $\mathcal{L}'_s$ as a metric of our attack's efficacy, since this value is closely related to the computational complexity required to extract the correct unknown key K'. This parameter also quantifies the amount of *correct* information our attack is able to learn about the secret key, i.e., the attack's entropy. We show in the next section that using information from multiple known-key profiles often decreases the value of $\mathcal{L}$ and, correspondingly, the attack's overall entropy.

6.3 Multi-Profile Attacks

While our attack performs much better than a brute-force attack on AES itself, information about timing variations is sometimes not fully captured by the known-key profile $P(K)$ with respect to the unknown-key profile $P(K')$. This effect is well-known in the literature [28], and often leads to very high values of $\mathcal{L}'_s$. The typical solution to this problem has been to improve key enumeration techniques [32]. However, we seek to reduce the complexity even further by instead introducing *multi-profile attacks*, which use information from several known-key profiles to extract key candidates for an unknown-key profile. Similarly to before, let $P(K')$ represent a timing profile for an unknown key K'. Then, consider a set of known keys $K^n = \{K_0, K_1, \ldots, K_{n-1}\}$ and their corresponding timing profiles $P^n = \{P(K_0), P(K_1), \ldots, P(K_{n-1})\}$. We extend the definition of a candidate set $C^{i,L}_{K\to K'}$ from the previous section to the multi-profile case:

$$C^{i,L}_{K^n\to K'} = \bigcup_{j=0}^{n-1} C^{i,L}_{K_j\to K'} \quad (13)$$

$$\left|C^{i,L}_{K^n\to K'}\right| = \alpha^{i,L}\sum_{j=0}^{n-1}\left|C^{i,L}_{K_j\to K'}\right| = \alpha^{i,L}nL \quad (14)$$

For the single-profile case, each position i simply had L key candidates. With this multi-profile case, we increase the number of key candidates per position to $\alpha^{i,L}nL$ where $\alpha^{i,L}$ is a parameter accounting for the case where some candidate sets in $C^{i,L}_{K^n\to K'}$ contain the same key candidates. For brevity, we denote the mean value of $\alpha^{i,L}$ across all key positions i simply as α^L. At first glance, this modification does not look like an improvement – we increase the number of key candidates at each position, which in turn increases the brute-force complexity. However, we extend the previous section's brute-force attack and show that the additional known-key profiles improve performance.

The multi-profile brute-force attack is identical to the single-profile attack, with one drop-in difference. The set of key guesses $G_\mathcal{L}$ is defined identically except that we replace the single-profile candidate set $C^{i,L}_{K\to K'}$ with the multi-profile candidate set $C^{i,L}_{K^n\to K'}$. The attack then iterates identically, testing G_1, then $G_2 - G_1$, up to $G_j - G_{j-1}$. Now, we define $\mathcal{L}'_m$ as the smallest value of $\mathcal{L}$ such that $K' \in G_{\mathcal{L}'_m}$. Although the number of key candidates increases by a factor of $\alpha^{\mathcal{L}'_m}n$, the value of $\mathcal{L}'_m$ itself decreases from $\mathcal{L}'_s$. We show experimentally

in our results that $\alpha^{\mathcal{L}'_m} n\mathcal{L}'_m < \mathcal{L}'_s$, which indicates that the benefit in $\mathcal{L}'_m$ often outweighs the cost of $\alpha^{\mathcal{L}'_m} n$ additional key candidates. We now again state an expression for the worst case brute-force complexity of the multi-profile attack:

$$\left|G_{\mathcal{L}'_m}\right| = \left(\alpha^{\mathcal{L}'_m} n\mathcal{L}'_m\right)^{|I_d|} \tag{15}$$

An alternatively derivation can be expressed as follows. Denote the error ranking of the correct key hypothesis $h \in C^{i,L}_{K_j \to K'}$ as L^i_j. Then, within each index i, we will discover the correct key hypothesis at the *minimal* error ranking L^i_j out of all profiles j – we denote $L^i = \min_{j \in [0,n)} L^i_j$. However the worst case brute-force complexity is still limited by the *maximum* L^i across all indices. Therefore, the complexity can alternatively be stated as:

$$\left(\max_{i \in I_d} \min_{j \in [0,n)} L^i_j\right)^{|I_d|} \tag{16}$$

7 Results

We now proceed to instantiate the attacks described in the previous sections in several separate experiments. First, we attack OpenSSL `1.1.1d` using our improved `PRIME+PROBE` attack. We quantify this attack's success rate in terms of two parameters—the number of random plaintexts, and the number of traces captured for each plaintext. For each random plaintext, we summarize the corresponding traces into an averaged trace and use this average to eliminate key candidates. We do so because any single trace may be noisy, and averaging many traces lets us identify true negatives more accurately. We also use these averaged traces to run a standard `PRIME+PROBE` attack without our improvements, as a comparison. For this attack, we quantify the reduced AES security level by building candidate sets and key guess sets, analogously with the time-driven attack. We then report $\log_2 |G_{\mathcal{L}'}| = 16 \log_2(\mathcal{L}')$ as the reduced AES security level.

For our time-driven attack, we implicitly measure timing variations related to the SoC's memory hierarchy, although we note that the L1 cache can easily accommodate all required data for both AES-128 implementations that we test. Thus our attack makes no explicit assumptions about which level of the memory hierarchy a given set of data is currently located in, and rather just indiscriminately measures black-box encryption timings without manipulating the architectural state. In our first experiment, we attempt to classify whether the caches exhibit vulnerable address-dependent timing variations by using our own, casual AES-128 implementation. Then, we replicate the attack using the standard OpenSSL AES-128 implementation to show the timing side-channel vulnerability of standard cryptographic libraries executing on the Apple's A10 SoC. For each time-driven attack configuration, we calculate the parameter $\alpha^{\mathcal{L}'} n$ which measures the amount of additional, non-duplicate key candidates introduced by including more known-key profiles—this is calculated as the mean of:

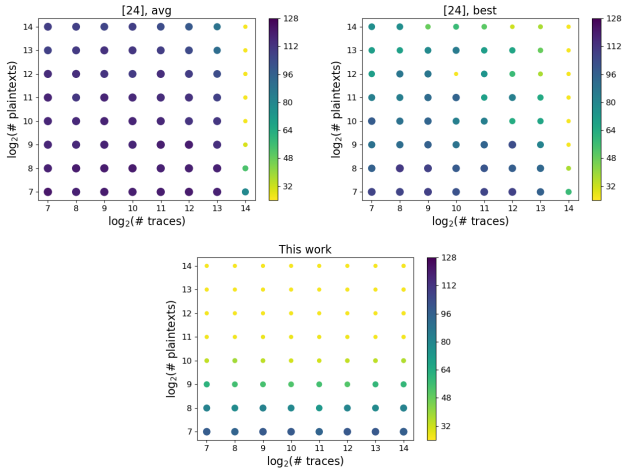


Figure 8: Results of our constraint-based statistical technique, compared to both the best and average performance of the original `PRIME+PROBE` technique [24]. Both figures show that our attack reduces the key search-space drastically for an exhaustive evaluation.

$$\frac{\left|C^{i,\mathcal{L}'}_{K^n \to K'}\right|}{\mathcal{L}'+1} \tag{17}$$

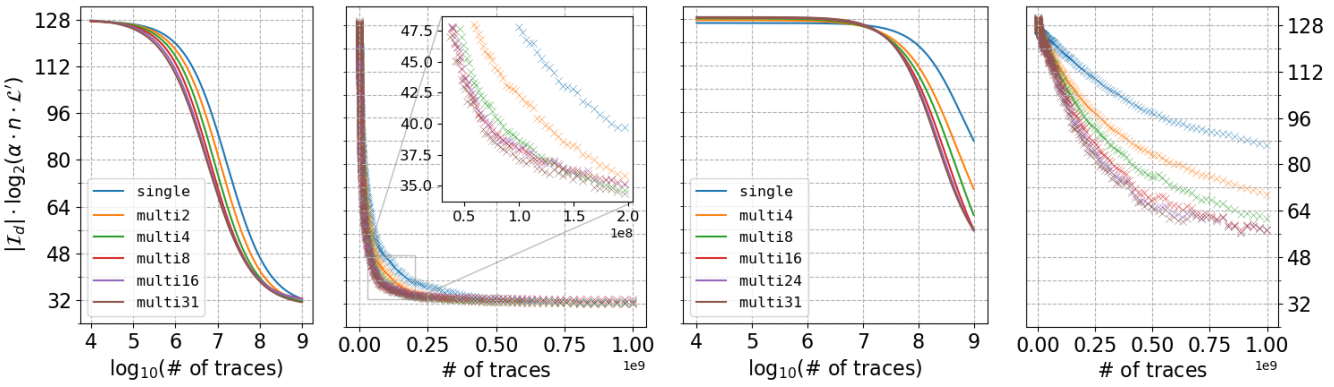
across all indices $i \in I_d$. We can then approximate the reduced security level of AES, measured in bits. The size of the key guess set $G_{\mathcal{L}'}$ can be calculated using equation 15 with $\log_2$:

$$\log_2 |G_{\mathcal{L}'}| = |I_d| \cdot \log_2\left(\alpha n \mathcal{L}'\right) \tag{18}$$

7.1 `PRIME+PROBE` Results

Using Project Sandcastle (Section 4.3) we compile a full Linux system for the iPhone 7, which includes the target of our access-driven attack—OpenSSL `1.1.1d`. We specifically compile OpenSSL with the `no-asm` configuration flag to ensure that the ARM AES instructions are not used, as attacking these is out of scope for this work. This is a standard choice for earlier works [14,21,23] in this field as well. We then boot the Linux kernel using the kernel parameters `isolcpus=1`, `nohz_full=1`, which removes the second CPU core from the scheduler and causes all processes to run on the first core. Finally, we explicitly launch our victim and attack processes on the second core using `taskset 0x2`.

To quantify the success rate of our `PRIME+PROBE` attack, we gathered a large dataset containing measurements for one random key. Specifically, this dataset includes 16384 traces collected for each of 16384 random plaintexts with a naturally unaligned t-table ($o \bmod B = 16$). We then randomly subsample from this dataset to analyze other attack configurations. Our results are shown in Figure 8 which displays the reduced AES security level along both hyperparameters.



(a) BootROM attack models (b) BootROM security levels (c) OpenSSL attack models (d) OpenSSL security levels

Figure 9: The security level (in bits) of AES under our timing attack with varying number of traces and attack configurations, executing at the BootROM level (a, b) and at the OpenSSL level (c, d).

As expected, increasing the number of random plaintexts and traces per plaintext generally decreases the security level of AES-128. We find that, generally, collecting traces for *more* plaintexts is better than collecting more traces per plaintext, primarily because our technique can build more constraints in these circumstances. The results show that we consistently outperform classical techniques [24], even when they perform at their best. The difference between our attack and the earlier work is significant because brute-force is possible on our reduced security level whereas the conventional method fails to sufficiently lower the security level, unless a large amount of data is collected.

7.2 Time-Driven Attacks at BootROM Level

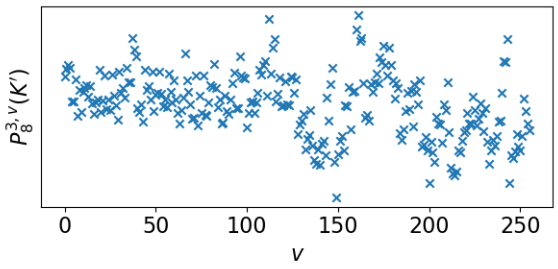
Figures 9 (a) and 9 (b) summarize the experimental results of our time-driven attack at the BootROM level. Our data for this stage was collected using our toolkit, specifically utilizing the (asynchronous) profiling program in Listing 4. We are interested in visualizing our attack success rate as the number of traces increases—however, independently profiling for each number of traces would take a long time. Instead, we profile with the maximum number of traces (10^9) and then read out partial data via USB to simulate profiling with less traces. This technique allows us to track our attack's efficiency with a varying number of traces, but does introduce some noise from USB IO. To compensate, we perform a number of unmeasured, random AES encryptions after reentering the profiling task to ensure all AES data is resident in the cache.

From this data, we draw a number of conclusions. First, we find that increasing the number of collected traces for each profile generally decreases the AES security level across all attack configurations. For the higher-order (large n) configurations, the security level drops faster (with less traces) as compared to the single-profile configuration. This effect is most significant around 10^7 traces, where the `multi32` attack configuration lowers the AES security level by 22 more bits than the `single` configuration. We do find that all configurations eventually reach a minimum security level of 32 bits, which indicates that 75% of the unknown key bits K' can be deduced using this specific attack strategy. The remaining key space can easily be searched exhaustively—using an Intel® Core™i7-7700HQ processor, the brute force attack takes only 288 CPU-seconds. We thus conclude that our attacks successfully break AES at the BootROM stage of execution.

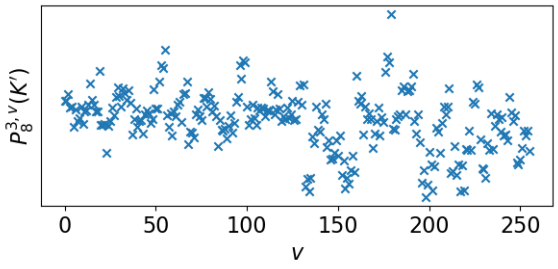
7.2.1 Cache Bank Conflicts

It is well documented in the literature that cache attacks can only extract a maximal amount of key information, depending on the technique used. Osvik et al. showed that their access-driven attack is only able to extract the top 4 bits of each key byte, mainly because the attack depends on tracing cache hits and misses on 64-byte cache blocks [24]. Other works show that attacks *within* the cache block granularity are feasible as well—*CacheBleed* [35] exploits timing information from *cache bank conflicts* to extract 60% of the bits in an RSA private key. Jiang et al. showed that cache bank conflict timing attacks are applicable to AES as well [21]. Since iTimed keeps all AES data in the L1 cache during the profiling phase, we believe that our attack measures timing variations inherently caused by cache bank conflicts.

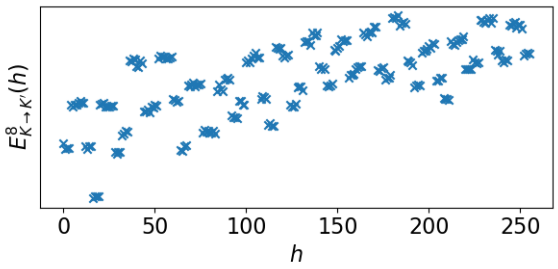
Various observations from the data support this cache bank conflict hypothesis. The timing patterns for this experiment tend to be very regular, with clear clusters at predictable locations of M_i^d as seen in Figure 7. The known-key profile, for example, has six clusters at even divisions of 32, and then four more clusters at even divisions of 16. While these patterns are distinct (and exploitable for timing attacks), they are also very small—the maximum range in the timing information for the known-key profile is only about 0.0035 timer cycles (145.83 picoseconds). Since this range is much less than the period of a single clock cycle, we believe the origin of these timing



(a) Timing profile for a known key K



(b) Timing profile for an unknown key K'



(c) Error measure between $P(K)$ and $P(K')$

Figure 10: $P(K)$, $P(K')$, and $E^d_{K\to K'}(h)$ for an example OpenSSL dataset. For this example, we have $d = 8$ and $i = 3$. Note the irregularity of the profiles and error measure as compared to Figure 7—this irregularity lowers the effectiveness of our attacks and increases the required number of traces.

variations is microarchitectural in nature. We also find that profiles collected at the BootROM level can be used in cross-device attacks against a separate iPhone 7, which suggests that our attack captures timing variations from a consistent, device-independent microarchitectural side channel.

7.3 Time-Driven Attacks on OpenSSL

Figures 9 (c) and 9 (d) show the results of our attack on the OpenSSL `1.1.1d` AES implementation. Similar to the BootROM experiment, we find that increasing the number of collected profiles generally decreases the security level of OpenSSL AES. Multi-profile configurations again outperform the `single` configuration, but the effect is much more significant for this experiment. We find that, at maximum, the `multi31` configuration lowers the AES security level by 32 more bits than the `single` configuration. We were unable to determine an exact lower bound for this experiment due to high profiling times—collecting 32 profiles with 10^9 traces each took two weeks. We believe that fully visualizing the reduced OpenSSL security level would require 32 profiles with about 10^{11} traces each, and require about 200 weeks to profile. Based on our experience with this attack, we believe that the `single` configuration is bounded at about 80 bits while the `multi31` configuration is bounded at about 48 bits. Using the same Intel® processor as in the previous experiment, we estimate the brute-force attack time for our proposed `multi31` configuration needs 6065.5 CPU-hours which is feasible given enough cores. By contrast, the conventional single-profile brute force attack would require over 3.5 billion CPU-years which is arguably impractical.

Figure 10 shows that the OpenSSL profiles are less regular than the BootROM profiles (Figure 7). There are no visible clusters that could be used to efficiently compare key hypotheses. Instead, the attack relies on the timing information's peaks and troughs. The maximum range of these timing variations is approximately .03 timer cycles (1.25 nanoseconds), which is an order of magnitude increase compared to the BootROM experiments. We believe this increase is explained by MMU address translation overhead, since the Linux kernel utilizes a full virtual memory system. Accesses to the translation lookaside buffer (TLB), which caches virtual to physical address translations, could also contribute to these timing variations [11].

8 Conclusion and Future Work

Hardware security research on iPhones is notoriously difficult. This paper is the first effort to enable hardware security experiments on the Apple iPhone SoCs. Extending a new public BootROM exploit, we reverse-engineered a significant part of the BootROM, exposed useful functions and interfaces within, and increased the experimental scope of our platform to whole system stacks, including hardware, driver, and operating system layers. Our effort greatly lowers the difficulty of implementing future hardware security experiments on Apple's SoCs. Using our tool, we implemented the first SCAs on the target platform, showed practical secret-key extraction on AES, and even improved the state-of-the-art of SCA by addressing platform-specific challenges and proposing statistical enhancements. Future extensions of this work may include attacking other algorithms such as public-key cryptosystems and evaluating remote fault or physical side-channel attacks with the help of our tool.

9 Ethical Disclosure

This work began on September 27th, 2019, when checkm8 was released to the public. We contacted Apple's product security team on July 11th, 2020, to report our findings prior to submitting the paper or revealing it on any other public forum.

References

[1] Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on aes (short paper). In *International Conference on Information and Communications Security*, pages 112–121. Springer, 2006.

[2] Brandon Azad. KTRW: The journey to build a debuggable iPhone. 36C3, 2019.

[3] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[4] Drew Branch. Debugging ios applications: A guide to debug other developers' apps. Medium, 2017.

[5] Joan Daemen and Vincent Rijmen. The block cipher rijndael. In *International Conference on Smart Card Research and Advanced Applications*, pages 277–284. Springer, 1998.

[6] L. Deshotels, C. Carabaș, J. Beichler, R. Deaconescu, and W. Enck. Kobold: Evaluating Decentralized Access Control for Remote NSXPC Methods on iOS. In *IEEE Symposium on Security and Privacy (SP)*, pages 399–413, 2020.

[7] Lorenzo Franceschi-Bicchierai. Key iphone source code gets posted online in 'biggest leak in history'. Motherboard, 2018.

[8] Lorenzo Franceschi-Bicchierai. The prototype iphones that hackers use to research apple's most sensitive code. Vice, 2019.

[9] 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):1–27, 2018.

[10] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. Ecdsa key extraction from mobile devices via nonintrusive physical side channels. In *ACM SIGSAC Conference on Computer and Communications Security*, page 1626–1638, 2016.

[11] 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.

[12] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321. Springer, 2016.

[13] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[14] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, 2015.

[15] Joshua Ho and Brandon Chester. The iphone 7 and iphone 7 plus review: Iterating on a flagship. anandtech, 2016.

[16] Apple Inc. Apple open source, https://opensource.apple.com/.

[17] Apple Inc. *Security enclave processor for a system on a chip*. Number US8832465B2. 2012.

[18] Apple Inc. *Dynamic voltage and frequency management based on active processors*. Number US9304573B2. 2013.

[19] Apple Inc. Apple platform security guide. Spring 2020.

[20] The iPhone Wiki. Bootrom exploits, https://www.theiphonewiki.com/wiki/Bootrom#Bootrom_Exploits.

[21] Zhen Hang Jiang and Yunsi Fei. A novel cache bank timing attack. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 139–146. IEEE, 2017.

[22] Alex Kovrizhnykh. Technical analysis of the checkm8 exploit. Digital Security, 2019.

[23] 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, Austin, TX, August 2016. USENIX Association.

[24] 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.

[25] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. 2002.

[26] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaking sgx by software-controlled voltage-induced hardware faults. In *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 1–6. IEEE, 2019.

[27] Chester Rebeiro and Debdeep Mukhopadhyay. Boosting profiled cache timing attacks with a priori analysis. *IEEE Transactions on Information Forensics and Security*, 7(6):1900–1905, 2012.

[28] Raphael Spreitzer and Benoît Gérard. Towards more practical time-driven cache attacks. In *IFIP International Workshop on Information Security Theory and Practice*, pages 24–39. Springer, 2014.

[29] Raphael Spreitzer and Thomas Plos. On the applicability of time-driven cache attacks on mobile devices. In *International Conference on Network and System Security*, pages 656–662. Springer, 2013.

[30] 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.

[31] Luca Todesco. The One Weird Trick SecureROM Hates. Power of Community, 2019.

[32] Nicolas Veyrat-Charvillon, Benoît Gérard, Mathieu Renauld, and François-Xavier Standaert. An optimal key enumeration algorithm and its application to side-channel attacks. In *International Conference on Selected Areas in Cryptography*, pages 390–406. Springer, 2012.

[33] Daimeng Wang, Zhiyun Qian, Nael Abu-Ghazaleh, and Srikanth V Krishnamurthy. Papp: Prefetcher-aware prime and probe side-channel attack. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[34] 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.

[35] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: A timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

Appendix A Code Snippets

A.1 DFU Mode Functionality

Listing 1: Relevant parts of the USB core

```
1  // data phase variables
2  static uint8_t *dp_buf;
3  static uint32_t dp_len, dp_rcvd;
4
5  void usb_core_handle_receive(
6      uint8_t *rx_buffer, bool setup_pkt,
7      int length, bool *data_phase){
8   ...
9   // data packet handler
10  if(!setup_pkt && length > 0) {
11   copy_len = dp_len - dp_rcvd;
12   if(length <= copy_len) {
13     copy_len = length;
14   }
15
16   memcpy(dp_buf, rx_buffer, copy_len);
17   dp_buf += copy_len;
18   dp_rcvd += copy_len;
19   *data_phase = true;
20
21   if(dp_rcvd == dp_len &&
22       intf_num >= 0 &&
23       intf_num < num_interfaces &&
24       interfaces[intf_num]->handle_data
25         != NULL) {
26      interfaces[intf_num]->handle_data();
27      dp_rcvd = 0;
28      dp_len = 0;
29      dp_buf = NULL;
30      *data_phase = false;
31     }
32  }
33  ...
34  // setup packet handler
35  if(setup_pkt) {
36   if(intf_num < num_interfaces &&
37        interfaces[intf_num]->handle_setup
38          != NULL) {
39
40     interfaces[intf_num]->handle_setup(
41       rx_buffer, &dp_buf);
42   }
43   ...
44 }
```

Listing 2: DFU mode setup handler

```
1  int dfu_handle_setup(
2      struct usb_request *req,
3      uint8_t **out_buffer) {
4    ...
5    if(type == '\x01') { // DFU download
6      len = req->wLength;
7      if(len == 0) {
8        status = SYNC_MANIFEST;
9      }
10     else {
11       if(len > 0x800) { // error
12         dfu_state = DFU_IDLE;
13         return -1;
14       }
15       *out_buffer = io_buffer;
16     }
17     bytes_expecting = len;
```

```
18        return len;
19      }
20      ...
21    }
```

Listing 3: DFU mode data handler

```
1   void dfu_handle_data(int len) {
2     if(bytes_expecting == len) {
3       if(bytes_received + len
4            <= img_buf_len) {
5         memcpy(img_buf + bytes_received,
6                  io_buffer, len);
7         dfu_state = DFU_IDLE;
8         bytes_received += len;
9         bytes_expecting = 0;
10        return;
11      }
12      status = FAIL;
13      dfu_done = true;
14      event_notify(&dfu_event);
15      return;
16    }
17    dfu_state = DFU_ERROR;
18    return;
19  }
```

A.2 Attack Programs

Listing 4: Time-driven profiling program

```
1   /* Main entry */
2   PAYLOAD_SECTION
3   void entry_async(uint64_t *base)
4   {
5    uint8_t msg_old[16];
6    uint8_t key_sched[176];
7    uint64_t timing;
8
9    // get initial params
10   uint8_t *msg = (uint8_t *) base[0];
11   uint8_t *key = (uint8_t *) base[1];
12   struct aes_cnst *c =
13      (struct aes_cnst *) base[2];
14   struct data *data =
15      (struct data *) base[3];
16   uint32_t num_iter = (uint32_t) base[4];
17
18   expand_key(key, key_sched, 11, c);
19   for(i = 0; i < num_iter; i++)
20   {
21    // generate a new msg
22    get_random(msg, 16);
23    memcpy(msg_old, msg, 16);
24
25    // encrypt it and measure time
26    start = get_ticks();
27    aes128_encrypt(msg, key_sched, c);
28    timing = get_ticks() - start;
29
30    // update counters
31    update_data(data, msg_old, timing);
32   }
33
34   event_notify(&data->ev_done);
35   task_exit(0);
36  }
```

Appendix B Algorithms

B.1 AES-128

For this work, we use AES-128 to encrypt a 16-byte input M using a 16-byte key K. During this encryption, AES utilizes four lookup tables—mainly T_0, T_1, T_2, T_3 (each of which contains 256 4-byte constants) and S (which contains 256 1-byte constants). The t-tables implement the SubBytes, ShiftRows, and MixColumn operations whereas the S-box implements SubBytes and ShiftRows [5]. We index the plaintext M as bytes $M = M_0 \| \ldots \| M_{15}$ and the key K in four-byte chunks $K = K_0 \| \ldots \| K_3$. The key is expanded into 10 round keys $K^i, 1 \leq i \leq 10$ with $K^0 = K$. Then, utilizing an intermediate byte-indexed state x initialized as $x^0 = M \oplus K$, we iterate for $0 \leq i < 9$:

$$(x_0^{i+1} \| \ldots \| x_3^{i+1}) = T_0[x_0^i] \oplus T_1[x_5^i] \oplus T_2[x_{10}^i] \oplus T_3[x_{15}^i] \oplus K_0^{i+1}$$
$$(x_4^{i+1} \| \ldots \| x_7^{i+1}) = T_0[x_4^i] \oplus T_1[x_9^i] \oplus T_2[x_{14}^i] \oplus T_3[x_3^i] \oplus K_1^{i+1}$$
$$(x_8^{i+1} \| \ldots \| x_{11}^{i+1}) = T_0[x_8^i] \oplus T_1[x_{13}^i] \oplus T_2[x_2^i] \oplus T_3[x_7^i] \oplus K_2^{i+1}$$
$$(x_{12}^{i+1} \| \ldots \| x_{15}^{i+1}) = T_0[x_{12}^i] \oplus T_1[x_1^i] \oplus T_2[x_6^i] \oplus T_3[x_{11}^i] \oplus K_3^{i+1}$$

Then in the final round, we set $i = 9$ and calculate

$$(x_0^{10} \| \ldots \| x_3^{10}) = \left(S[x_0^9] \| S[x_5^9] \| S[x_{10}^9] \| S[x_{15}^9]\right) \oplus K_0^{10}$$
$$(x_4^{10} \| \ldots \| x_7^{10}) = \left(S[x_4^9] \| S[x_9^9] \| S[x_{14}^9] \| S[x_3^9]\right) \oplus K_1^{10}$$
$$(x_8^{10} \| \ldots \| x_{11}^{10}) = \left(S[x_8^9] \| S[x_{13}^9] \| S[x_2^9] \| S[x_7^9]\right) \oplus K_2^{10}$$
$$(x_{12}^{10} \| \ldots \| x_{15}^{10}) = \left(S[x_{12}^9] \| S[x_1^9] \| S[x_6^9] \| S[x_{11}^9]\right) \oplus K_3^{10}$$

and return x^{10} as the ciphertext.