iTimed: Cache Attacks on the Apple A10 Fusion SoC

Gregor Haas, Seetal Potluri, and Aydin Aysu
Department of Electrical and Computer Engineering
North Carolina State University
{ghaas, spotlur2, aaysu}@ncsu.edu

***Abstract*—This paper proposes the first cache timing side-channel attack 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). Using this toolkit, we then implement an access-driven cache timing attack (in the style of `PRIME+PROBE`) as a proof-of-concept illustrator.**

The advanced hardware control enabled by our toolkit allowed us to reverse-engineer key microarchitectural details of the Apple A10 Fusion's memory hierarchy. We find that the SoC employs a randomized cache-line replacement policy as well as a hardware-based L1 prefetcher. We propose statistical innovations which specifically account for these hardware structures and thus 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 by 50 more bits than a straightforward adaptation of `PRIME+PROBE`, while requiring only half as many side channel measurement traces.

I. Introduction

With rare exceptions [1] [2], 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 [3], partial source code releases [4], or even illegally leaked source code [5] 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 [6]. 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 [7], 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 [9], 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 than for 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 [10], secure co-processor [11], 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 [12] or VoltJockey [13] 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 [3]. 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 [14].

Contributing to the ongoing effort to build quality, open-source iPhone research tools [7] [9], 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. Utilizing these toolkits, we present a proof-of-concept access-driven (as in [15]) cache timing SCA. Our toolkit allows us to bootstrap a research-friendly software environment on Apple hardware and reverse-engineer key details of the caching microarchitecture. Then, accounting for these, we create novel statistical analyses which significantly improve the success rate of `PRIME+PROBE` on t-table AES-128. To the best of our knowledge, this is the first such SCA on an Apple SoC.

A. Contributions

The central contributions of this paper are:

- Building off of the checkm8 exploit, we implement an extensible BootROM toolkit to base extensive hardware security experiments on. This toolkit is open-sourced at https://github.com/iTimed-Toolkit.
- Using this toolkit, we develop and implement an access-driven timing side channel attack based on the `PRIME+PROBE` strategy. We show that a straightforward implementation of `PRIME+PROBE` may not leak sufficient information due to Apple's cache replacement policies and prefetcher.
- To successfully implement this attack, we propose novel attack and statistical techniques that account for the A10's microarchitecture. We find that our modified `PRIME+PROBE` attack can lower the OpenSSL `1.1.1d` AES-128 implementation's security level by up to 50 more bits than a standard `PRIME+PROBE` attack, while requiring only half as many traces. To the best of our knowledge, this is the first such access-driven SCA on an Apple SoC.

II. 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.

A. 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, this 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.

B. checkm8

On September 27th, 2019, independent iOS security researcher axi0mX released a proof-of-concept tool named *checkm8*[1], which exploits a use-after-free vulnerability in the DFU mode's USB core implementation. Using this vulnerability, it is possible to craft a buffer overflow attack that overwrites a function pointer on the heap (for technical details, see [16]). Then, using the DFU mode's normal functionality, an arbitrary shellcode can be uploaded and executed with full permissions in the BootROM. This breaks all 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 IV, 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 IV that these read/write/execute primitives can be combined and extended in powerful ways to build a more advanced research platform.

C. 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

[1] https://github.com/axi0mX/ipwndfu

specifically target an iPhone 7 (model $A1778$) 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[2]—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[3] configuration: two power efficient cores, and two high performance cores [17]. 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.

D. Target Algorithm for SCAs

For this work, we analyze AES-128 which encrypts 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 [18]. 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

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

and return x^{10} as the ciphertext.

E. Timing Attacks on AES

Cache timing attacks can be broadly classified into three categories, ordered by adversary strength: time-driven attacks, trace-driven attacks, and access-driven attacks. Time-driven and trace-driven attacks were first theoretically discussed in [19], experimentally demonstrated later by Bernstein in [20] and by Acıiçmez et al. in [21], respectively, and then improved with better statistical analysis [22], [23]. These attacks mainly define their threat model adversary as a passive observer.

Access-driven attacks, by contrast, assume that the adversary can actively manipulate the state of the cache. These were first demonstrated with the seminal `PRIME+PROBE` technique [15], but have become much more popular recently. Discoveries of techniques such as `FLUSH+RELOAD` [24], `EVICT+RELOAD` [25], and `FLUSH+FLUSH` [26], 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 [27] and subsequently extended to others [23]. More recently, Lipp et al., at USENIX 2016, ported access-driven attacks to ARM Android devices [28]. 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 [29]. 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.

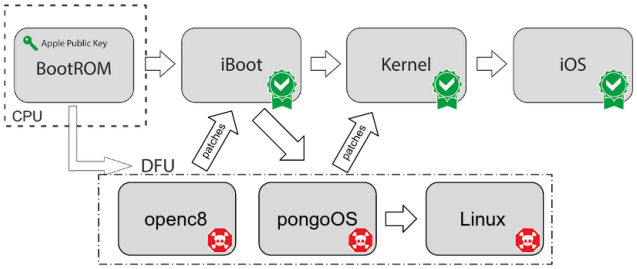

Fig. 1: Summary of Apple's secure boot chain (top, [16]), and how each component of our research platform interacts with it (bottom). Our tooling enhances checkm8 by addressing its limitations and combines our own extensions (openc8) with pongoOS and Project Sandcastle.

III. Threat Model

Our access-driven attack (Section V), follows the standard synchronous threat model for access-driven timing SCAs [15]. Specifically, we assume that the adversary has the ability to synchronously trigger AES encryptions with known plaintexts. For this specific attack instantiation, 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. We note that cross-core access-driven attacks are possible (i.e. through L2), but we did not pursue this attack style in order to simplify our initial microarchitectural reverse-engineering. Furthermore, the attack assumes knowledge of the virtual address of the t-tables—this is a standard assumption in the literature [15] and is not difficult to learn in practice.

[2] https://deviceatlas.com/blog/most-popular-iphones
[3] https://www.arm.com/why-arm/technologies/big-little

IV. Tooling

While checkm8 is a powerful exploit technique, the proof-of-concept released by axi0mX is not suitable for extensive hardware security research. We fully re-implement the base checkm8 exploit with novel extensions to improve both reliability and extensibility. We also discuss two other tools which were not originally intended for hardware security research, and show how we incorporate these into our research platform. Figure 1 shows a summary view of our research platform. When discussing openc8, we refer to the computer that runs the toolkit as the *host* and the target iPhone as the *device*.

A. Expanding the checkm8 Toolkit

We created our own open-source toolkit (named *openc8*) 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*.

1) Reliability: We first improved the tool's success rate, both in terms of successfully exploiting checkm8 and system stability. checkm8 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 caused by the need for stable partial USB requests. 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[4] with a MAX3421E USB host controller. The Arduino driver for this host controller is open source and rather minimal, 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.

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

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

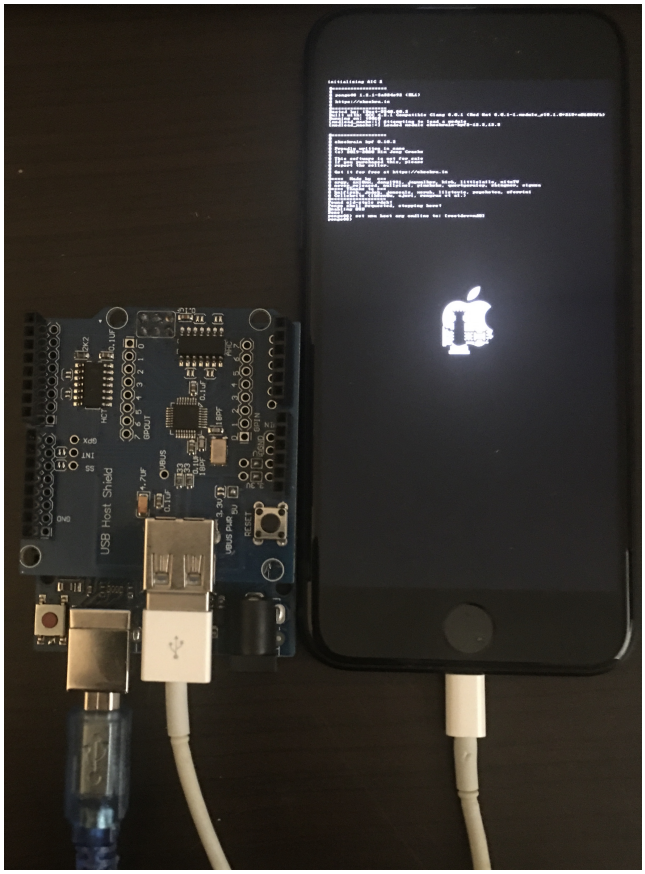

Fig. 2: Arduino with MAX3421E USB host shield connected to our test iPhone 7. The Arduino is a USB proxy between the host and device and generates correct partial requests when necessary. This setup successfully addresses checkm8's reliability issues. This iPhone is running *pongoOS*, briefly discussed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

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.

3) Execution Framework: In our toolkit, we follow the same general technique as axi0mX for arbitrary code execution. After exploiting the device, we upload a *handler* shellcode which replaces the default USB data handler in the DFU mode implementation. This USB handler, in most cases, will simply forward incoming USB transactions to the existing DFU handler thus preserving the functionality of DFU mode. However, specially formatted USB requests are instead forwarded to the shellcode, for request-specific processing. At this point, our implementation diverges from

that of axi0mX. Rather than implementing a static set of functions in the shellcode (such as the read/write/execute functionality in the original checkm8), we implement an extensible system for installing and uninstalling so-called payloads—short programs which implement some desired functionality extension for the BootROM.

4) Payload Infrastructure: openc8 additionally includes full build and execution systems for these payloads. 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. All iPhones which we investigate in this work are ARMv8-A devices—therefore, we include cross compiler support for this architecture so that experimental 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 function calls are translated to use absolute jumps rather than relative jumps, so payloads can be installed anywhere in the device's memory space.

B. 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* [30], 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*[5]. 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 II-A) 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.

1) Integration with openc8: openc8 fully supports both building and booting to pongoOS, which is a necessary functionality since checkra1n does not fully support all platforms which we use. In openc8, pongoOS is compiled as a payload similarly to all of our other experimental code. However a very specific set of patches must be made to iBoot in order for pongoOS to boot successfully. The details of these patches are not known except to the checkra1n developers, but an attempt can still be made to apply them. Although the host-side checkra1n executable is obfuscated such that static reverse-engineering is not possible, all USB traffic between checkra1n and the device is completely unencrypted. Using Wireshark[6], we snooped this USB traffic and extracted checkra1n's patching shellcode. Although we could have attempted to directly statically reverse-engineer this, we found that the functionality is quite complex. The shellcode is recursive, reinitializes the DFU mode several times, and includes many hard-coded constants. Therefore, we simply saved this shellcode binary and send it to the device at the appropriate time.

C. 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.

V. iTimed: Access-Driven Attack on Apple A10

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 [15], 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` [24] and `FLUSH+FLUSH` [26]. However, `PRIME+PROBE` attacks are still relevant today and the most recent publications exclusively use them as the canonical example[31].

A. Notation

We closely follow the notation presented by Osvik et. al. in `PRIME+PROBE` [15]. Access-driven attacks must

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

[6]https://github.com/wireshark/wireshark

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:

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

Encrypting a message M with a key K will cause memory accesses to certain t-table entries. We define an oracle $\mathcal{Q}_K(M, L, y)$, which equals 1 if encrypting M with K will access index y in t-table L. Thus, by repeatedly querying $\mathcal{Q}$ 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 $\mathcal{Q}$ is unrealistic and instead base their attack on an unreliable oracle $\mathcal{M}(M, L, y)$. Specifically, $\mathcal{M} \approx \mathcal{Q}$ such that, for many (K, M, L, y), the expectation of $\mathcal{M}$ is higher when $\mathcal{Q}_k(M, L, y) = 1$ than when $\mathcal{Q}_k(M, L, y) = 0$:

$$E\left[\mathcal{M}_K \mid \mathcal{Q}_K = 1\right] > E\left[\mathcal{M}_K \mid \mathcal{Q}_K = 0\right] \quad (2)$$

B. 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 $\mathcal{Q}_K(M, L, y) = 0$, we would expect that all of the attacker's data is still present in cache set $x = C(L, y)$. By contrast, if $\mathcal{Q}_K(M, L, y) = 1$, 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

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

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

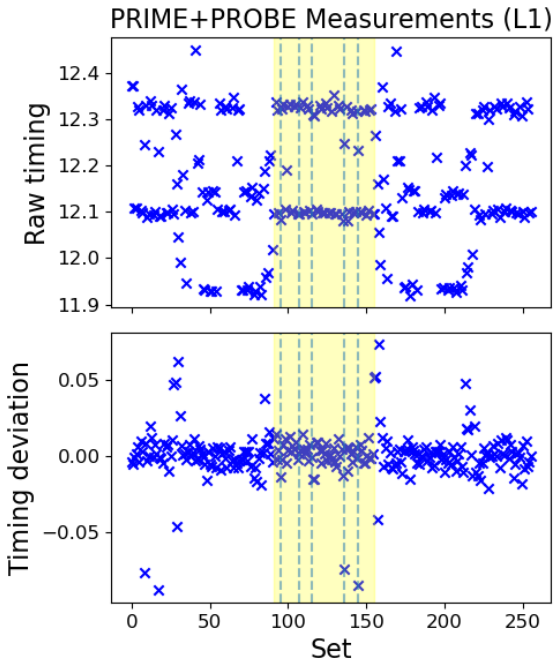

Fig. 3: PRIME+PROBE timing measurements for each set with a known plaintext and key. The top figure shows raw timing measurements, while the bottom figure normalizes each measurement by the average timing of a set. The highlighted region shows the range of the t-tables (that is, $C(0, 0) \rightarrow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

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 [28]. 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 [32] 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 3 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.

2) A10 SoC Hardware Prefetcher: In the top chart of Figure 3 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.

Prefetching is a common technique used in cache architectures which increases spatial locality. Prefetchers learn the CPU's current *working set* (group of addresses which are commonly accessed together) and store all of these addresses in the cache when one of them is accessed—even if the addresses reside in different cache data blocks. Prefetchers have been proposed as active cache timing SCA defenses [34]. We therefore reveal that a successful access-driven timing SCA on the A10 SoC has to modify both the `PRIME+PROBE` attack technique and statistical analysis in order to defeat Apple's prefetcher implementation. We discuss these two aspects in the next two subsections respectively.

C. 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 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 use, 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.

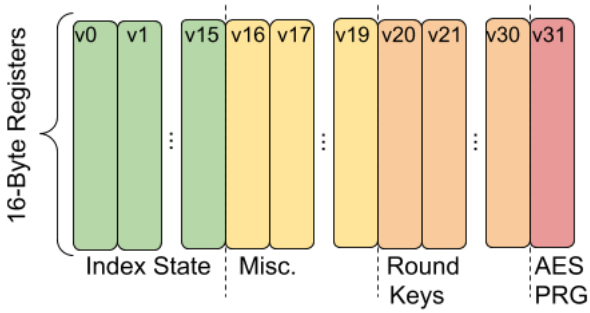

Fig. 4: Register map of our randomized `PRIME+PROBE` implementation. By using the A10's floating-point registers as state storage and a PRG, we can defeat the prefetche without any additional memory accesses.

Figure 4 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.

D. Statistical Modifications

While the original `PRIME+PROBE` statistical technique [15] is functional on the A10 SoC, its efficacy is greatly reduced (see Section VI). The hardware prefetcher causes false positives when encrypting, which decreases the distance between $E[\mathcal{M}_K \mid \mathcal{Q}_K = 1]$ and $E[\mathcal{M}_K \mid \mathcal{Q}_K = 0]$. 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 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.

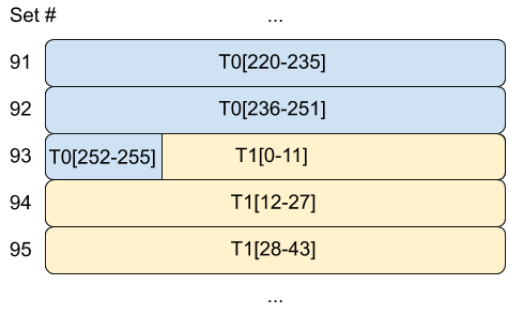

Fig. 5: Visual representation of t-tables which are not aligned on the cache block boundary. We find that such t-tables often leak more information than aligned t-tables.

For the dataset in Figure 3, for example, we would set $T^+ = -0.01$ and $T^- = -0.05$—see Figure 7 for a comprehensive performance evaluation. We then define a scoring function E for individual timing measurements T_x^M:

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

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 5 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.

VI. Results

We now proceed to instantiate the attack described in the previous sections on real Apple hardware. We attack the industry-standard OpenSSL `1.1.1d` using our improved `PRIME+PROBE` attack, and quantify the 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. 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 defining a simple brute-force attack. The key candidates for each key index are scored using either our constraint-based approach or the classic `PRIME+PROBE` approach, and then sorted by score. We then find the maximum position L of a correct key candidate across all key indices, and report $16 \log_2 L$ as the reduced AES security level. This models an adversary who performs a simple breadth-first search across key candidates, informed only by cache side-channel leakage information.

A. *PRIME+PROBE Results*

Using Project Sandcastle (Section IV-C) 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 [25], [35], [28] 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, encrypted by OpenSSL 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 summarized in Figure 6 which displays the reduced AES security level along both hyperparameters. As

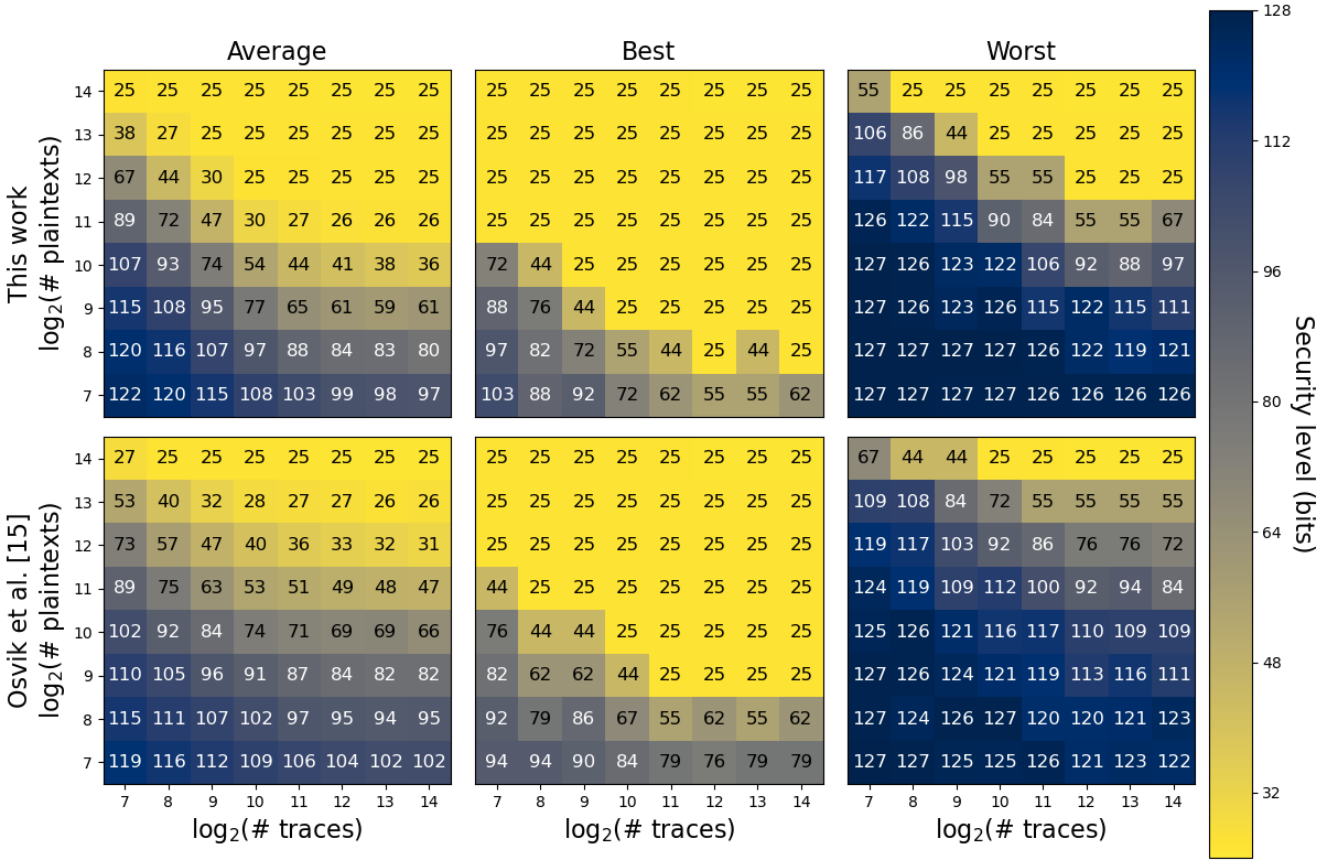

Fig. 6: Results of our constraint-based statistical technique, compared to the average, best, and worst performance of the original PRIME+PROBE technique [15]. Our attack can reduce the key search-space for an exhaustive evaluation.

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.

The results show that we outperform classical techniques [15], even when they perform at their best and especially when they perform at their worst. When comparing the average performance of both attacks, we find the best speedup for the configuration with 1024 plaintexts, averaged from 8192 traces each—our attack recovers 31 more bits of key material than the classic attack. When comparing the best performance for both attacks, we find the best improvement for the configuration with 256 plaintexts, averaged from 4096 traces each. We recover 37 more bits of key material in this case. Finally, when comparing the worst-case performance for both attacks, we find that we can recover 50 more bits of key material under the configuration with 4096 plaintexts, averaged from 4096 traces each. Note that, generally, our attack brings another "row" (as in Figure 6) into the brute-forceable range. This represents a $2\times$ reduction in the number of side channel traces required.

We also explore the effect of threshold choice on the attack's guessing entropy. Specifically we focus on the configuration with 2048 plaintext measurements, averaged from 2048 traces each. We then set $T^+, T^- \in [0.05, -0.15]$ such that $T^- < T^+$, and run our constraint-based attack multiple times. We report the average-case performance in Figure 7. We find that our attack has a narrow band of thresholds for which it leads to successful key extraction. For higher T^+, T^- (bottom left of Figure 7) we observe generally increasing success rates as thresholds increase. With these configurations, we classify many measurements as negatives (often falsely). This leads to high guessing entropies, which then generally decrease as the threshold band moves to a more correct range. For lower T^+, T^- (top right of Figure 7), we observe that the attack fails completely. This is because these configurations classify almost each measurement as a positive, which means that our attack cannot build any constraints.

VII. Conclusion and Future Work

Hardware security research on iPhones is notoriously difficult. This paper proposes the first complete infrastructure to enable general-purpose hardware security experiments on the Apple iPhone SoCs. Extending a new public BootROM

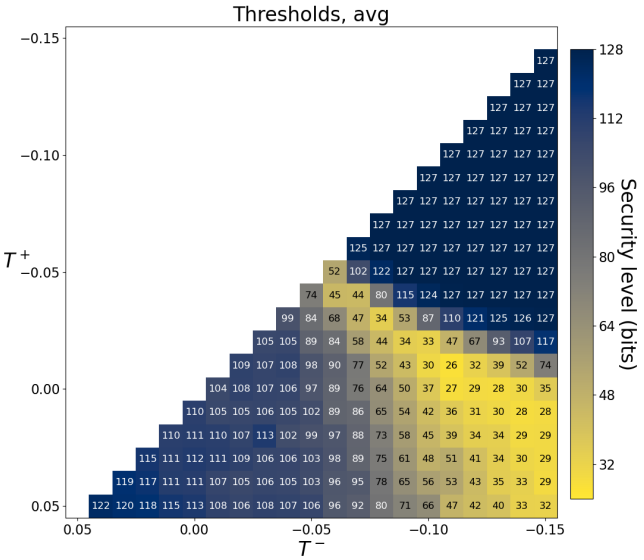

Fig. 7: Average performance of our attack on 2048 plaintexts, averaged from 2048 traces each, with varying values of T^+ and T^-. Note that the attack performs at its worst for low T^+, T^-—this is because we mistakenly classify *every* measurement as a positive in this configuration, and thus our attack can extract no constraints.

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 cache timing 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.

Our results show that the novel constraint-based attack outperforms the classic PRIME+PROBE attack [15] under various hyperparameters and attack configurations. We again note that we only pursue a PRIME+PROBE attack style as it forms a canonical access-driven example. Recent research has exclusively used PRIME+PROBE, which indicates that it is still a primary and relevant access-driven attack style [31] However, we have observed vulnerable cache flush timings in our experiments, paving a way towards other, more recent cache attacks [24], [26].

Based on results, we argue that cross-core attack styles (i.e. through L2) are possible as well. The highly-controllable research platform created by our toolkit can allow reverse-engineering of the A10's memory hierarchy at these higher levels, including details such as replacement policies and any other prefetchers. Then, with knowledge of key details of the A10's memory hierarchy, it may be possible to execute access-driven attack through the iOS software environment on iPhones. We thus emphasize that the purpose of this specific work is to determine what kinds of modifications must be made to a classic PRIME+PROBE attack in order for the attack to succeed.

Finally, we could explore other novel avenues of side-channel research. Side-channel attacks on microarchitectural structures such as branch predictors [36] or TLBs [37], which have mostly been demonstrated on Intel x86 processors, could be possible on mobile Android and iOS devices as well. Attacks based on speculative execution, such as Meltdown [38] and Spectre [39] are possible on Apple SoCs as well[7], but Apple's mitigations for these vulnerabilities have not yet been analyzed in the literature. Finally, implementations of fault attacks such as VoltJockey [13] or ClkScrew [12] could be possible with future work on reverse-engineering Apple's DVFM driver. Rowhammer attacks, which have previously been shown on ARM Android devices [40], could be possible as well.

VIII. Ethical Disclosure

We took the necessary steps for 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 preliminary findings prior to submitting the paper or revealing it on any other public forum.

IX. Acknowledgements

We thank anonymous reviewers for their feedback on the paper. This research is supported in part by the National Science Foundation (NSF) Division of Computer and Network Systems (CNS) under Grant No. 1850373 and Grant No. 1943245.

References

[1] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "Ecdsa key extraction from mobile devices via nonintrusive physical side channels," in *ACM SIGSAC Conference on Computer and Communications Security*, 2016, p. 1626–1638.

[2] O. Lisovets, D. Knichel, T. Moos, and A. Moradi, "Let's take it offline: Boosting brute-force attacks on iphone's user authentication through sca," Cryptology ePrint Archive, Report 2021/460, 2021, https://eprint.iacr.org/2021/460.

[3] A. Inc., "Apple platform security guide," Spring 2020. [Online]. Available: https://manuals.info.apple.com/MANUALS/1000/MA1902/en_US/apple-platform-security-guide.pdf

[4] ——, "Apple open source, https://opensource.apple.com/."

[5] L. Franceschi-Bicchierai, "Key iphone source code gets posted online in 'biggest leak in history'," 2018, motherboard. [Online]. Available: https://www.vice.com/en_us/article/a34g9j/iphone-source-code-iboot-ios-leak

[6] D. Branch, "Debugging ios applications: A guide to debug other developers' apps," 2017, medium. [Online]. Available: https://blog.securityevaluators.com/debugging-ios-applications-a-guide-to-debug-other-developers-apps-c041311498eb

[7] B. Azad, "KTRW: The journey to build a debuggable iPhone," 2019, 36C3. [Online]. Available: https://bazad.github.io/presentations/36C3-2019-KTRW_The_journey_to_build_a_debuggable_iPhone.pdf

[7] https://support.apple.com/en-us/HT208394

[8] L. Franceschi-Bicchierai, "The prototype iphones that hackers use to research apple's most sensitive code," 2019, vice. [Online]. Available: https://www.vice.com/en_us/article/gyakgw/the-prototype-dev-fused-iphones-that-hackers-use-to-research-apple-zero-days

[9] L. Deshotels, C. Carabaş, 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)*, 2020, pp. 399–413.

[10] A. Inc., *Dynamic voltage and frequency management based on active processors*, 2013, no. US9304573B2.

[11] A. Inc, *Security enclave processor for a system on a chip*, 2012, no. US8832465B2.

[12] A. Tang, S. Sethumadhavan, and S. Stolfo, "Clkscrew: Exposing the perils of security-oblivious energy management," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1057–1074.

[13] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "Voltjockey: Breaking sgx by software-controlled voltage-induced hardware faults," in *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2019, pp. 1–6.

[14] T. iPhone Wiki, "Bootrom exploits, https://www.theiphonewiki.com/wiki/Bootrom#Bootrom_Exploits."

[15] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*. Springer, 2006, pp. 1–20.

[16] A. Kovrizhnykh, "Technical analysis of the checkm8 exploit," 2019, digital Security. [Online]. Available: https://habr.com/en/company/dsec/blog/472762/

[17] J. Ho and B. Chester, "The iphone 7 and iphone 7 plus review: Iterating on a flagship," 2016, anandtech. [Online]. Available: https://www.anandtech.com/show/10685/the-iphone-7-and-iphone-7-plus-review/3

[18] J. Daemen and V. Rijmen, "The block cipher rijndael," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 1998, pp. 277–284.

[19] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel." 2002.

[20] D. J. Bernstein, "Cache-timing attacks on aes," 2005. [Online]. Available: http://palms.ee.princeton.edu/system/files/Cache-timing+attacks+on+AES.pdf

[21] O. Acıiçmez and Ç. K. Koç, "Trace-driven cache attacks on aes (short paper)," in *International Conference on Information and Communications Security*. Springer, 2006, pp. 112–121.

[22] C. Rebeiro and D. Mukhopadhyay, "Boosting profiled cache timing attacks with a priori analysis," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 6, pp. 1900–1905, 2012.

[23] R. Spreitzer and B. Gérard, "Towards more practical time-driven cache attacks," in *IFIP International Workshop on Information Security Theory and Practice*. Springer, 2014, pp. 24–39.

[24] Y. Yarom and K. Falkner, "Flush+reload: a high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 719–732.

[25] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 897–912.

[26] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 279–299.

[27] R. Spreitzer and T. Plos, "On the applicability of time-driven cache attacks on mobile devices," in *International Conference on Network and System Security*. Springer, 2013, pp. 656–662.

[28] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 549–564. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp

[29] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.

[30] L. Todesco, "The One Weird Trick SecureROM Hates," 2019, power of Community. [Online]. Available: https://iokit.racing/oneweirdtrick.pdf

[31] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+probe 1, javascript 0: Overcoming browser-based side-channel defenses," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/shusterman

[32] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 300–321.

[33] D. Wang, Z. Qian, N. Abu-Ghazaleh, and S. V. Krishnamurthy, "Papp: Prefetcher-aware prime and probe side-channel attack," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[34] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani, "Prefetch-guard: Leveraging hardware prefetches to defend against cache timing channels," in *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2018, pp. 187–190.

[35] Z. H. Jiang and Y. Fei, "A novel cache bank timing attack," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 139–146.

[36] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers' Track at the RSA Conference*. Springer, 2007, pp. 225–242.

[37] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 955–972.

[38] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.

[39] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[40] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1675–1689.