Let's Take it Offline: Boosting Brute-Force Attacks on iPhone's User Authentication through SCA

Oleksiy Lisovets, David Knichel, Thorben Moos and Amir Moradi

Ruhr University Bochum, Horst Görtz Institute for IT Security, Bochum, Germany,
{firstname.lastname}@rub.de

Abstract. In recent years, smartphones have become an increasingly important storage facility for personal sensitive data ranging from photos and credentials up to financial and medical records like credit cards and person's diseases. Trivially, it is critical to secure this information and only provide access to the genuine and authenticated user. Smartphone vendors have already taken exceptional care to protect user data by the means of various software and hardware security features like code signing, authenticated boot chain, dedicated co-processor and integrated cryptographic engines with hardware fused keys. Despite these obstacles, adversaries have successfully broken through various software protections in the past, leaving only the hardware as the last standing barrier between the attacker and user data.
In this work, we build upon existing software vulnerabilities and break through the final barrier by performing the first publicly reported physical Side-Channel Analysis (SCA) attack on an iPhone in order to extract the hardware-fused device-specific User Identifier (UID) key. This key – once at hand – allows the adversary to perform an offline brute-force attack on the user passcode employing an optimized and scalable implementation of the Key Derivation Function (KDF) on a Graphics Processing Unit (GPU) cluster. Once the passcode is revealed, the adversary has full access to all user data stored on the device and possibly in the cloud.
As the software exploit enables acquisition and processing of hundreds of millions of traces, this work further shows that an attacker being able to query arbitrary many chosen-data encryption/decryption requests is a realistic model, even for compact systems with advanced software protections, and emphasizes the need for assessing resilience against SCA for a very high number of traces.

Keywords: iPhone · SCA · Passcode Recovery

1 Introduction

Over the last decade, smartphones entirely reshaped the way we communicate while drastically increasing the amount of user-related data collected and stored on device or uploaded into the cloud. With the advancement of the Internet of Things and increased interconnection of everyday-life devices, this trend is reinforced even further. Smartphones became a replacement for various smart cards including credit cards; they process health-related data, and serve as an access key in various applications. As a result, compromising the security and data protection mechanisms of smartphones dramatically impacts user privacy. Here, the user authentication plays a key role as its bypass would lead to full control over the device. While biometric-based authentication methods like fingerprint matching or face recognition were introduced over the years, using a (numeric) passcode is still commonly the fall-back option if these methods fail. Next to Samsung and Huawei devices, Apple's iPhone is the most sold smartphone in the world showing a market share of 14% in the first quarter of 2020 [Goa20]. With the introduction of Touch ID in iPhone 5s and later with Face ID for the iPhone X, Apple has started integrating biometric-based user authentication methods in their devices while setting a passcode still remains as a prerequisite to enable Touch ID or Face ID. Whereas a 4-digit numeric passcode was the default option for older system versions, its length has been extended to contain 6 numeric digits with the launch of iOS 9. Nevertheless, iOS always offers an option for manually setting an arbitrary-length (≥ 4) numeric/alphanumeric passcode. We should highlight that it is always possible – and in case of a cold boot even required – to unlock the iPhone using the passcode instead of the configured biometric authentication methods. Therefore, the passcode is still the default fall-back solution for user authentication, enabling an adversary to alternatively target this method in order to maliciously gain access to the device's system and user data.

When the user enters the passcode, it is entangled with a device-specific key called User Identifier (UID). The UID is an encryption key unique for each device and is fused into hardware during manufacturing. The result of running the aforementioned function is then used to decrypt/encrypt user data stored on the device. The entanglement between passcode and the UID key restricts brute-force attempts to be performed on the (same) device. In addition, a single execution of the aforesaid cryptographic function is designed to take about 80 milliseconds, resulting in an expected time of over 22 hours for an on-device brute force search of a 6-digit numeric passcode.

Current methods, like the one presented by Markert et al. in [MBG+20], try to optimize the on-device search of a user's passcode through an ordered list of common PINs, as PIN verification is tied to a specific phone. By utilizing Side-Channel Analysis (SCA) attacks, we show that the coupling of the brute-force search to a specific iPhone can be levered out completely, enabling a significantly faster passcode recovery.

Since the first introduction of SCA attacks as a threat to cryptographic implementations [Koc96, KJJ99], researchers and practitioners have reported successful key-recovery attempts mainly on their-own-designed devices. This picture has changed when they have been shown highly effective by targeting real world devices, even in a blackbox scenario, where no details about the implementation are known to the adversary. For example, in [EKM+08, KKMP09] SCA attacks were used to completely break the remote keyless entry system based on KeeLoq technology employed by several car manufacturers at that time. In [ORP13], by means of SCA, the authors reported the recovery of the Yubikey 2's secret key – a One-Time Passwords (OTPs) token used for two factor authentication – enabling the malicious generation of valid OTPs, even after returning the token to the owner. In 2011, the result of a power analysis attack on the contactless smartcard DESFire MF3ICD40 was reported [OP11], resulting in a complete recovery of its 112-bit key. As this smartcard was employed in several large payment and public transport systems around the world at that time (e.g., Czech railway, Australian myki card, Clippercard in San Francisco) it evidently emphasizes the relevance of SCA attacks in real-world scenarios. Further, in [SRH16], Saab et al. show two ways of recovering AES keys in the context of Intel's AES-NI cryptographic instruction set extension by placing a magnetic field probe in close proximity of two capacitors on a motherboard hosting an Intel Core i7 Ivy Bridge microprocessor.

An example for a successful SCA attack performed on a smartphone is presented in [BFMT16], where Belgarric et al. successfully recovered the key used in the implementation of the ECDSA signature scheme in Android's standard cryptographic library. In their work, they leverage the electromagnetic emanation of the CPU to distinguish between different elliptic curve operations in the context of Weierstrass and Koblitz Curves. Another EM-based SCA attack on a smartphone is reported in [VMC19], where the authors successfully extract the secret AES key of the CPU's hardware coprocessor. The work especially focuses on issues arising when performing such an attack on a modern system and involves desoldering the DRAM placed on top of the main SoC. However, the manufacturer and model of the device under test are not disclosed.

Academic publications dealing particularly with physical attacks on iPhones are hard to find. To the best of our knowledge only a 2016 work by Sergei Skorobogatov [Sko16] describes a real-world implementation attack on an iPhone. In more detail, the author describes how to perform a real-world mirroring attack on an iPhone 5c, enabling to bypass the limit of passcode retry attempts by restoring a previous state of the NAND flash memory. Beyond that, only informal reports of successful hardware exploitation of iPhone devices exist. For instance, an article from *The Intercept*, published in 2015 in course of the Snowden Leaks, mentions that already in 2011, EM-based SCA attacks were performed by the CIA in order to extract the Group Identifier (GID) key from iPhone 4 devices in order to decrypt and analyze the boot firmware for vulnerabilities [SB15]. More details can be found in [int15].

In previous works by Sogeti [sog11b] and Elcomsoft [elc11], the software implementation of iPhone data protection has been reverse engineered. Furthermore, they utilized a BootROM vulnerability to perform on-device passcode cracking by booting a custom Secure Shell (SSH) ramdisk with a patched kernel, allowing them to instruct the Advanced Encryption Standard (AES) engine to use the UID key from userspace.

In this work, we utilize a BootROM vulnerability to deploy a custom payload in bootloader context, which allows us to communicate with the AES engine and perform a successful SCA attack on an iPhone 4 recovering the complete 256-bit UID key fused into the hardware design of an individual device's processor. Despite the fact that no detailed information about the attack procedure is included in the leaked documents [SB15, int15], it is likely that the attacks supposedly performed by the CIA to extract the GID key from an Apple A4 processor, are very similar to our attack presented in this work. The hardware AES engine can be configured to use either of the two keys, GID or UID, so that the attack procedure for extracting them is identical. In our attack, having the UID key in hand, the authentication process becomes decoupled from the device, enabling a significantly faster and scalable brute-force search of the passcode utilizing highly parallelized computation on Graphics Processing Units (GPUs). Since we show that existing software vulnerabilities enable the collection and processing of a high number of measurement traces, our work underlines the practical relevance of assessing the resilience of a device against SCA, even for several hundreds of millions of traces, which is often questioned by the research community.

For the collection of the SCA measurements, the device evidently needs to be physically accessed by the adversary. Afterwards, however, the adversary does not have to be in the possession of the iPhone anymore, as the storage can be dumped and the passcode search is performed completely offline. In order to give an overview on the performance of our attack, we refer to the required time for a trivial on-device brute-force search of an 8-digit numeric passcode which is reduced from 92 days to less than 3 hours by performing the search on 2 NVIDIA RTX 2080 TI GPUs [NVI18].

The rest of the paper is structured as follows. First, we give an overview of the attack scenario and our approach in Section 2, before all necessary background is clarified in Section 3. In Section 4, we describe how we recovered the UID key by performing SCA attacks on an iPhone 4. Once recovered, we use this key to perform a parallel brute-force search of the passcode, as detailed in Section 5. A discussion on the applicability of this attack to newer iPhone series is given in Section 6. Finally, we conclude the paper in Section 7.

2 Attack at a Glance

The attack procedure described in the following is based on the assumption that code execution by means of a software exploit, i.e., a BootROM exploit, is achieved. The adversary then has oracle access to the AES engine realized as a dedicated hardware co-processor on the CPU and can query arbitrary data to be encrypted in a chosen-plaintext SCA attack. For our device under test, the necessary BootROM exploit is publicly available.

2.1 Attack Scenario

First, the adversary maliciously gains physical access to the victim's device. Without any persistent modification of the system, she then boots a custom ramdisk providing SSH access and downloads the System Keybag from device (further described in Section 3.2). Furthermore, she performs an SCA attack in order to extract the device-specific UID key. For user authentication, the UID key is entangled with the user's passcode in order to derive the so-called passcode key which is in turn used to unwrap further keys from the System Keybag. This can then be used to unlock individual files stored encrypted on the filesystem. Using existing vulnerabilities, we show that the attacker can easily extract this file from the device – again without any permanent changes on the system – which, together with the UID key, enables the recovery of the user's passcode by means of an offline brute-force search. After successful recovery of the passcode, the system can be normally booted and the adversary can simply log into the device, allowing complete access to all data, e.g. by running an iTunes backup. As the necessary modifications of the mainboard only consist of removing the metal shielding of the CPU (which can be simply unclipped and clipped back), they can easily be reversed and the phone can still be operated normally after the attack, leaving the victim with no direct chance to recognize any malicious activity. The power analysis attacks which we performed in addition to the EM-based attacks, require the disassembly of inductors and capacitors from the board, but even in that case it is possible to reverse the modifications, although greater care and effort is required. Admittedly, the attack procedure requires not only physical access and minor modifications of the hardware, but also a considerable amount of time to acquire the necessary amount of measurements (in our case, it took three weeks in total to recover the UID). Nonetheless, we can think of highly relevant attack scenarios where an attacker is able to invest the time for this attack and where it is worthwhile to do so. Apart from finding or stealing a device and extracting all the sensitive user data, an attacker could also run a malicious phone shop where she extracts the UID keys in advance before handing out the phone to the customer and might sell them to malicious third parties. In fact the break-even point, where the time spent to perform the physical attack is less than the time saved when doing offline instead of on-device brute-force search is reached for passcodes of 8 digits and more.

In this work we present the aforementioned attack on an iPhone 4; however the same setup (with minor changes) can be used to collect SCA measurements and extract the keys from an iPhone 4s and iPhone 5/5c. We further believe that similar research can be done on devices up to the iPhone X/iPhone 8. In fact, due to a couple of publicly-known vulnerabilities [@ax, Xu] of such newer devices (explained in detail in Section 6), it might be possible to obtain oracle access to the relevant AES engine and collect the SCA measurements necessary for an attack. However, since we have not practically attempted such experiments yet, we cannot make any claims about the success/difficulty of the attacks on newer devices, especially considering that newer iPhones reportedly contain DPA countermeasures.

2.2 High-Level Description

At first, we execute the SHA-1 Image Segment Overflow (SHAtter) exploit to disable signature checks in the initial Secure Boot process, enabling the execution of custom codes. Afterwards, we deploy a payload integrated into the second-stage bootloader, enabling us to query an arbitrary amount of chosen data to the AES hardware engine for encryption and decryption, and to re-configure one of the General Purpose Input/Output (GPIO)-pins (e.g. volume-down button) as output for triggering the SCA measurements. By performing a Correlation Power Analysis (CPA), we exploit correlation between the recorded SCA traces and secret intermediate values of the AES encryption, allowing us to recover the complete 256-bit UID key. We show that being in possession of this key allows an offline recovery of the user passcode by running a parallelized brute-force search on several GPUs. Having recovered the passcode in turn enables a decryption of all user files previously stored on the device and potentially (if the user is signed in to iCloud on the device) data stored on the user's iCloud.

3 Background

3.1 Secure Boot

As described in [App12], the Apple iPhone 4 implements a secure boot mechanism to establish a boot-chain-of-trust which starts at the BootROM and works its way up to the operating system kernel and even further to application level. Here we focus only on the first boot components relevant to our attack.

On power up, the BootROM, which is an immutable piece of software fused into the System on Chip (SoC), is executed. This is the root of the trust chain. The BootROM functionality is kept minimal as it is the most trusted code and a vulnerability in the BootROM cannot be fixed with a software update. Its task is to load, validate and execute the first stage bootloader either from Non-Volatile Random Access Memory (NVRAM) or over Universal Serial Bus (USB). The latter is used in cases where the former fails or the user enters the Device Firmware Upgrade (DFU) mode using a special button combination.

The first stage bootloader runs in the Static Random Access Memory (SRAM) with its main responsibility being the initialization of low-level hardware components such as Dynamic Random Access Memory (DRAM) and to load, validate and execute the next stage of the bootloader. As it is executed in DRAM, the second stage is provided with much more memory capacities. It is responsible for initializing higher-level hardware components (such as the screen) and further for loading – amongst other parts – the boot logo, the devicetree and a ramdisk (in case of a recovery/update boot). Finally, the second stage bootloader loads, validates, initializes and executes the kernel.

Every component, which is loaded before the kernel is executed (including the kernel itself), is shipped as an img3 image, which is cryptographically signed and encrypted with Cipher Block Chaining (CBC)-AES-256. Each img3 image contains an Initialization Vector (IV) and key, which are encrypted using a device model specific hardware fused Key Encryption Key (KEK) also called Group Identifier (GID) key. After the signature of the img3 image is verified, the GID key is used to decrypt the corresponding image specific IV and key, which are then in turn used to decrypt the image itself.

The GID key (as well as the UID key) is never exposed to the Central Processing Unit (CPU) directly. Instead, decryption oracle queries are made to a dedicated hardware AES engine, which uses the GID/UID key internally. More precisely, there is a software-based selection of the key slot the Advanced Encryption Standard (AES) engine uses for encryption.

On newer iPhones (starting from the iPhone 4s), the second stage bootloader even disables selection of the GID as key to the AES engine before executing the kernel (enforced through hardware registers), ruling out any usage by the operating system. Other key slots, such as the one containing the UID key or user supplied keys, can still be selected by the operating system.

3.2 iPhone Data Protection and User Authentication Mechanisms

A detailed overview of the iPhone's Data Protection and User Authentication Mechanisms can be found in [App12]. All necessary concepts for this work are described in the following.

Before written to the flash memory, every file is – per default – encrypted with a 256-bit per-file-unique key utilizing the AES engine running in CBC mode. Note that, disabling encryption of the filesystem by the user is not possible. A per-file key itself is wrapped with a key corresponding to a certain protection class and stored in the file's metadata.

The system contains different file classes with various access rights, shortly explained below.

- Files of the *Complete Protection* class can only be accessed while the iPhone is unlocked. Their in-memory keys are discarded after the device has been locked for 10 seconds.
- The *Protected Unless Open* class allows creating new files while the device is locked, but once the file is closed, it cannot be reopened until the user unlocks his device.
- The *Protected Until First User Authentication* class prevents the files from being accessed on a fresh boot until the user unlocks the iPhone for the first time by the passcode.
- Finally, *No Protection* is the default class for all files not assigned to a specific other class. Note that, even in this case, the corresponding files are stored in encrypted form. This prevents attackers from accessing the files by desoldering and dumping the flash memory.

The class keys in turn are wrapped by a key derived by combining the device-specific UID key and the user passcode (if set, except for the *No Protection* class) and stored in a file referred to as the *System Keybag*.

According to Apple's whitepaper the System Keybag is unlocked, i.e., the class keys are unwrapped, by means of Password-Based Key Derivation Function 2 (PBKDF2) processing the user passcode by a Pseudo Random Function (PRF) being AES making use of the UID key [App12]. The PBKDF2 internally iterates the PRF for a high number of times. The iteration count is chosen to lengthen a single unlock attempt to approximately 80 milliseconds, resulting in the iteration count being set to $50,000$ in case of the iPhone 4.

By default, iOS limits users to perform only a total of 10 attempts (to unlock the iPhone with passcode) with increasing time delays in between, after which – if enabled– the device wipes the system, rendering all data inaccessible. Note that this is a software-induced restriction which can be bypassed by utilizing known BootROM exploits.

On first setup, the user is prompted to choose a 4-digit numeric passcode (or 6-digit starting from iOS 9) as the default user authentication method. Although iOS offers options to use longer numeric or even alphanumeric passcodes, to the best of our knowledge and based on our personal survey asking 3,864 iPhone users, the majority of users just stays with the default option.

3.3 Side-Channel Analysis

Instead of targeting the cryptographic algorithms as in classic cryptanalysis, physical attacks aim at recovering the secrets stored in or processed by the so-called cryptographic device as a particular realization of cryptographic algorithm(s). Side-Channel Analysis (SCA) attacks, as a passive and non-invasive class of physical attacks, have absorbed the lion's share of attention in the scientific community due to their high efficiency. Further, such attacks leave no sign on their back indicating the device has been compromised. SCA attacks exploit dependencies between physical properties of the implementation and the processed secret data. Next to the execution time [Koc96], other exploitable side channels have been introduced over time, including but not limited to power consumption [KJJ99] and Electro-Magnetic (EM) radiations [GMO01]. In short, nowadays it is well known that a cryptographic device would be vulnerable to various SCA attacks unless the implementation is equipped with countermeasures dedicated to each attack vector.

Correlation Power Analysis. In this work, we mainly use CPA, where the measured power/EM traces are correlated to the result of a hypothetical power/EM model over the key-dependent intermediate values of the underlying cryptographic algorithm. This process is conducted in a divide-and-conquer fashion allowing the attacker to recover the secret key in small portions, e.g., byte by byte in case of the AES. As a side note, since Pearson's correlation coefficient estimates the linear dependency of two random variables, the feasibility of a CPA attack depends on the linear dependency (similarity) of the hypothetical power/EM model to the actual leakage of the attacked cryptographic device. In such a case, the correlation associated to the correct key guess should show a distinguishable distance to that of the other key candidates. Apart from collecting low-noise SCA measurements, the difficultly of CPA attacks indeed lies on choosing an appropriate intermediate value and finding a fitting hypothetical power/EM model [MS16].

Leakage Assessment. In this work, we apply the fixed-versus-random t-test [SM15], making use of a statistical test based on the student's t-distribution. In short, two groups of SCA measurements are collected: 1) when the cryptographic device (with a secret key) is supplied by a fixed input (plaintext in case of encryption) and 2) when random input is given to the device (with the same secret key). The first-order fixed-versus-random t-test examines whether these two groups of SCA measurements are distinguishable from each other through their sample mean (average). If so, it is said that very likely there is an attack which can exploit such a distinguishability to recover the secret. Since the result of a t-test is a confidence level (probability) of the aforementioned distinguishability, the higher the t-statistics is, the higher leakage (stronger distinguishability) is predicted.

4 SCA Attacks on iPhone

4.1 iPhone Preparation

For the collection of SCA measurements we applied a set of non-permanent soft- and hardware modifications to the iPhone. These modifications are described in the following.

4.1.1 Hardware Preparation

First, we disassembled the iPhone, removed its mainboard from the case, and disconnected all peripherals in order to gain access to the CPU. Afterwards, we removed the metal shield protecting the CPU which enables placing an EM probe directly at the top of the chip. Next, we built a Universal Asynchronous Receiver Transmitter (UART) connector [Ess] utilizing a PodBreakout [pod] connector and an FT232RL USB-to-Serial Breakout Board [FT2]. This yields a connector with two USB cables, one for communicating with the iPhone and one for UART to be connected to a Personal Computer (PC).

Additionally, we removed the volume buttons from the case and connected wires to the volume-down button for easy access. This is done for the purpose of providing a fast and low latency interface to the CPU as these buttons are directly connected to the SoCs GPIO pins.

Afterwards, we dismounted the battery connector, enabling to operate the mainboard with an external DC Power Supply set to 4.0 V (which is slightly above the normal supply voltage of 3.7 V) draining on average about 100 mA during the measurements. Note that this was an ad-hoc choice which is not expected to influence the results of the attack, as the relevant ICs are powered via further voltage regulators.

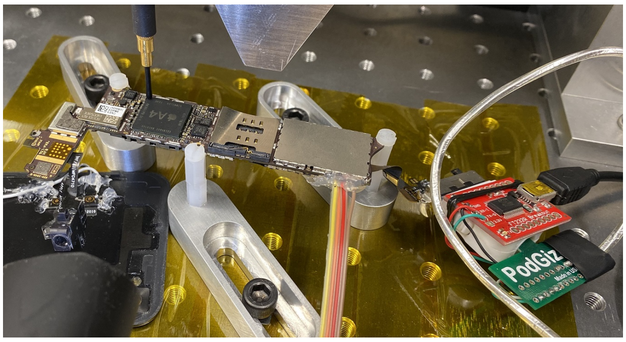


Figure 1: iPhone 4 mainboard mounted on a stage

4.1.2 Payload Execution

As we need arbitrary code execution to perform our measurement, we make use of public exploits. The crucial steps required to enable execution of a custom payload in the second-stage bootloader are described in the following.

As an initial step, the iPhone is forced to boot in DFU mode. Therefore, it has to be first connected to a PC using a USB cable. In order to enter DFU mode, the power button and the home button are pressed simultaneously for 10 seconds. After the power button is released, the home button should still be held pressed for another 15 seconds. The screen stays black when DFU mode is entered successfully.

After the iPhone entered DFU mode, we used the SHAtter exploit to disable signature checks in BootROM. To this end, we utilized *ipwndfu* [@ax], which is a python tool providing BootROM exploits for several iOS devices. Subsequently, we used *irecovery* [lib], an open source tool for communicating with iOS bootloaders over USB, to transfer a patched first- and second-stage bootloader.

The applied patch disables signature checks in the first-stage bootloader allowing us to execute a modified second-stage bootloader. In the second-stage bootloader – which provides a proprietary recovery console – one of the commands (namely *go*) is redirected in order to force the program flow to jump to the *loadaddress* which is the location in memory where data uploaded over USB is stored. As a result, applying this patch allows uploading and executing custom payloads on the device.

Finally, we again used *irecovery* to transfer a custom payload binary and to interact with the second-stage bootloader recovery console. By executing the previously patched *go* command, we ran the uploaded payload.

4.1.3 Measurement Payload

The custom payload executed in the second-stage bootloader enables querying encryption/decryption of chosen plaintexts/ciphertexts by the on-chip AES engine. Furthermore, it enables us to choose which key slot (GID, UID, or custom) is used by the AES engine. In the following, the structure and functionality of the inserted code is described.

First, the GPIO interface for the volume-down button is reconfigured to act as an output port. This allows driving the corresponding exposed pins high/low by utilizing

Memory Mapped Input/Output (MMIO), i.e., by simply writing a value to a specific address in memory.

Next, the bootloader's builtin AES routine, responsible for communicating with the hardware AES engine, is patched to set the GPIO pin on high at the start of the encryption/decryption, and on low right after its termination. This configuration enables tabbing the exposed pin and detecting the time instances associated to the activity of the AES engine. Therefore, we could easily use the tabbed signal to trigger the oscilloscope collecting SCA measurements.

Finally, we replaced a recovery console command to enter a custom *measurement* mode. The measurement mode is a custom function which first disables the bootloader's cooperative scheduler by patching the yield-function to return immediately. Subsequently, it enters an infinite loop which waits to receive a command over UART and executes its functionality accordingly. We developed the custom measurement mode to ease the operations necessary for SCA measurements. Due to the slow communication via UART we configured the measurement mode to limit the UART communications while allowing to collect several SCA measurements. To this end, we followed the concept introduced in [SM15], meaning that the PC sends a few custom commands to the iPhone via UART, the measurement mode configures the AES engine accordingly, generates random input (plaintext/ciphertext) and runs the AES encryption for a certain number of times, and finally sends back a checksum to the PC. During this time the oscilloscope is repeatedly triggered and collects SCA traces. Since the PC and iPhone are synchronized (via the initial commands), the PC calculates the randomly-generated plaintexts/ciphertexts as well and associates them to the traces collected by the oscilloscope. This process greatly accelerates the SCA measurement process.

4.2 The Apple A4

Our target device – the iPhone 4 – uses the Apple A4 SoC, which is also used in the iPod Touch fourth generation, the iPad first-generation and the Apple TV second-generation. The Apple A4 provides a 32-bit ARMv7-A CPU manufactured on Samsung's 45 nm fabrication process [chi10], clocked at 800 MHz (or 1 GHz in case of iPad) with Package on Package (PoP) used to provide 256 MB Random Access Memory (RAM) (or 512 MB for the iPhone 4).

Due to the PoP, the RAM is located on top of the CPU as can be seen in Figure 2. It is indeed impossible to put an EM probe very close to the CPU surface to monitor its direct emanations. Instead, we are only able to put the probe either on top of the packaging or on the Printed Circuit Board (PCB) next to the chip, measuring the emanations associated to the power distribution network around the die.

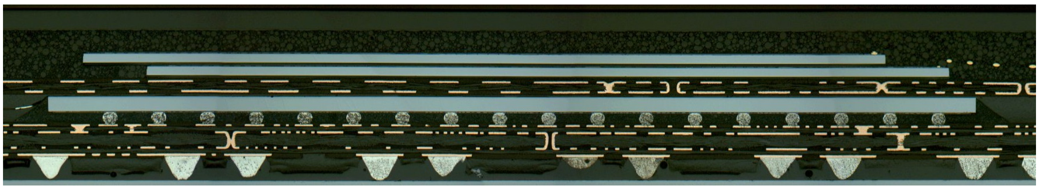


Figure 2: Cross-section of the A4 processor + RAM Package on Package (PoP) (taken from iFixit A4 teardown [ifi10])

4.3 Measurement Setup

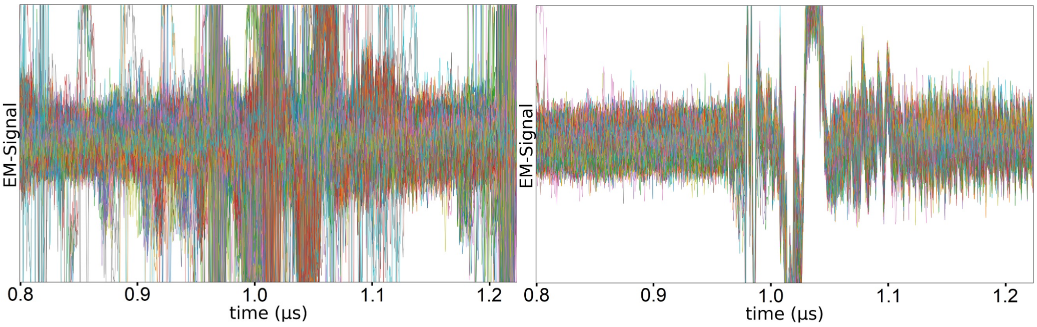
We use a **Langer EMV-Technik RF-B 0,3-3** EM probe, which has a flat head with a diameter of 2 mm allowing to capture frequencies in the range of 30 MHz to 3 GHz. The probe is connected to a **Langer EMV-Technik PA 303 SMA** amplifier (with similar bandwidth), which amplifies the EM signal by 30 dB, before the signal is monitored by the oscilloscope.

We employed a **Teledyne LeCroy WaveRunner 8254M** with 2.5 GHz bandwidth for monitoring the signals and recording the SCA traces. The traces have been sampled at a sampling rate of 40 GS/s, which is the machine's maximum capacity.

4.4 Preprocessing of the Traces

Figure 3a shows a part of the superimposition of 500 collected EM traces, where we detect a heavy misalignment, complicating any straightforward statistical analysis. Although we configured the measurement mode to run on a single-threaded core in bootloader context, it can clearly be seen that the trigger signal, which we provided to indicate the start and end of encryption/decryption is not fully synchronized to the activity of the AES engine. It actually implies that the code running on the CPU core does not work synchronously either with the co-processors or with the IO peripherals, which controls the GPIO pin we used for triggering the oscilloscope.

As a consequence, one part of our preprocessing tries to align the traces. Here, we realized that the misalignment often appears in groups, i.e, multiple traces are shifted by a similar offset. Thus, our main approach is to find those clusters and coarsely align them to one group by shifting them so that the strongest peaks overlap. Afterwards, for a more fine grained alignment, we chose an arbitrary *reference trace* and align the other traces by an appropriate temporal offset so that they match as closely as possible to the reference trace. We used the minimum euclidean distance as the metric to find the best matching offset between two traces.



(a) before preprocessing **(b)** after preprocessing

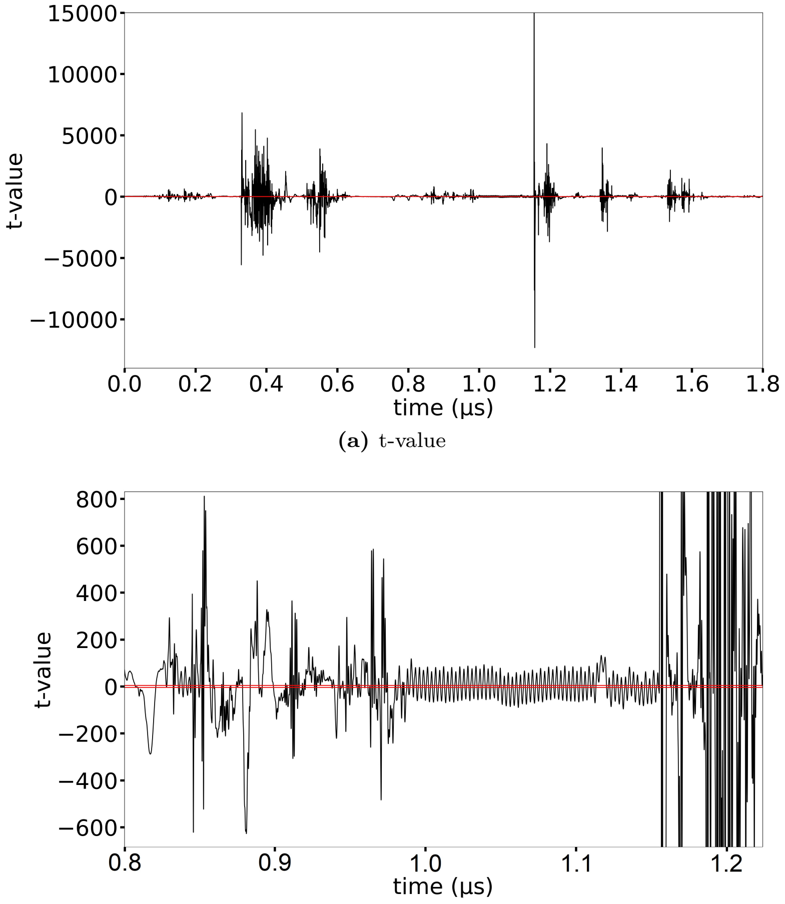
Figure 3: Superimposition of 500 EM traces before and after preprocessing

As after alignment, a significant amount of traces still showed some noisy peaks going out of bounds, we additionally filtered out traces whenever these peaks occurred during the time interval on which we performed our analysis, thereby discarding around 20% of the collected traces. The result after applying our alignment and filtering process on those 500 traces can be seen in Figure 3b.

As a side note, we noticed that the jitter and misalignment is greatly reduced when decrypting multiple blocks consecutively. Hence, in our attacks and analyses we always collected the traces associated to decrypting 8 blocks in CBC mode.

4.5 Leakage Assessment

First, we performed the fixed-versus-random t-test (see Section 3.3) on the aligned traces when the EM probe was placed arbitrarily on the SoC. We collected 10,000,000 traces while the input (ciphertext for the decryption function) was randomly interleaved between a fixed value and a random input.



(a) t-value

(b) t-value (zoomed)

Figure 4: Non-specific t-test

Figure 4a shows the corresponding t-value over the time covering the decryption of 8 blocks of AES in CBC mode. Very large t-values can be seen in several portions of the traces. Most notably, the period between 0.3 μs and 0.6 μs corresponds to the communication between the CPU and the AES engine through the dedicated Input/Output (I/O). The chunks around 1.2 μs, 1.4 μs and 1.6 μs are predicted to be relevant to the similar communication in the reverse direction, i.e., AES engine to the CPU. We assume that the decryptions take place in between 0.8 μs and 1.2 μs, which still show a considerably-high t-value (see Figure 4b). Hence, we concentrated on this period to conduct the attacks.

4.6 Power Model

The iPhone AES engine allows to specify a user key for its operation, thus we search for an appropriate power model in a known-key scenario, i.e., we collected traces for which we know the underlying key. This way we are able to easily examine different hypothetical power models since all cipher's intermediate values are known to us.

Based on the information from Apple's whitepaper describing how the AES engine is used, we made certain assumptions about how the AES hardware might have been designed. We know that the AES engine needs to be fast since during the operation of the iPhone, key derivation as well as encryption/decryption of files are performed quite often. A slow AES engine would lead to serious efficiency penalties. It should also support both encryption and decryption of different variants of AES with 128-, 192- and 256-bit key. Furthermore, due to the way data protection is designed for iPhones, we know that the AES key is changed frequently (each file has a separate key).

Therefore, we assume a round-based implementation, as it is a good trade-off between flexibility, performance and physical area on the SoC. More precisely, our assumption is that the AES engine performs a cipher round in a clock cycle.

Based on these assumptions, we considered different design architectures for the AES decryption, and examined various hypothetical models over different intermediates values. Examples include Hamming Weight (HW) of the cipher state after each round operation (e.g., SubBytes (SB), ShiftRows (SR), AddRoundKey, and MixColumns (MC)) and Hamming Distance (HD) between consecutive cipher states for each aforementioned round operation.

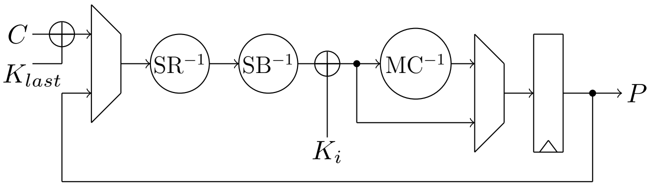


Figure 5: Guessed design architecture for the round-based AES decryption module

Figure 5 depicts the guessed round-based design architecture, for which we could observe high correlation by the aforementioned power models. We determined the best fitting power model as the HD of the consecutive 128-bit values stored in the state register (see Figure 5), if we follow the operations inversely, i.e., from the last round of decryption. The last value stored in the state register is the plaintext P, and the second-last one is $\mathrm{SR} \circ \mathrm{SB}(P \oplus K_1)$ since the MixColumns' inverse (MC^{-1}) is omitted in the last round in this design architecture. Therefore, the HD of the state register in the last decryption round can be written as

$$\mathrm{HW}\big(\mathrm{SR} \circ \mathrm{SB}(P \oplus K_1) \oplus P\big), \tag{1}$$

where the first round key is denoted by K_1. The same model for the second-last round yields

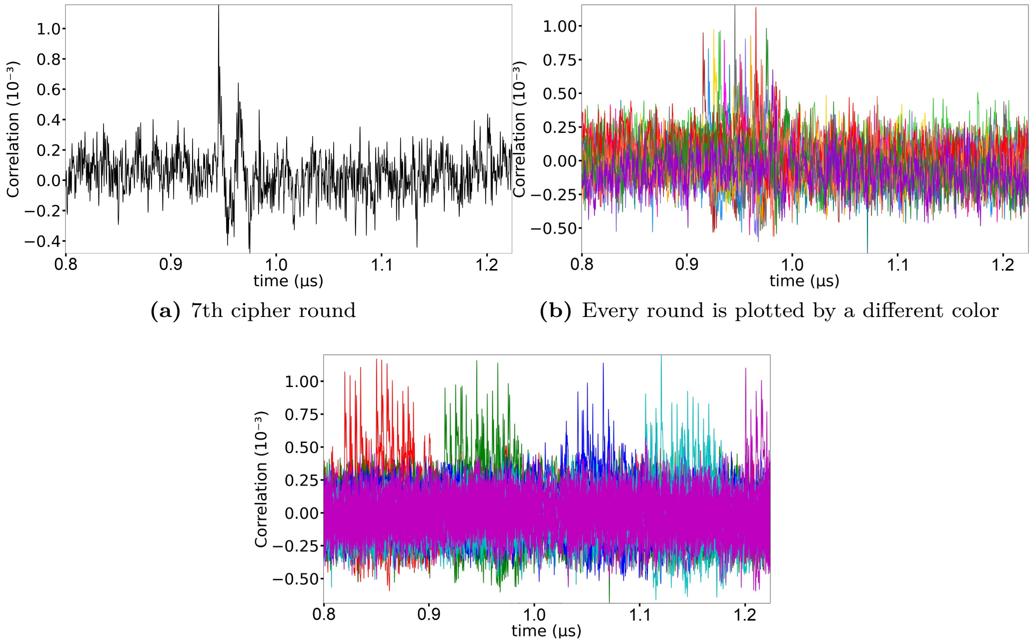
$$\mathrm{HW}\Big(\mathrm{SR} \circ \mathrm{SB}\big(K_2 \oplus \mathrm{MC} \circ \mathrm{SR} \circ \mathrm{SB}(P \oplus K_1)\big) \quad \oplus \quad \mathrm{SR} \circ \mathrm{SB}(P \oplus K_1)\Big), \tag{2}$$

where K_2 denotes the second round key. The same model can similarly be derived for the other cipher rounds. Note that since we focus on the last rounds of decryption, we write the equations over the plaintext and using the encryption operations instead of their inverse.

The result of correlating 80,000,000 measured and aligned EM traces with the aforesaid power model (when the AES engine's inputs are selected randomly) are shown in Figure 6. The cipher rounds can be easily detected as depicted in Figure 6b. We further repeated this procedure for all encryption blocks within the trace. As shown in Figure 6c, we can clearly distinguish the distinct time periods in which an AES operation is performed.

4.6.1 Full-Chip Scan

After we found a promising hypothetical power model, we tried to optimize the probe's position on the SoC. We divided the SoC surface (which is around 7.3 mm square) into a



(a) 7th cipher round

(b) Every round is plotted by a different color

(c) All rounds of a decryption block are plotted by the same color

Figure 6: Multiple correlation traces associated to the selected HD power model, using 80,000,000 traces

grid of 25×25 spots, at each of which we collected 150,000 EM traces. After aligning the traces at each spot individually, we estimated the correlation of the intermediate values using the above-explained HD power model for all cipher rounds and for all decryption blocks. Maximum resulting correlations for each spot are shown by a 3D heatmap on the left-hand side of Figure 7. In case a strong signal stands out at a certain spot, neighboring positions are expected to have a similar correlation value as well. As shown by the 3D heatmap, maximum correlations belong to the cases where the probe is placed at the border of the SoC. This is indeed in line with our expectation with respect to its PoP technology, as explained in Section 4.2. Based on this experiment, we identified the position (x, y)=(4.1 mm, 1.1 mm) as the most suitable spot. We have examined other spots with high correlation as well, but the traces at those positions contained more noise compared to those measured at the selected spot. For the key-recovery attacks, explained in the following, we placed the EM probe at this position, shown on the right-hand side of Figure 7, and collected 500,000,000 traces when the AES engine was supplied by random inputs.

4.6.2 Key-Recovery Attacks

All above given results were based on the HD model over consecutive 128-bit cipher-intermediate values. In order to perform an attack, we need to consider a smaller portion to decrease the attack complexity, i.e., being able to search for a small part of the key e.g., a byte. This is trivially achieved by taking the HD of a byte of the same intermediate value. More precisely, we refer to Equation (1), where by guessing an 8-bit portion of K_1, the HD of the corresponding 8-bit consecutive intermediate value can be estimated. Following this process, we can recover the first 128-bit round key in a byte-by-byte fashion.

Since the underlying AES engine realizes the AES-256 function and the targeted UID

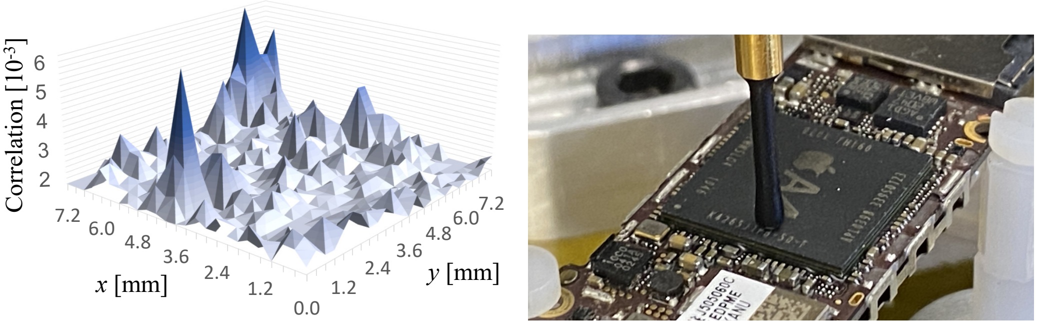


Figure 7: 3D heatmap of the SoC plotting highest correlation in time using the selected HD power model and 150,000 traces at each spot (left) and probe position (right)

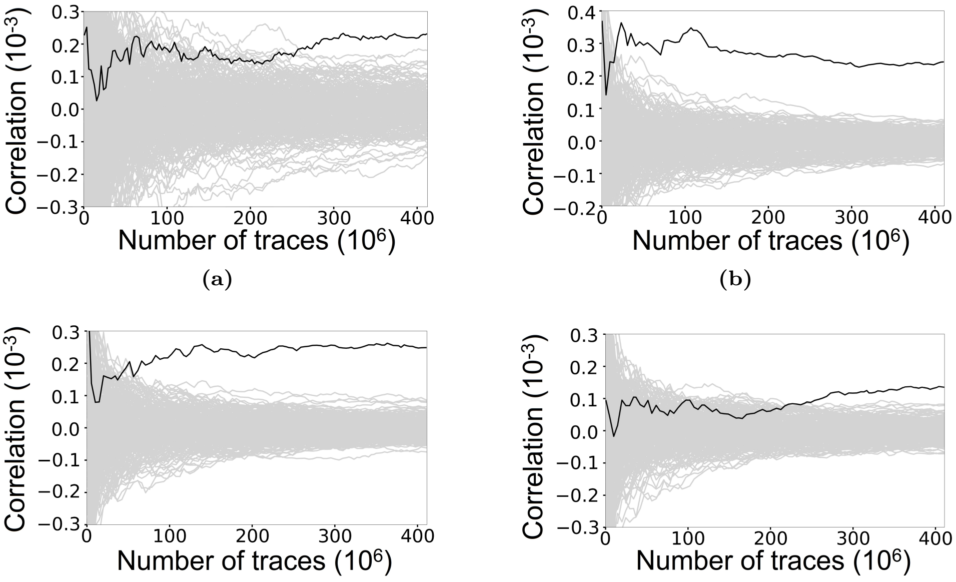
is a 256-bit key, we need to extend the attack to one more round, i.e., the round before the last decryption round. As given in Equation (2), knowing the first round key K_1, HD of the consecutive 8-bit values can be estimated by only guessing an 8-bit portion of the second round key K_2. Therefore, the same attack, i.e., with complexity of 2^8 for each byte of K_2 can be conducted at the second to last round.

Figure 8 shows the result of a couple of attacks targeting different key bytes at either the last or the second last decryption rounds. Notably, we required a large number of aligned traces to reveal the correct key guesses. For some key bytes, we needed around 30,000,000 traces, and in some other cases this number reached 270,000,000. As the AES engine is run in CBC mode, each recorded EM trace contains a series of single AES decryptions; in case a block did not yield a distinguishable correlation, a different one was considered. This helped us to limit the number of required traces since not all decryption blocks in a noisy trace are affected.

In short, we have conducted three sets of attacks: one when the key was known to us (user supplied key), one to recover the UID key and the last one to reveal the GID key. For each of these cases, which led to successful key recovery, we required not more than 300,000,000 traces. This is much more than what has been reported in literature with respect to the SCA attacks on unprotected cryptographic implementations, e.g., [MS16, EKM+08]. We predict that either the implementation contains a form of Differential Power Analysis (DPA) countermeasure or this high number of required traces is due to the high noise level and low signal amplitude. As stated, the EM probe could not be placed close to the CPU die and we encountered different noise sources in our measurements. Nevertheless, our results (despite a high number of required measurements) in fact confirm that such SCA attacks can still be considered as serious threats even to extremely compact embedded systems fabricated with nano-scale process technologies and running at a high clock frequency[1].

In summary, data acquisition of plaintexts, ciphertexts and traces for 500,000,000 EM measurements took about two weeks in total, while conducting all attacks to fully recover a 256-bit AES key using a machine with 40 CPU cores, 64 GB of memory and 2 NVIDIA RTX 2080 TI took around one additional week. Note that we expect the required time to be shorter when performing the attack multiple times on further devices of the same type due to the experience obtained during each device's analysis and the natural optimization of the measurement procedure. We are also confident that improvements of the measurement setup are possible which may reduce the data and time complexity of this attack.

[1] Assuming a round-based implementation, based on our SCA measurements and correlation peaks identifying consecutive cipher rounds in Figure 6, we conclude that the AES engine is supplied by a clock at a frequency of around 200 MHz.



(a)

(b)

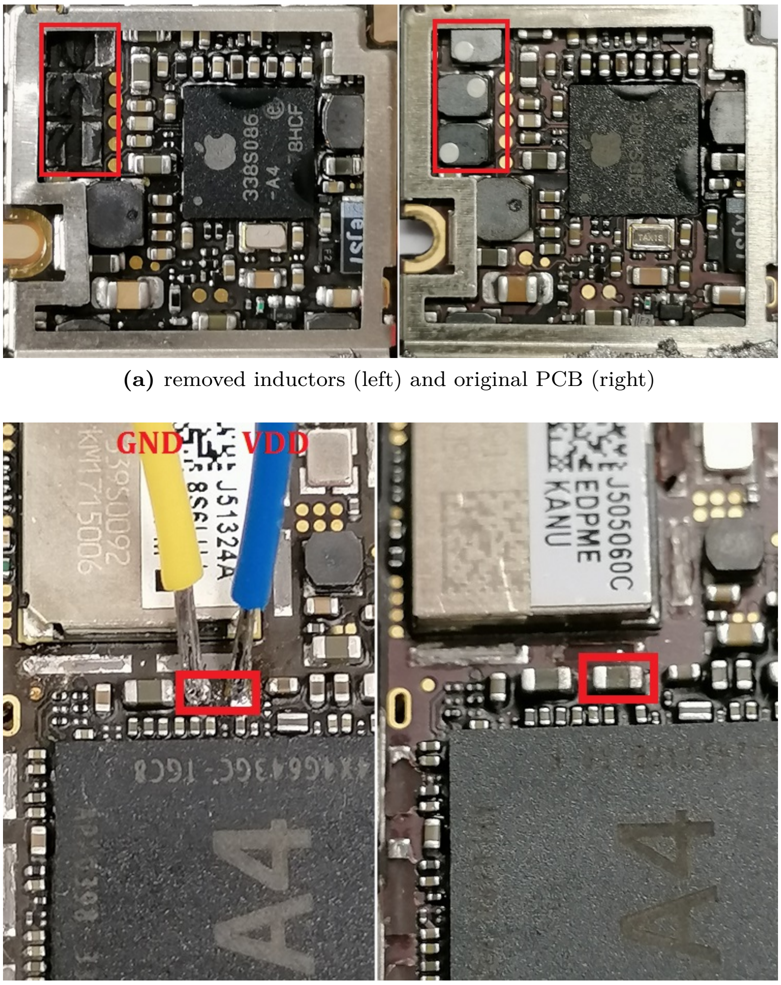
(c)

(d)

Figure 8: Exemplary CPA attack results

4.6.3 Power vs. EM

Given the large number of measurements required to recover the key of the AES engine via EM measurements, it is fair to wonder whether the distance between the probe and the active area of the die was simply too large to capture the radiated information in maximum quality. Since the PoP packaging prevents us from placing the EM probe any closer to the actual AES core, we decided to also investigate the power consumption of the A4 processor. Conceptually, power measurements do not require such a close physical proximity to the computing cells. Therefore, we placed a 1Ω shunt resistor in the `VDD` path of the core power supply of the A4 processor. Of course, the core power supply of the CPU is not directly accessible, e.g., via the battery connector. Instead, it is generated on the PCB via a Buck converter circuit from the main power supply. Measuring the power consumption in front of this converter (i.e., measuring in the main power supply at the converter's input) would yield no successful analysis as any small instantaneous voltage drop on its output side (i.e., in the core power of the chip) gets quickly compensated by the internally stored charges and only the charging cycle of the converter could be seen on its input. Therefore, after we had identified the position of the Buck converter circuit on the PCB, we removed its inductors, effectively cutting the core power supply open, see Figure 9a. Afterwards, we removed one of the larger capacitors on the PCB which we previously identified as a smoothing capacitor for the core voltage. Then, we soldered two cables to the SMD pads of the (now missing) smoothing capacitor and powered the core of the A4 processor through these pads via an external DC Power Supply at 1.35 V, see Figure 9b. Please note that the pictures in Figure 9 show two different devices and not the same board before and after modification. Although we destroyed the inductors of the Buck converter during the removal process, it is generally possible to perform both adaptations of the PCB carefully enough to keep all pieces intact in order to revert the changes later on. We also need to mention here that the iPhone struggled to boot after altering the PCB in the described manner. However, after patching the bootloader to operate in a reduced power mode we did not experience any bootloader crashes and could



(a) removed inductors (left) and original PCB (right)

(b) artificial core power supply (left) and original PCB (right)

Figure 9: PCB adaptations required for measuring the CPU’s core power.

perform the desired measurements. The reduced power mode also leads to decreased operating frequencies and therefore may be beneficial for physical adversaries anyway.

As mentioned before, a 1Ω shunt resistor needs to be placed in the `VDD` path of the power supply. This was realized with an auxiliary board connected between the external power supply and the iPhone, see Figure 10. We employed a DC Blocker (`BLK-89-S+` from Mini-Circuits[2]) to remove the DC shift and an AC amplifier (`ZFL-1000LN+` from Mini-Circuits) to increase the amplitude of the measured signal. Then, we measured the voltage drop across the CPU’s core via a coaxial cable connected to the amplifier’s output. We have recorded the measurements using our digital oscilloscope configured to a bandwidth limit of 1 GHz and a sampling rate of 2.5 GS/s. Similar to the EM measurements, we could not identify any AES-like sequence of power peaks in the traces. Even more problematic was the absence of any communication or IO peaks in the traces which could have been used to achieve the same re-alignment that was previously detailed for the EM analysis. Hence, we had to perform the attacks on the raw traces without any pre-processing. Nevertheless, Figure 11 shows two successful recoveries of different key bytes via CPA using the same power model as for the EM analysis. Here, about

[2] https://www.minicircuits.com

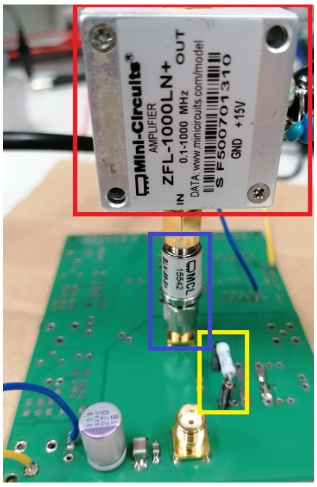


Figure 10: Auxiliary board with AC amplifier (red), DC blocker (blue) and 1Ω shunt resistor (yellow).

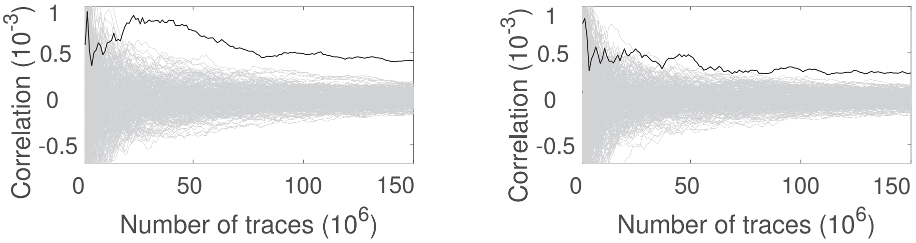


Figure 11: Exemplary CPA attack results exploiting the power consumption.

20 and 75 million traces were required to isolate the correct key candidates from the incorrect ones. However, attacks on other key bytes required up to 200 million traces for an unambiguous recovery. In that regard, we can conclude that the attacks exploiting the power consumption are hardly any more effective than the EM attacks, especially when considering the additional effort to modify the PCB. Yet, both side channels can be used in practice to successfully recover the keys processed by the hardware AES engine.

4.7 Reversing the Modifications

As described in Section 4.1.1, for collecting the EM traces, the case of the iPhone has to be opened in order to expose the battery, the mainboard, and the CPU. The battery is dismounted and the volume buttons are removed from the case. Note that the necessary modifications can be made on a separate set of hardware allowing the target device to remain mostly unmodified. For example, wires can be soldered to a spare volume-button-cable, which is then (in place of the original part) temporarily attached to the mainboard using the available connector. Furthermore, the on-board battery connector does not need to be modified, when instead a modified battery cable is used. In addition, the metal shielding has to be removed. As long as its mount on the board is left in place, this can also simply be clipped back on afterwards. For the power measurement described in Section 4.6.3, some inductors and a large capacitor had to be removed from the PCB. These elements can be soldered back onto the board after performing the attack. As a result, the modifications necessary to perform the attacks are entirely reversible, enabling an attacker to completely hide the malicious key recovery from the victim.

4.8 Possible Countermeasures

In order to protect future devices from attacks similar to the one presented in this work, several suitable options are available ranging from software protections over architecture-level defenses to classical DPA countermeasures. For the newer iPhone series (from 5s onward), Apple itself has already applied an additional architecture-level barrier to protect the most sensitive hardware-fused keys from extraction, namely by introducing the Secure Enclave, a dedicated processor unit, and prohibiting usage of the SEPUID (used for the entanglement with the passcode) from the application processor [App21]. This way, a software exploit on the application processor is not sufficient to get oracle access to the relevant crypto engine and is hence not sufficient to perform a chosen-plaintext SCA attack.

Naturally, possible protection mechanisms of the crypto engine itself include classical DPA countermeasures like *masking* and *hiding* which aim to either split the sensitive information into a number of shares using randomly chosen masks which shifts leakages to higher orders, or reduce the signal-to-noise ratio and make the measured traces hence less informative [MOP07]. According to [App21] and beginning with Apple's A9 processor, the company has also started to integrate DPA countermeasures into their designs, although no further details on the kind of countermeasures that were implemented are given. Further possible countermeasures could include the introduction of tamper protection, making it harder to physically modify the device without the victim's notice [App21].

5 Offline PIN Recovery

Up to this point, we recovered the UID key by performing an SCA attack. We now show that being in possession of this key, the brute-force attempts completely decouples from the device, enabling a highly parallelized search for the user's passcode by a GPU cluster.

5.1 Dumping the System Keybag

As discussed in Section 3.2, the System Keybag contains wrapped class keys protected by the UID key tangled with the user passcode via a key derivation function. For each passcode guess, the key derivation function should be executed to calculate the guessed derived key. Afterwards, the correctness of the guessed derived key is examined by attempting to unlock every class key from the keybag. If every class key could be successfully unlocked, the correct passcode is found.

In order to extract the System Keybag from the device, we used msftguy's *ssh-rd* tool from [msf]. It is a tool written in Java, which automatically downloads the iPhone firmware from Apple's servers; extracts, decrypts and patches bootloaders as well as the kernel, and creates a ramdisk providing an SSH server, which is then booted on the device. As this is similar to one-time booting a live Linux distribution from USB on a computer, no modifications are made to the filesystem. Note that it does not harm if the booted ramdisk uses a possibly older version of iOS than the installed one.

Next, we established an SSH connection to the device in order to download the System Keybag which is stored at /private/var/keybags/systembag.kb on normal boot or /mnt/keybags/systembag.kb on a ramdisk boot (if we mount the user data partition to /mnt).

5.2 Highly Parallelized Passcode Recovery utilizing GPUs

For the purpose of recovering the user's passcode, we implemented a parallelizable program in *OpenCL* whose speed scales with the number of GPUs of a single host, as well as with the number of hosts. Utilizing OpenCL offers independence from the underlying hardware, i.e., our implementation can be executed on several architectures including NVIDIA, AMD, Intel, ARM Mail or any other hardware supporting OpenCL. Our implementation is split into

(1) parsing the System Keybag,

(2) generating the passcode batch,

(3) pre-processing the passcode batch,

(4) deriving the key based on the passcode batch, and

(5) verifying the derived key batch.

Step (1) and (5) are performed on CPU, since parsing the keybag needs to be performed only once and unwrapping class keys – following RFC 3394 – is not computationally intensive and can be stopped once a single key failed to unwrap. Note that, in order to verify the correctness of the derived key batch, we considered unlocking all 10 class keys from the System Keybag.

The other steps are performed on GPU with the batch size being calculated as GPU_workgroup_size × 64 × GPU_factor with the GPU_workgroup_size being 1024 for our NVIDIA RTX 2080 TI GPUs and the GPU_factor being manually set to 64 in order to balance performance and processing time per batch loaded onto the GPU. For our setup, this results in a batch containing $2^{22} = 4,194,304$ passcode guesses which takes about 14 minutes to be processed. It is possible to further decrease the GPU_factor to process a smaller number of passcode guesses, i.e., when only searching for a 4-digit passcode in a single batch. However, these values are determined as optimal when searching for a number of passcode guesses higher than those fitting into a single batch.

Step (2) generates a batch of passwords to be stored in memory. In our program, we limited the generation of passcodes to numeric ones, while its extension to cover alphanumeric passcodes is straightforward. However, we would like to note that finding alphanumeric passcodes becomes more efficient when combined with dictionary attacks or other password-search techniques as described in [AHW18], which are out of the scope of this work.

Next, step (3) performs PBKDF2 to compress the arbitrarily length string (i.e., passcode) to a sequence of 32 bytes. Unlike described in [App12], the key derivation function is not implemented as PBKDF2 with AES as the PRF. It rather consists of a single iteration of the regular PBKDF2 algorithm with SHA1-HMAC as PRF followed by Apple's custom Key Derivation Function (KDF).

Step (4) performs Apple's custom KDF which consists of the PBKDF2-derived 32-byte user key, the UID key, an iteration count, and an internal 32-byte state (as reverse engineered by Sogeti in ramdisk_tools/AppleKeyStore_kdf.c in [sog11a]). First, the internal state is initialized with the user key. Afterwards, the following is executed in a loop: The IV (which is initially set to zero) is salted with the current iteration count and XORed to the user key, which is then encrypted utilizing AES-128 in CBC mode keyed by the UID key. The output is then XORed to the state and the last block (of two blocks in total) is used as the new IV. This is repeated as many times as specified by the iteration count, before the state is returned as output (50,000 times in case of our target iPhone 4).

Since this is the computationally most intense operation, we implemented a bitsliced version of AES which processes 64 blocks in parallel. This is even further parallelized on multiple GPU threads, thus computing 64 work groups of size 1024, each of which computes 64 blocks at a time, resulting in a total of 4,194,304 blocks being processed in parallel.

Finally, step (5) is performed on multiple CPU threads and uses the previously computed keys to unlock all class keys from the System Keybag. Only if all class keys are successfully unlocked, the corresponding passcode is considered to be the correct one. As most of the time the unlocking of the first class key from the System Keybag fails, this step usually terminates rapidly.

5.3 Results

Table 1 shows a comparison between the required time to find numeric passcodes in the worst-case scenario for different passcode lengths when performing the brute-force search on device and on a GPU cluster. Given the ability to execute arbitrary code, short numeric passcodes can be reasonably searched on-device. Assuming the verification of a single passcode guess takes 80 ms as described in [App12], finding 6-digit passcodes requires 22 hours (in worst-case) if verification is performed on device. Our practical experiments with Sogeti's iphone-dataprotection toolkit from [sog11a] on an iPhone 4 show that verifying 2,000 passcodes takes 6 minutes (180 ms per attempt). This means that the actual numbers given in Table 1 for the on-device search are higher by a factor of 2.25. According to Sogeti, the time to crack a passcode may vary depending on the device [sog11b]. It should be noted that Elcomsoft reported 73.5 ms per attempt on an iPhone 5 which is slightly faster than the stated 80 ms. Our work decouples passcode search from hardware limitations of the target device and enables performing the corresponding search programs on arbitrary platforms. Using a single NVIDIA RTX 2080 TI, we already achieved a speedup by a factor of 380, making it possible to search for a 10-digit passcode in a reasonable time by simply utilizing a 2-year-old gaming setup. Employing a GPU cluster with 8 instances of NVIDIA RTX 2080 TI, accelerates the search even further. With this advanced setup, searching for an 11-digit passcode would take around 30 days in the worst-case scenario while renting these resources would cost around 2000 EUR. As this scales linearly, the worst-case time to find longer passcodes using more GPU instances can be easily estimated. Overall, taking into account the time required to perform the SCA attack in order to extract the UID key (see Section 4.6.2), our attack outperforms the trivial on-device search approach if the target numeric passcode is at least 8 digit long.

Table 1: Worst-case passcode search time

digits	iPhone	RTX 2080 TI	8×RTX 2080 TI
4	13 minutes	2 seconds	< 1 second
6	22 hours	3 minutes	26 seconds
7	9 days	35 minutes	4 minutes
8	92 days	5 hours	43 minutes
9	925 days	58 hours	7 hours
10	25 years	24 days	3 days
11	253 years	243 days	30 days
12	2536 years	2439 days	304 days

6 Applicability to Newer iPhone Series

Generally, this attack is applicable to newer iPhone series as well. Without major changes the same procedure can be performed on iPhone 4s and iPhone 5/5c, when using checkm8 [@ax] instead of SHAtter.

Starting with the iPhone 5s, the iPhone features a Secure Enclave coprocessor (SEP), which is responsible for data protection. Instead of using the Application Processor (AP) UID key, the SEP has a separate UID key (SEPUID), which is entangled with the user passcode for file encryption. Thus, to perform the same analysis, it is required not only to get code execution on the AP, but also on the SEP.

For the iPhone 5s, iPhone 6, iPhone 6s, iPhone 7, iPhone 8 and iPhone X (and all other devices with the same CPU), vulnerabilities in both, the AP BootROM and SEP BootROM are publicly known (namely checkm8 and blackbird [Xu]) providing a way to execute arbitrary code with the highest possible privileges on the AP and the SEP. Therefore, it is generally possible to create payloads with similar capabilities as described in Section 4.1.3 to be executed on the SEP rather than the AP. The corresponding exploits for such vulnerabilities are not publicly available for all listed iPhone generations yet. However, at least for the iPhone 7 there exists a publicly available tool called checkra1n [aea], which exploits the known vulnerabilities to gain code execution on the SEP. We would like to mention that we have already initiated follow-up research to examine if a similar SCA attack is possible on iPhone 7 and later. In our initial attempts, we were able to gain code execution on the SEP using checkra1n and to query the SEP's AES engine to encrypt chosen data. Although this allows collecting SCA measurements from newer hardware, it is hard to predict whether extracting the UID/SEPUID key using the SCA measurements would be similar to (or harder/easier than) that on the iPhone 4, especially since Apple claims to have introduced DPA countermeasures to protect the hardware AES engines starting from the A9 processor generation, i.e., from the iPhone 6*s* series onward [App21].

7 Conclusions

Utilizing public software exploits for known vulnerabilities in order to enable oracle access to the AES engine, we extracted both hardware fused 256-bit AES keys, namely the UID and the GID of an iPhone 4 through Side-Channel Analysis (SCA) attacks processing the Electro-Magnetic emanation and power consumption of the AES engine embedded on the underlying A4 processor. Independent of the implications of our attacks, to the best of our knowledge, no successful SCA attack on an iPhone has been reported in academic literature so far. We, for the first time, presented the success of a corresponding key recovery process despite a compact System on Chip with PoP packaging. Although we need a large amount of traces to recover the complete AES keys, we showed that, if the software barriers are overcome, i.e., there are exploits available to execute custom code on the device, the model of a physical attacker being able to query arbitrary many chosen-plaintext encryptions (or chosen-ciphertext decryptions) is absolutely realistic, emphasizing the need for sufficient protection against SCA – even for hundreds of millions of traces.

Having the UID key in hand, it becomes possible to conduct offline brute-force attacks recovering the user's passcode. Using a highly parallelized GPU implementation, we established a scalable method to highly increase the performance of the passcode search procedure. As the performance linearly scales with the number of utilized GPUs, the search time can be shortened depending on the value of the data versus the budget. We showed that a 10-digit numeric passcode can be revealed in a reasonable time employing a common gaming setup, while a large-scale adversary might even be able to cover 12-digit numeric passcodes.

A possible remedy would be to choose stronger passcodes (either numeric or alphanumeric); however that may result in user inconvenience, as the passcode needs to be entered once in a while despite biometric authentication. We stress that this vulnerability cannot be mitigated on affected devices via a software update, since the leakage originates from the integrated AES engine. In short, our attack emphasizes the need for implementing sophisticated countermeasures against physical attacks, particularly against SCA attacks, for mobile devices, even when the device is a highly compact embedded system and the adversary's knowledge is restricted to a blackbox model.

New software exploits may allow the adversary to interact with the AES engine on the newer iPhone generations. This enables SCA measurements to be collected from such devices and key-recovery attacks to be examined. Trivially, investigating the applicability of such attacks is among our future works.

Acknowledgments

We initiated a communication with Apple in a responsible disclosure about our findings on 5 Oct 2020. We would like to thank Apple for their support and kind communication during this process. The work described in this paper has been supported in part by the German Research Foundation (DFG) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[aea] @argp et al. checkra1n exploit. Link [Online; accessed on 16-March-2021].

[AHW18] Sudhir Aggarwal, Shiva Houshmand, and Matt Weir. New Technologies in Password Cracking Techniques. In *Cyber Security: Power and Technology*, pages 179–198. Springer, 2018.

[App12] Apple. *iOS Security Guide*, May 2012.

[App21] Apple. *iOS Security Guide*, February 2021.

[@ax] @axi0mX. Github: ipwndfu. Link [Online; accessed on 18-September-2020].

[BFMT16] Pierre Belgarric, Pierre-Alain Fouque, Gilles Macario-Rat, and Mehdi Tibouchi. Side-Channel Analysis of Weierstrass and Koblitz Curve ECDSA on Android Smartphones. In *CT-RSA 2016*, volume 9610 of *Lecture Notes in Computer Science*, pages 236–252. Springer, 2016.

[chi10] Chipworks Confirms Apple A4 iPad chip is fabbed by Samsung in their 45-nm process, 2010. Link [Online; accessed on 22-September-2020].

[EKM$^+$08] Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T. Manzuri Shalmani. On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoqCode Hopping Scheme. In *CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 2008.

[elc11] Slides: Evolution of iOS Data Protection and iPhone Forensics: from iPhone OS to iOS 5, August 2011. Link [Online; accessed on 22-September-2020].

[Ess] Stefan Esser. iPhone UART cable. Link [Online; accessed on 30-October-2020].

[FT2] FT232RL USB to Serial Breakout Board - robotshop. Link [Online; accessed on 30-October-2020].

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic Analysis: Concrete Results. In *CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2001.

[Goa20] Laurence Goasduff. Gartner Says Global Smartphone Sales Declined 20% in First Quarter of 2020 Due to COVID-19 Impact, 2020. Link [Online; accessed on 18-September-2020].

[ifi10] Apple A4 teardown, 2010. Link [Online; accessed on 22-September-2020].

[int15] Differential Power Analysis on the Apple A4 Processor. *The Tntercept*, 2015. Link [Online; accessed on 23-March-2021].

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[KKMP09] Markus Kasper, Timo Kasper, Amir Moradi, and Christof Paar. Breaking KeeLoq in a Flash: On Extracting Keys at Lightning Speed. In *AFRICACRYPT 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 403–420. Springer, 2009.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[lib] libimobiledevice. GitHub: libirecovery. Link [Online; accessed on 18-September-2020].

[MBG$^+$20] Philipp Markert, Daniel V. Bailey, Maximilian Golla, Markus Dürmuth, and Adam J. Aviv. This PIN Can Be Easily Guessed: Analyzing the Security of Smartphone Unlock PINs. In *Symposium on Security and Privacy, SP 2020*, pages 286–303. IEEE, 2020.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[MS16] Amir Moradi and Tobias Schneider. Improved Side-Channel Analysis Attacks on Xilinx Bitstream Encryption of 5, 6, and 7 Series. In *COSADE 2016*, volume 9689 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 2016.

[msf] msftguy. GitHub: ssh-rd. Link [Online; accessed on 18-September-2020].

[NVI18] NVIDIA. *Geforce RTX 2080 Ti*, 2018. Link [Online, access 21-March-2021].

[OP11] David Oswald and Christof Paar. Breaking Mifare DESFire MF3ICD40: Power Analysis and Templates in the Real World. In *CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2011.

[ORP13] David Oswald, Bastian Richter, and Christof Paar. Side-Channel Attacks on the Yubikey 2 One-Time Password Generator. In *RAID 2013*, volume 8145 of *Lecture Notes in Computer Science*, pages 204–222. Springer, 2013.

[pod] PodBreakout - sparkfun. Link [Online; accessed on 30-October-2020].

[SB15] Jeremy Scahill and Josh Begley. The CIA Campaign to Steal Apple's Secrets. *The Intercept*, 2015. Link [Online; accessed on 21-March-2021].

[Sko16] Sergei Skorobogatov. The bumpy road towards iPhone 5c NAND mirroring. *CoRR*, abs/1609.04327, 2016.

[SM15] Tobias Schneider and Amir Moradi. Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations. In *CHES 2015*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513. Springer, 2015.

[sog11a] iPhone dataprotection toolkit, 2011. Link [Online; accessed on 22-September-2020].

[sog11b] Slides: iPhone data protection - HackInTheBox Amsterdam, 2011. Link [Online; accessed on 22-September-2020].

[SRH16] Sami Saab, Pankaj Rohatgi, and Craig Hampel. Side-Channel Protections for Cryptographic Instruction Set Extensions. *IACR Cryptol. ePrint Arch.*, 2016:700, 2016.

[VMC19] Aurélien Vasselle, Philippe Maurine, and Maxime Cozzi. Breaking Mobile Firmware Encryption through Near-Field Side-Channel Analysis. In *ASHES@CCS 2019*, pages 23–32. ACM, 2019.

[Xu] Hao Xu. Attack Secure Boot of SEP. Mobile Security Conference (MOSEC) 2020. Link [Online; accessed on 28-September-2020].