The Decline and Dawn of Two-Factor Authentication on Smart Phones

Tomáš Rosa

tomas.rosa@rb.cz

Raiffeisenbank, a.s.
Hvězdova 1716/2b, 140 78 Praha 4

Abstract

In this paper, we focus on two-factor authentication methods employing smart phones. Regarding this platform, it is well-known there are several risks that shall be evaluated carefully when designing such applications. It actually turns out this situation probably signalizes an emerging fall of the two-factor authentication as we know it, for instance, from certain contemporary banking applications. Smart phones, on the other hand, provide not only new threats and vulnerabilities. They also promise to deliver an excellent mix of computational power, rich peripheral devices, and amazing applications right into the client’s palm. After having successfully mastered this part of mobile devices evolution, we can hope to see a dawn of the two-factor authentication as we need it.

1 Introduction

When it comes to an authentication of a subject (e.g. a bank client) or even a message originated at this subject (e.g. payment order issued), there are three well-known factors that the verifier (e.g. the bank) can use to assure itself about the subject identity. These factors are: what the subject *knows* (password, PIN, etc.), what the subject *has* (smart card, smart phone, etc.), and what the subject *is* (fingerprint, retina scan, etc.) [32].

By carefully combining several authentication factors together, we can get two- or even three-factor schemes. It is widely believed that the more factors we use the better assurance of the subject identity we have. In banking area, two-factor authentication schemes became de facto standard. Such schemes usually combine first and second factor from the aforementioned list. That means clients are authenticated by what they know and what they have. In practice, the first factor is usually of a form of PIN or password that the client types, for instance, into a web-based internet banking application. The second factor is usually of a form of mobile phone that is known to be able to receive short messages directed to a particular mobile phone number. During authentication procedure, the bank generates a one-time password (OTP) and sends it via Short Messages Service (SMS) to that particular cell phone number. If the client is able to retype this OTP into the web application, the second authentication factor is regarded as successfully verified (i.e. the client has the mobile phone).

Security of the aforementioned scenario of two-factor authentication involving the mobile phone and the web-based application relies on several assumptions. One of them requires it is practically hard for an attacker to compromise the operating environment of both the particular phone and the web browser where the client part of the serving application runs. In nowadays, however, we are witnessing two important changes, both of them being connected with the increasing computational power of mobile devices. We see a fast convergence of palm computers and mobile phones, the result of which is usually referred to as a smart phone. Smart phones are becoming fully capable of running standard web-based applications like mobile banking or even company application intranets. Therefore, instead of two independent operating environments, the attacker needs to compromise only one – the smart phone. Furthermore, several research studies hand in hand with practical observations show clearly that such a compromise is quite feasible for a reasonably skilled and motivated attacker [9], [12], [14], [17], [24], [38], [43]. From a purely theoretic viewpoint, we are still having a two-factor authentication scheme. Its security is, however, considerably weaken, since that cornerstone independency assumption does no longer apply. Apparently, we shall expect a fall of the two-factor authentication as we know it in the very near future.

Smart phones, on the other hand, do not provide only new vulnerabilities and threats. They also promise to deliver an excellent mix of computational power, peripheral devices, and applications right into the client hand. After studying this phenomenon for a while, we can conclude there are still ways on how to provide two-factor authentication with an adequate level of security. All we need is to be prepared to withdraw the current design approach and to devise a brand new paradigm almost from the scratch. After having successfully mastered this evolution, we can hope to see a dawn of the two-factor authentication as we need it. Of course, it is not an easy way, since there is a huge amount of several aspects to care about. Even worse, we can see a need for various supporting devices that are yet to be developed. On the other hand, we believe this is a doable adventure and we hope this article provides certain light and encouraging guidance on its whole beginning.

In the following text, we use both the terms user and client interchangeably to refer to a subject which identity or message is being authenticated. We define basic model of three threats that are connected with smart phones. For each threat, we then present a typical effective way on how the risk can be mitigated. We do not strive to improve the smart phone platform itself here. We basically accept the particular threat can occur and we search for possible ways on how to mitigate the risk. To do that we mainly employ techniques of distributed implicit PIN verification, redundancy-less encryption schemes, intensive explicit sensitive data wiping, and finally we also touch approaching technologies such as TrustZone, TEE, CAP/DPA, and NFC-based authentication tokens bearing an independent display and simple keyboard. We examine this topic from the sole viewpoint of the code that is running on the particular smart phone device, since we believe that this is the part that deserves the great attention now. To explain the countermeasures, we use easy-to-follow examples illustrating what can happen when something in our design goes wrong. Our aim is to explain the main principles while omitting unimportant technical details for the sake of readability.

For the sake of completeness, we note we are searching for a two-factor authentication based on what the client *knows* and what the client *has*. To minimize our assumption on how far a general client is able to memorize random strings, we will assume the first factor is a decimal PIN of at least 4 digits. The second factor is then – naturally – the particular hardware device itself represented by a device secret key. Regarding the particular mobile platforms, we are focused on the two most popular and promising operating systems, i.e. Google Android and Apple iOS. These platforms are used just for examples, since the main results presented bellow are general. Furthermore, our examples will be also related to using two-factor authentication in the banking area.

The rest of this paper is organized as follows: In Part 2, we introduce a simple threat model of three threats that are connected with smart phones. In Parts 3, 4, and 5, we examine an example approach on how we can design a plausible two-factor authentication when facing the threats defined before. Part 6 presents an overview of expected technology evolution that can provide important security tools in a near feature. Finally, we conclude in Part 7.

2 Smart Phone Threat Model

The key part of understanding all those risks imposed by the smart phone platform is devising a proper model of emerging threats. We present a model that we believe is descriptive enough while keeping things still very easy to understand even for people outside the security area. This is important to be able to discuss this topic with e.g. our business-oriented colleagues.

Definition: *Let the After-Theft Attack (ATA) be any attacking scenario that assumes the attacker has unlimited physical access to the user's smart phone.*

The name of this approach comes from the usual situation which it can occur in, that is after a physical device robbery. It is well-known paradigm in information security saying the particular piece of HW stolen is usually not the most precious thing the victim has lost in such a case. If the smart phone was running, for instance, incorrectly designed mobile banking application, the client should be afraid of loosing much more money than just the price of a new mobile device. Our task is to mitigate the risk, so, ideally, the price of the mobile hardware is the total loss in such a case.

Definition: *Let the File-Skimming Attack (FSA) be any attacking scenario that assumes the attacker is able to get a full copy of the data file container operated by the authentication application at some time.*

Definition: *Let the On-the-Fly Attack (OFA) be any attacking scenario that assumes the attacker is able to launch their privileged code running on the user's smart phone transparently during the time the legitimate user performs the authentication procedure. The term privileged means that the code attains at least the same rights as the authentication process including the ability to establish and use a data connection with the attacker.*

To play a bit with the model given above, we show a simple reasoning about the mutual relations among ATA, FSA, and OFA. The primary distinction in between ATA and FSA is that in the later scenario we do not prescribe the unlimited physical access to the smart phone. FSA threat can occur, for instance, when the attacker steals a weak-protected backup copy of the user's phone data. ATA usually leads to stronger assumptions, since the application container is often just one out of many data items the attacker can get in ATA. Even if there is a HW-assisted encryption of the mobile device filesystem, the after-theft attack assumption is usually at least as strong as the file-skimming attack model [43]. For the purpose of our simple model, we will, therefore, consider that by defeating ATA we shall have automatically defeated FSA as well.

When evaluating the ATA risk, it is important to realize that, unfortunately, there are a lot of well-described and precisely designed forensics techniques that share the same basic objective – to quickly get as much as possible data out of a mobile phone of an unknown holder [21], [22], [43]. These techniques also include HW probing methods that allow to physically connect to the non-volatile Flash memory of the device and to dump its whole content [6]. By carefully profiling JTAG probes connection [7] together with the scanning software for the particular smart phone device processor (based on several sample test devices), the attacker can, sometimes, hope to be able to physically dump even the content of the volatile RAM before its data los due to a power off. RAM memory samples may also be easily accessible for so-called rooted (Android) [22] or jailbroken (iOS) [21], [27], [43] devices. This is another unfortunate synergy between a relatively honest initiative (to bring more unrestricted apps to the mobile) and criminals (to quickly rip-off sensitive user data) [14]. On the other hand, it should be noted that those weaknesses exploited for e.g. iOS jailbreaking [17], [27], are already out there and can be exploited independently on whether there is any jailbreaking initiative or not.

Looking from other perspective, there already are forensics tools and approaches that work even on those devices with encrypted Flash memory content [4], [43]. Apparently, the after-theft attack is really a powerful threat. Similarly to the situation with e.g. payment cards abuse, we can expect attacker groups will specialize on performing various kinds of ATA as soon as there are enough promising potential targets (e.g. mobile banking applications) in widespread use. Therefore, the risk of ATA shall be evaluated really carefully during the application design.

Furthermore, it is easy to see that the ability to mount on-the-fly attack is a stronger assumption than file-skimming attack. In other words, if an attacker is able to succeed with FSA approach then they would succeed with OFA as well. To do so the attacker would simply reduce OFA scenario to FSA and then follow the former approach. In terms of logic implication, the ability to succeed with FSA implies the ability to succeed with OFA. Therefore, simple logic reasoning says that if we have (somehow) designed a successful countermeasure against OFA, we must have defeated FSA as well. For this reasoning, it is important to note that we are not trying to prevent the particular threat to happen, since that would require improving the smart phone platform itself. We actually accept it may happen and we strive to mitigate the risk.

We have seen defeating ATA or OFA also guarantees FSA mitigation. The relationship in between ATA and OFA is, however, not such simple. It may be tempting to regard OFA as a stronger assumption than ATA, but this would be dangerously misleading. The distinction is that the physical access to the smart phone can dissolve certain countermeasures that worked well under OFA scenario. Several parts of the attacked environment that were by principle inaccessible under the OFA scenario may then suddenly become accessible under ATA leading to a successful attack scenario [43]. The OFA can, on the other hand, benefit by capturing certain on-the-fly data elements like client’s PIN, secret key derivatives, etc. which may be by principle inaccessible in a device that is no longer operated by its legitimate user.

Therefore, we shall always investigate both ATA and OFA independently during the application design. If we are aiming for the really highest security, we shall consider even a deliberate combination of ATA and OFA, denoted as ATA $\cup$ OFA. In the worst case scenario, attackers could manage to spread a malware infection across a large amount of mobile handsets. This code would behave according to OFA scenario capturing all interesting data in the on-the-fly style. Later on, when such mobile phone gets stolen, it (with a high probability) contains some data that were already carefully collected under OFA. The attacker would then apply ATA to finally gain these data and exploit them for an attack.

Of course, such a reasoning will not bring us a “magic” countermeasure in itself. It can, however, be rather helpful when engineering our security measures to quickly verify their synergy effects. For instance, if we know we have already solved OFA resistance by e.g. using TrustZone processor mode (cf. Part 6), then we do no longer need to investigate FSA separately – it must have been already solved. On the other hand, we also know that, despite the effort of employing such sophisticated technique, we still have to verify ATA resistance independently, since there is no that simple synergy effect.

3 FSA Risk Mitigation

Let us recall two important observations presented in Part 2: We automatically defeat FSA by previously mitigating ATA or OFA. Furthermore, ATA is such important threat that it has to be defeated anyway. Even if we already solved FSA or OFA resistance, we still need to verify ATA resistance independently. Basing on these observations, we will not discuss FSA countermeasures separately. We encourage the reader to directly consult the countermeasures against ATA as the minimum requirements instead. We can reasonable believe this is the right way to approach two-factor authentication design for smart phones.

4 ATA Risk Mitigation

4.1 What to Avoid

Recall we are searching for a two-factor authentication scheme, where first factor is client's PIN and the second factor is a device master key that represents the "identity" of the device itself. The PIN as well as the device master key must be set up during a personalization phase. Since there is an overwhelming amount of possible personalization scenarios, each of them being more or less correct, we omit discussion of this procedure here. It should be obvious for any reasonably skilled developer on how to do that. Instead, we focus on certain properties of the authentication mechanism itself, which were proven to be no-obvious or neglected even by expectably skilled architects [24].

We start by a simple example of a typical naïve implementation that can be, unfortunately, really found in practice. Let us assume there is a bank using RSA-based authentication scheme with a public key certificate of the client registered at the bank while the private key is stored on the mobile device encrypted by the PIN. Let us further assume that the programmer strived to provide the maximum protection for the private key, so the encryption scheme involves not only client PIN but also several system-level keys. For an attacker working under ATA, it follows, however, from [43], [44] we must assume the attacker can directly defeat any system-level encryption keys. Therefore, the only unknown for the ATA attacker remains the client PIN.

The natural question in this situation is: Can we successfully use a brute force to discover the PIN? The answer is obvious: Yes, of course! We may hear several arguments why brute force is allegedly impossible here, so let us review them. Certain developers argue that there is a PIN try counter in the mobile application. But such an obstacle would be only relevant if the attacker was using the user interface of the mobile application to deduce the PIN. This is, however, non-sense, since we must assume the ATA attacker can have a direct access to the PIN-encrypted key store data. Therefore, the brute force procedure goes totally around the mobile application user interface, so any PIN try counter is clearly ineffective.

The only one question is how to confirm whether our particular PIN guess (starting by 0000 and counting) is correct or not. Actually, there are two possibilities. Since there is a local PIN try counter, it may happen there is some kind of PIN fingerprint that is used by the application to decide whether the value entered by the user is valid or not. In the second case, there would be a combination of the local PIN try counter with a distributed implicit PIN verification described bellow, so there is no direct local template for the PIN verification. We can, however, still find enough *latent* PIN fingerprints for the brute force (cf. the following discussion).

More educated developer may argue that there is a PIN try counter, but this time it is at the bank side. It would work like this: The mobile application uses whatever PIN the user types in to decrypt the private key. Then, whatever value of the private key is obtained, it is used in the authentication protocol to try to authenticate to the bank. If the authentication succeeds, the PIN is assumed to be valid. Otherwise, it is assumed to be incorrect and the PIN try counter is decreased at the bank side. As we will see, this is the whole idea of the distributed implicit PIN verification which – in itself – is correct and highly desirable. Unfortunately, the implementation of this idea is totally wrong here due to the cryptographic mechanism chosen!

Even if we do not store any other fingerprint of the PIN, the RSA private key cryptogram in the key store itself is enough to mount the brute force attack locally. The principal idea to understand here is a *plaintext redundancy*. Recall, in this case, the plaintext of the PIN-based encryption is the RSA private key itself. Looking into [37], it is easy to see there is a huge amount of redundancy in the private key encoding. We can see the ASN.1 description on the following list:

```
RSAPrivateKey ::= SEQUENCE {
     version Version,
     modulus INTEGER, -- n
     publicExponent INTEGER, -- e
     privateExponent INTEGER, -- d
     prime1 INTEGER, -- p
     prime2 INTEGER, -- q
     exponent1 INTEGER, -- d mod (p-1)
```

```
        exponent2 INTEGER, -- d mod (q-1)
        coefficient INTEGER, -- (inverse of q) mod p
        otherPrimeInfos OtherPrimeInfos OPTIONAL
    }
```

To confirm whether the particular PIN guess is correct or not, it may actually suffice to verify those ASN.1 formatting rules are fulfilled (under particular encoding scheme used – BER, DER, etc.) [11]. It means the resulting plaintext for the correct PIN guess shall begin by `SEQUENCE` containing all those particular elementary items.

Even if the programmer decided to omit those ASN.1 formatting tags, or if we want yet-better checking mechanism, we can use the sole algebraic properties of the RSA key itself. If we verify e.g. that for the decrypted private keys it holds:

p and q are primes of expected lengths,

$coefficient*q \bmod p = 1$,

then we can be practically certain the PIN guess leading to such decryption is correct.

We have shown that even if we strip the RSA down to the bone, we cannot resist local brute force attack on the PIN under the after-theft (or even file-skimming) attack assumption. The problem here is the *plaintext redundancy* which guarantees the possibility to distinguish correct and incorrect PIN guesses [32]. Such a redundancy must be avoided and it is quite hard to that for RSA. Some techniques are known (e.g. to store a random seed and re-generate the whole RSA key on-the-fly) but they lead to a painfully inefficient code at the mobile device.

4.2 Distributed Implicit PIN Verification

Let us now summarize the requirements for the PIN verification scheme that are necessary to defeat after-theft attack:

1. Distributed implicit PIN verification: At the mobile device, the authentication scheme computes certain authentication OTP (one-time password) for each and every PIN value entered. The bank side is then queried to decide whether the OTP is accepted or not. The PIN value is regarded as the correct one only if the OTP is accepted by the bank. There is a PIN try counter at the bank (!) side that prevents brute force attacks on PIN.
2. On the mobile device, there are no local direct or indirect PIN fingerprints that might allow successful local brute force attack on the PIN.

In this way, we get a distributed implicit PIN verification with no local fingerprints, which is the key to successfully defeat ATA style crime. Let us again emphasize it must be verified really carefully whether the second requirement is fulfilled. If we need to use the PIN to encrypt certain key, for instance, for HOTP [23] or TOTP [42], it must be ensured there is no redundancy in the key (which plays the role of plaintext here) that could be exploited as a latent PIN fingerprint. In the previous part, we have already seen that RSA keys are highly inconvenient for this purpose. We shall, however, pay attention also to e.g. 3DES key parity bits, etc. [32]. Furthermore, it is important to realize that even the OTP value itself (even the already used one) conveys enough information to successfully mount the PIN-guess attack. Therefore, these values shall be carefully wiped out of the mobile device memory as soon as possible. Especially, we shall not store OTP values in non-volatile memory!

Furthermore, we shall not place the PIN used for the two-factor authentication discussed here to be the same as the device-PIN that may be used to unlock the mobile device screen. It follows from [4], [43] that this particular value can be successfully brute-forced under ATA assumption. This vulnerability is rooted directly in the particular operating system design and cannot be improved by our application. The independent device password lock can be, on the other hand, used to protect the second factor, i.e. the device master key. If our clients do not mind managing two secrets (i.e. the authentication PIN and the device-unlock password) it can improve the overall security significantly.

4.3 DoS Prevention – Partial OTP Verification

The requirements given in the previous part are necessary to prevent ATA. They are, however, not sufficient to prevent both ATA and denial-of-service (DoS) attacks. We have already seen that there must be a PIN try counter at the bank side that limits PIN-guess attack strategy. Furthermore, this limitation must be quite aggressive, provided we have only 4-digit long PIN value offering 10000 possible values.

Unfortunately, trivial implementation of the PIN try counter mechanism opens the door for DoS. By pretending PIN-guess scenario, the attacker could easily block client's access to their bank account. To prevent both ATA and DoS, we need to be able to distinguish the following situations:

1. the OTP was generated by somebody who already has access to the original device master key, but the PIN value was incorrect,
2. the OTP was generated by somebody who does not have access to the device master key.

In case one, we assume this is a PIN guessing attack, so we apply the PIN try counter together with appropriate login blocking/delaying policy. In the second case, however, we assume this is probably DoS attack and we do not block the account. We shall, however, slow down our response delay a bit anyway. This delay is to be able to estimate the upper abound of DoS success rate later on. Note this delay could be in principle set independently on a delay used when the client makes a typo in PIN, since we really can distinguish these two cases (incorrect device key vs. correct device key with wrong PIN). On the other hand, keeping this delay reasonably small and the same for both cases simplifies the implementation considerably. Anyway, for our illustrative estimates, we assume there is at least two-second delay after a failing OTP verification. Furthermore, we shall also ensure the authentication query cannot be restarted until the particular reply delay has already elapsed.

From the mathematical viewpoint, it is reasonable to view the OTP code as a communication channel sample that in fact conveys two verification codes. In particular:

$$OTP = DVC \otimes PVC,$$

where *DVC* is a device verification code which purpose is to verify the code was computed using the correct device master key. *PVC* stands for a PIN verification code which purpose is to verify the PIN value itself. The operator ⊗ denotes any reasonable way of embedding those two codes into one codeword. In the simplest scenario, it may be a straightforward concatenation of decimal strings. At the bank side, we first verify the *DVC* part while the *PVC* is verified (and PIN try counter is applied) only if the former *DVC* check succeeded.

In the aforementioned OTP scheme, there are transparent and easy-to-understand rules for choosing *DVC* and *PVC* lengths. The first one must limit DoS attack strategy. If we assume there is at least two seconds delay after an incorrect *DVC*, then *DVC* of six digits (and longer) can provide adequate protection. Of course, we shall still monitor our clients' activity to be able to detect sustained login attempts.

The size of *PVC*, on the other hand, shall protect the account against an attacker that has already managed getting the device master key (e.g. by ATA approach). So, *PVC* shall be at least 4 digits long. The limiting factor is the overall OTP length which should conform to the particular usability criteria – e.g. 12 digits long OTP seems to be near the upper acceptance limit. The situation depends, of course, on whether we are designing a stand-alone authentication token where clients are supposed to also retype their OTPs into e.g. internet banking, use it in voice conversations, etc., or whether the OTP is used internally and automatically by the mobile application only. In the second case, the upper limit is considerably higher.

The construction and justification of particular OTP generation algorithm is beyond the scope of this overviewing paper. To give at least a rough idea, we may consider using two HOTP [23] codes. The first one (*DVC*) would be based on the device master key only, while the second one (*PVC*) would also reflect the client's PIN. The integrity check provided by *DVC* shall also cover the *PVC*, therefore guarantying certain kind of whole OTP integrity. The operator ⊗ would be then a simple concatenation. Note that the PIN dependency of the *PVC* part can also be indirect in such a way that the second master key used for this particular HOTP computation can be directly protected under PIN-based encryption. Of course, following our discussion above, we must ensure there is no usable plaintext redundancy in this key then.

4.4 Ephemeral Data Wiping Issues

We have emphasized we shall not store any direct or indirect PIN fingerprints to prevent brute force attacks on PIN under after-theft attack scenario. Recall this also includes eventual fingerprints left in volatile RAM, since we shall admit that by using JTAG probes, by exploiting the behaviour of rooted or jailbroken phones, or even by abusing a vulnerability of a honest application installed to e.g. monitor peripheral interfaces accessible to the attacker (cf., for instance, Apple External Accessory framework [26]), the ATA-style attacker can obtain RAM snapshots even for passcode-protected devices [22], [43]. Furthermore, we have already warned these potential fingerprints also embrace OTP values generated by using the PIN, no matter whether such OTP was already used or whether it remains yet-unused by the client.

It follows that:

1. OTPs shall not be stored in non-volatile memory,
2. OTPs shall be wiped out of RAM as soon as possible.

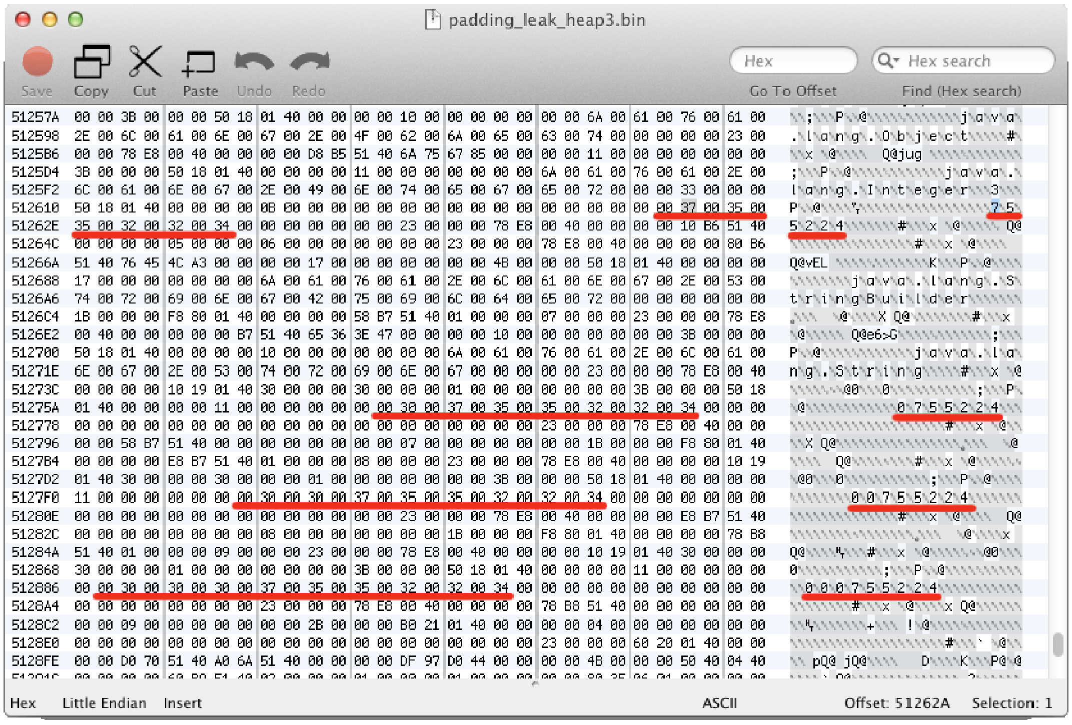
Although this may seem as trivial requirements to obey, the author has already met several practical examples showing the reality may actually be quite different. As an example, let us consider the reference implementation of the HOTP scheme listed in RFC 4226 [23]. The same also applies to the TOTP scheme according RFC 6238 [42]. To pad the resulted OTP by zeros from the left, the following fragment of Java code is employed:

```
result = Integer.toString(otp);
while (result.length() < digits) {
     result = "0" + result;
}
return result;
```

The `result` is holding a reference to `java.lang.String`. The function of this piece of code is obvious and simple: After having converted the OTP from integral value into a string, it appends padding zeros (one-by-one) from left until the resulting code has at least the pre-specified amount of digits. The good point of the whole implementation is that it can be easily ported to Android and we may reasonably assume some programmers will already do that. What is, however, a bit worse is that the code is seriously flawed!

To fully understand where the issue is, it would require a deeper excurse into Java bytecode and eventually also Dalvik virtual machine executable. Long story short, the first pitfall is the unfortunate choice of using `String` to hold the resulting OTP. In Java, every `String` instance is by-standard immutable object, so to wipe such variable out of memory requires at least invocation of Java reflection API which may not always provide a desired result. Secondly, the concatenation command silently allocates two extra objects that remain hidden behind those simple "plus" and "equals" operators. The first allocation creates `java.lang.StringBuffer` to perform the concatenation. The second allocation creates `String` object to hold the new string `result` while overwriting the reference pointing to the `result` of the previous iteration. Therefore, with each zero digit appended, there is at least one copy of the sensitive part of the OTP left in volatile RAM in such a way we do not have any reference pointing to them, so we cannot even try to wipe this data out! We omit the implementation-dependent elaboration on how many OTP copies are exactly created for the sake of simplicity. Anyway, for the attacker, even one copy is far enough.

It is an easy exercise for even beginner-level hacker to show that these data are already available from memory dumps. In Figure 1, we can see a memory dump from a sample application running on Google Nexus S I9023XXKF1 with Android version 2.3.6 build GRK39F. Those sensitive OTP copies are clearly marked – it is the "755224" value according to the test vectors listed in [23]. We have artificially shortened the original OTP integer during the computation to expose the padding function in action. It should be emphasized that the particular Dalvik memory snap was done several hours after the user has already "closed" this demo application. It was observed that if there is enough resources, the Android usually keeps the hosting Linux process of each "closed" application loaded for quite a long time (in order of days). This should probably improve start-up timing when the application is re-started. From the user perspective, however, the application is simply done. This shows another important motivation to do sensitive data wiping properly, since the client may not be fully aware of the after-theft attack risk. Note that similar risk applies for iOS as well. Actually, Apple declares clearly that: "*The system keeps suspended apps in memory for as long as possible, removing them only when the amount of free memory gets low.*" [31]. Again, the reason is to improve application launch time significantly. Unfortunately, attacker's chance to gain sensitive data from a stolen phone is improved significantly as well.



```
51257A  00 00 3B 00  00 00 50 18  01 40 00 00  00 00 10 00  00 00 00 00  00 00 6A 00  61 00 76 00  61 00   ..;...P..@............j.a.v.a.
512598  2E 00 6C 00  61 00 6E 00  67 00 2E 00  4F 00 62 00  6A 00 65 00  63 00 74 00  00 00 00 00  23 00   ..l.a.n.g...O.b.j.e.c.t.....#.
5125B6  00 00 78 E8  00 40 00 00  00 00 D8 B5  51 40 6A 75  67 85 00 00  00 00 11 00  00 00 00 00  00 00   ..x..@......Q@jug.............
5125D4  3B 00 00 00  50 18 01 40  00 00 00 00  11 00 00 00  00 00 00 00  6A 00 61 00  76 00 61 00  2E 00   ;...P..@............j.a.v.a...
5125F2  6C 00 61 00  6E 00 67 00  2E 00 49 00  6E 00 74 00  65 00 67 00  65 00 72 00  00 00 33 00  00 00   l.a.n.g...I.n.t.e.g.e.r...3...
512610  50 18 01 40  00 00 00 00  0B 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 37 00  35 00   P..@......................7.5.
51262E  35 00 32 00  32 00 34 00  00 00 00 00  00 00 23 00  00 00 78 E8  00 40 00 00  00 00 10 B6  51 40   5.2.2.4.......#...x..@......Q@
51264C  00 00 00 00  05 00 00 00  06 00 00 00  00 00 00 00  23 00 00 00  78 E8 00 40  00 00 00 00  80 B6   ................#...x..@......
51266A  51 40 76 45  4C A3 00 00  00 00 17 00  00 00 00 00  00 00 4B 00  00 00 50 18  01 40 00 00  00 00   Q@vEL.............K...P..@....
512688  17 00 00 00  00 00 00 00  6A 00 61 00  76 00 61 00  2E 00 6C 00  61 00 6E 00  67 00 2E 00  53 00   ........j.a.v.a...l.a.n.g...S.
5126A6  74 00 72 00  69 00 6E 00  67 00 42 00  75 00 69 00  6C 00 64 00  65 00 72 00  00 00 00 00  00 00   t.r.i.n.g.B.u.i.l.d.e.r.......
5126C4  1B 00 00 00  F8 80 01 40  00 00 00 00  58 B7 51 40  01 00 00 00  07 00 00 00  23 00 00 00  78 E8   .......@....X.Q@........#...x.
5126E2  00 40 00 00  00 00 00 B7  51 40 65 36  3E 47 00 00  00 00 10 00  00 00 00 00  00 00 3B 00  00 00   .@......Q@e6>G............;...
512700  50 18 01 40  00 00 00 00  10 00 00 00  00 00 00 00  6A 00 61 00  76 00 61 00  2E 00 6C 00  61 00   P..@............j.a.v.a...l.a.
51271E  6E 00 67 00  2E 00 53 00  74 00 72 00  69 00 6E 00  67 00 00 00  00 00 23 00  00 00 78 E8  00 40   n.g...S.t.r.i.n.g.....#...x..@
51273C  00 00 00 00  10 19 01 40  30 00 00 00  30 00 00 00  01 00 00 00  00 00 00 00  3B 00 00 00  50 18   .......@0...0...........;...P.
51275A  01 40 00 00  00 00 11 00  00 00 00 00  00 00 30 00  37 00 35 00  35 00 32 00  32 00 34 00  00 00   .@............0.7.5.5.2.2.4...
512778  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  23 00 00 00  78 E8 00 40  00 00   ....................#...x..@..
512796  00 00 58 B7  51 40 00 00  00 00 00 00  00 00 07 00  00 00 00 00  00 00 1B 00  00 00 F8 80  01 40   ..X.Q@.......................@
5127B4  00 00 00 00  E8 B7 51 40  01 00 00 00  08 00 00 00  23 00 00 00  78 E8 00 40  00 00 00 00  10 19   ......Q@........#...x..@......
5127D2  01 40 30 00  00 00 30 00  00 00 01 00  00 00 00 00  00 00 3B 00  00 00 50 18  01 40 00 00  00 00   .@0...0...........;...P..@....
5127F0  11 00 00 00  00 00 00 00  30 00 30 00  37 00 35 00  35 00 32 00  32 00 34 00  00 00 00 00  00 00   ........0.0.7.5.5.2.2.4.......
51280E  00 00 00 00  00 00 00 00  00 00 00 00  00 00 23 00  00 00 78 E8  00 40 00 00  00 00 E8 B7  51 40   ..............#...x..@......Q@
51282C  00 00 00 00  00 00 00 00  08 00 00 00  00 00 00 00  1B 00 00 00  F8 80 01 40  00 00 00 00  78 B8   .......................@....x.
51284A  51 40 01 00  00 00 09 00  00 00 23 00  00 00 78 E8  00 40 00 00  00 00 10 19  01 40 30 00  00 00   Q@........#...x..@.......@0...
512868  30 00 00 00  01 00 00 00  00 00 00 00  3B 00 00 00  50 18 01 40  00 00 00 00  11 00 00 00  00 00   0...........;...P..@..........
512886  00 00 30 00  30 00 30 00  37 00 35 00  35 00 32 00  32 00 34 00  00 00 00 00  00 00 00 00  00 00   ..0.0.0.7.5.5.2.2.4...........
5128A4  00 00 00 00  00 00 00 00  23 00 00 00  78 E8 00 40  00 00 00 00  78 B8 51 40  00 00 00 00  00 00   ........#...x..@....x.Q@......
5128C2  00 00 09 00  00 00 00 00  00 00 2B 00  00 00 B0 21  01 40 00 00  00 00 04 00  00 00 00 00  00 00   ..........+....!.@............
5128E0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  23 00 00 00  60 20 01 40  00 00   ....................#...` .@..
5128FE  00 00 D0 70  51 40 A0 6A  51 40 00 00  00 00 DF 97  D0 44 00 00  00 00 4B 00  00 00 50 40  04 40   ...pQ@.jQ@.......D....K...P@.@
```

Figure 1: Leakage from the reference implementation of HOTP.

The lesson learned tells us we shall be extremely careful about using existing code fragments, despite being labelled as “reference implementations”. It is important to understand the primary purpose of such reference code is to provide test vectors for the particular cryptographic scheme, not to serve as a robust tamper-resistant design. Especially on Android, we shall consider implementing the most sensitive parts in native binary code [25] to avoid further issues inherently connected with the Garbage-collected environment of Dalvik VM.

5 OFA Risk Mitigation

Let us first further precise the term “privileged code” used in the on-the-fly attack definition in Part 2 above. On Android, for instance, we may consider a Linux process running with the same real and effective user-id as the application under the attack. Since Android security policy is based on application sandboxing by giving them separate user-ids [25], having the same user-id automatically grants the attacking process the rights of the victim application.

Furthermore, since the victim application probably also uses some kind of data connection, we can assume that attaining the user-id of e.g. general banking application is sufficient to mount OFA (against this application) according to the aforementioned definition. What is important to realize here is that, contrary to common believe, we are not calling for having a root access (user-id = 0) to the victim device. Of course, having the root access is sufficient condition to mount OFA on Android, but it is not a necessary assumption.

Regarding Android’s simple sandbox, bearing the same user-id allows the attacking process to e.g. fully trace the victim application. Since we may reasonably assume that a vast majority of such applications will be written in Java compiled for the Dalvik Virtual Machine [25], [35], the attacker can enjoy the comfort of the virtual machine runtime support to inspect the code while looking for interesting ephemeral data variables like PIN, temporary key derivatives, one-time passwords, transaction data, etc. On iOS, the escape from the sandbox can be more adventurous [5]. Anyway, it was already shown several times that it is a doable task. Finally, in the iOS Darwin environment, the attacker can (besides rather classic binary hacks [33]) enjoy at least Android-comparable support provided by the Objective-C runtime layer [3], [34], [43]. In this case, it is not only possible to freely inspect attributes and invoke methods of arbitrary victim application objects – it is even possible to override their behavior at runtime. This makes those popular and strong hacking techniques based on existing code patching almost automatic and trivial. It is, therefore, important to realize that the runtime environments of both Android and iOS (i.e. the two major smart phone operating systems) make the attacker's job in the on-the-fly attack considerably easier. Unfortunately, this is the price paid for the comfort of application programmers. This, on the other hand, is not to say we should harden OFA by lowering programmer's comfort.

Although we promised not to dig deeply into technical details in this paper, we should give at least rough ideas on how OFA can be mounted. The cornerstone principles are code exploits and social engineering. We use the term *code exploit* according to its well-known informal definition, roughly stating that it is any approach aimed at abusing a vulnerability in the victim code to perform certain attacking job (e.g. to execute unauthorized code, escalate the privilege level, escape from a sandbox, etc.). From the aforementioned elaboration of privileges needed for OFA, it follows that these exploits need not to be strictly focused on the operating system kernel, which is often silently assumed. It should be clear it is sufficient to find an exploitable weakness right in the victim application itself to achieve the appropriate privileges and hence to mount the OFA (on the same application). Important conclusion is that programmers of e.g. banking applications shall, therefore, not regard code exploits as being solely problem of the operating system only, since it is their problem, too. This further emphasizes the necessity of proper penetration tests.

On Android, there is an interesting inter-process communication phenomenon based on so-called *intents* [25]. Actually, the whole ecosystem of all application modules on a particular smart phone resembles a distributed system with ad hoc servers and clients. It was shown, it is important to fully understand consequences of this behavior in order to develop secure applications [8], [12]. Basic survey of iOS vulnerabilities including user-land application vectors is given in [27]. In [9] and [36], potential risk of Apple's URL Schemes was discussed, which exposes certain similarities to the Android's *intent* danger noted above. The *intents* are, however, much more general mechanism, so the attacking scenarios are considerably broader.

Regarding the risk of exploits, we have to also note there can be obstacles with patch distribution on both Android and iOS [14] platforms. The situation is such that we cannot ensure that each and every mobile platform that is conforming to the necessary operating system requirements (and hence capable of running our application) is also patched properly. All we can do right now is to hope that the situation will get better as smart phone manufacturers fully recognize the risks connected with their passivity. In a worst case, we would need to programmatically limit running our security-critical applications on platforms and operating system versions that are known to be seriously vulnerable. We shall also devise reliable patch management of our mobile application itself.

Social engineering is connected especially with the Android platform due to the unfortunate decision to leave privilege-granting process on the final user [12], [14], [25]. It well known that users, when it comes to security, often do not do the right settlements. Some of them simply hope the risk management is not their concern here, some ones are too focused on having a new miraculous application, etc. The illusion of "universally better" security connected with the iOS platform can be, however, dangerous as well. It was already demonstrated before, that it is practically feasible to pollute the *App Store* by a malicious application [29]. Furthermore, in [27], there are known iOS vulnerabilities documented such that to catch an infection it was enough to tap on a wrong link in the Safari Mobile browser.

Regarding the power of OFA, it is obvious we cannot fully defeat this threat by some miraculous design of the mobile application itself, since the attacker can see all that our application sees and operate over each and every attribute and method our application implements. We can (and shall) enforce the best software engineering practice to minimize risk of exploits in our own code, but this is the far most point we can go this way. We then have to rely on the security of other applications running on the same smart phone as well as on its operating system integrity and proper management of all this stuff by a generally inexperienced user. While this is perhaps a doable job in a near time window (several years from now), it is probably not an everlasting approach.

To unify our further approach, we use the term *independent token generator* to refer to a cryptographic tool that we suggest to employ to defeat the OFA strategy. We deliberately use this general term as we do not want to suggest any physical appearance of the generator, yet. In Part 6, we show that there are two promising ways on how to practically implement this tool. One of them assumes a classic autonomous hardware device that exists outside the smart phone device. Another one, however, uses so-called TrustZone processor mode to create this generator virtually, on the same physical device that is running the smart phone environment itself. We omit a discussion of possible SIM-based arrangements, because of two reasons. At first, there is a strong relation to a particular GSM operator in a particular country, so these are not unified and general approaches. At second, user-land applications on both Android and iOS are currently unable to exchange direct APDU (according to ISO/IEC 7816) commands with the UICC/SIM card.

Now, we are going to postulate the basic security requirements for the independent token generator. Precise formulation allows us to further investigate certain nuances of the particular implementation later on. It also allows us to show that provably defeating the joint ATA ∪ OFA strategy requires the most rigorous implementation of the token generator – in a form of a stand-alone hardware device (cf. Part 6.3).

The *independent token generator* shall at least:

1. generate one-time passwords (OTP) for a two-factor authentication scheme,
2. securely manage generator master key used for the purpose of req. 1,
3. securely handle client's PIN used for the purpose of req. 1,
4. allow the client to see any user-input data that are used for the OTP computation in a way that cannot be compromised by a smart phone malware (transaction data review),
5. allow the client to approve or deny the OTP computation in a way that cannot be compromised by a smart phone malware.

In the following part, we will investigate selected promising technologies that can be used for such independent token generator implementation. We will also point out those slight nuances linked to the particular way the postulates given above are fulfilled.

6 Promising Technologies

6.1 TrustZone

TrustZone is a security flag ship developed and promoted by ARM Ltd. Let us recall this is the company that sells ARM processor core designs as intellectual properties (commonly called *IP cores*) to particular integrated circuit manufacturers that eventually provide *system on chip* (SoC) designs for smart phones. Practically speaking, our two-factor authentication application for either Android or iOS will eventually find itself running on certain ARM IP core platform. In particular, in the time of writing this article, TrustZone technology was incorporated into all Cortex-A IP core series [28].

The TrustZone principle is based on introducing another control bit called *Non-secure* (NS) that is stored in *Secure Configuration Register* (SCR) managed by the existing, well-known (for system programmers) control coprocessor CP15. Low value of NS indicates the running application belongs to so-called *secure world*, hence it is allowed to access secure parts of the chip. It may be tempting to view this mode as just "protected mode plus" leading to a rather classical hypervisor system design. This would be, however, highly misleading. At first, the secure world is orthogonal to those existing processor modes [2], therefore appearing as a virtual processor core in itself. At second, the NS flag is propagated to the *Advanced eXtensible Interface* (AXI) bus which is the main interconnection network inside SoC designs. Thanks to this design, the secure world is not limited to the particular processor core as it extends towards the system peripherals as well. Therefore, for instance, a cryptographic coprocessor can recognize that it is communicating with a secure world application, ergo allowing a sensitive key operation, etc.

Illustration of cooperation in between the *normal world* and *secure world* code is show in Figure 2 according to [2]. It is assumed the secure world will be using its own simple operating system that manages application service modules called *trustlets*. Our token generator is a clear candidate for such a trustlet. The only part directly exposed to the normal world is the secure monitor that plays a role of gatekeeper between those two worlds.

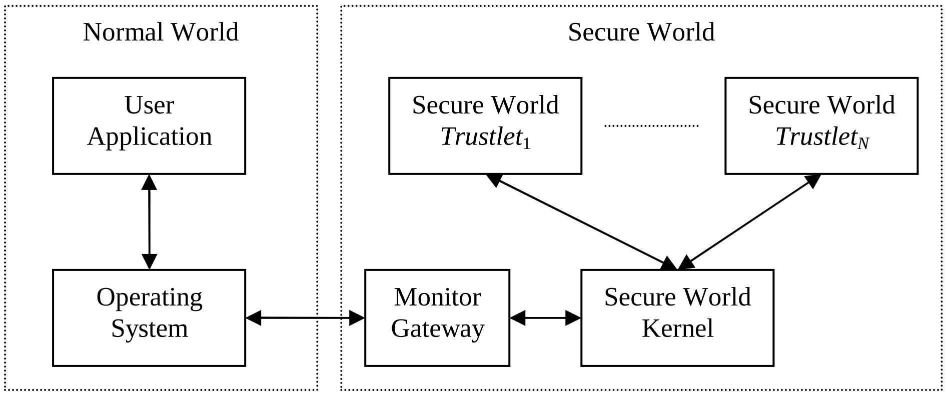


Figure 2: TrustZone Operating Environment.

We should, however, recognize that the monitor is by no means the only one component that must be carefully checked for exploitable vulnerabilities. In security, it is always important to recognize the appropriate model and to follow it during risk analysis. In this particular case, we actually have a situation similar to client-server applications. It is well understood that a vulnerable server application can compromise the whole server environment. Despite all its benefits, the TrustZone in itself does not save us from carefully evaluating every piece of code that is running in the secure world. On the other hand, the secure world clearly promises to be much more rigor and manageable environment – similarly, for instance, to a contrast in between ordinary client workstation and the bank server. There should be a moderate set of trustlets, each and every one undergoing a deep security evaluation before being allowed to run on the secure virtual processor core.

The following table illustrates possible arrangement of a successful one-time password computation. We assume there are some input user data that can be used to e.g. authenticate client transaction details.

Normal World	**Secure World**
i) collect input data	
ii) invoke OTP generator trustlet	
	iii) display input data for verification
	iv) display safety greeting message and ask for the input data confirmation by entering PIN
	v) compute OTP
	vi) return OTP to the normal world
vii) use the OTP	

Table 1: Token generator command flow.

Now, let us discuss security of the arrangement of Table 1 with respect to our threat model. We proceed by verification that this design fulfils our postulates given in Part 5, so it can resist OFA. The first postulate is trivial, so we will start with the second one. Let us assume that the generator master key (whatever it is) is stored in such a way it is inaccessible from any normal world code including (!) its kernel. It is important to realize this key is no longer operated by the normal world part of the mobile application, so it is inaccessible for a possible normal world malware running with the same privileges.

Furthermore, it is reasonable to use the same *distributed implicit PIN verification* scheme for managing the OTP generation and verification as we have already shown for ATA prevention in Part 4. This way, we are aiming to provide not only OFA resistance, but also ATA resistance as well. If we want to use a different cryptographic scheme, we must carefully verify an attacker cannot succeed with ATA approach, since we already have to achieve ATA resistance in any way. From Part 2, we also know that OFA resistance in itself does not automatically imply ATA resistance – so we always have to check twice.

We must also constantly bear on our mind that almost every non-volatile data element becomes accessible when the attacker has a total physical control over the smart phone device. Unfortunately, this seems to be true even for trustlet data. There are certain constructions based on encrypting the secret data elements by using a cryptographic coprocessor that is accessible in the secure world only (recall that the NS flag is observable on the internal AXI bus). This way, however, we are becoming to be closely dependent on the particular SoC design, hence our application and security arguments are no longer general. For certain proprietary applications or limited-variability platforms like iOS devices, this may not be a problem. Further discussion of this approach is, however, beyond the scope of this paper.

We continue with postulate no. 3. One important secure world feature is that it can (and will) load its own keyboard and display drivers, hence eliminating any possible attacker's patches aimed to capture or modify the user input/output data. We shall, however, investigate carefully what this exactly says about security. We can derive the following statements that shall be guaranteed for the secure world environment. Generally, the trustlet can be sure that:

1. its keyboard input is already freshly typed by the user with no attacker's capture or modification,
2. its display output is not captured or modified by an attacker.

We shall realize that guarantee no. 1 in itself does ***not*** say that attacker's code running in the normal world cannot display a fake PIN request dialog to convince the user to enter the PIN right into attacker's hands. Therefore, to prevent the PIN capture threat, we need to carefully blend our algorithm using both aforementioned assurances. Possible arrangement is given in step iv) of Table 1. Let us assume that, in some trustlet configuration menu, the client can set up a personalized secret welcome message that is displayed each time the trustlet needs sensitive input data like PIN. Similarly to the case of the generator master key, let us further assume that this message is stored in such a way it is inaccessible from any normal world code including (!) its kernel. By guarantee no. 2, we are assured that this message cannot be captured when being displayed by the trustlet. Therefore, attacker working under OFA assumption has no chance to learn this secret greeting anyway. So, we can use it as a hint for the user to be able to distinguish a legitimate PIN request from a fake attacker's dialog. Working this way, we have also achieved certain resistance against the joint strategy ATA $\cup$ OFA, as we have hardened stealing the PIN via OFA and gaining it later on via ATA.

Note, however, that there is still certain residual risk of the joint strategy ATA $\cup$ OFA which is rooted in the fact that the OTP value is eventually returned back to the normal world application, cf. step vi) in Table 1. In this environment, it could be captured by attacker's code using the OFA approach. Later on, during the following ATA phase, the attacker would pick up this value and use it to mount a brute force attack on the PIN (cf. discussion in Part 4).

Postulate no. 4 is ensured by guarantee 2 stated above, while postulate no. 5 is achievable due to the assumption that both secret generator master key and the safety PIN challenge message are inaccessible for any normal world code. Therefore, the only way to compute OTP is to invoke the trustlet running in the secure world and this code is assumed to fulfil the security policy given by our postulates.

So far, it seems TrustZone is really interesting concept present in almost any modern smart phone and that we can readily start our own trustlet development. The first part is true, but the second one, unfortunately, is not. In time of finalizing this paper (spring 2012), there was no official public support for writing TrustZone applications in both Android and iOS development tools. Furthermore, there were no public rules set for trustlet evaluation and certification, neither was it clear how this process should look like at all. Basing on informal discussion following [1], this situation should start changing right this year (2012). Hopefully, the dream will eventually come true.

6.2 Trusted Execution Environment (TEE)

This is a standard developed and promoted by the Global Platform Inc. which is well-known subject in the area of smartcard applications. This standard is clearly inspired mainly by the former model of using smartcards as secure elements sitting besides unsecured rich execution environments like PC/Mac main processors. It has, however, achieved certain maturity and it aims to provide a unified application access to various secure environments that are accessible on the particular platform [16]. Detailed discussion of this concept, which is mainly standard-based activity that is still under construction, is beyond our scope here.

We have included this term mainly to show how it compares with the TrustZone technology mentioned above, since sometimes we can see these two notions being confused. The best way to understand this subject is perhaps to regard TrustZone as a physical realization of a particular trusted execution environment covered by the TEE standard. What is, however, important to bear on mind is that TrustZone is an existing technology that is already implemented in almost every reasonable smart phone device, while TEE is just a general standard-oriented activity. On the other hand, the application standard framework is what is currently missing in TrustZone, so we can hopefully see these two initiatives going hand in hand to improve smart phone security. What is worth mentioning with respect to TEE is that it theoretically can extend TrustZone's secure world domain behind the SoC boarder that is even behind the reach of its internal AXI bus [2].

6.3 NFC-coupled Smart Device

We have introduced TrustZone as a robust technology allowing us to solve ATA and OFA resistance without the necessity of using another external HW device. We have also achieved certain resistance against the strongest joint strategy ATA $\cup$ OFA. There was, however, still certain residual risk of successful attack under this assumption. If we want to achieve really robust and defendable resistance against ATA $\cup$ OFA, then there is obviously nothing that could save us from the necessity of using a fully-fledged independent HW device. The only question is: Which one to chose?

There are two promising technologies that are perceived separately, yet. The first one is an interesting approach based on using existing EMV chip payment card scheme [39] for generation of OTP based on a shared secret in between the payment card and the bank. There are two almost identical approaches which, however, bear different names. VISA uses so-called DPA (Dynamic Passcode Authentication) while MasterCard has their CAP (Chip Authentication Program). In practice, this whole concept is often dubbed as CAP/DPA [39]. Unfortunately, the documentation of these schemes is still confidential, so the sole publicly available sources are reverse engineering notes like [10]. There is apparent limitation of CAP/DPA – it is exclusively targeted for a two-factor authentication in the bank area. To remove the necessity of carrying a separate card reader for CAP/DPA-based authentication, payment card association are considering using smartcards with embedded display and keyboard in the near future.

The second promising technology is NFC (Near Field Communication) [30]. We have already talked about NFC at ISS 2008 [20]. Let us recall that a device equipped with NFC controller can behave either as a contactless card reader, contactless card emulator, or to work in so called point-to-point active mode. Contactless cards accessible by NFC device working as card reader include mainly smartcards according to ISO/IEC 14443. Interestingly, we can note that our prediction at ISS 2008 [20] saying NFC-ready devices, especially mobile phones, will be popular tools for hacking activities at the HF-band RFID came true [15].

Putting these two pieces together, we can get a really interesting independent token generator. With the mobile phone, it would communicate by using the NFC interface. The benefit of NFC when compared to e.g. Bluetooth is that the OTP generator can be powered right through the NFC interface, so there is no need for any battery management. This in turn makes the device considerably thin at a reasonable cost. The OTP generator itself would be a contactless CAP/DPA smartcard with embedded autonomous display and keyboard. The embedded display which is driven by the smartcard processor (independently on the smart phone) allows secure checking of transaction data before entering the PIN. The on-card keyboard ensures PIN confidentiality and the CAP/DPA scheme finally should ensure quite positive acceptation in the banking area.

If we should design a two-factor authentication for non-banking area, we can still use almost the same approach. We would only replace the CAP/DPA scheme (realized as e.g. Java Card application) with another scheme designed specially for the particular purpose. The key important ideas of the whole design would, however, stay the same:

1. tamper resistant key store with on-chip OTP computation,
2. independent display,
3. independent keyboard,
4. NFC used for powering and communication with the smart phone.

Regarding the bank area, there is also another possibility, this time rooted in NFC mobile payment applications. For such purpose, there is a secure element hosted right in the mobile phone (it can be either part of UICC/SIM card or a separate silicon die) that implements the payment card application. To conduct a payment, the secure element interfaces with the NFC controller working in the contactless card emulator mode, so finally appearing as a kind of contactless payment card to the outside world. There is also an interface channel in between the secure element and the main application processor allowing the client to enter so-called passcode which can be seen as an analogue of the classic EMV offline PIN to some extent.

In the aforementioned setup of NFC mobile payment, it would be also possible to include a CAP/DPA application on the secure element. The autonomous display and keyboard needed for secure independent token generator can be finally substituted by a carefully designed trustlet module running on the secure virtual processor core (cf. TrustZone discussion above). This way, we can get another implementation of a robust independent token generator that can withstand even the joint ATA $\cup$ OFA strategy.

7 Conclusion

In this basic study, we were focused on a relatively small part of the whole information system (e.g. mobile banking) that is running on the client smart phone. There are several other parts of the system that already do actively contribute to its security. These can, sometimes, even compensate certain weaknesses of the mobile platform. Their discussion is, however, beyond the scope of this elaboration. Our aim was to investigate on how far we can effectively go right on the mobile device under very decent assumptions on its security. The conclusion is that the two-factor authentication with an adequate level of security is still achievable.

On the other hand, it is definitely challenging task to design such mechanism securely. First of all, we have to fully understand the relevant threat model. Such a simple model was presented here with certain accent on so-called after-theft attack. Resistance to this attack is achievable even with a general SW arrangement (i.e. without using any exotic HW features or external devices) and it is considered as a *must* for e.g. today's banking application. Resistance to the on-the-fly attack is of different kind, as it is only partially achievable with a general SW implementation. To do that, we must not only avoid exploits in our own application. We also have to carefully educate clients to manage their smart phones properly and, finally, we have to accept the residual risk. This seems to be a bearable task, but for a quite limited time period (several years). Hopefully, the situation will be better in the future as we can look forward to have applications employing TrustZone secure virtual processor core as well as various supporting devices, like NFC-capable authentication smartcards with autonomous display and keyboard.

References

[1] Anderson, J., Cade, B., and Stuart, T.: *Mobile Security – EMV and Beyond*, MasterCard Global Risk Management Conference, Prague, 2011

[2] *ARM Security Technology - Building a Secure System using TrustZone Technology*, whitepaper, ARM Limited, 2009

[3] Bachman, J.: *iOS Applications Reverse Engineering*, Swiss Cyber Storm, 2011

[4] Bédrune, J.-B. and Sigwald, J.: *iPhone Data Protection in Depth*, HITB Amsterdam, 2011

[5] Blazakis, D.: *The Apple Sandbox*, Black Hat DC, 2011

[6] Breeuwsma, M.-F., de Jongh, M., Klaver, C., van der Knijff, R., and Roeloffs, M.: *Forensic Data Recovery from Flash Memory*, Small Scale Digital Device Forensics Journal, Vol. 1, No. 1, June 2007

[7] Breeuwsma, M.-F.: *Forensic Imaging of Embedded Systems Using JTAG (boundary-scan)*, Digital Investigation 3, pp. 32 - 42, 2006

[8] Chin, E., Felt, A.-P., Greenwood, K., and Wagner, D.: *Analyzing Inter-Application Communication in Android*, MobiSys'11, 2011

[9] Dhanjani, N.: *New Age Application Attacks Against Apple's iOS (and Countermeasures)*, Black Hat Barcelona, 2011

[10] Drimer, S., Murdoch, S.-J., and Anderson, R.: *Optimised to Fail: Card Readers for Online Banking,* Financial Cryptography '09, 2009

[11] Dubuisson, O.: *ASN.1 - Communication Between Heterogeneous Systems*, Morgan Kaufmann Academic Press, 2001

[12] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S.: *A Study of Android Application Security,* Proc. of the 20th USENIX Security Symposium, 2011

[13] Fairbanks, K.-D., Lee, C.-P., and Owen III, H.-L.: *Forensics Implications of Ext4*, Proc. of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research, ACM, 2010

[14] Felt, A.-P., Finifter, M., Chin, E., Hanna, S., and Wagner, D.: *A Survey of Mobile Malware in the Wild*, SPSM'11, 2011

[15] Francis, L., Hancke, G., Mayes, K., and Markantonakis, K.: *Practical Relay Attack on Contactless Transactions by Using NFC Mobile Phones*, IACR ePrint Archive, 2011/618, 2011

[16] *GlobalPlatform Device Technology - TEE System Architecture*, ver. 1.0, GPD_SPE_009, December 2011

[17] Halbronn, C. and Sigwald, J.: *iPhone Security Model & Vulnerabilities*, HITB KL, 2010

[18] Hay, R. and Amit, Y.: *Android Browser Cross-Application Scripting*, CVE-2011-2357, IBM Rational Application Security Research Group, 2011

[19] Heider, J. and Boll, M.: *Lost iPhone? Lost Passwords!*, Fraunhofer SIT Report, cf. also [41], 2011

[20] Hlaváč, M. and Rosa, T.: *RFID: What's in our pockets anyway?*, Information Security Summit '08, 2008

[21] Hoog, A. and Strzempka, K.: *iPhone and iOS Forensics – Investigation, Analysis and Mobile Security for Apple iPhone, iPad, and iOS Devices*, Elsevier, 2011

[22] Hoog, A.: *Android Forensics – Investigation, Analysis and Mobile Security for Google Android*, Elsevier, 2011

[23] *HOTP: An HMAC-Based One-Time Password Algorithm*, RFC 4226, 2005

[24] http://androidandme.com/2012/02/applications/google-wallet-hacked-again-no-root-access-required-this-time/

[25] http://developer.android.com

[26] http://developer.apple.com

[27] http://theiphonewiki.com

[28] http://www.arm.com/products/processors/technologies/trustzone.php

[29] http://www.bbc.co.uk/news/technology-15635408

[30] http://www.nfc-forum.org/home/

[31] *iOS App Programming Guide*, Apple Developer Guide, Apple Inc., 2011

[32] Menezes, A.-J., van Oorschot, P.-C., and Vanstone, S.-A.: *Handbook of Applied Cryptography*, CRC Press, 1996

[33] Miller, C. and Iozzo, V.: *Fun and Games with Mac OS X and iPhone Payloads*, Black Hat Europe, 2009

[34] Miller, C. and Zovi, D.-A.-D.: *The Mac Hacker's Handbook*, Wiley Publishing, Inc., 2009

[35] Octeau, D., Enck, W., and McDaniel, P.: *The ded Decompiler*, NAS-TR-0140-2010, Networking and Security Research Center, The Pennsylvania State University, 2011

[36] Oudot, L.: *Planting and Extracting Sensitive Data Form Your iPhone's Subconscious*, HITB Amsterdam, 2011

[37] *PKCS #1 v2.1: RSA Cryptography Standard*, RSA Laboratories, June 14, 2002

[38] Rosa, T.: *Android Ecosystem Integrity - Possible Malware Cross-Infection Vector*, seminar note, http://crypto.hyperlink.cz/rosa_android_gm_v1b.pdf, 2011

[39] Rosa, T.: *Unleashing EMV Cards For Security Research*, Santa’s Crypto Get-Together in Prague, http://crypto.hyperlink.cz/files/EMV_unleashed_rosa_v1.pdf, 2010

[40] *Secure Coding Guide*, Apple Developer Guide, Apple Inc., 2012

[41] Toomey, P.: *"Researchers Steal iPhone Passwords In 6 Minutes" - True, But Not the Whole Story*, Security Blog, http://labs.neohapsis.com/2011/02/28/researchers-steal-iphone-passwords-in-6-minutes-true-but-not-the-whole-story/ , 2011

[42] *TOTP: Time-Based One-Time Password Algorithm*, RFC 6238, 2011

[43] Zdziarski, J.: *Hacking and Securing iOS Applications*, O'Reilly Media, 2012

[44] Zovi, D.-A.-D.: *Apple iOS 4 Security Evaluation*, Black Hat USA, 2011