SecurePay: Strengthening Two-Factor Authentication for Arbitrary Transactions

Radhesh Krishnan Konoth
Vrije Universiteit Amsterdam
r.k.konoth@vu.nl

Björn Fischer
Vrije Universiteit Amsterdam
code@bjorn-fischer.de

Wan Fokkink
Vrije Universiteit Amsterdam
w.j.fokkink@vu.nl

Elias Athanasopoulos
University of Cyprus
eliasathan@cs.ucy.ac.cy

Kaveh Razavi
Vrije Universiteit Amsterdam
kaveh@cs.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

***Abstract*—Secure transactions on the Internet often rely on two-factor authentication (2FA) using mobile phones. In most existing schemes, the separation between the factors is weak and a compromised phone may be enough to break 2FA. In this paper, we identify the basic principles for securing any transaction using mobile-based 2FA. In particular, we argue that the *computing system* should not only provide *isolation* between the two factors, but also the *integrity* of the transaction, while involving the user in confirming the *authenticity* of the transaction. We show for the first time how these properties can be provided on commodity mobile phones, securing 2FA-protected transactions even when the operating system on the phone is fully compromised. We explore the challenges in the design and implementation of SecurePay, and evaluate the first formally-verified solution that utilizes the ARM TrustZone technology to provide the necessary integrity and authenticity guarantees for mobile-based 2FA. For our evaluation, we integrated SecurePay in ten existing apps, all of which required minimal changes and less than 30 minutes of work. Moreover, if code modifications are not an option, SecurePay can still be used as a secure drop-in replacement for existing (insecure) SMS-based 2FA solutions.**

1. Introduction

Today's Two-Factor Authentication (2FA) schemes for secure online banking and payment services often use smartphones for the second factor during initial authentication or subsequent transaction verification. As a result, all current solutions are vulnerable to sophisticated attacks and offer only weak security guarantees (see also Table 3). Specifically, attackers may compromise the phone (including the kernel [1]–[3]) and break the second factor. This is true for mobile-only banking services, but also for solutions that use a separate device (typically, a PC) to initiate a transaction.

Starting with the former, users increasingly rely exclusively on mobile applications for using their bank services, purchasing products, or booking trips [4], [5]. Using web-based payment services through a smartphone brings convenience, as users can now access them anytime and anywhere—even when access to a personal computer is not possible. However, such convenience comes at a cost to security guarantees offered by 2FA. Specifically, 2FA works if and only if the two factors remain independent and isolated, because it requires a compromise of *both* factors to initiate fraudulent transactions—a difficult task when devices are decoupled. This is not the case, however, when a *single* device, such as a smartphone, serves both factors, since the attacker needs to compromise only the potentially vulnerable smartphone for breaking 2FA.

Worse, even PC-initiated transactions are not safe if the attacker obtains root privileges on the mobile device. In that case, attackers can replace or tamper with the mobile apps, intercept messages and display misleading information. Unfortunately, compromising smartphones is a realistic threat especially given a compromised PC, because even though the phone and PC are physically separate, the devices are often not independent [6].

Surprisingly, despite ample proof that today's phone-based 2FA is weak in practice [6]–[8] and despite a range of proposed solutions [9]–[14], this is not at all a solved problem. In practice, the issues may be as basic as a lack of separation between the two factors. However, even if there is a strong separation, there are other, equally fundamental issues. For instance, even research solutions tend to focus on a limited threat model that excludes fully compromised phones where an attacker obtains root access, infects the kernel and/or the corresponding app. Given that 2FA is often used in high-value interactions (e.g., banking) and full system compromises are a common occurrence [15]–[19], such a limited threat model is wholly insufficient.

Getting what appears to be a simple issue right is remarkably hard and we will show that even the most state-of-the-art solutions are lacking. When studied closely, 2FA exhibits many subtle issues in both the bootstrap phase (generating and registering keys) and operational phase (performing transaction). Given the many failed attempts at secure solutions, verifying the correctness of a solution in all possible corner cases is difficult for a human analyst. Instead, we propose the use of automated proofs to *guarantee* the security properties of our solution.

In terms of basic principles, we say that a transaction's *authenticity* is ensured if we prevent an attacker from initiating a transaction on behalf of the user (or server)

without being noticed. Meanwhile, a transaction's *integrity* is preserved if an attacker is not able to modify the content of the messages exchanged or displayed. Given these basic principles, we argue that even if existing solutions separate the two factors in 2FA, they tend to focus on authentication and ignore integrity, even though secure transaction requires both authenticity and integrity.

In this paper, we present SecurePay, a novel, principled design to regain the strength of 2FA even in the presence of fully compromised devices, while retaining the convenience of mobile devices by securing a minimal amount of core functionality for handling the second factor in the secure world provided by the Trusted Execution Environment (TEE). All other code runs in the normal world. SecurePay applies both to mobile-only transactions *and* to transactions initiated on a personal computer with the mobile device serving as the second factor only. Unlike previous work [9], [10], [12]–[14], [20], [21], SecurePay builds on the solid foundation of a minimalistic protocol for guaranteeing authenticity *and* integrity for any transaction-based system, for which we additionally provide a formal security proof.

The protocol is deliberately minimalistic to adhere to Saltzer and Schroeder's principles of Economy of Mechanism, Least Common Mechanism, Least Authority, and Privilege Separation [22]. It is the first solution to include just the minimum of generic 2FA functionality in the TEE to cater to *all* banking, payment, and similar services—allowing each to secure their transactions with verifiable OTPs (one-time passcodes) in a 2FA solution.

In particular, SecurePay's secure world provides three essential functions. First, it is responsible for generating SecurePay public-private key pairs. The private key *never* leaves the secure world, while banking and similar services will use the public key to encrypt the verification OTPs for transactions. Second, it will decrypt the verification OTPs that it receives from banking and similar services (via the mobile app in the normal world as an intermediary). Since the SecurePay private key never leaves the TEE, it is the only entity capable of decrypting these messages. Finally, it is capable of displaying the encrypted messages to the user in a secure and unforgeable manner. In other words, when users see messages displayed by SecurePay's trusted code, they can be certain that it was generated by the trusted components and that it was not tampered with. SecurePay ensures this through a software-only solution based on a secret that is shared between the user and the TEE.

These three functions allow any payment or similar service to implement secure transactions. To back this up, we incorporated SecurePay in ten different apps with minimal effort. Moreover, on the client-side, SecurePay may also serve as a drop-in replacement for existing (unsafe) solutions such as SMS—without any code changes whatsoever. To the best of our knowledge, SecurePay is the first generic system that supports arbitrary transaction services without requiring additional hardware or significant changes on the client side, while ensuring the authenticity and integrity of the transactions initiated from a PC or smartphone, even in the case of fully compromised devices. We designed SecurePay as an effective and practical solution, utilizing TEE features available in all modern devices.

Contributions We make the following contributions:

1) We analyze current 2FA-based techniques for protecting Internet banking, show why they are weak, and identify the key functionality that we need to isolate.
2) We present SecurePay, a generic 2FA design capable of guaranteeing the integrity and authenticity of any transaction (financial or otherwise) initiated from a mobile app or PC even in the face of a complete system compromise—where current solutions target *only* authenticity for more limited threat models.
3) We implemented and evaluated SecurePay on an actual smartphone[1].
4) We present a secret-based trusted-UI without the requirement of additional hardware.
5) We provide a formal proof of SecurePay's security guarantees.

2. Background

In this section, we discuss the necessary background information for both our threat model and the SecurePay design.

2.1. Mobile transactions and 2FA

Two-factor authentication (2FA) is a well-established mechanism for hardening authentication schemes. Typical designs for web-based transactions require users to demonstrate not only their knowledge of some secret credentials (such as a password), but also their possession of some artifact (such as mobile phone). Since mobile phones are easily the most popular choice for a large class of 2FA implementations, we limit ourselves to phones only. In virtually all such schemes, after a user submits the credentials to a server, the server sends an OTP to the user's phone, often in the form of a short sequence of digits. The assumption is that the ability to also submit the OTP proves possession of that phone. In principle, 2FA secures clients against the misuse of passwords that may have leaked [23], [24]. An adversary who steals a user's password and/or compromises the user's personal computer is still unable to impersonate the victim, as long as the service can transmit the second factor safely to the smartphone, that is assumed to be non-compromised.

Besides initial authentication, 2FA also commonly protects sensitive transactions, such as bank transfers. In particular, e-banking and e-commerce services enforce a 2FA procedure whenever the *already authenticated* user performs a particular action, such as transferring money from one bank account to another, or verifying an on-line purchase. For instance, the Bank of America's SafePass offers security based on 2FA. Without it, customers can transfer only small amounts of money, while for higher-value transfers, the bank sends a 6-digit OTP as an SMS text message to the user's phone. In this context, 2FA no

1. While this sounds simple enough, TEEs in phones are normally not accessible to researchers. It was only possible thanks to a vendor's support.

longer functions as an enhanced user authentication mechanism, but rather as additional procedure to verify the *authenticity* of the transaction. There is a subtle but important difference between these two uses of 2FA. Specifically, for on-line transactions just verifying the *authenticity* of the transaction is not enough, and *integrity* must be preserved also, for instance to prevent strong and stealthy attackers from modifying the user-issued transaction without the user's knowledge.

Unfortunately, in the absence of strong guarantees of separation between the factors, and authenticity and integrity of the transaction, an attacker who compromises the mobile can also intercept and/or tamper with the OTP. Worse, all 2FA solutions for mobile devices today are vulnerable to strong attackers capable of compromising the system completely by obtaining root access, infecting the kernel, or replacing the banking app with a malicious repackaged version. Such strong attackers may be able to initiate a transaction without the user's knowledge, breaking the *authenticity* of the transaction, or hijack the user's transaction by displaying misleading information on the display, breaking the *integrity* of the transaction.

2.2. Separating the factors

Fortunately, modern devices offer strong isolation primitives in the form of a TEE, such as ARM's TrustZone. As we shall see, if carefully used, they can restore the isolation between the factors in a 2FA scheme. A TEE offers a hardware-supported secure environment that protects both code and data from all other code—with respect to authenticity and integrity. Even the kernel of the operating system in the normal world is unable to view or tamper with anything in the secure world.

However, even with a TEE, facilitating a secure transaction using 2FA in the face of a strong attacker is deceptively difficult, and all existing solutions show weaknesses in important cases. For instance, in Section 6.6 we will see that solutions such as the TrustPay design are vulnerable to *man-in-the-mobile* attacks even though it uses a TEE, because isolation is only the first step; integrity and authenticity of the transaction are equally important. Similarly, we will also discuss the weaknesses of VButton [25] design that result from the lack of a Trusted UI indicator.

Moreover, the naive solution of simply allowing each and every bank, payment service, or transaction application to run custom code in the TEE is undesirable, as doing so violates the Principle of Least Authority as well as that of Privilege Separation [22], and increases both the number of bugs and the attack surface in the TEE (jeopardizing the security of the entire system). Instead, we should keep the amount of code running in the TEE to a minimum and offer the least amount of functionality possible to cater to all possible 2FA applications (adhering to the Principle of Least Common Mechanism).

Besides TEEs, it is also possible to separate the factors by an additional (external) hardware device. For instance, the only solutions with a similar threat model to ours, such as ZTIC [9], require such additional hardware in the form of a USB stick. These devices need constant updates to support new service providers. Moreover, ZTIC protects only PC-initiated transactions and previous work has shown that extending it to mobile-only scenarios is difficult [26].

2.3. Trusted Execution Environment (TEE)

A TEE is a secure execution environment that runs in parallel with the operating system of a smartphone or similar device. Several hardware vendors have introduced different hardware-assisted TEEs: Intel Identity Protection Technology, Intel Software Guard eXtension, ARM TrustZone etc. [27], [28]; however, in this paper we focus on TrustZone, as mobile devices tend to use ARM processors. Most such devices support a TrustZone-based TEE [29], which they implement using a set of proprietary methods. TEEs manage *trusted* applications which provide security services to *untrusted* applications running on the commodity operating system of the mobile device. For this purpose, the GlobalPlatform (GP) consortium is developing a set of standards for TEEs [30]–[32]. It includes APIs for the creation of trusted applications [33], as well as for interacting with other trusted applications securely [34]. Each trusted application should be able to run independently and should be prevented from accessing additional resources of other (trusted) applications. Nowadays, TEE developers implement TEEs in compliance with the GP API specifications.

ARM TrustZone [35]–[37] provides a TEE by enabling the system to run in two execution domains in parallel: the *normal* and the *secure* world (Figure 6). The current state of the system is determined by the value of the *Non Secure (NS)* bit of the secure configuration register. The secure domain is privileged to access all hardware resources like CPU registers, memory and peripherals, while the normal world is constrained. There is an additional CPU mode called *monitor*, which serves as a gatekeeper between the two domains. This monitor mode is also privileged, regardless of the status of the NS bit. Likewise, the memory address spaces is divided into regions, which are marked as secure or non-secure using the TrustZone Address Space Controller. Finally, peripherals can be marked as accessible to the secure world by programming the TrustZone Protection controller.

To switch from the normal to the secure world, the system must initially switch to monitor mode by executing a *Secure Monitor Call (SMC)*. Essentially, this allows the monitor mode to verify whether switching from one world to the other should be permitted. If the request is determined valid, it modifies the NS bit accordingly and completes the world switch.

3. Threat model and assumptions

We assume a strong and stealthy attacker, who has obtained the user's credentials (e.g., via a password leak, or a compromised device), and fully compromised the user's smartphone. In other words, all the code in the normal world, including the operating system kernel, should be considered malicious. The attacker seeks to perform malicious transactions on behalf of the victim. For example, the attacker may replace the banking app in the user's device with a malicious one. While strong, this is a realistic threat model on today's smartphones and probably *should* be the threat model for highly sensitive

applications such as payment systems. Many exploits exist that allow attackers to run malicious apps on a user's phone in a stealthy manner [6], [38], [39] and escalate the privilege to root [16], [18], [19]. As we discuss in Section 8, previous work fails to protect against such strong attacker models.

We also assume that users have secured their accounts with 2FA, so that the attacker must bypass 2FA restrictions (stealthily) for every malicious transaction. As the attackers have compromised the mobile device, they have access not just to the user's credentials, but also to the second factor OTP which we assume the payment service to deliver *only* to the compromised device. As attackers may have compromised the phone entirely, denial-of-service attacks are not relevant. Hardware-level attacks such as bus snooping [40], [41] and cold boot [42], [43] are out of scope as they require physical access to the phone.

Bootstrap We assume that TEE provides a secure boot process to ensure the integrity of the executables running in the secure world. In fact all modern devices achieve the secure-boot by implementing a chain of trust [44], [45]. On a device-reset event, the boot code from ROM verifies and loads the secure bootloader. The secure bootloader initializes TEE and loads the non-secure bootloader – after verifying its integrity. Finally, the non-secure bootloader verifies and loads the normal world OS.

4. Design

In this section, we first describe the requirements for a secure and compatible mobile-based 2FA before we explain SecurePay's design.

4.1. Requirements for a secure and compatible design

Transaction-based systems use traditional phone-based 2FA solutions to guarantee the authenticity of a transaction and the 2FA works only as long as one of the factors is not completely compromised. The main weakness of such 2FA-based solutions under our threat model is that the two factors are not sufficiently isolated and as a result, these solutions cannot guarantee both the authenticity and the integrity of the transaction.

As an example, consider Alice, who uses e-banking (using either her mobile phone or PC) to transfer some money to Bob. When Alice is about to make the money transfer, in the ideal case her bank sends an additional short code (generally referred to as a One-Time Password or OTP) with the transaction summary to Alice's phone to verify the requested transaction. However, if the phone is compromised, the attacker can (i) silently initiate a transaction, read the OTP and send it to the bank to confirm a fraudulent transaction breaking the *authenticity* of the transaction, or (ii) display a falsified transaction summary to the user (that matches the user's expectation) and trick her to confirm a different, fraudulent transaction—breaking the *integrity* of the transaction.

Thus, we identify the following key requirements that must be satisfied for any design for secure 2FA:

1) Isolation: we must ensure the separation of the domains manipulating the two factors in 2FA.
2) Integrity: attackers should not be able to tamper with (or read and display in modified form) a transaction's OTP messages as sent by the payment service.
3) Authenticity: users must be looped in to enforce the authenticity of the transaction.
4) Secure bootstrapping: users must be able to securely register the device to the service they wish to engage in 2FA authentication.

Besides these strict requirements, we increase both security and usefulness of our solution by three additional constraints:

5) Least common mechanism: the TEE should support the minimum functionality needed to support most applications and no more [22].
6) Provable security: given the pivotal role of 2FA for many highly sensitive services, we demand a formal proof of our design's security guarantees.
7) Compatibility: to facilitate adoption, we demand that it should work with existing services.

4.2. SecurePay

We propose SecurePay, our design of a secure and compatible 2FA solution that satisfies all of the aforementioned requirements. To satisfy (1), SecurePay uses TEE, such as ARM's TrustZone for creating a hardware-enforced isolated environment. To satisfy (2), SecurePay uses off-the-shelf public-key cryptography and the TEE for protecting the integrity of the transaction. To satisfy (3), SecurePay relies on a software-based secure display implementation, the output of which can only be produced by legitimate code which is recognizable as such by the user. To satisfy (4), SecurePay provides a tamper-resistant mechanism enforced by the TEE that allows users to securely register with a service provider that allows authentication through 2FA.

Furthermore, to satisfy our softer requirements, SecurePay provides a minimal TCB that runs in the TEE (trusted app) and we have formally verified that its protocol provides *authenticity* and *integrity* for transactions. To provide compatibility, SecurePay is capable of utilizing SMS as the communication channel between the user and service provider, and provides a normal-world component, the SecurePay app, to communicate the received encrypted SMS to SecurePay's trusted app (TA) in the TEE. Thus the service providers do not have to modify their mobile app to utilize SecurePay. Upon receiving the encrypted OTP and transaction summary, the user can invoke the SecurePay app to display the decrypted message on the secure screen (fully controlled by the TEE).

To further explore the compatibility aspects, SecurePay provides two modes of operation, one that is a fully transparent drop-in replacement for existing SMS-based schemes, and another which requires a small modification to the service providers' apps but offers full integration to simplify the user interaction.

For the drop-in replacement mode, Figure 1 shows the workflow as a sequence of steps. First, the user initiates a transaction from an app on the phone (or in the browser running on her PC as shown in Figure 4). The service provider receives this request and responds

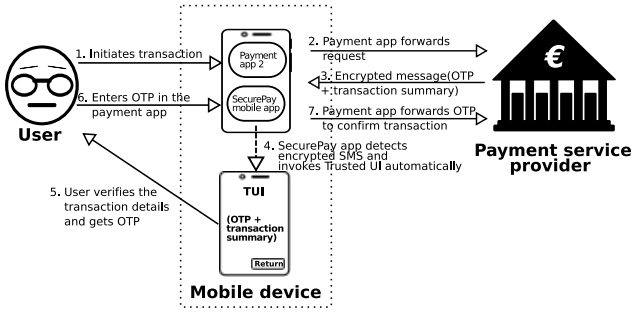

Figure 1: The work-flow of SecurePay-based transactions in transparent (drop-in replacement) mode.

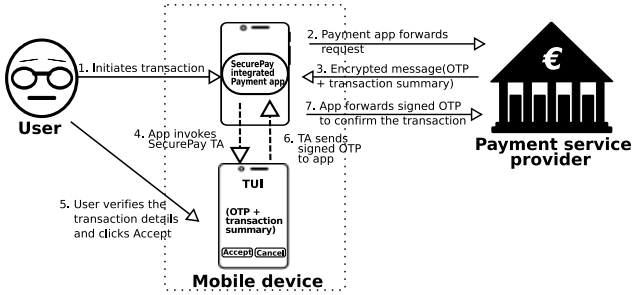

Figure 2: The work-flow for full integration of SecurePay (SecurePay integrated mode).

with an SMS containing an encrypted message (using the public key of the user's SecurePay) that includes the transaction summary and an OTP. The SecurePay's secure app receives this SMS and launches the trusted UI to display the transaction summary and the OTP to the user. Once the user verifies the authenticity of the transaction, she can switch to the service provider's app and enter the OTP, exactly like she would do with existing SMS-based OTPs. This OTP is then forwarded to the service provider and if it matches the one sent by the provider earlier, the transaction completes successfully. This version of SecurePay does not require any modification on the service provider's app on the phone, but it does require the user to memorize (or note down) the OTP and enter it in the service provider's mobile app (or web interface) to confirm the transaction. Fortunately, studies in psychology and HCI [46] have shown that humans can remember without difficulty on average 7 items in their short memory. Table 1 shows that most services are using fewer digits (5 to 6) as OTP.

SecurePay can lift this requirement and hide the OTP entirely with a small modification of the service provider's app to fully integrate SecurePay. Figure 2 shows the necessary steps for confirming a transaction in this version of SecurePay. The main difference is that the service provider's app directly communicates with the SecurePay trusted app using the SecurePay library (discussed in Section 5). Similar to the previous version, the user initiates a transaction using the service provider's app. The service provider then sends an SMS with an encrypted summary and an OTP to the phone. Given that we do not want to increase SecurePay's code base, we let the provider's app (instead of SecurePay) receive this information (via SMS or Internet) and forward it to the SecurePay trusted app, which decrypts the message and shows the user the

TABLE 1: Type of OTP used by different services

Service Provider	Type of OTP	Length of OTP
Google	digits	6
ING Bank	digits	6
Bitfinex	digits	5
Bank of America	digits	6
Citibank	digits	6
Deutsche Bank	digits	6
Dhanlaxmi Bank	digits	6
Axis Bank	digits	6
Alpha Bank	digits	6
TransferWise	digits	6

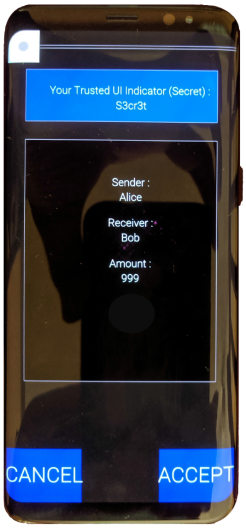

Figure 3: Full integration of SecurePay: the user simply presses accept or cancel on the trusted screen (SecurePay integrated mode).

transaction summary (but no OTP). The only thing the user needs to do is accept or reject this transaction after looking at the summary (see Figure 3). The SecurePay trusted app then transparently signs the OTP and sends it to the service provider's app, which in turn forwards the signed OTP to the service provider. Upon receiving the signed OTP, the provider completes the transaction if the OTP matches the one sent earlier. This version of SecurePay provides more convenience for the user, but requires a small modification of the service provider's app (around 20 lines of code on average and little effort, as we will show in our evaluation).

Next, we discuss implementation details of SecurePay before analyzing its security guarantees and evaluating its performance.

5. Implementation

The architecture of SecurePay is depicted in Figure 6. In our prototype on Android 8, the mobile operating system runs in the normal world and manages all mobile apps, while Kinibi, Trustonic's TEE, runs in the secure world and manages all trusted apps. We tested a full implementation of SecurePay on a Samsung Galaxy S8 mobile device. In this section, we first discuss the implementation of SecurePay's components, then introduce its

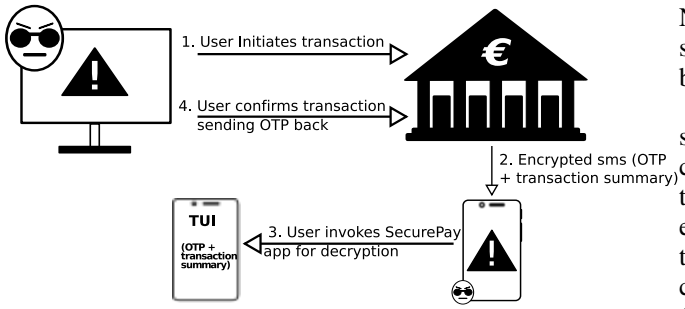

Figure 4: Shows how a SecurePay-enabled user issues transactions securely through a PC even if both PC and associated mobile device are infected by malicious code.

secure bootstrapping process, and finally, explain in detail how a user can initiate and complete a transaction securely even if all (normal-world) software on the user's devices (both PC and mobile) is fully compromised.

5.1. SecurePay components

SecurePay contains two main components: the SecurePay trusted app (TA) and the SecurePay Android library that enables any mobile app to communicate to the SecurePay TA. The SecurePay TA runs inside the secure world, beyond the reach of normal apps running in the normal world. The mobile app, running in the normal world, can access the TA only through the APIs implemented by SecurePay Android library.

The SecurePay Android library is an Android Archive (AAR) file which can be linked to any Android app. It implements the API that allows apps in the normal world to access the functionalities provided by the SecurePay TA and comprises 1,546 LoC. Internally, the library uses the Global Platform (GP) TEE client API to implement these APIs.

For the drop-in replacement mode, we built the SecurePay mobile app, the normal world component of the SecurePay (Figure 1), using the SecurePay Android library. The end user can use the SecurePay mobile app to (i) generate the key pair, (ii) retrieve the public key, and (iii) decrypt and display the SMS on the secure screen. Once the secure screen is visible, the user is sure that the SecurePay TA is in control and the content of the screen can be trusted. How SecurePay implements its secure screen will be explained later in the section.

In SecurePay integrated mode, the SecurePay TA transparently sends the decrypted OTP back to the payment app (Figures 2 and 3), once the user verifies and accepts the transaction. This can be implemented using the SecurePay Android library, but in this case, the service provider has to modify its app to receive the encrypted transaction summary and OTP from the server, and to invoke the SecurePay TA to decrypt and display it on the secure screen. In practice, doing so took less than 30 minutes in all ten apps we tried (Section 6). Then, the user can verify the transaction details and press *accept* or *cancel* on the secure screen (Figures 3 and 2). If the user accepts the transaction, the SecurePay TA signs the OTP with the private key and sends it back to the service provider. If not, the SecurePay TA terminates the session. Note that the signing of the OTP by the SecurePay TA serves to prevent the attacker from trying to guess or bruteforcing the OTP reply to the server.

Note that in both modes, the OTP only leaves the secure world if the user *accepts* the transaction by either clicking on the button on the secure screen or entering the OTP in the normal world app. This is how SecurePay ensures the authenticity of the transaction, while the integrity of the transaction is ensured using public-private cryptography, a secure screen and secure bootstrapping (which we discuss in the next subsection).

The SecurePay trusted app The trusted core of SecurePay comprises 4,565 LoC running in the *Kinibi* secure world—a GP-compliant TEE which implements secure storage APIs and many common cryptographic APIs. As a consequence, SecurePay should work out of the box with any GP-compliant TEE.

Specifically, the TEE Internal API defined by the GlobalPlatform Association and implemented by Kinibi supports most common cryptographic functions such as message digests, symmetric ciphers, message authentication codes (MAC), authenticated encryption, asymmetric operations (encryption/decryption or signing/verifying), key derivation, and random data generation. On top of these primitives, Kinibi implements a powerful *secure storage* layer which guarantees the confidentiality and integrity of sensitive general-purpose data, such as key material, as well as the atomicity of all operations on secure storage.

Using these APIs, the SecurePay TA supports three minimal functions. First, it can generate an asymmetric key pair of which the private key never leaves the TEE. Second, it can display the QR code of the public key on the secure screen (Figure 5). Third, it can decrypt messages encrypted with the public key when requested to do so from an (unprivileged) user app in the normal world and securely display it to the user on a secure display—with guaranteed authenticity and integrity, even if the attacker has administrator access to the phone. We now explain these functions in more detail.

`Generate_keys()`: The mobile OS automatically invokes this function at first boot (or full device reset). When normal-world code invokes the function, the SecurePay TA first checks whether a key pair already exists in its Kinibi-enforced GP-compliant secure storage and only if the pair does not exist, will it generate a new RSA key pair (using Kinibi's cryptographic API). Of this keypair, it returns only the public key to the normal world.

`Display_public_key()`: When invoked, the SecurePay TA checks whether a key pair already exists in its secure storage. If it exists, it extracts the public key component and displays its QR code on the secure screen (Figure 5).

`Display_summary()`: When invoked, the SecurePay TA decrypts a message using the private key stored in secure storage and displays it on the secure screen. It is used to handle the OTP from the transaction service (such as a bank). Recall that after the user initiated a transaction and the mobile app has sent the transaction details to, say, her bank, the banking service encrypts the transaction details and a freshly generated OTP using the user's public key and sends it back to the mobile device. Now, the SecurePay mobile app invokes `Display_summary()` to

display the transaction summary and OTP on the secure screen. The API takes a boolean input parameter which decides whether the trusted app should return the *signed* OTP to the normal world if the user clicks on the *accept* button (Figure 3). To minimize confusion for the user, if the parameter is set to false, the SecurePay TA only displays a *return* button instead of *accept* and *cancel*.

Secure screen The main challenge in realizing a trusted user interface (TUI/secure screen/trusted screen) is ensuring that users can tell if they are actually dealing with a trusted application, and not with a user interface injected and controlled by a malicious app [27]. Since a switch from normal-world to secure world code is done via a GP TEE client API which internally calls the `SMC` instruction, an attacker who has full control over the victim's device can easily bypass the switch and project an attacker-controlled user interface instead—tricking the user into believing that the active interface is now the trusted one.

Existing approaches are not suitable for SecurePay and typically require additional hardware. For instance, to realize a TUI, TrustOTP (TOTP) [13] shares a single screen between the normal and the secure world, but with two different frame buffers, of which one is accessible only from the secure world. Moreover, to make sure that the attacker cannot bypass world switching, TrustOTP uses a separate non-maskable interrupt, triggered by a special button on the phone for passing control to the secure world. Unfortunately, such solutions do not work for SecurePay. Since the GP compliant TEE is expected to run multiple trusted apps in the secure world, a single interrupt is insufficient, while adding separate hardware interrupts for each of them is impractical. In addition and equally important in practice, current COTS phones lack such special-purpose buttons to begin with.

As an alternative, one could also use a single piece of additional hardware, such as a specialized LED, as an indicator of whether the display is controlled by the normal or the secure world [47]. By configuring a GPIO port to be only accessible from the secure world and connecting to a special LED on the phone, the user knows that if the LED is on, the secure world is in control of the display. Again, as smartphones today do not have such a LED-based indicator, this is not a practical solution for our purposes either.

Hence, SecurePay implements a software-only solution whereby the trusted code authenticates its output to the display by means of an easily recognizable shared secret. The example secret is an image or secret text that is known only to the user and the SecurePay TA. Examples are shown in Figures 5 and 3, where the simple logo in the top left corner serves as a simple example of a secret image and "*S3cr3t*" as an example of the secret text (the "Trusted UI Indicator") in the figure. The secrets are explicitly loaded into the TEE at first boot (or full system reset), when the device was assumed to be in a pristine state and they are stored in the Kinibi's *secure storage* layer which guarantees both confidentiality and integrity of the data. Since only the trusted code knows the secrets, the user knows that if the device displays the secret image and/or the text on screen, the trusted code must be in control of the screen and the frame buffer, and no other code can access it. Even the Android OS

Figure 5: Registering with the bank by showing the QR code of public key on the trusted screen

literally does not have any access to the hardware or the frame buffer during the period that the secure screen (TUI) is active — meaning that malware cannot capture the data displayed on the screen or simulate touches, even if the phone is rooted.

5.2. SecurePay registration and bootstrap

Enabling SecurePay with an actual service involves communicating the public key to the service. In case the user owns several devices, all devices must register with the SecurePay-protected service. Registration takes place when the user installs the client part (i.e., the mobile application) on the device. For successful registration, the user must communicate the public key securely to the service—in terms of integrity, not necessarily confidentiality.

Since we assume that the mobile device may be already compromised at registration time, we must prevent attackers from registering their own public keys with a user's account, either by initiating a binding request themselves, or by replacing the public key with their own when the user's binding request is in transit. Like all secure transaction systems, SecurePay requires a secure bootstrap procedure to handle this. Various solutions are possible, ranging from custom hardware extensions to in-person registration at a physical office (whereby a public key displayed on the phones secure display is manually bound to the account number).

Initial registration in our design simply assumes the presence of a secure terminal—for instance, at an ATM machine or a physical branch office. After installing the SecurePay mobile app, the user can invoke the SecurePay TA to display the QR code of the public key on the trusted screen as shown in the figure 5. Note that the user has to make sure the display is currently controlled by the trusted app by verifying the personalized image or secret text before sharing it to the bank. Finally, the bank simply scans the QR code to retrieve the public key safely from the user's device.

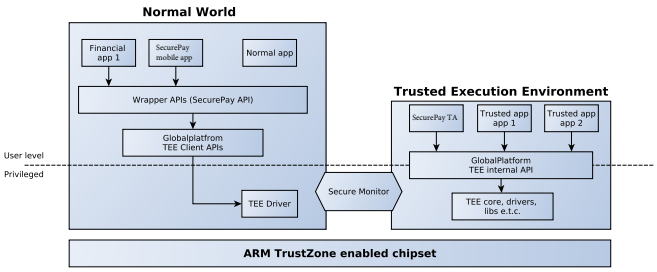

Figure 6: Multiple apps can use the same SecurePay TA.

Note that the registration for SecurePay is comparable to or simpler than that of many other payment services. For example, to enable e-banking, many banks require physical presence at a branch office and/or hardware tokens. More importantly, the threat model for SecurePay is considerably stronger than that of existing systems—protecting the user against attacks launched from a fully compromised device, where the attacker controls even the device's operating system kernel.

6. Evaluation

6.1. Security of mobile transactions

An attacker can get privileged access on a victim's device in two ways: exploit a software/hardware vulnerability or trick the user to install a repackaged version of a mobile app. Many reports [48], [49] show that cyber criminals are often successful in tricking the users into installing a repackaged version of financial apps using social engineering techniques. For the purposes of this paper, we exploited a hardware vulnerability of the Nexus 5 phone [18] to get root access and replace the official financial app with a repackaged version. The latter hijacks transactions and sends money to attacker-controlled accounts. Once a transaction is completed, the malicious app displays a fake transaction summary to the user on the infected device instead of the details of what actually happened. Countering such attacks, that can take place when a bank transaction is carried out solely by using a compromised device, is extremely hard.

SecurePay can help the user (victim) stop any hijacked transaction from even happening in the first place. With SecurePay enabled, once the e-banking service receives a user's transaction, it encrypts the transaction details and a freshly generated OTP with the user's public key stored at the bank's server and sends back the result to the financial app. The repackaged version of the app receives the encrypted message, but, unless the private key has leaked from the trusted storage, decrypting it is not possible. The only way to decrypt the message is to relay it to SecurePay, which, being in the secure world, controls the private key. However, once SecurePay decrypts the message, it forwards the plain text to the secure display. The user is able to inspect the modified transaction and signal an abort message to the e-banking service by entering an invalid OTP. Note that the bank needs to generate a new OTP for every transaction request it receives in order to prevent the attacker from reusing an old OTP.

6.2. Security of non-mobile transactions

Many well-known banking trojan horses like Zeus [50], Dyre [51], and Dridex [52] use malicious plugins or API hooking techniques to modify the HTTP responses received by a browser or to silently perform illegal operations on behalf of the user [53]. This is commonly known as a Man-in-the-Browser (MitB) attack [54].

Let us assume that the user is making a financial transaction from an infected (non-mobile) host, such as a PC, but SecurePay powers the user's mobile device and financial application. For example, Alice initiates a transaction to transfer $100 to Bob using her browser running on her PC. An attacker, through a MitB attack, modifies the transaction to $1,000 to be transferred to an attacker-controlled account. Once the e-banking service receives the transaction, it encrypts the transaction summary and a freshly computed OTP, and sends the encrypted message back to Alice's smartphone, using a push notification or SMS. Since Alice runs the SecurePay mobile app, the message is handled by the SecurePay TA and the transaction summary is displayed, along with the OTP, on the trusted display. Alice reviews the transaction, and since it has been modified, aborts the transaction.

We stress that in this scenario, the attacker may well have full control over the user's mobile device and even her PC and web account credentials. However, in spite of this, thanks to SecurePay, it is still not possible for any hijacked transaction to actually take place.

6.3. Verification using TAMARIN

We formally verified SecurePay's *authenticity* and *integrity* security properties using TAMARIN v1.4.1 (see Appendix A). The TAMARIN [55] prover supports the automated, unbounded, symbolic analysis of security protocols. It features expressive languages for specifying protocols, adversary models, and properties, and efficient support for deduction and equational reasoning. A security protocol is specified through multi-set rewrite rules and facts. A rewrite rule takes a number of facts and rewrites them to other facts. Initially, the state contains no facts and only rewrite rules that do not require input facts can be applied. An exception is the generation of a fresh nonce, which is always possible.

With regards to SecurePay we have two such initiator rules. The first rule models the initiation of a binding request for a new device. In this case, the fresh nonce is the private key of the device to be added. Since this rule can always be applied, the proof is performed for infinitely many devices. The second rule is the initiation of a new transaction, which can also be performed infinitely many times. The nonce is the transaction data, which is the initial input from the user to perform a transaction. This also means that the models hold for infinitely many transactions.

There are two flavors of facts. Persistent facts remain part of the state after they are consumed by a rewrite rule, hence they can never be removed. Linear facts are removed from the state when they are consumed by a rewrite rule. The latter may be used to model multiple

steps of a role. Each rewrite step produces a linear fact that is consumed by the successor step. We give an example:

```
rule step 1 :
  [ Fr ( ~nonce ) ]
 ==>
  [ Step1Completed ( ) , Out ( ~nonce ) ]

rule step 2 :
  [ Step1Completed ( ) , In ( response ) ]
 ==>
  [ Step2Completed ( ) ]
```

The first rewrite rule generates a fresh nonce and sends it into the network using the fact `Out (~nonce)`. The second step waits for a response from the network using the fact `In (response)`. The second rewrite rule is only ready to be performed after the first rewrite rule has been performed (modelled using the fact `Step1Completed`). Furthermore, persistent facts can be used to model the completion of a SecurePay binding request. Upon completion of the binding request the bank will create the fact `!PublicKeyForAccount (account, publicKey)`. This fact can be consumed (many times) to encrypt messages sent from the bank to the device holder. The usage of persistent facts in the model allows that the complete SecurePay protocol (binding requests and transactions) can be verified in a single model.

The adversary model employed by TAMARIN is the well-known Dolev-Yao model [56]. The intruder learns every message which is sent over the network, can change its content and may generate new messages from the knowledge obtained so far. In terms of TAMARIN this means that an intruder observes all `In`-facts and can produce an arbitrary number of `Out`-facts. All other facts are not observable by the intruder. Perfect encryption is assumed, meaning that the intruder does not learn anything from an encrypted message for which she does not own the key. Since in SecurePay the normal world is compromised, all messages to or from the normal world are compromised. We model this fact by exposing all messages that are traversing through the normal world to the network, i.e. the intruder.

We identify the following entities, which are involved in the protocol: i) the human performs binding requests and transactions, ii) the trusted app generates key pairs and displays messages on the secure screen, and iii) the bank processes binding requests and verifies transactions.

Our TAMARIN specification of SecurePay separates the multi-set rewrite rules of the trusted zone, the bank and the human entity by prefixing all rule names. An overview of all rewrite rules and their intended relations are depicted in Figure 7 (some facts are omitted for readability). Rewrite rules that do not consume any fact can be executed arbitrarily many times, hence we consider infinitely many devices and accounts. Similarly, because the fact `HumanInitiatesTransaction` only consumes persistent facts, it can be executed arbitrarily many times if we witness a single `HumanOpensAccount`. It follows that we also consider arbitrarily many transactions. Note that the rewrite rules may not appear in the same order in every trace. Because the intruder can create an arbitrary number of messages from previously obtained knowledge,

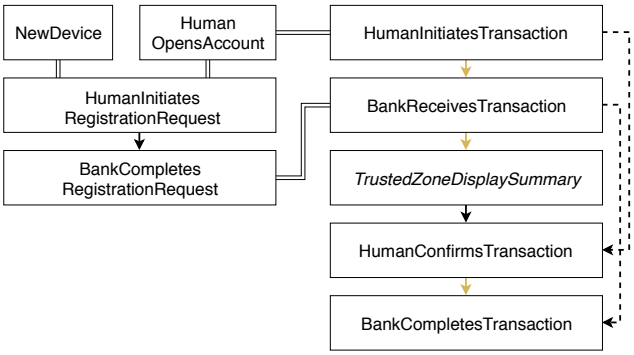

Figure 7: A visualization of the Tamarin specification of SecurePay. Boxes denote rewrite rules. Arrows between boxes denote facts, pointing from producer to consumer. Double lines denote persistent facts. Yellow arrows depict messages sent through the network or the normal world, i.e. interceptable by the intruder. Black lines are other linear facts, not observable by the intruder. Dashed arrows depict facts that denote a successor step within a role.

we consider a multitude of rule interleavings. We verified the security properties for all possible cases.

Unlike many security protocols, SecurePay has a control flow. Namely, for each transaction, the user decides to input the correct or a wrong OTP and the bank decides to accept or reject a transaction based on the received OTP. We model this behavior by restricting the application of rewrite rules using equality reasoning. An example for the above is `BankCompletesTransaction`. This rule can only be executed if the OTP contained in the network message matches the one that is generated by the fact `BankReceivesTransaction`.

TAMARIN imposes the security properties on a global view by inspecting the trace. Additional to input and output facts, rewrite rules can specify action facts. Whenever a rule is applied, the action facts are appended to the initially empty trace. The trace is used to verify security properties, which are called lemmas in TAMARIN. Lemmas are expressed using first-order logic formulas which must hold in all traces that are reachable from the initial configuration. To prove authentication and integrity, we specify lemma 1.

$$\forall t. TransactionCompleted(t) \rightarrow (\exists . HonestTransactionInitiated(t)) \quad (1)$$

Both `HonestTransactionInitiated` and `TransactionCompleted` are facts and contain the transaction details including the account owner. Intuitively, the lemma means that a transaction must only complete if it was initiated by the account owner.

TAMARIN automatically verifies security properties, expressed in first-order logic, for all possible execution traces, in a backward fashion. Hereby it employs constraint solving. It is either concluded that a given property holds for all execution traces that are possible from the initial protocol configuration, or a counter-example is produced. Since this verification problem is undecidable, inevitably such a backward run may not terminate. To achieve termination, the user may formulate and, with the

help of TAMARIN, prove so-called source lemmas that construct the possible sources for a fact. Source lemmas are solved using induction on the trace length.

To achieve termination of SecurePay's verification, one source lemma was needed. The lemma shows all possible sources for the encrypted message that is decrypted by the trusted zone. For all messages concerning a transaction, either the message comes from the bank or the intruder must know the OTP (that is contained in it). Using induction, TAMARIN can prove this lemma automatically.

To prove that replay attacks are not possible, we specify lemma 2.

$$\begin{aligned} &\forall t, t', i, j. TransactionCompleted(t)@i \wedge \\ &\quad TransactionCompleted(t')@j \wedge i \neq j \\ &\qquad \rightarrow \\ &(\exists k, l. HonestTransactionInitiated(t)@k \wedge \\ &\quad HonestTransactionInitiated(t')@l \wedge k \neq l) \end{aligned} \tag{2}$$

The variables i, j, k, l are time variables of the logical clock that is part of Tamarin. They uniquely identify the respective facts, that are proceeding the @ symbol. By definition, two facts cannot occur at the same time. Using inequality, we assume two arbitrary but distinct transactions and verify that the account owner(s) must have initiated two distinct transactions. We make no assumptions about t and t', therefore, we also verify replay attacks accross different accounts.

Using TAMARIN, we verified *authentication* for SecurePay: every transaction must be initiated by the human that owns the account. Furthermore, we verified *integrity*: a transaction can only complete if both the human and the bank agree on the transaction details. The proof holds for an unlimited number of devices and transactions. Additionally, we verified that a replay attack is not possible.

6.4. Performance evaluation

We evaluate the performance overhead of our system by integrating the SecurePay normal-world library into a native library and then bundling it with a custom Android banking app. Since transactions are relatively infrequent events, throughput is not of paramount importance. Instead, we demand that each operation involved in registration and transaction verification takes a "reasonable" time—no longer than one or two seconds, say. For this, we measure the time taken by the core operations of SecurePay using the `System.nanoTime()` function, available in the Java library. We invoke each core operation 1,000 times and report the average value in seconds. We conduct this experiment on a Samsung Galaxy S8.

SecurePay takes 1.34 seconds to generate a 2,048 bit RSA key pair and to retrieve the public key from the secure world, and 1.91 seconds to generate an RSA key pair and the display QR code of the public key on a trusted screen. Note that key generation happens only once. Finally, it takes 1.29 seconds to decrypt and display a transaction summary of 100 bytes on the trusted screen, including SMS retrieval from the inbox, a world switch, decrypting and displaying the message. The performance of SecurePay is directly proportional to the performance of each component in the TEE (such as the cryptographic services, secure-storage services, trusted UI, etc.). We expect that for all practical applications, SecurePay can be enabled on commodity smartphones with little additional overhead.

TABLE 2: Open-source apps modified to utilize SecurePay

Android apps	LoC added	Time taken
Wordpress login	20	$<$ 30 minutes
InboxPager login	20	$<$ 30 minutes
Openshop.io login	20	$<$ 30 minutes
OpenRedmine login	20	$<$ 30 minutes
Quill login	20	$<$ 30 minutes
Yaaic login	20	$<$ 30 minutes
Seadriod login	20	$<$ 30 minutes
Slide login	20	$<$ 30 minutes
Kandriod login	20	$<$ 30 minutes
Photobook login	20	$<$ 30 minutes

6.5. Integration effort

Any mobile vendor implementing a TEE according to the GP specification can use the SecurePay TA [57]. Financial app developers can easily integrate SecurePay into their apps by linking the provided user-space library.

The apps that are already using SMS-based mobile 2FA do not require any change in their mobile app. However, the developers need to add 45 lines of code on the server side to encrypt the transaction summary and/or OTP using the user's public key. For the developers who want to use the second (fully integrated) model, we measured the effort that it requires to integrate SecurePay in a mobile application. For this purpose, we picked 10 open-source Android apps that have a *login activity* from Github [58], [59] and recorded the time required for a full integration of SecurePay. To ensure the diversity we picked the apps randomly from the following categories: shopping, business, social network, productivity, etc.

Table 2 shows that the app developer can link the SecurePay Android library and fully integrate SecurePay using only 20 LoC. 16 of these LoC are to configure the app to use the SecurePay TA, while 4 are to invoke the relevant SecurePay API (mentioned in the implementation section 5.1). The table also shows that it took one researcher (unfamiliar with the target app) less than 30 minutes to add SecurePay support to any app.

6.6. Comparison with similar efforts

TrustPAY [11] proposes a design to ensure security and to protect the privacy of a mobile payment (m-payment); however, under the threat model considered in the current paper, it fails to protect both. In this part, we explain how TrustPAY works, possible attacks against it and how SecurePay successfully deals with such problems. We depict the underlying protocol of TrustPay in Figure 8 (taken directly and unmodified from TrustPAY) for protecting an m-payment transaction. In short, TrustPAY works as follows. Any normal world (NW) app can use TrustPAY to make a secure m-payment following a series of steps:

1) The TrustPAY component of the NW app requests a new/existing RSA key pair to the TrustPAY

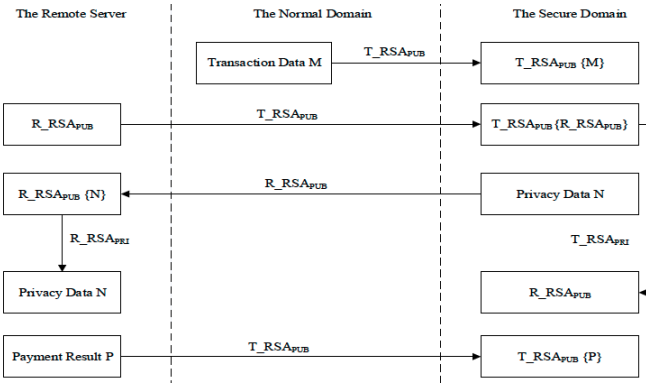

Figure 8: TrustPAY for m-payments.

trusted app (TA), which runs inside the secure world (TA checks for an existing key pair; in case this is not found, it generates a new RSA key pair).

2) TA saves the newly generated private key (T_RSA_{PRI}) inside the secure world and shares the public key (T_RSA_{PUB}) at the NW app.
3) Once the user places an order, the financial app in NW encrypts the order information using T_RSA_{PUB} key and sends the encrypted order information to the TA.
4) The TA decrypts the order information using its T_RSA_{PRI} key and displays it on the Trusted UI.
5) TrustPAY requests the bank for its public key by sharing T_RSA_{PUB}.
6) The bank encrypts its public key (R_RSA_{PUB}) with T_RSA_{PUB} and sends it to the TA.
7) The user can now verify the order details displayed on the Trusted UI and if she wants to pay the order, the user needs to enter the account number, password and verification code on the same Trusted UI.
8) Finally, the TA encrypts the user's private data with the public key of the bank, and sends the information back to the bank.

Let us analyze how an attacker can leak the user's private data (for example the account number or password) and hijack an in principle TrustPAY-protected transaction. We consider TrustPAY and the threat model assumed in this paper, i.e., the attacker already has root access on the device. In this case, when TrustPAY requests the bank's public key, the attacker can send an attacker-controlled RSA public key to the TA (instead of the bank's public key). This means that, when the user confirms the payment, TA encrypts the user's private data and confirmation status of the order using the attacker-controlled public key. Now the attacker can decrypt the user's private data, modify the transaction/order confirmation status, and encrypt it with the bank's public key, before the normal app relays it to the remote server. The remote server then decrypts the request with the bank's private key and confirms the fraudulent m-payment. Moreover, it should be noted that the attacker can also display the user-expected transaction/order details on the Trusted UI by just encrypting it with TA's public key (T_RSA_{PUB}), which is accessible to NW. As we have already discussed above, SecurePay can protect the user from such *man-in-the-mobile* attacks. Moreover, SecurePay can also be used for PC-initiated financial transactions, which are not supported by TrustPAY.

In concurrent work, VButton [25] provides a system for enabling a mobile service provider to verify the authenticity of a user-driven operation originated from an untrusted client device. VButton requires integration at both client- and server-side to validate each user-driven operation. Moreover, it neither provides a Trusted UI indicator nor a secure way to register the public key to the service provider/attestation server. Without a Trusted UI indicator and a secure bootstrap protocol, VButton is susceptible to timing-based and MitB attacks. In the timing-based attack, an attacker can show a different value to the user in the untrusted UI and switch to the Trusted UI with the correct value right before the user confirms the transaction. This can be mitigated with a check by the developer to ensure the user has had enough time in the Trusted UI, but this is not explored in the paper. In the MitB attack, the attacker could register their own public keys with a user's account, either by initiating a binding request themselves or by replacing the public key with their own when the user's binding request is in transit. Furthermore, the lack of a secure registration process also makes VButton vulnerable to relay attack as mentioned in the paper [25]. Compared to VButton, SecurePay's design is simple, complete, practical and formally proven to be secure. SecurePay can even be used as a secure drop-in replacement for existing (insecure) SMS-based 2FA without requiring any code change at the client-side.

Most of the hardware-based solutions [10], [20], [60] that are available in the market also fail to protect users from the aforementioned attacks, because these solutions can only be used to ensure the authenticity of the action and ignore its integrity properties. Moreover, these solutions have a variety of drawbacks. First, most of them are only for PCs. Second, hardware solutions are cumbersome in firmware upgrades. Third, they typically cost tens of dollars per token [61] and, fourth, they are inconvenient to carry around.

ZTIC [9] proposes a hardware-based solution for defending against *man-in-the-middle*. Compared to SecurePay, ZTIC ensures the integrity of PC-initiated banking transactions only. Furthermore, ZTIC requires a predetermined list of banks and additional modifications to the client, such as installing HTTP parsing profiles, user credentials, and X.509 certificates for supporting each new service. SecurePay does not require changes of the client code, requires no extra hardware, and protects both PC-initiated and mobile-initiated transactions.

We depict how SecurePay compares to related solutions using a series of key properties in Table 3. To the best of our knowledge, SecurePay is the only system that (1) requires no change at the client side to support a new financial service, (2) ensures the integrity *and* authenticity of transactions even for fully compromised clients, (3) does not require any additional hardware (beyond the TEE already present in almost all smartphones today), and finally, (4) protects from strong attack vectors such as MitB and MitM.

2FA Soln.	Authenticity	Integrity	MitM	MitB	Mobile	PC	Generic client	No hardware cost
TOTP [13]	✓	✗	✗	✗	✓	✓	✗	✓
TrustPAY [11]	✓	✗	✗	✗	✓	✗	✗	✓
VButton	✓	✓	✗	✗	✓	✗	✗	✓
ZTIC [9]	✓	✓	✗	✓	✗	✓	✗	✗
RSA SecurID [20]	✓	✗	✗	✗	✓	✓	✗	✗
Yubikey [10]	✓	✗	✗	✗	✓	✓	✗	✗
E.dentifier2 [12]	✓	✗	✗	✗	✗	✓	✗	✗
Authenticator [14], [21]	✓	✗	✗	✗	✓	✓	✗	✓
SecurePay	✓	✓	✓	✓	✓	✓	✓	✓

TABLE 3: Comparison: SecurePay ensures the integrity and authenticity of a transaction, protects from *man-in-the-mobile* (MitM) and MitB attacks, supports both PC and mobile platforms, requires no change of the client for supporting a new service provider, and does not require additional hardware.

7. Discussion

We discuss other security aspects of SecurePay.

Availability: SecurePay does not guarantee availability at all. We assume that the mobile app runs on a compromised device. For instance, attackers can simply turn off the code that implements the SecurePay API. In that case, any forthcoming transactions will fail. Even so, no malicious transactions are possible.

Replay attack: We assume that (i) the remote service generates a unique OTP for each transaction request it receives, (ii) the remote service accepts the OTP only once, and (iii) the OTP expires after a short period to protect from replay attack. In Section 6.3 we formally verified that SecurePay is not vulnerable to replay attack.

SIM-jacking: SIM-jacking is an attack where the attacker convinces a victim's carrier to switch victim's phone number over to a SIM card that the attacker owns to bypass the current phone-based 2FA. SIM-jacking attacks have been widely used to hack into social media accounts, steal cryptocurrencies, and break into bank accounts [62]–[64]. SecurePay is not vulnerable to such attacks because SecurePay is not dependent on the SIM card.

Insecure TEEs: SecurePay assumes a secure implementation of TEE. If the TEE implementation has bugs, the attackers can exploit them to steal the private key from the TEE. Orthogonal to this work, formal verification can be used to ensure TEE is free of software bugs [65].

Microarchitectural attacks on TEEs: Defending against microarchitectural attacks on TEE is orthogonal to this research. Currently, SecurePay assumes a secure implementation of TEE. For instance, precautions such as constant-time software and microarchitectural resource flushing are known techniques again cache and speculation attacks. The attacks based on power/voltage glitching can be mitigated by following the standard practice of disallowing access to power/voltage regulators from the normal world. In the case of Rowhammer [18], [19], [66], [67], to the best of our knowledge, there is no real-world attack that can compromise TEE. The only known attack triggers uncontrolled flips in TEE only when normal world's memory is allocated next to TEE. This is almost never the case in real devices (including our test phone). Nevertheless, even this weak attack can be mitigated by adding guard rows [68], [69].

8. Related work

2FA has been used to authenticate and protect financial transactions for many years. Multiple different ways to implement 2FA have been used: SMS-based, software-based, and hardware-based.

The most widely adopted approach nowadays is SMS-based Mobile 2FA[2], probably because it has practical advantage over some other methods in that it requires no additional hardware to store and handle the secondary authentication token. Services using SMS-based 2FA send an OTP and transaction summary in the form of an SMS to the user's mobile device, so that the user can verify the transaction details and confirm the transaction by entering the received OTP.

Recently, the National Institute of Standards and Technology at the US Department of Commerce stated that since SMS messages can be intercepted and redirected, implementers should consider an alternative authentication mechanism [7].

Besides SMS, over the last few years software-based 2FA implementations for authentication and transaction verification have become very popular. Software-based OTPs are usually generated by means of a form of software application. This could be an app running on the smartphone that generates OTPs from the *seed record* along with the device clock and an OTP generating algorithm. Google's *Authenticator* [21] and Microsoft's *Azure Authenticator* [14] are examples of such solutions and can be enabled for dozens of web services like Google, Microsoft Online, WordPress, Joomla, Amazon Web Services, Facebook and Dropbox. However, as we discussed earlier, software-based 2FA solutions cannot protect the user in the scenario where her mobile device is compromised.

As an alternative to such systems, hardware-based 2FA solutions rely on a separate piece of hardware, equipped with a small screen that is capable of generating OTP and displaying it. Today, several hardware-based solutions are available in the market, such as Yubikey [10], RSA SecurID [20] and E.dentifier2 [12]. Hardware-based 2FA is considered to be better than SMS-based and software-based solutions. However, it comes with an additional cost and causes inconvenience. Moreover, these solutions are used for authentication purposes – not for ensuring the integrity of a transaction.

Alexandra et al. [8] analyzed potential attacks against mobile 2FA and provided possible solutions against those attacks. Research [6] has shown how synchronization features and cross-platform services can be used to elevate a regular PC-based Man-in-the-Browser to an accompanying Man-in-the-Mobile threat and bypass SMS-based

2. https://twofactorauth.org/

2FA. Our work provides protection from *all* these attack vectors.

Lenin et al. [70] propose a design for secure e-commerce transactions, but fail to protect from MitM attacks. Note that since the underlying Nizza architecture provides a software-level secure execution domain, one could port SecurePay to it to support any type of electronic commerce application (in addition to AppCore's cart-based ones). Norman et al. [71] demonstrate how to implement a secure graphical user interface to provide isolation between clients to prevent spying on each other, but does not protect users from MitB/MitM. The problem for SecurePay is different, since here we assume the entire (normal world) system may be compromised, including even the operating system kernel, and the objective is to guarantee that the user can distinguish outputs from the trusted app from those of regular programs.

A series of academic efforts involve the development of trusted applications for security solutions. Azab et al. [72] propose a system that provides real-time protection of the OS kernel using TrustZone (TZ). Santos et al. [73] use TrustZone to build a trusted-language runtime to protect the confidentiality and integrity of .NET mobile applications running in the normal world. Li et al. [74] propose a verifiable mobile advertisement framework to detect and prevent advertisement frauds using TrustZone. Marforio et al. [75] propose a location-based second-factor authentication mechanisms for payment at point-of-sale. Pirker at al. [76] propose a framework to protect the privacy of the user when a payment is made by mobile apps. In contrast, our work ensures authenticity and integrity of a transaction–rather than privacy. Truz-Droid [47] proposes a design to integrate the TEE with the mobile operating system to allow any app to leverage the TEE and builds a prototype on a Hikey board. Unlike SecurePay, Truz-Droid requires modification of the operating system, additional hardware support (a LED controlled by the TEE), and still, neither provides an easily adoptable drop-in solution for the banking apps nor supports PC-initiated transactions.

TrustOTP [13] has shown how to convert smartphones into secure OTP tokens. TrustOTP can be used to protect authenticity of any transaction but unlike our work, it does not also protect the integrity of a financial transaction. Moreover, TrustOTP has to be updated with the OTP generating algorithm used by each service provider, while SecurePay is decoupled from the OTP generating algorithm used by the service provider. Meanwhile, TrustPAY [11] proposes a payment system to ensure security and to protect privacy of mobile transactions. However, as we discussed in section 6 in more detail, certain flaws in their design allow the OS running in the normal world to leak the user's private information and to modify transaction details. Similarly, as discussed in the section 6, VButton proposes a system to enable a mobile service provider to verify the authenticity of user-driven operation; however, it lacks the Trusted UI indicator and the secure public-key registration process which are required to protect from timing-based, MitB and relay attacks.

Kellner et al. [77] claim that there is tremendous popularity among regular users for customizing their devices through jailbreaks. Jailbreaks remove vital security mechanisms, which are necessary to ensure a trusted environment that allows to protect sensitive data, such as login credentials and transaction numbers (TANs). The study shows that all but one banking app, available in the App Store, can be fully compromised by trivial means without reverse-engineering, manipulating the app, or other sophisticated attacks. Hence, the study pleads for more advanced defensive measures for protecting user data. The formally verified SecurePay design is a practical solution for this problem.

Finally, regarding the TEE, various vendors offer their own TEEs: OP-TEE [78], Trustonic [79], QSEE [80], SierraTEE [81], T6 [82], and MobiCore [83], and each of these TEEs comes with an SDK which helps developers to build trusted apps for the secure world.

9. Conclusions

In this paper, we explored the risks associated with using a single mobile device for payments, even when enhanced authentication, such as 2FA is in place. We stressed that strong attackers can compromise the potentially vulnerable device and render 2FA completely useless. In parallel, we argued that sensitive applications, such as payment systems, gain limited security with using 2FA for several actions—not just for signing in, but also for issuing sensitive transactions. Following up, we defined a strong threat model, where stealthy attackers compromise smartphones for hijacking user-initiated payments that are otherwise protected with 2FA. We therefore identified the necessary requirements for facilitating a system, that leverages 2FA for securing the user's actions, even when compromised. The key property of our analysis is that 2FA should not be considered for protecting authenticity only, but also for the integrity of individual actions (i.e., the contents of a financial transaction). We, finally, presented SecurePay, a fully working prototype, based on commodity technologies such as ARM's TrustZone, for realizing smartphones that allow users to perform Internet banking (and similar transaction activities) securely, even when their device is compromised.

10. Acknowledgments

We thank the anonymous reviewers for their valuable comments and input to improve the paper. This research was supported by the MALPAY consortium, consisting of the Dutch national police, ING, ABN AMRO, Rabobank, Fox-IT, and TNO. This paper represents the position of the authors and not that of the aforementioned consortium partners. This work further received funding from European Union's Horizon 2020 research and innovation program under grant agreements No. 786669 (ReAct), No. 830929 (CyberSec4Europe), the Netherlands Organisation for Scientific Research under grant agreement 016.Veni.192.262, and the RESTART program of the Research Promotion Foundation, under grant agreement ENTERPRISES/0916/0063 (PERSONAS).

References

[1] J. Gu, V. Zhang, and S. Shen, "ZNIU: First android malware to exploit dirty cow vulnerability," 2017. [Online]. Available: https://blog.trendmicro.com/trendlabs-security-intelligence/zniu-first-android-malware-exploit-dirty-cow-vulnerability/

[2] D. Goodin, "Attackers exploit 0-day vulnerability that gives full control of android phones," 10 2019. [Online]. Available: https://arstechnica.com/information-technology/2019/10/attackers-exploit-0day-vulnerability-that-gives-full-control-of-android-phones/

[3] E. Xu and J. C. Chen, "First active attack exploiting cve-2019-2215 found on google play, linked to sidewinder apt group," 1 2020. [Online]. Available: https://blog.trendmicro.com/trendlabs-security-intelligence/first-active-attack-exploiting-cve-2019-2215-found-on-google-play-linked-to-sidewinder-apt-group/

[4] Board of Governors of the Federal Reserve System, *Consumers and Mobile Financial Services*, 2016, https://www.federalreserve.gov/econresdata/consumers-and-mobile-financial-services-report-201603.pdf.

[5] R. Jones, "Mobile banking on the rise as payment via apps soars by 54% in 2015," 2016. [Online]. Available: https://www.theguardian.com/business/2016/jul/22/mobile-banking-on-the-rise-as-payment-via-apps-soars-by-54-in-2015

[6] R. K. Konoth, V. van der Veen, and H. Bos, "How anywhere computing just killed your phone-based two-factor authentication," in *Proceedings of the* 20^{th} *International Conference on Financial Cryptography and Data Security (FC)*, 2016.

[7] P. A. Grassi, J. L. Fenton, E. M. Newton, R. A. Perlner, and A. R. Rege, "Digital identity guidelines – authentication and lifecycle management," 2017. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-63b

[8] A. Dmitrienko, C. Liebchen, C. Rossow, and A.-R. Sadeghi, "On the (in)security of mobile two-factor authentication," in *Proceedings of the* 18^{th} *International Conference on Financial Cryptography and Data Security (FC)*, 2014.

[9] T. Weigold, T. Kramp, R. Hermann, F. Höring, P. Buhler, and M. Baentsch, "The Zurich trusted information channel – an efficient defence against man-in-the-middle and malicious software attacks," in *Proceedings on the* 1^{st} *International Conference on Trusted Computing - Challenges and Applications (TRUST)*, 2008.

[10] Yubico, "Yubikey." [Online]. Available: https://www.yubico.com/

[11] X. Zheng, L. Yang, J. Ma, G. Shi, and D. Meng, "Trustpay: Trusted mobile payment on security enhanced ARM TrustZone platforms," in *Proceedings of the* 21^{st} *IEEE Symposium on Computers and Communication (ISCC)*, 2016.

[12] ABN AMRO, "E.dentifier2." [Online]. Available: https://www.abnamro.nl/en/images/Generiek/PDFs/Overig/edentifier2_usermanual_english.pdf

[13] H. Sun, K. Sun, Y. Wang, and J. Jing, "Trustotp: Transforming smartphones into secure one-time password tokens," in *Proceedings of the* 22^{nd} *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[14] Microsoft, "Microsoft Authenticator." [Online]. Available: https://www.microsoft.com/en-us/store/p/microsoft-authenticator/9nblgggzmcj6

[15] H. Meng, V. L. L. Thing, Y. Cheng, Z. Dai, and L. Zhang, "A survey of android exploits in the wild," in *Computers & Security*, 2018.

[16] CVE-2016-5195, "A privilege escalation vulnerability in the linux kernel," 2019. [Online]. Available: https://dirtycow.ninja/

[17] Github Repository, "A collection of android exploits," 2018. [Online]. Available: https://github.com/sundaysec/Android-Exploits

[18] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the* 23^{rd} *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[19] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. P. Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "GuardION: Practical mitigation of dma-based rowhammer attacks on ARM," in *Proceedings of the* 15^{th} *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2018.

[20] EMC^2, "RSA SecureID Hardware Tokens." [Online]. Available: https://www.rsa.com/en-us/products/identity-and-access-management/securid-hardware-tokens

[21] Google, "Google Authenticator." [Online]. Available: https://github.com/google/google-authenticator

[22] J. H. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 9 1975.

[23] C. Herley and D. Florencio, "How to login from an internet café without worrying about keyloggers," in *Proceedings of the* 2^{nd} *Symposium on Usable Privacy and Security (SOUPS)*, 2006.

[24] J. Hong, "The state of phishing attacks," *Communications of the ACM*, vol. 55, no. 1, pp. 74—-81, 1 2012.

[25] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan, "VBtton: Practical attestation of user-driven operations in mobile apps," in *Proceedings of the* 16^{th} *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.

[26] D. A. Ortiz-Yepes, R. J. Hermann, H. Steinauer, and P. Buhlerr, "Bringing strong authentication and transaction security to the realm of mobile devices," in *IBM Journal of Research and Development*, 2014.

[27] R. van Rijswijk-Deij and E. Poll, "Using trusted execution environments in two-factor authentication: comparing approaches," in *Proceedings of the* 1^{st} *Open Identity Summit (OID)*, 2013.

[28] F. Zhang and H. Zhang, "SoK: A study of using hardware-assisted isolated execution environments for security," in *Proceedings of the* 5^{th} *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2016.

[29] P. Gullberg, "Trusted execution environment, trustzone and mobile security," *OWASP göteborg: security tapas*, 2015.

[30] GlobalPlatform, *TEE System Architecture Version 1.1.0.10*, 2018, GPD_SPE_009.

[31] S. A. Bailey, D. Felton, V. Galindo, F. Hauswirth, J. Hirvimies, M. Fokle, F. Morenius, C. Colas, J.-P. Galvan, G. Bernabeu *et al.*, "The trusted execution environment: Delivering enhanced security at a lower cost to the mobile market," GlobalPlatform Device Technology, Tech. Rep., 2011.

[32] J.-E. Ekberg, K. Kostiainen, and N. Asokan, "The untapped potential of trusted execution environments on mobile devices," in *Proceedings of the* 35^{th} *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[33] GlobalPlatform, *TEE Client API Specification Version 1.0*, 2010, GPD_SPE_007.

[34] GlobalPlatform, *TEE Internal API Specification Version 1.0*, 2014, GPD_EPR_017.

[35] T. Alves and D. Felton, "TrustZone: Integrated Hardware and Software Security," Tech. Rep., 2004.

[36] J. Winter, "Experimenting with ARM TrustZone – or: How i met friendly piece of trusted hardware," in *Proceedings of the* 11^{th} *International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2012.

[37] ARM, "ARM security technology: Building a secure system using TrustZone technology," Tech. Rep., 2009.

[38] J. J. Drake, "Google Android - "Stagefright" Remote Code Execution (CVE-2015-1538)," 2015. [Online]. Available: https://www.exploit-db.com/exploits/38124

[39] D. Vyukov, "Use-After-Free Remote Code Execution Vulnerability (CVE-2016-7117)," 2016. [Online]. Available: https://www.exploit-db.com/exploits/38124

[40] G. Gogniat, T. Wolf, W. Burleson, J. P. Diguet, L. Bossuet, and R. Vaslin, "Reconfigurable hardware for high-security/ high-performance embedded systems: The SAFES perspective," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, pp. 144–155, 2 2008.

[41] FuturePlus System, *DDR2 800 bus analysis probe*, 2016, https://www.yumpu.com/en/document/view/41592980/fs2334-ddr2-800-mt-s-dimm-analysis-probe-futureplus-systems.

[42] T. Müller and M. Spreitzenbarth, "FROST: forensic recovery of scrambled telephones," in *Proceedings of the* 11^{th} *International Conference on Applied Cryptography and Network Security (ACNS)*, 2013.

[43] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold boot attacks on encryption keys," in *Proceedings of the* 17^{th} *USENIX Security Symposium (USENIX SEC)*, 2008.

[44] Samsung, *Trusted Boot*, https : / / docs.samsungknox.com / whitepapers/knox-platform/trusted-boot.htm.

[45] N. Johnson, "Qualcomm's chain of trust," 2018. [Online]. Available: https://lineageos.org/engineering/Qualcomm-Firmware/

[46] G. A. Miller, "The magical number seven, plus or minus two: some limits on our capacity for processing information," *Psychological review*, vol. 101, no. 2, p. 343, 4 1994.

[47] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du, "TruZ-Droid: Integrating trustzone with mobile operating system," in *Proceedings of the* 16^{th} *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.

[48] C. Castillo, "Phishing attack replaces android banking apps with malware," 2013. [Online]. Available: https://www.mcafee.com/blogs/other-blogs/mcafee-labs/phishing-attack-replaces-android-banking-apps-with-malware/

[49] F. Assolini, "SMiShing and the rise of mobile banking attacks," 2016. [Online]. Available: https://securelist.com/smishing-and-the-rise-of-mobile-banking-attacks/75575/

[50] J. Wyke, "What is Zeus?" 2011. [Online]. Available: https://www.sophos.com/en-us/medialibrary/PDFs/technical%20papers/Sophos%20what%20is%20zeus%20tp.pdf

[51] Symantec, "Dyre: Emerging threat on financial fraud landscape," Symantec, Tech. Rep., 2015.

[52] Dick O'Brien, "Dridex: Tidal waves of spam pushing dangerous financial trojan," Symantec, Tech. Rep., 2016.

[53] L. Kharouni, "Automating online banking fraud," 2012. [Online]. Available: https://www.trendmicro.de/cloud-content/us/pdfs/security-intelligence/white-papers/wp_automating_online_banking_fraud.pdf

[54] P. Gühring, "Concepts against man-in-the-browser attacks," 2006. [Online]. Available: http://www.cacert.at/svn/sourcerer/CAcert/SecureClient.pdf

[55] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin, "The tamarin prover for the symbolic analysis of security protocols," in *Proceedings of the* 25^{th} *International Conference on Computer Aided Verification (CAV)*, 2013.

[56] D. Dolev and A. C.-C. Yao, "On the security of public key protocols," in *Proceedings of the* 22^{nd} *Symposium on Foundations of Computer Science (SFCS)*, 1981.

[57] GlobalPlatform, "Certified products." [Online]. Available: https://globalplatform.org/certified-products/

[58] Open source Android apps in Github, 2016. [Online]. Available: https://github.com/pcqpcq/open-source-android-apps

[59] Github, "The world's leading software development platform." [Online]. Available: https://github.com/

[60] A. Blom, G. de Koning Gans, E. Poll, J. de Ruiter, and R. Verdult, "Designed to fail: A usb-connected reader for online banking," in *Proceedings of the* 17^{th} *Nordic conference on Secure IT Systems (NordSec)*, 2012.

[61] ENCAP Security, "SMiShing and the rise of mobile banking attacks," 2016. [Online]. Available: https://securelist.com/smishing-and-the-rise-of-mobile-banking-attacks/75575/

[62] L. Franceschi-Bicchierai, "How criminals recruit telecom employees to help them hijack sim cards," 2018. [Online]. Available: https://www.vice.com/en_us/article/3ky5a5/criminals-recruit-telecom-employeessim-swapping-port-out-scam

[63] B. Barrett, "How to protect yourself against a sim swap attack," 2018. [Online]. Available: https://www.wired.com/story/sim-swap-attack-defend-phone/

[64] B. Krebs, "Busting sim swappers and sim swap myths," 2018. [Online]. Available: https://krebsonsecurity.com/2018/11/busting-sim-swappers-and-sim-swap-myths/

[65] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "SeL4: Formal Verification of an OS Kernel," in *SOSP*, 2009.

[66] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand pwning unit: Accelerating microarchitectural attacks with the gpu," in *Proceedings of the* 39^{th} *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[67] P. Frigo, E. Vannacci, H. Hassan, V. van der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi, "TRRespass: Exploiting the Many Sides of Target Row Refresh," in *S&P*, 2020.

[68] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A. Sadeghi, "CAn't Touch This: Software-only mitigation against rowhammer attacks targeting kernel memory," in *Proceedings of the* 26^{th} *USENIX Security Symposium (USENIX SEC)*, 8 2016.

[69] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks," in *OSDI*, Oct. 2018.

[70] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth, "Reducing tcb complexity for security-sensitive applications: three case studies," in *Proceedings of the* 1^{st} *ACM European Conference on Computer Systems (EuroSys)*, 2006.

[71] N. Feske and C. Helmuth, "A nitpicker's guide to a minimal-complexity secure gui," in *Proceedings of the* 21^{nd} *Annual Computer Security Applications Conference (ACSAC)*, 2005.

[72] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Martin, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world," in *Proceedings of the* 21^{st} *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[73] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM TrustZone to build a trusted language runtime for mobile applications," in *Proceedings of the* 19^{th} *international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2014.

[74] W. Li, H. Li, H. Chen, and Y. Xia, "AdAttester: Secure online mobile advertisement attestation using trustzone," in *Proceedings of the* 13^{th} *Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2015.

[75] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Capkun, "Smartphones as practical and secure location verification tokens for payments," in *Proceedings of the* 21^{st} *Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[76] M. Pirker and D. Slamanig, "A framework for privacy-preserving mobile payment on security enhanced ARM TrustZone platforms," in *Proceedings of the* 11^{th} *International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2012.

[77] A. Kellner, M. Horlboge, K. Rieck, and C. Wressnegger, "False sense of security: A study on the effectivity of jailbreak detection in banking apps," in *Proceedings of the* 4^{th} *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[78] Linaro, "OP-TEE." [Online]. Available: https://www.op-tee.org/

[79] Trustonic, "What is a Trusted Execution Environment (TEE)?" [Online]. Available: https://www.trustonic.com/news/technology/what-is-a-trusted-execution-environment-tee/

[80] Qualcomm, "Qualcomm Trusted Execution Environment (QSEE)." [Online]. Available: https://www.qualcomm.com/

[81] Sierraware, "SierraTEE Trusted Execution Environment." [Online]. Available: https://www.sierraware.com/open-source-ARM-TrustZone.html

[82] TrustKernel, "T6." [Online]. Available: https : // www.trustkernel.com/products/tee/t6.html

[83] Giesecke and Devrient, "MobiCore." [Online]. Available: https://www.gi-de.com/

Appendix A.
SecurePay's TAMARIN model and its security properties

```
theory secpay
begin

builtins: asymmetric-encryption, hashing


rule trusted_zone_generates_keypair:
  [
    Fr(~privateKey)
  ]
  --[ NewDevice(~privateKey) ]->
  [
    !TrustedZonePrivateKey(~privateKey),
    !TrustedZonePublicKey(pk(~privateKey)),

    // we assume that the public key is leaked
    Out(pk(~privateKey))
  ]

rule human_opens_account:
  [
    // user may open arbitrary many accounts
  ]
  --[ NewAccount($accountNr) ]->
  [
    !UserAccount($accountNr)
  ]

rule human_initiates_binding_request:
  [
    !TrustedZonePublicKey(publicKey),
    !UserAccount(accountNr)
  ]
  --[ BindingRequestInitiated(publicKey) ]->
  [
    ShowPublicKey(publicKey, accountNr)  // physically show the public key at a
                                         // secure terminal (in form of QR code)
  ]



rule bank_receives_binding_request:
  [
    ShowPublicKey(key, accountNr)
  ]
  --[ BankCompletesBindingRequest(key) ]->
  [
    !PublicKeyForAccount(accountNr, key)
  ]


rule human_initiates_transaction:
  let
    // we assume that transaction_details might be deducible
    transaction = <accountNr, $transaction_details>
  in
    [
      !UserAccount(accountNr)
    ]
  --[ HonestTransactionInitiated(transaction) ]->
    [
      HumanWaitsForTrustedDisplay(transaction),
      Out(transaction)
    ]



rule bank_new_transaction:
  let
    transaction = <from_account, transaction_details>
  in
    [
      In(transaction),
      Fr(~otp),
```

```
      !PublicKeyForAccount(from_account, pkAccount)
    ]
  --[ BankTransactionInitiated(transaction, ~otp) ]->
    [
      Out(aenc{'transaction', ~otp, transaction}pkAccount),
      BankWaitsForOTP(<~otp, transaction>)
    ]


// this rule is used for binding requests and transactions
rule trusted_zone_display_summary:
  let
    decryptedMessage = adec{decryptRequest}privateKey
  in
      [
        In(decryptRequest),
        !TrustedZonePrivateKey(privateKey)
      ]
  --[ TrustedZoneSummaryDisplayed(decryptedMessage) ]->
      [
        DisplayTransactionOnSecureScreen(decryptedMessage)
      ]



rule human_confirms_transaction:
    [
      DisplayTransactionOnSecureScreen(<'transaction', otp, <accountNr, transaction_details>>),
      HumanWaitsForTrustedDisplay(<accountNr, transaction_details>)
    ]
  --[ HumanConfirmsTransaction(accountNr, transaction_details) ]->
    [ Out(otp) ]



rule bank_completes_transaction:
  [
    In(otp),
    BankWaitsForOTP(<otp, transaction>)
  ]
  --[ TransactionCompleted(transaction) ]->
  []

//
// source lemma
// required for termination
//
//

lemma types [sources]:
  "(All otp user transaction_details #i. TrustedZoneSummaryDisplayed(<'transaction', otp, user,
     transaction_details>) @i
    ==>
    (
      (Ex #j. BankTransactionInitiated(<user, transaction_details>, otp) @j)  |
      (Ex #j. KU(otp) @ j & j < i)
    )
  )"


//
// security property
//
//

lemma all_accepted_transactions_must_come_from_human:
  "All from details #i. TransactionCompleted(<from, details>) @i ==> (Ex #j.
    HonestTransactionInitiated(<from, details>) @j)"

lemma replay_attack_not_possible:
  "All from1 details1 from2 details2 #i #j. TransactionCompleted(<from1, details1>) @i &
    TransactionCompleted(<from2, details2>) @j & not #i = #j ==> (Ex #k #l.
    HonestTransactionInitiated(<from1, details1>) @k & HonestTransactionInitiated(<from2,
    details2>) @l & not #k = #l)"



//
```

```
// sanity checks for binding
//
//

#ifdef SANITY
lemma new_devices_can_be_created:
  exists-trace
    "Ex x #i. NewDevice(x) @#i"



lemma human_can_initiate_binding_request:
  exists-trace
    "Ex x #i. BindingRequestInitiated(x) @#i"


lemma bank_can_complete_binding_request:
  exists-trace
    "Ex x #i. BankCompletesBindingRequest(x) @#i"
#endif


//
// sanity checks of transaction
//
//

#ifdef SANITY
  lemma human_can_initiate_transaction:
    exists-trace
      "Ex x #i. HonestTransactionInitiated(x) @#i"


  lemma human_can_confirm_transaction:
    exists-trace
      "Ex x y #i. HumanConfirmsTransaction(x,y) @#i"


  lemma bank_honest_transaction:
    exists-trace
      "All n m #i. BankTransactionInitiated(n, m) @i ==> (Ex #j. HonestTransactionInitiated(n)
    @j)"


  lemma transaction_can_complete:
    exists-trace
      "Ex x #i. TransactionCompleted(x) @i"


  lemma two_different_transactions_can_complete:
    exists-trace
      "Ex x y #i #j. TransactionCompleted(x) @#i & TransactionCompleted(y) @#j & (not (x = y))"
#endif

end
```